Model-Based GN&C Simulation and Flight Software Development for Orion Missions beyond LEO

Ryan Odegard
Draper Laboratory
17629 El Camino Real
Houston, TX 77578
rodegard@draper.com

Zoran Milenkovic
Draper Laboratory
1555 Wilson Blvd, Suite 501
Arlington, VA 22209
zmilenkovic@draper.com

Joel Henry
NASA Johnson Space Center
2101 NASA Parkway
Houston, TX 77058
joel.r.henry@nasa.gov

Michael Buttacoli
NASA Johnson Space Center
2101 NASA Parkway
Houston, TX 77058
michael.buttacoli@nasa.gov

Abstract—**For Orion missions beyond low Earth orbit (LEO), the Guidance, Navigation, and Control (GN&C) system is being developed using a model-based approach for simulation and flight software. Lessons learned from the development of GN&C algorithms and flight software for the Orion Exploration Flight Test One (EFT-1) vehicle have been applied to the development of further capabilities for Orion GN&C beyond EFT-1. Continuing the use of a Model-Based Development (MBD) approach with the Matlab®/Simulink® tool suite, the process for GN&C development and analysis has been largely improved. Furthermore, a model-based simulation environment in Simulink, rather than an external C-based simulation, greatly eases the process for development of flight algorithms. The benefits seen by employing lessons learned from EFT-1 are described, as well as the approach for implementing additional MBD techniques. Also detailed are the key enablers for improvements to the MBD process, including enhanced configuration management techniques for model-based software systems, automated code and artifact generation, and automated testing and integration.**

TABLE OF CONTENTS

1. INTRODUCTION

The Orion spacecraft is being developed by NASA to take humans back to deep space. The first test mission, EFT-1, will demonstrate many capabilities for extending the human presence in space beyond LEO. For the Orion GN&C flight software (FSW), which is responsible for navigating, guiding, and controlling the vehicle, a model-based development process has been implemented. Advantages of MBD in human spaceflight algorithm development on Orion have been previously described [1, 2]. Specifically, distinct benefits from MBD have been realized to date on Orion: removal of the steps of converting GN&C algorithmic pseudo-code documentation to flight software, increases in the amount of time spent exercising flight code, improvements in the processes for producing algorithm documentation and reviewing algorithm functionality, implementation of automated checking of software standards, and development of an automated testing framework and report generation.

As part of that EFT-1 FSW development process, several challenges and areas of improvement were identified. These included configuration management difficulties, code integration and graphical merging shortcomings, development complexities related to the simulation environment, slow simulation execution speed, efficiency of generated code, and long model build times. To address the challenges and lessons learned from EFT-1, a number of improvements have been implemented. The first has been to develop a medium fidelity, model-based, simulated 6-degree-of-freedom (DOF) environment using Simulink [3] in which to exercise flight algorithms. The past use of C-based simulation environments required additional overhead and complicated interfaces to the Simulink environment. The Simulink simulation is also tailored to on-orbit flight regimes, which allows faster execution speeds. Another advantage has been the use of Simulink model variants, which improves build time and allows quickly exercising different functionality without modifying the models. A collection of process improvements has been implemented to address the challenges of configuration management and

978-1-4799-1622-1/14/$31.00 ©2014 IEEE

code development processes. The process improvements have served as enablers for the Simulink model development approach for GN&C. Centralization to a NASA repository has proven beneficial and accessible by various team members. The use of the git [4] distributed version control tool has demonstrated great flexibility as a revision control system tool. It has also simplified the work-flow and removed dependencies on a host of custom scripts used for EFT-1 development. A number of automation capabilities have been utilized to speed up the software development process. Central to this is the use of an automated build and test server, which has enabled many improvements over EFT-1 in the ability to automatically and continuously test the models and the generated code. Other improved automation processes include change request integration, regression and unit testing, code generation, build artifact generation, and metrics collection. Previous challenges in performing three-way model merges have been alleviated by the use of recent capabilities in Simulink model merging tools, as well as process improvements for communicating conflicts among team members.

The following is a summary of challenges encountered during EFT-1 software development that are addressed in this paper.

Challenge: Provide an environment where model-based FSW can be developed quickly and without TCP/IP interfaces between C and Simulink.
Challenge: Enhance collaboration between geographically-dispersed teams operating on different secure networks.
Challenge: Speed up model development.
Challenge: Implement a usable graphical merging capability for Simulink models.
Challenge: Ease the burden of integrating multiple branches of development at once, leading to long integration cycles.
Challenge: Speed up and automate testing and integration for individual developers.
Challenge: Maintain and manage a working, compiled, and testable version of the auto-generated FSW code concurrently with model development.

2. SIMULINK ON-ORBIT SIMULATION

The primary improvement in the model-based development process for the Orion GN&C FSW development has been to build a medium fidelity, model-based simulation environment using Simulink. This differs from the previous approach for the EFT-1 GN&C development that used high fidelity, C-based simulations. For development prior to the preliminary design review (PDR), the medium fidelity simulation allows faster development while maintaining adequate accuracy for the maturity required. Furthermore, running the C-based simulation with Simulink requires multiple processes communicating via TCP/IP socket connections, which adds complexity and I/O overhead to the simulation execution process. There are initialization steps on both the simulation side and the Simulink side that take minutes to execute prior to each simulation run. Running both the C-based simulation and the Simulink models also requires developers to be proficient in multiple programming languages for development and debugging, whereas a Simulink only environment alleviates that requirement to a large degree. This allows for designers to better understand each other's work, provides the ability to share developer resources if needed, fosters more opportunities for reuse across the team, and provides a common look and feel to the subsystem design that benefits the review and code audit steps in the verification and validation process. Also, engineers new to the project do not need significant training on initialization scripts and simulation tools prior to starting algorithm development. Overall, having the simulation models in Simulink reduces the complexity of interfacing with the FSW models, streamlining and simplifying the simulation initialization and execution processes.

The Simulink-based simulation has been dubbed FLASHE (FLight Algorithm Simulation Environment), and the collection of flight software models is called RAMSES (Rapid Algorithm Matlab/Simulink Engineering Simulation). Figure 1 shows the top level Simulink model that contains the FLASHE simulation and the RAMSES flight software models. Each has a corresponding data logging block for recording output variables to file for a simulation run.

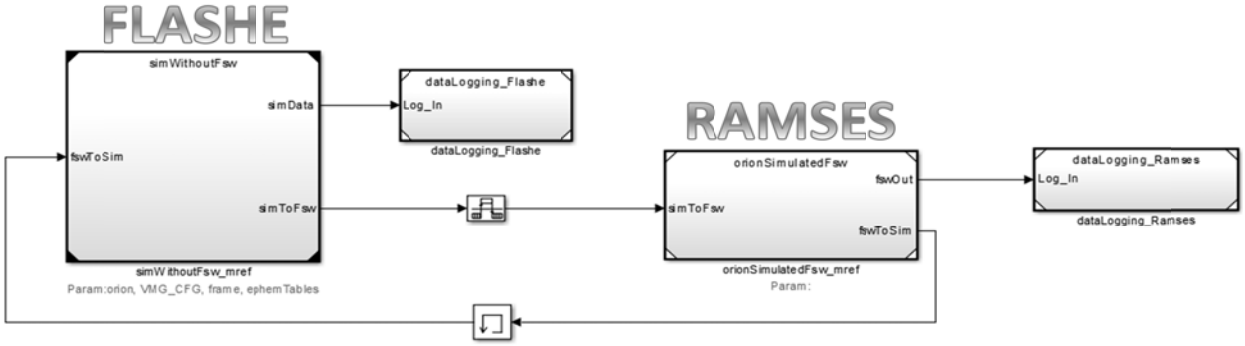

Figure 1. Top-level simulation model.

FLASHE includes a number of models that represent the Orion vehicle and its environment. These include dynamics models to represent the effects of gravitational forces, vehicle forces and torques from thruster and engine firings, atmospheric and aerodynamic effects, and solar pressure effects for quiescent flight outside LEO. Simulink provides a variety of options for performing differential equation integration, including such solvers as ode15s, ode23tb, ode45, etc., with both fixed-step and variable-step options. Because the RAMSES FSW models are required to execute as a fixed rate, as they would on the vehicle target processor, a custom, 4th order Runge-Kutta integration method is implemented in the vehicle dynamics and found to be sufficient in terms of accuracy and computational speed.

There are also models that represent the sensors on the Orion vehicle used for navigation purposes, such as Inertial Measurement Units (IMUs), Global Positioning System Receivers (GPSRs), Star Trackers (STs), and Barometric Altimeters (BAROs). Mostly inherited from development done on EFT-1, these models emulate the sensors on the vehicle, which the navigation software uses to estimate the vehicle state.

The effector models in FLASHE include a Reaction Control System (RCS) thruster model, Auxiliary (Aux) jet model, and a main engine model with Thrust Vector Control (TVC). The Simulink implementation of the RCS and Aux models reuse the same model reference block by passing in different model reference parameters that specify the thrust locations, directions, and magnitudes. The RCS model is also reused between the Orion Service Module (SM) and Crew Module (CM) models. The main engine and TVC models represent the expected behavior and performance when Orion executes large magnitude burns.

Finally, FLASHE has also been configured with a simple target vehicle model to simulate rendezvous and docking with another spacecraft.

All of the model references in FLASHE utilize model reference parameters that are collected in a workspace structure organized hierarchically that corresponds to the model hierarchy. The parameters as passed into the models can be changed between runs without requiring recompilation of the models.

Data Logging

Logging of data in Simulink has been a challenge for Orion GN&C development. The lack of data configurable logging capabilities in Simulink led to a custom file-writing model for EFT-1. This model can be reconfigured to allow logging Simulink model output data to file. The two main advantages are that it does not require changes to the algorithmic models themselves in order to alter what data signals are logged, and it writes data to disk during the run. Native Simulink capabilities do not support either of these options. For long simulation runs with many models, it is critical to be able to configure which data are logged, because there is a significant impact on run speed when large amounts of data are logged during the run. For the more recent FLASHE and RAMSES development, an alternate approach is taken primarily due to the length of time required to update the custom data logging model. With recent support for 64-bit MATLAB installations, it was decided to focus less on writing data to disk during the run, and develop a fast, reconfigurable data logging capability utilizing native Simulink capabilities. This approach relies on regeneration of a Simulink model in which the Simulink logging specifications are set programmatically for both FLASHE and RAMSES outputs. The variables to log and the desired rates are set in one or more text-based data record files, and the logging model can be regenerated within seconds. This is an improvement over the previous custom recording model, which takes several minutes to rebuild. The data record files containing the signals to log can be either leaf-level elements or bus signals, and can support arrays of bus signals, as of MATLAB 2013a.

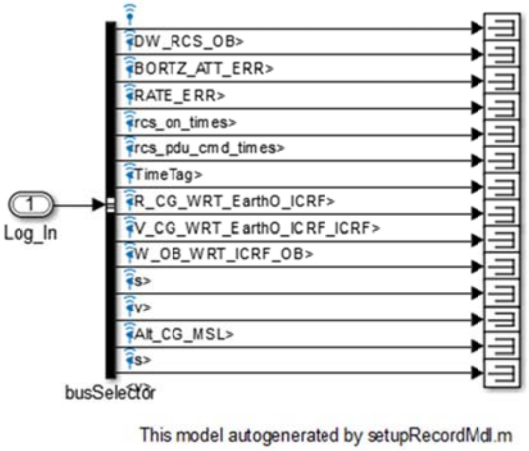

Figure 2. Automatically generated logging model based on configurable signals and recording rates.

3. SIMULINK VARIANTS

One key feature of Simulink that has been leveraged in model development for FLASHE and RAMSES is the use of model reference variants. Introduced in MATLAB version 2009A, model variants provide a data-driven method for reconfiguring a model with a different implementation. For the Orion GN&C project, model variants are used to easily switch between models without modifying blocks or interfaces. For example, one variant of a gravity model implements a point-mass, a second implements a gravity model that accounts for the oblateness of the Earth (J2 effect), and a third implements higher-order gravity effects. By setting variables in the MATLAB workspace, a conditional setting of the Simulink block determines which variant gets executed.

The primary way in which model variants have been used is to create empty "stub" reference blocks for some of the models. This allows the exclusion of a set of code from being compiled and run for a certain simulation. The main use of this concept in FLASHE is the creation of stub sensor models. Instead of representing how the sensor hardware behaves according to the environmental conditions and dynamics, the stub models merely pass through the "true" state of the vehicle, as calculated in the dynamics models. When the details of the performance of the sensors is not important for a certain set of GN&C analysis, then this "perfect navigation" setup saves the developer time in building (and rebuilding) the complex sensor models during development.

Another variant implementation in FLASHE has been two alternatives for modeling the planetary ephemeris information. The first variant uses a MATLAB implementation of the SPICE system [5], computed during the simulation, for establishing planetary body ephemerides. A second variant uses a table lookup routine that removes any run-time dependence on the SPICE libraries. This offers flexibility in running lower overhead or higher fidelity models with just the change of a variable in the MATLAB workspace.

On the RAMSES side, model variants have been used to include model code in the project repository for all Orion phases of flight, but not require compilation and execution of all the code for simulation runs. During previous iterations of development prior to model variants, the Simulink architecture for integrating GN&C code for flight phases ranging from ascent to on-orbit to entry and landing proved challenging from the perspectives of interface management and build and execution speed. Even quiescent on-orbit simulations carried the overhead of atmospheric algorithms designed for highly dynamic periods. With model variants, simulations involving flights around the moon do not require the inclusion of models that are built for ascent abort logic and control, nor entry, descent, and landing algorithms. Yet the overall architecture is maintained and managed easily because the interfaces between those models are unaltered with the use of "stub" model variants. From a flight software integration and maintenance perspective, this has proven to be of great benefit.

4. FLIGHT SOFTWARE MODELING

The GN&C flight software is broken up into a series of functional "domains" that compartmentalize portions of the logic and code. The algorithms for GN&C are specific to a given functional domain, so each is grouped into one of the domains shown in Table 1.

Table 1. List of GN&C Flight Software Domains

Flight Software Domain Abbreviation	Description
CNC	Crew Module Controls
CNL	Launch Abort System Controls
CNE	Engine Controls
CNP	Propulsion System Controls
CNS	Service Module Controls
GDA	Ascent Guidance
GDE	Entry Guidance
GDO	Orbit Guidance
GMP	GN&C Mass Properties
NVA	Absolute Navigation
NVE	Ephemeris Navigation
NVR*	Relative Navigation

* NVR is not implemented as rendezvous and docking are not planned for near term Orion missions.

The flight algorithms for each of these domains are developed in Simulink with the exception of the CNE and CNP domains, which are written directly in C++. The rest of the domains are integrated into the RAMSES modeling for the development and testing of those flight algorithms. As shown in Figure 3, the "integrated" model variant for RAMSES consists of all of these domains.

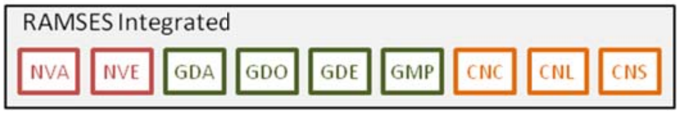

Figure 3. RAMSES "Integrated" variant inclusive of all the GN&C FSW.

As mentioned in the previous section, the use of model variants in the modeling of the GN&C flight software in Simulink has allowed greater fine-tuning of flight-phase-specific simulation performance. Specifically, much of the on-going work for Orion is geared toward further exploration of the Moon and beyond, which includes primarily on-orbit GN&C functionality. As a result, much of the development and analysis is independent of the algorithms related to atmospheric flight. Therefore, an "orbit" model variant is established, which includes the domains shown in Figure 4. By using the orbit model variant, there are 73 fewer models that need to be built and run in the simulation.

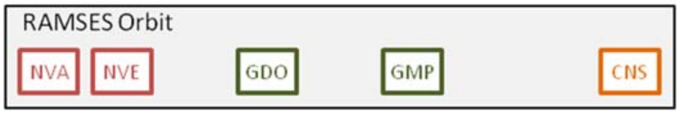

Figure 4. RAMSES "Orbit" variant inclusive of only the on-orbit GN&C FSW.

In addition to the "integrated" and "orbit" variants of the FSW, there is also a big advantage in being able to run perfect navigation. As mentioned in the previous section describing the FLASHE variants for stubbing out the sensor models, perfect navigation can also be used to exclude the navigation flight software, which comprises approximately 60% of the total GN&C code. In effect, the navigation state sent to the guidance and controls domains represents the "truth" state, rather than the state that would be estimated on-board from incorporating measurements from the navigation sensors. Not only does perfect navigation provide time savings when compared to full absolute navigation, but perfect navigation capability is essential for development and analysis of guidance and control algorithms.

There are actually two additional navigation-related variants that can be exercised in FLASHE/RAMSES. One is to use the perfect navigation state in the guidance and controls FSW, but to run the sensors and navigation algorithms in parallel. This allows assessment of the navigation performance but without effect on the control of the vehicle. Secondly, a stochastic navigation variant is setup, which provides a means of non-deterministically perturbing the true state to represent the type of estimation done by the navigation flight software. This implementation provides an incrementally higher level of fidelity for guidance and controls analysis, without the need to include the navigation sensors or flight software models in the simulation. Table 2 summarizes the navigation variants used in FLASHE/RAMSES. By using perfect navigation, there are 168 fewer models that need to be built and run in the simulation.

Table 2. Summary of navigation model variants.

Nav Variant Name	Active Nav Pathway (to G&C)	Passive Nav Pathway (for telem)	Sensor Models Used?	Description
PerfectNav	Perfect	n/a	Off	Perfect Nav
PerfectNav_ NVAProcessing	Perfect	NVA	On	NVA and sensors running for telemetry
RealNav	NVA	Perfect	On	NVA running and used by G&C
StochasticNav	Stochastic	Perfect	Off	Truth state perturbed

Figure 5 shows the switching logic implementation in Simulink for configuring the perfect navigation variant in RAMSES.

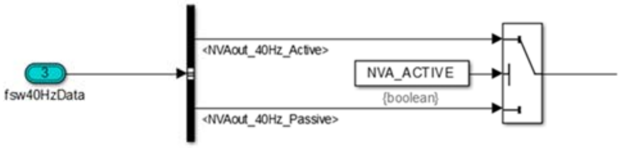

Figure 5. Perfect navigation switching.

The other benefit of being able to integrate all of the flight software models into the same RAMSES architecture is for the common GN&C executive software that controls the sequencing and execution of all of the GN&C domains. Within the GN&C Command Interface (GCI) flight software domain lives the executive sequencer and GN&C command handler software. This domain is responsible for triggering and configuring the appropriate algorithms at the appropriate time, based on data-configurable input files. Because this software executes during all phases of flight, it has been crucial to be able to maintain a common GCI code base for the integrated set of FSW code. If, alternatively, there were separate repositories or Simulink implementations for different flight phases, it would be stressing on the maintenance of the common GCI software.

Another set of common code that has benefitted from the model-based design approach using Simulink model variants is the tool set that is used to data-configure GCI, known as the Phases, Segments, Activities, and Modes (PSAM) Tool [6]. The acronym PSAM refers to the hierarchical portions of the mission timeline. The major divisions of the mission are the *phases*, such as ascent, on-orbit, and entry. The phases are subdivided into *segments*, which comprise vehicle-level goals, such as performing an on-orbit burn, or coasting in a particular configuration. Within the GN&C subsystem there are specific objectives within a segment that are referred to as *activities*. Within a burn segment, the GN&C activities include maneuvering to the burn attitude, performing a main engine burn, executing a trim burn if necessary, and assuming a post-burn attitude. For each activity, the GN&C domains are in specific *modes* that control which algorithms are executing. Furthermore, the configuration parameters for the algorithms, such as thresholds, limits, deadbands, etc., are specified for a given activity. The GN&C activities, modes, and parameters are sets of data that are configured with the aid of the PSAM Tool, and loaded into GCI for execution throughout the mission. For an integrated set of RAMSES FSW models, the maintenance of the PSAM Tool enjoys the same benefits of a common code base as does the GCI domain.

5. COMPATIBILITY WITH EXTERNAL SIMS

While the FLASHE simulation environment has shown numerous benefits in doing rapid FSW development for orbit algorithms, more formal analysis and testing after PDR for the Exploration Mission-1 (EM-1) to the moon will require other higher fidelity C-based simulations. For this reason, and because FLASHE is not intended to support ascent or descent and landing phases of flight, an interface to NASA's Trick Simulation Environment [7] has been updated to work with the 2013a version of MATLAB. Data being passed between the Trick simulation and the FSW in Simulink are exchanged via TCP/IP socket connections using custom s-functions, as was done for EFT-1.

There are two main aspects of the interface between Trick and RAMSES. The first is that the input/output data must be maintained, and can then support multiple other Trick-based simulations. There are likely two Trick-based simulations that will require interfacing with the GN&C FSW in RAMSES, so the development in Simulink and with FLASHE will ensure the Trick interface is maintained. The current FLASHE regression testing includes a check that these interfaces do not break as development progresses.

The second main aspect of the RAMSES interface to the Trick simulations is that the FSW functionality remains consistent across simulation environments. The FLASHE regression testing is also exercising a subset of the functionality that is unique to the Trick simulation, in that it is part of an ascent profile which is not the primary role of FLASHE. Thus, the GN&C FSW, as developed in RAMSES, is maintained for operation in both Simulink- and Trick-based simulation environments.

6. CONFIGURATION MANAGEMENT

One of the biggest challenges in the Orion GN&C FSW development, both before PDR and leading up to the EFT-1 flight test, was configuration management. While MBD affords many benefits over the course of a project, having a large, geographically dispersed team across many organizations makes working with graphical models particularly challenging. The first hurdle in working with many organizations is dealing with separate information technology (IT) and security requirements; finding a method in which all team members can share information has never been a trivial problem. Secondly, the set of tools for managing the work of many team members must be both robust and easy to use in order to facilitate deploying flight software. While solutions were found for both of these problems for EFT-1, they were not ideal and recent work has largely improved these aspects of the GN&C development process.

Members of the Mathworks team have described configuration management techniques in the context of Simulink model development [8]. The Orion GN&C team, however, has learned two important lessons for successfully carrying out Simulink model development with dispersed organizations: establish easy access to a configuration controlled repository and utilize tools that ease distributed development.

Simulation model and flight software development for EM-1 has utilized servers at NASA's Johnson Space Center (JSC) to centralize code and model repositories. While organizations often limit internal network access due to security and intellectual property reasons, the roles of Orion contractors with NASA already include the contractual agreements required for a collaborative work environment on JSC servers. As a result, there have been far fewer, and more minor, issues related to GN&C developers having access to the JSC repositories.

The configuration management tool that greatly aids the Orion GN&C configuration control process has been git, an open-source, distributed revision control system. Git has been successfully used for large projects such as the Linux kernel and has a large user base. Because git is a distributed source code management system, there is no dependency upon a server connection to manage file revisions. Furthermore, because each developer has a complete history of a repository locally, actions in git are faster than a centralized system. In fact, compared to the EFT-1 code management database, git is one to two orders of magnitude faster on common operations such as creating a new repository and checking the status of file modifications.

With other revision control tools used in the past, there was a need to develop a set of custom scripts that managed a "view" of the source code. These scripts provided two main functions: enabling interfacing with different simulation environments, and segregating the source code under configuration management from the generated artifacts. When working in a view, the custom scripts managed the interactions between the simulation and the RAMSES models in Simulink to simulate a mission run. Also, the execution of Simulink models involves the generation of executable artifacts (i.e. .mex* files). When present in a configuration controlled area, these mex files are potentially a hassle to separate from the maintenance of the source code. The use of git makes this problem obsolete with the ability to ignore certain files or types of files. Therefore the dependence on custom scripts and "views" were completed eliminated by using git.

In addition, git is particularly adept at non-linear development, where branches and merges are common, easy, and fast. This has made collaboration for FLASHE and RAMSES development much easier than in the past. Previously, exchanging source code between organizations was done via one of two inconvenient methods: post source file(s) to a secure website where it could be downloaded by another developer, or wait for the periodic synchronization of databases between organizations, which was as infrequently as once per week. With access via simple ssh connections to the common JSC servers and with git merging capabilities, exchanging and sharing work is significantly more efficient.

Another improvement in the FLASHE/RAMSES development for EM-1 has been the code integration process. The two process approaches used previously were

1) monthly submissions of code changes to an integration team, and 2) a small team responsible for all changes directly incorporated into the primary FSW code database. The first approach was used early on in the project when there were more GN&C developers doing parallel development without efficient means of recombining the work. The monthly integration process was a large and challenging effort for reasons including limited tools for merging distributed changes, long integration periods, and simultaneous code changes to graphical models that were difficult to merge. It was not uncommon for the process of integrating all of the code changes to take up to a month, which left essentially no time for developers to include their new updates into the latest baseline. This process was inefficient for all involved. When the EFT-1 flight test mission was established, the GN&C team moved into more of a production mode with a limited team involved in making direct changes to the GN&C models and code. Members of the small team could see updates immediately, but the larger community was required to wait for baselines, which were generally months apart. For developers who were not on the small team, their work was done on top of a baseline release. That work was then sent to a team member who could integrate it into the configuration management tool, but those steps were accomplished via email or posting to a common area, completely outside of the CM tool. If any other changes had occurred in parallel, those had to be manually merged or reapplied. None of these interactions utilized features which configuration management tools are intended to provide.

The process that has been implemented more recently with FLASHE/RAMSES development has been an automated continuous integration process. Scripts running in the JSC lab monitor for requests for integration from GN&C developers. This request comes via a web-based issue submittal page where the developer specifies the location of his or her repository as well as the git branch containing the work. Then a series of regression tests are automatically started. If the developer's work merges properly and passes the regression tests, then it is automatically merged into the "central" repository for others to retrieve. If the merge with the developer's branch or the regression tests fail, logs are created and made available on the issue page for review.

The automatic and continuous nature of the integration process takes advantage of the best aspects of the previous integration processes, while also adding automation that saves a lot of time. Like the process used for the small production team on EFT-1, the developer is responsible for keeping up to date with progress and work being integrated into the central repository. For example, if a developer has not merged the latest work into his or her branch prior to requesting integration, it is possible that the merge will contain conflicts. Instead of the responsibility for cleaning up the merge conflicts being the job of an integrator, it is the responsibility of the developer to properly merge in the latest from the central repository to ensure the automatic integration process runs to completion. Because integration takes place continuously, a developer can pull the latest as often as multiple times per day, with low overhead, in order to incorporate small changes more frequently to make tasks that take a long time to complete easier to manage. Another improvement over past processes is that the regression testing is completely automated, as long as the data used to verify the regression tests do not change. For the production team on EFT-1, it was important that changes were not introduced that inadvertently caused simulation runs and tests to fail. At times this could be a burden because of the amount of time it took to build and execute all the models and tests. With an automated process, it is simpler and faster for a developer to rely on an automated regression and verification process to take the time to build and execute the models, while the developer's time and computer resources can be freed up for other productive uses.

7. MODEL MERGING

Another common challenge with model-based development, specifically with graphical models, is model merging. For ASCII, text-based files, there is a long history, familiarity, and simplicity with merging multiple changes to the same file. Finding the differences in one or more lines of source code is relatively straightforward, so merging can be automatic when there are no conflicts between the changes, or can be compared side-by-side when the changes conflict with one another. Although the .mdl format for Simulink models are text-based, it is not straight-forward to merge these text files and still have a functioning graphical model in Simulink. As a result, model differencing and merging requires being able to view the Simulink models themselves, not the text files.

In the past, the Orion GN&C team experimented with third-party differencing and merging tools called SimDiff and SimMerge by Ensoft [9]. Recent releases of MATLAB have introduced tools for merging Simulink models. The Simulink tools are capable of finding and merging small to medium sized differences in models but cannot automatically track large changes, which is true even of text-based merging. For Orion, the GN&C flight software team has not required specialized licenses for model development; only MATLAB, Simulink, and Stateflow are needed. The Simulink merging tool, however, is only available with the Simulink Report Generator toolbox, so license costs are a deterrent and hindrance for many GN&C developers. As a result of this limitation, the GN&C team has customized the interaction between git and Simulink .mdl files, and also developed a process that facilitates conflict communication between model developers.

When conflicts occur when merging text files in git, the syntax convention of "<<<<<<" and ">>>>>>" is used to indicate modifications made by the developer doing the merge, the work being merged into, and the common ancestor. For Simulink .mdl files, these modifications inserted by git prevent an executable model from being opened in Simulink. A customization of the attributes of git has been implemented to instead create three separate .mdl file instances: "theirs," "mine," and "ancestor." With this specific handling of merge conflicts of .mdl files, the Simulink model differences can be viewed either side by side, or with other model merging tools.

There are some very beneficial aspects of git being a distributed revision control system whose utilities have been described in previous sections. One challenge in Simulink model development without a centralized repository on a server that maintains "checkin/checkout" write privileges is avoiding model conflicts altogether. There is nothing to prevent developers using git to both modify the same .mdl file simultaneously, which is more difficult to merge (regardless of what merge tools are available) than a plain text file. To address that issue, a script has been developed that gives insight to all developers what .mdl files are currently being modified. This script works by querying the branches of GN&C developers' repositories and finds .mdl files that have been modified. This list of modified model files is updated multiple times per day and posted to a webpage wiki for developers to review. Although there is no guarantee that .mdl conflicts are eliminated, this capability and the information derived are useful for identifying potential or existing conflicts early. An important aspect of the process regardless of the tools is communication amongst developers. For example, if a developer sees in the .mdl list another person working on the model he or she wishes to modify, a phone call can clarify if work schedule arrangements should be needed in order to cleanly merge both sets of model work. The other main procedural approach used on EFT-1 has been to assign a single point of contact to each algorithm, to avoid unnecessary model merging conflicts.

8. Simulation Validation, Verification, and Documentation

As with any software, ensuring that the behavior of the code meets expectations is a tedious and detail-oriented task. The Orion GN&C FSW for EFT-1 has undergone a very thorough process that certifies that the software has been built correctly and that it meets the specifications laid out in the requirements. For the FLASHE simulation, the fidelity of the software need not be as high as it is for a human-rated spacecraft, but determining what fidelity to implement is an important question and a task on its own. For example, there are two models for determining the ephemerides of the celestial bodies: the ephemeris interpolation tables, and the SPICE libraries. The SPICE libraries are of the highest fidelity, while the ephemeris tables—which interpolate between SPICE data—have less fidelity. The analyst should trade simulation execution speed and model fidelity in order to determine which of the models to use.

Prior to determining whether a model is valid, a level of expectation must be set forth for each model, as described in Table 3. The cornerstone of FLASHE model validation is the question of whether the model produces the correct results. At a minimum, the unit tests should prove that the fundamental algorithms in the model produce the correct result. The unit test data can be compared against analytical answers, results from other simulations with validated models, hardware test data, flight test data, etc. In the aforementioned example of the ephemeris tables, the results can be compared to SPICE data. A template for building unit tests was developed. This template provides a unified interface for calling unit tests throughout the FLASHE code base, allowing the entire validation test suite to be called from a single top-level function.

Table 3. FLASHE model standards.

Validation Criteria	Description
Model Reference	Each model should be allow independent, stand-alone configuration, revision management
Structured Parameters	Each model reference block should utilize structured model reference parameters for constants that are used within the model
Uses standard utilities	Each model should use utilities from the standard library
Default initialization data file	Each model should have a standard parameter data initialization file used to configure the model
Unit tests	Each model should have a standard unit test suite used for validation purposes
Documentation	Each model should have a standard set of documentation

In addition to testing code, the unit test template contains example documentation based on MATLAB's native m-file-to-HTML publishing capability. The MATLAB Report Generator toolbox also has capabilities to generate HTML files containing hyperlinked images of models in Simulink and Stateflow. These reports are collected and linked together for easy distribution and review. Furthermore, the website wiki used for the project can directly tie in to the repository of artifacts, making it possible to review current documentation on any model merely by navigating to the project webpage, thus removing the need to maintain equivalent documentation in separate locations.

While having unit tests for each model provides proof for the accuracy of the overall simulation, it is not practical for each developer to run the gamut of validation tests. Rather, for day-to-day simulation development and code updates, a set of regression tests have been designed. The idea behind the regression suite is two-fold. The first objective is to exercise each simulation model at least once prior to submitting changes for integration. The second objective is to test an integrated simulation run, providing a way to check that interfaces have been properly built. Thus, the regression tests provide an answer to the verification question: "Does the simulation continue to perform as expected?" A set of verified regression data is available for comparison to the developer. The data also serve as gate-keepers for integration of submitted changes.

9. Automated Build and Test Services

As mentioned in Section 6, a continuous, automatic integration process has been implemented. This approach has provided substantial time savings and robustness for development. It includes automated builds and testing, and when each task of work correctly passes the regression testing, the main repository is updated automatically.

The popular and open web based service Redmine is being used as a centralized front end for communicating and coordinating regular change traffic with the git repositories. The Redmine interface provides a simple and intuitive mechanism to create, assign and update software Change Requests (CRs). Furthermore, email notification alerts are sent as the state of a CR changes throughout its lifetime. Any CR written into Redmine maintains a current state. These CR attributes can be queried through an HTTP-based protocol called the REST (REpresentational State Transfer) API. This feature has been utilized by building several bash shell scripts that check the state of all Redmine CRs using curl to communicate through the REST API. When a CR is changed to "ready for testing," a merge between the CR and the main-line repository occurs in a temporary candidate area. After a successful git merge, the candidate software is built and tested. If the merge, build, and test procedures go smoothly, the script will push the candidate repository changes back up to the main-line repository. Regardless of the pass/fail outcome, the shell scripts will update the CR status in Redmine and attach a log of the entire procedure from start to finish using the REST API.

Continuous Services

The services of Redmine and the REST API described provide a simple interface for single-shot builds. In order to provide continuous and repeating services, a third open source tool is being utilized called Jenkins. Jenkins offers a clean and intuitive front end for scheduling command line processes. The tool tracks the execution time of any job it launches, and automatically sends emails out in the event of a failed job. Jenkins detects pass or fail based on the return status of the bash script.

Jenkins is configured to run the CR merge, build, and test script every 15 minutes throughout the day. Thus, a developer who has finished their CR work can merely go to the Redmine interface and set their CR state to "ready for testing." Ideally within 15 minutes, the scheduled Jenkins job picks up on the changed state and commences the testing process. One important note is that within any configured job, Jenkins will not launch a new process unless the last job has already completed.

Nightly Builds

Jenkins jobs are also configured to perform nightly builds of the project. These jobs use the same underlying engine as the merge, build, and test described above, only they do not query Redmine for new CRs, nor are any merges performed. The test suite for nightly builds is larger than the CR process. This is to exercise a gamut of unit tests and other more extensive integrated testing without being a burden on resources during work hours. Should any one of these jobs fail, email notification goes out to the project leads with information on the reason for failure. This allows troubleshooting to take place before other CRs are introduced on top of a broken build.

The results of these nightly builds are not merely discarded. They represent a significant time and CPU investment, and thus the build artifacts they produce during successful executions are copied back to public areas. These build artifacts can be used by other users through a process coined "winking in" or "winkin" as a quick way to boot strap a newly cloned source code repository.

Winking In

Building large MATLAB/Simulink projects such as FLASHE/RAMSES requires MATLAB to do a significant amount of processing to generate its .mex object files. Early in the project it was realized that the MATLAB build artifacts from one build area could be copied into a second area, enabling code in the second location to run without any further building by MATLAB. Copying files is significantly faster than building them from scratch. As long as the second area contains the same source code as the first area, a user can entirely forego the build process by just copying the artifacts from the first area and globally updating embedded path information in some of the text files. This procedure of copy and update has been coined "winkin". Winkin of the build artifacts (around 33,000 files) from a pre-built area such as the nightly build area is an order of magnitude faster than performing an entire rebuild. While it can take up to two hours to build all the Simulink models serially, and up to about 45 minutes using MATLAB's Parallel Computing Toolbox, winking in the build artifacts takes only about five minutes.

10. Autocode

One of the biggest benefits of the MBD process is the ability to automatically generate production code directly from the source models very early in the software design cycle. This gives the project the ability to put many "miles" on the software and find potential software issues starting as early as pre-PDR. Commonly referred to as "autocoding," the GN&C flight software process uses the Simulink Coder™ product to generate C++ flight code from the Simulink models in RAMSES.

Focusing too much on the software performance early in the project can slow down development if excessive emphasis is put on the low level code. Generating code can also take a long time and slow development if tested excessively. The generation and verification of the autocode, however, has been automated via the build and test server. Instead of requiring developers to run all of the regression runs on the the compiled autocode, the build and test server handles the steps automatically in the nightly runs. Using the Software In The Loop (SIL) mode in Simulink, the production C++ code is generated, compiled, wrapped in an s-function, and executed closed-loop with the FLASHE simulation. The same CR regression suite is run with the same logged data, and the pass/fail criteria are used to determine the success of the simulation. Figure 6 illustrates the only visual difference that would be seen for the different simulation modes. After PDR, the software verification will become a more integrated part of the FSW development process. The GN&C FSW team is planning to add Processor In The Loop (PIL) runs for as well, leveraging the automated build and test server. While SIL mode uses the same compiler that is used for the Simulink models, PIL mode uses an emulated GreenHills Integrity target that matches the Orion flight computer target.

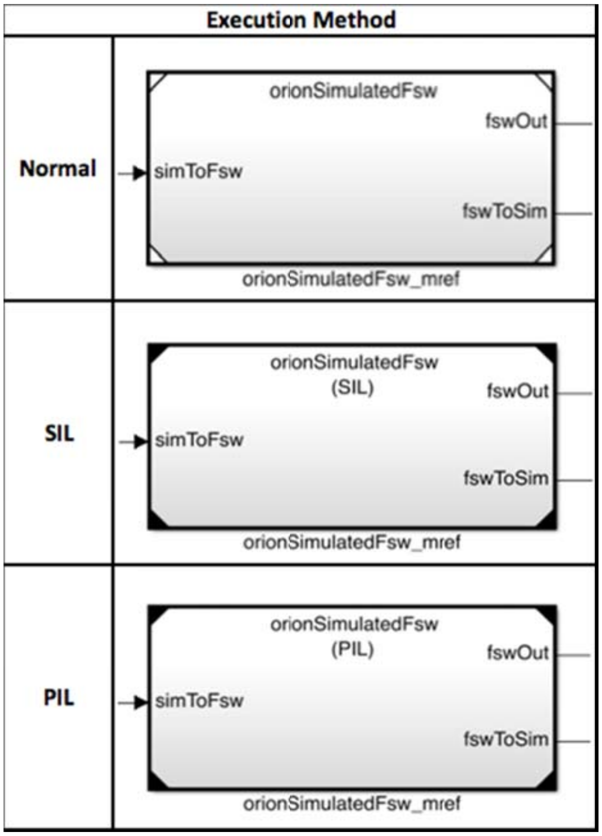

Figure 6. The GN&C RAMSES FSW model in Normal, SIL, and PIL modes.

11. METRICS

An important part of any large software project is metric tracking. A major advantage in automated regression testing is the ability to continuously collect metrics on the FLASHE and RAMSES models through the entire period of development. A lesson learned from the EFT-1 development is that better metrics tracking may have helped to avoid issues and rework during the unit testing, software standards compliance, and integration phases of the project. A lot of time and effort can be saved at the back end of a project by managing software metrics early on. For example, code with a high complexity is very difficult to fully test. To comply with the Class-A software standards, all flight code must be fully covered and have unit tests proving 100% Modified Condition Decision Coverage (MCDC). If the code is too complex, 100% MCDC may be unachievable and the code may need to be modified to comply. Tracking and managing the complexity of the code from the pre-PDR stage will greatly reduce this potential rework and the need to change proven algorithms toward the end of the development cycle.

Collecting software metrics early in the project will help provide trends that can help identify problems early. It becomes simpler to pin-point software changes that make either positive or negative impact to design parameters (like stack). As the metrics collection tools are under development, there is a limited amount of data that to provide useful trends. **Error! Reference source not found.** is an illustration of an example of how trends could be used to identify a large code impact.

The automated build and test server is used to collect weekly metrics. These metrics are continuously collected and stored in a SQLite [10] database. SQLite is in the public domain, is ACID (Atomicity, Consistency, Isolation, Durability)-compliant and implements most of the SQL standard. This permits optimum querying for extraction of large data sets with little chance of data corruption. Fifty-four unique metrics are currently being collected for the GN&C FSW.

When targeting a real-time operating system with limited power, the software must be optimized appropriately. The Orion spacecraft uses a relatively low powered Power PC 750FX clocked at 400 Mhz with a 100 Mhz FSB (Front-Side Bus). The GN&C partition is only allocated 7.5 ms each second to execute all capability, and has a stack allocation of less than 500K. The CPU allocation is less than 80% of that for the EFT-1 flight, and the GN&C partition will have more software capability in future missions. Tracking the relative performance will spotlight changes made to the code that have negative effects on the performance. Visibility into which algorithms are utilizing the most CPU processing will aid in prioritizing optimization efforts. Being able to conduct code optimization in the early stages of the software development eases CPU allocation issues later in the program.. Furthermore, because keeping simulation speed as high as

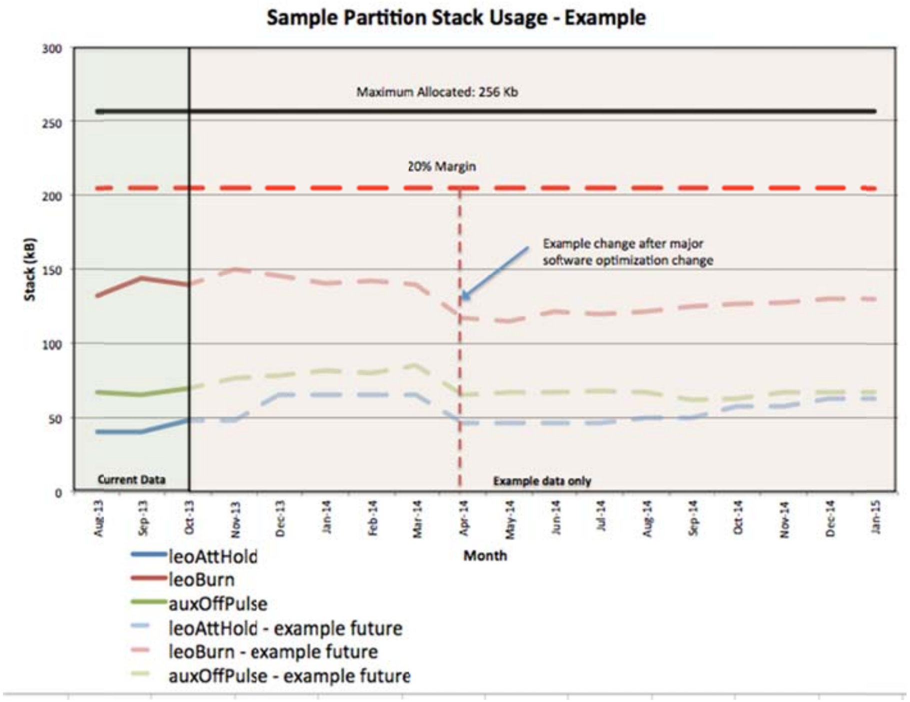

Figure 7. Example illustration of software metrics trends.

possible is very important for development, metrics that record the build and run times for each regression test are useful in determining changes that have slowed down simulation build and execution times. This is particularly true in the context of the lessons learned from EFT-1 development. Table 4 shows examples of some of the types of metrics collected on the GN&C flight software.

Table 4. Example metrics collected for the GN&C FSW.

Metric	Description
Code Cyclomatic Complexity	Code complexity should be managed early on to ease unit testing effort
Unit Test reports	Useful for monitoring the unit testing progress of the entire project
SLOC	SLOC is a useful project management metric for resource planning
Profile Reports Per Regression Run	Useful for showing the methods that are utilizing the CPU the most
Stack usage per regression run	Stack is limited and needs to be managed to stay within a specific allocation
Build time	Major changes in build time will be recognized and isolated to individual CRs
Run speed	Simulation speeds should be as high as possible to ease development
Model Advisor Compliance	The pass/fail data for each model on modeling standards compliance

12. SUMMARY

The Orion GN&C team has used its experience in model-based development of flight software to produce effective flight code for the first Orion test flight. The team has extended that experience to build upon a number of lessons learned to improve the development environment and processes for a larger set of GN&C functionality and software required for missions beyond LEO. The following is a summary of challenges encountered during EFT-1 software development and resulting solutions as discussed in this paper.

Challenge: Provide an environment where model-based FSW can be developed quickly and without TCP/IP interfaces between C and Simulink.
Solution: A new simulation is built natively in Simulink via model-based design principles.

Challenge: Enhance collaboration between geographically-dispersed teams operating on different secure networks.
Solution: Utilize git distributed version control.

Challenge: Speed up model development.
Solution: Create a process that allows for parallel development on separate branches, with continuous integration and eliminate reserved checkouts.

Challenge: Implement a usable graphical merging capability for Simulink models.
Solution: Design and implement custom scripts that tap into underlying git capability to extract "mine," "theirs," and "ancestor" instances, and allow the developer to 3-way compare these models.

Challenge: Standardize the simulation model design, testing, validation, verification, and documentation.
Solution: Experiences from previous projects have been applied in the areas of model development. A standard API to unit test drivers has been created that facilitates creating, running, and documenting unit tests and associated models.

Challenge: Ease the burden of integrating multiple branches at once, leading to long integration cycles.
Solution: Responsibility of merging and correcting code is placed on each individual developer, which allows code changes to be integrated into the baseline and available for distribution within a matter of hours.

Challenge: Speed up and automate testing and integration for individual developers.
Solution: Utilize automated scripts and tools to merge and test code. Automatically integrate if all of the regression data are unchanged.

Challenge: Maintain and manage a working, compiled, and testable version of the autocode concurrently with model development.
Solution: Utilize automated nightly scripts on build and test server to autocode and test.

A model-based simulation environment in Simulink improves the process for development of on-orbit flight algorithms. Furthermore, a series of process improvements contribute greatly to the work flow of the team, including enhanced configuration management techniques for model-based software systems, automated code and artifact generation, and automated testing and integration.

ACKNOWLEDGMENTS

To acknowledge all the contributors to the Orion GN&C model and process development is not possible since it was a product of a large, diverse team. However, the authors wish to acknowledge Chris Rossi from Draper Laboratory and Leon Gefert from the Glenn Research Center.

REFERENCES

[1] Tamblyn, Scott, Henry, Joel, and King, Ellis, *A Model-Based Design and Testing Approach for Orion GN&C Flight Software Development*. IEEE Aerospace Conference. Big Sky, Montana, 2010.

[2] Henry, Joel R. and Jackson, Mark C., *Orion GN&C Model Based Development Experience and Lessons Learned*. AIAA GN&C Conference, 2012.

[3] The Mathworks online: http://www.mathworks.com/

[4] git online: http://git-scm.com/

[5] NASA's Navigation and Ancillary Information Facility online: http://naif.jpl.nasa.gov/naif/toolkit.html

[6] Odegard, Ryan G., et. Al., *Configuring the Orion Guidance, Navigation, and Control Flight Software for Automated Sequencing*. IEEE Aerospace Conference. Big Sky, Montana, 2011.

[7] Lin, Alexander S., Penn, John M., *Trick Simulation Environment 07*, NASA Tech Briefs, June 2012; 17-18.

[8] Walker, Gavin, Friedman, Jonathan, and Aberg Rob., *Configuration Management of the Model-Based Design Process*. Proceedings of SAE World Congress & Exhibition, 2007.

[9] Ensoftcorp online: http://www.ensoftcorp.com/simdiff/all-simdiff-and-simmerge-editions/

[10] SQLite online: http://www.sqlite.org/

BIOGRAPHY

***Ryan Odegard** is an aerospace engineer at the Charles Stark Draper Laboratory. He has worked with NASA on guidance, navigation, and control architecture design, as well as systems engineering concept design, database design, and reliability analysis. His background includes multidisciplinary system design optimization for complex systems and architecture development. Currently a Member of the Technical Staff at Draper Laboratory, Ryan earned his S.M. from the Massachusetts Institute of Technology in Aeronautics and Astronautics under a Draper Laboratory Fellowship, and has a B.S. from the University of Arizona in Mechanical Engineering.*

***Zoran Milenkovic** received a B.S. in Aerospace Engineering from Iowa State University in 2004 and an M.S. in Aerospace Engineering from the University of Houston in 2010. His work at the Draper Laboratory has been focused on the area of guidance and navigation for rendezvous and proximity operations for the Space Shuttle, the Orion MPCV, and the Sierra Nevada Dream Chaser amongst others. In 2006 he joined the Draper Laboratory team having previously worked at Muniz Engineering and Lockheed Martin.*

***Joel Henry** is the GN&C Software Functional Manager for Orion at NASA's Johnson Space Center in Houston, Texas. He has a background in mechanical engineering and computer science. Joel has a BSME from the University of Texas at Austin and is a Licensed Professional Engineer in the state of Texas.*

***Michael Buttacoli** is an Aerospace Engineer with the National Aeronautics and Space Administration. He has worked in the NASA community for 15 years developing software for Mission Control, Trajectory Design, and ISS Rendezvous and Proximity operations. He is now a part of the GN&C Engineering team for the Orion project. Mike holds a B.S. degree in Aerospace Engineering from Purdue University.*