Model-Based Verification and Validation of Spacecraft Avionics

M. Omair Khan\textsuperscript{1} and Michael Sievers\textsuperscript{2}
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109-8099, USA

Shaun Standley\textsuperscript{3}
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109-8099, USA

Verification and Validation (V&V) at JPL is traditionally performed on flight or flight-like hardware running flight software. For some time, the complexity of avionics has increased exponentially while the time allocated for system integration and associated V&V testing has remained fixed. There is an increasing need to perform comprehensive system level V&V using modeling and simulation, and to use scarce hardware testing time to validate models; the norm for thermal and structural V&V for some time. Our approach extends model-based V&V to electronics and software through functional and structural models implemented in SysML. We develop component models of electronics and software that are validated by comparison with test results from actual equipment. The models are then simulated enabling a more complete set of test cases than possible on flight hardware. SysML simulations provide access and control of internal nodes that may not be available in physical systems. This is particularly helpful in testing fault protection behaviors when injecting faults is either not possible or potentially damaging to the hardware. We can also model both hardware and software behaviors in SysML, which allows us to simulate hardware and software interactions. With an integrated model and simulation capability we can evaluate the hardware and software interactions and identify problems sooner. The primary missing piece is validating SysML model correctness against hardware; this experiment demonstrated such an approach is possible.

We implemented a SysML model and simulation of a typical command processing infrastructure and avionics hardware. Test cases were then run using the simulation and the test results compared to those obtained from identical tests run on a hardware testbed. The simulation produced test results that precisely matched those of the avionics hardware demonstrating the future potential of this approach.

Nomenclature

\begin{itemize}
  \item \textit{SysML} = Systems Modeling Language
  \item REU = Remote Engineering Unit
  \item MOS = Mission Operations System
  \item GDS = Ground Data System
  \item S/C = Spacecraft
  \item FSW = Flight Software
  \item FPGA = Field-Programmable Gate Array
\end{itemize}

\textsuperscript{1} Systems Engineer, System Verification and Validation Group, MS179-260, Member.
\textsuperscript{2} Systems Engineer, Advanced Computer Systems and Technologies Group, MS156-142, Sr. Member.
\textsuperscript{3} Technical Group Supervisor, System Verification and Validation Group, MS179-260, Member.
1. Introduction

Verification and Validation (V&V) at JPL is traditionally performed on flight or flight-like hardware running flight software. For some time, the complexity of avionics has increased exponentially while the time allocated for system integration and associated V&V testing has remained fixed. There is an increasing need to perform comprehensive system level V&V using modeling and simulation, and to use scarce hardware testing time to validate models; the norm for thermal and structural V&V for some time.

A. V&V background, need, and the Avionics complexity problem

Space project systems consisting of ground stations, ground networks, spacecraft, and possibly probes and rovers deployed to the surface of other planets, have for some time been out pacing our ability to perform comprehensive verification and validation on the as-built hardware and software. Increased complexity, increased on-board autonomy, increased test risk, and increased use of spacecraft capabilities beyond the uses envisaged in the baseline mission plan have all contributed to a need for more verification and validation (V&V). The increased complexity is largely due to exponentially more capable on-board processors, more capable science instrumentation demanding higher data rates, and an ever present desire to manage and recover from the inevitable on-board faults that occur during deep-space missions. Spacecraft avionics have evolved from the simple sequencers used in early flights to highly complex, multi-processor configurations that can autonomously reconfigure themselves to work around faults and process huge volumes of data in real-time. We use the term “avionics” to refer to the electronics and flight software that form the brains and nervous system of a spacecraft.

Testing all permutations of electronics configuration and software operation for all modes, states, hierarchies, configurations, and behaviors is impossible. Even knowing what state the spacecraft is in at any point in time is not possible and mechanisms that attempt to revert to a “known” state cannot assure that all state variables are returned to a known value. Given this situation, the job of verification (e.g. was the system built according to specification?) and validation (e.g. does the as-built system meet the driving need of the mission?) is a daunting challenge.

Models already have a critical role in primary V&V; thermal and structural models that have been validated against hardware tests have been the venue for comprehensive verification for decades. Software simulators are used for verification of algorithms, operations processes, and for dry runs of test procedures before hardware testing. Unfortunately, some desired avionics tests cannot be performed on physical systems at all because requisite internal nodes are unreachable. In particular, fault injection tests are highly limited which means that system behaviors during and after faults under various operational modes cannot be tested at all on flight hardware. To keep up with the demand for V&V we must continue to integrate the use of validated models and testbeds to achieve more complete coverage of the system capabilities. The desired, though probably idealistic, end state is that the only Flight Model tests that will need to be performed are tests of fundamental workmanship (phasing, environments, etc.), end-to-end tests between interfacing systems (Deep Space Network compatibility, etc), and tests whose purpose is to validate the models and simulations used for primary V&V.

Our interest is in the development of simple simulations of project-level processes such as maneuvers, science passes, deployment and mobility operations, and other complex flight-to-ground interactions in nominal and fault cases that can validate project and system level requirements and validate system design before committing to hardware. Generally, having the system expressed formally as a static model makes the necessary V&V easier to identify early in the design process. The modeling semantic itself prevents many inconsistencies and errors in test design. Interfaces and behaviors are clearly and unambiguously expressed in a consistent way, making errors more obvious when they occur, and the possibility of generating simulations directly from models, essentially making the models ‘executable’, contributes to validating such a system model.
We explored the utility of Systems Modeling Language (SysML) and related tools for modeling, and for directly generating simulations by modeling and simulating avionics hardware and software. Verification tests were run on a hardware testbed, identical tests run on the simulation, and the results compared. The hardware model was validated by the corresponding hardware tests.

The approach described in this paper addresses many of the complexity issues by creating computer models of the system that are validated at each step of design decomposition. Our concept reduces the likelihood that design errors are detected late because validation is performed concurrent with design. Moreover planning V&V concurrent with design aids in defining an efficient set of V&V activities. Using computer models in design is not new, what we add is the concept of validating these models early in the lifecycle before hardware is available, making that validation part of the V&V process, and using validated models to replace testing on the physical system.

B. SysML Summary

Our V&V approach is based on the System Modeling Language (SysML) which is an extension of the Unified Modeling Language (UML) used for developing software. Like UML, SysML is a language constructed from a formally defined semantics that includes representations of elements, requirements, behaviors, test cases, parametrics and the relationships that link these. Because SysML has a formal semantics, models built within SysML have unambiguous meaning which leads to better communication among system designers. But the central point of SysML is that the formal semantics enables descriptions that are used for automated reasoning and analyses. Contrast this with a typical block diagram that uses arbitrary color codes and line styles to represent a system. In this latter situation, it might not be clear whether a line connecting two objects is data, control, an arbitrary association, generalization, specialization, or something else.

In brief, a model built using SysML comprises elements and associations. Elements are objects, e.g., classes, systems, subsystems, and other parts while associations are the relationships that connect elements. Elements consist of a number of items that reflect the type of entity and scope the associations that may be attached. Elements may represent physical system components (e.g., electronics and mechanical structures), software elements, or any combination. There are a number of defined associations that may be applied between entities such as data and control flows, inheritance, containment, and allocation. Not all associations may be applied to all elements and associations may be assigned features that enable simple rule checking. SysML semantics also allows for extensions of the language that also must follow rigorous definitions. For example, there are semantic specializations of elements defined for real-time computer systems.

It is important that the semantics of SysML are distinguished from tools provided by a number of vendors. The tools support design and analyses of models described in SysML. While the form and function of these tools may vary from vendor to vendor, the underlying SysML is invariant. Consequently, although we have implemented our V&V methodology with one tool, our approach is not tool specific.

II. Experiment Description

Our approach extends model-based V&V to electronics and software through functional and structural models implemented in SysML. We develop component models of electronics and software that are validated by comparison with test results which are then simulated enabling a more complete set of test cases than possible on flight hardware.

A Remote Engineering Unit (REU) was chosen as the device to simulate, based on the available testbed resources, existing test data, and the complexity of the assembly we would be able to model given the overall scope of the task. The REU was integrated with an existing avionics testbed. Existing REU functional and performance test cases were run to provide data suitable for validating similar runs on the model.

For our work we implemented a simple system architecture comprising models of ground command and telemetry, spacecraft telecommunications, spacecraft command and data handling, 1553 bus, and a remote engineering unit (REU) which is responsible for collecting analog and digital data from multiple sources on a spacecraft. We modeled the REU in considerable detail and validated the model by comparing model test results...
with bench test results received from a physical REU. The model consists of the two primary elements: Mission Operations/Ground Data System (MOS/GDS) and the Spacecraft (S/C) System as shown in Figure 1.

![Figure 1. Primary Elements of Simulation Model.](image)

Figure 1. Primary Elements of Simulation Model.

Figure 2 looks inside the S/C System and Figure 3 shows the Command and Data Handling (CDH) components. Our focus is on the REU which is shown connected to a 1553 Bus Controller. The REU receives commands generated in software (which is allocated to the CDH processor) and returns its status or the value of data sampled. There are also a number of memory locations within the REU that are used for control. The values placed into those memory locations are also sent on the 1553 bus.

![Figure 2: Spacecraft System](image)

![Figure 3: The Command and Data Handling System Internal Diagram](image)

Figure 2: Spacecraft System

![Figure 4](image)

Figure 4 shows that the REU consists of a watchdog timer, FPGA, EEPROM and SDRAM. The FPGA manages inputs and outputs to a number of subaddresses (SA), implements the 1553 remote terminal (RT), and a telemetry controller. The notation in Figure 4 indicates composition and not data flow. The numbers attached to the arrows in the composition associations indicate multiplicity. We do not attempt to show the internal structure of the REU due to the large number of interconnects. However, the tool used for this work provides an instantiation feature that makes all of the associations automatically and in a way that allows our simulation access to all internal elements.

American Institute of Aeronautics and Astronautics
III. SysML Model

We can model both hardware and software behaviors in SysML, which allows us to simulate hardware and software interactions. With an integrated model and simulation capability we can evaluate system interactions and identify problems sooner. The model was created using commercial software, which implements SysML semantics, and simulated using a simulation tool plugin, which executes the logic in the model.

A. General Structure

The hardware devices and software artifacts were modeled using SysML’s structural semantics (e.g. block definition and internal block diagrams as shown in Figures 1 - 4). The machine logic and software processes were modeled using SysML behavior semantics (e.g. activity and state machine diagrams). Combining these two paradigms, a spacecraft command and data handling system model was created that captured the devices/software involved and their respective behavioral logic, as shown in Figures 5 – 7. Interweaving simulation capabilities into the model allows for various scenario metrics (e.g. REU subaddress channel bit values) to be stored and moved among block structures according to the logic pathways dictated by the system’s behavior relationships. In this manner, we were able to create an event-driven simulation of the command and data handling system model.
Figure 5: Telecom/Avionics to Flight Software interface behavior

Figure 6: Flight Software to Bus Controller interface behavior
B. Simulation Basics

An example of this modeling/simulation process is shown in Figures 8 – 10. Figure 8 represents a simple system consisting of two blocks, which represent two interacting devices within a system. Value properties on the blocks represent device metrics of interest. Figure 9 shows the system’s internal block diagram that represents the communication paths between the two devices, as well as the overlaid activity diagrams that dictate each device’s behavior. Figures 8 and 9 represent the full structural and behavioral description of the system. Figure 10 shows that by adding simple coding logic (as dictated by the simulation engine) to the actions in the activity diagram we can move data between the two devices. Data flow that is simulated between the two blocks represents metrics captured by the value properties on the blocks. In this way we can move data between the devices during run time and store instantaneous data values on the respective block’s value properties.
A key advantage for using simulation techniques is that multiplicities are instantiated at run time. This allows for expressing complex structure using simple multiplicities in the model diagram, yet these are fully navigable during simulation execution. This feature was especially important for compact representation of the eight 1553 subaddresses of 32 channels each used in the REU (see Figure 4).

Applying this basic modeling/simulation paradigm to a larger model like the spacecraft command and data processing system model works more efficiently by introducing certain artifacts into the model. Adding a global
variables block greatly simplifies traversing the model tree during simulation runs. Storing the randomized runtime pointer addresses in the address block at the start of the simulation enables quickly jumping to key areas in the model tree rather than traversing from the top block in the model each time.

C. Command and Data Handling System Simulation

Using the aforementioned model simulation basics and increased efficiency techniques, the spacecraft communications and data handling model consisted of the system model (as shown in Figures 1 – 4) and simulation code placed within activity and state machine nodes to mimic system functionality. The simulation was then tested with various input commands to determine the validity of the model. The accuracy of the model was compared to actual device runs with the same input commands.

The behavior diagrams shown in Figures 5 – 7 hinge on the concept of sending, receiving, and unraveling signals encoded by a bit pattern. The bit pattern will instruct what actions, whether nominal or fault, the flight software and REU can take. This functionality was guided in the model by the logic relationships in the behavior diagrams and the simulation code embedded within the behavior nodes (as shown in Figure 10). All representative bits in the simulated system were simulated as strings due to the relative ease with which strings can be concatenated and parsed using simple javascript commands. These three elements were key to allowing manipulation of input commands in a manner similar to how they would be treated by the spacecraft avionics.

In order to send signals from a simulated MOS/GDS, a command and telemetry dictionary was referenced to act as a repository and translator for the various commands that the spacecraft is programmed to understand. The simulation allows the user to input a series of commands that he or she would like to put through the command and data handling model. The basic input commands are correlated with the command and telemetry dictionary to create a full CCSDS command sequence that includes necessary CCSDS overhead formatting as well as the embedded spacecraft command codeblocks. The simulated MOS/GDS aspect of the model was incorporated purely as simulation code, so there is no corresponding behavior diagram to indicate the overall operation flow.

Upon command generation and transfer by MOS/GDS, the simulation will mimic command reception and error detection, which are behaviors of the telecom and avionics subsystems of a S/C. For the limited scope of the simulation, command reception by the telecom subsystem was simulated as a simple pass through between the MOS/GDS transmitters and the S/C’s receiver blocks. More time and effort was dedicated to the error detection aspect of command reception, which is traditionally controlled by a FPGA. This consisted of checking the CCSDS formatting on the input command message and identifying the type of S/C command being sent. All logic used in simulating this functionality was based on existing hardware documentation. The overall functional logic for these processes is shown in Figure 5.

After initial error checking, the command is passed to FSW for further decomposition and processing. These processes consist of unraveling the command message based on the structure of the CCSDS formatting to isolate the embedded command codeblocks. These S/C commands are then placed on sequence engines and executed according to the type of command and information contained within it. These processes were simulated by parsing the command message in chunks (to mimic FSW algorithms) until the codeblock was isolated. This process was greatly facilitated by simulating the command message as a string because this simply meant utilizing string parsing and concatenation operations in various layers of complexity. The overall functional logic for these processes is shown in Figure 6.

Upon execution by the FSW sequence engine, the S/C commands are translated into their corresponding 1553 commands and sent to the REU for processing. The translation from S/C command to 1553 codeblock was done via simulation code, which used the same command and telemetry dictionary that was used in manufacturing CCSDS commands by the MOS/GDS behavior. These commands are then sent to the 1553 bus controller, which stacks these commands on its queue. At a simulated rate of 8 Hz, the bus controller executed each of its 1553 commands by broadcasting them to all avionics hardware devices. In turn, each device listens to the command, but only reacts if the command is meant for it. This process of events was simulated by string parsing and comparison to set values, such as those representing the device’s identification number.

The REU reacts to properly formatted 1553 commands either by updating or reporting its internal register values based on command instructions. How it reacted to various commands is based purely on the hardware logic, which
is pre-defined for the existing REU device. This aspect of simulating the hardware logic proved to be the most difficult aspect of the simulation as it was widely varying and many times resulted in cascading functionality on other hardware devices. This may seem trivial when testing the actual device, but proved to be quite complex when trying to simulate the variety of logical paths. We greatly limited our scope by only engaging REU-centric commands, but if multiple hardware devices (e.g. memory card) were present it would present an opportunity to incorporate more complex logical operations. REU logic was incorporated into the simulation by changing block value properties that corresponded to subaddress channel values. Each combination of these values would indicate an aspect of the avionics system’s current state of operations. Hardware logic was incorporated purely as simulation code. The overall functional logic for FSW, bus controller, and REU interactions is shown in Figure 6 and 7.

Throughout the S/C command and data handling system processes, timing is a very critical aspect that must be handle carefully and accurately. The most time critical processes in the simulation were the FSW sequence engine execution and 1553 bus controller messaging. The FSW sequence engine behavior was important because CCSDS commands come with a time tag that indicates when the command shall be executed. Generally the command is required to be processed within a few milliseconds of its indicated execution time or else the system will generate a fault. On the other hand, the 1553 bus controller must send messages at a fixed millisecond-scale time step. If these conditions are not met, the avionics system is left idle and the FSW would likely initiate a system reset. To account for the timer aspect of these processes, we incorporated a simulation clock that was only progressed after certain events. In this manner, the simulation would essentially “freeze” time to carry our concurrent operations and only step the clock once these operations were complete. The clock steps were mainly dominated by the 8 Hz messaging rate of the bus controller.

Aside from nominal operations of the simulated hardware and software, an equally important aspect of the simulation was in the off-nominal operations. A lot of the fault mitigation aspects of command reception and REU hardware logic were accounted for in the static behavior diagrams, but were not enacted with simulation code. This serves as a very important avenue for future exploration because it would serve as a key method for virtual testing of all the avionic system’s logical pathways, some of which would be damaging if run on the actual hardware. The ideal simulation would be to have a system that has the capability to inject faults into the nominal behavior based on probabilistic or user-based triggers, which opens up the possibility of running Monte-Carlo simulation on the model with varied input command and fault conditions.

**IV. Conclusion**

This work showed that it is possible to model avionics boxes and software in SysML, derive simulations directly from the models, and validate the simulations against hardware test results. Test cases were run using the simulation, and the test results compared to those obtained from identical tests run on a hardware testbed. The simulation produced test results that precisely matched those of the avionics hardware.

We deliberately began simulation work at a low level of assembly to constrain the scope of this activity, and found perhaps unsurprisingly that simulating complex hardware logic at this level is difficult in SysML. Doing this in more detail would require a much more dedicated modeling effort, as well as a multi-thread capable simulation engine. Our SysML simulation work so far is best suited for modeling mission scenario sequence and system interactions, whereas more focused simulation software or scripts should be used to handle repetitive processes that occur on time scales less than one milliseconds. In future efforts we will use SysML model simulation at a S/C subsystem level, and pass inputs to more capable box and card-level simulators to mimic device functionality.

We found that most of the value-added was gained by modeling at the systems level; the tools we used were not well suited to modeling lower level electronics. Ideally, we would like to use a more capable simulation engine that can handle fine-granularity behaviors; these tools exist, but were not integrated for this work. Despite these limitations, the concept of adding simulation to a model is sound and useful. Ideally we strive to have a library of behaviors that match a library of hardware and software components that can be selected and re-used. Elements of such a library would only require validation once, and the library would be usable in multiple projects.
Acknowledgments

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The authors would like to thank Melody Safavizadeh for her advice on the REU’s detailed functionality and testbed setup.

References


