Survey of Verification and Validation Techniques for Small Satellite Software Development

Stephen A. Jacklin
NASA Ames Research Center

Presented at the 2015 Space Tech Expo Conference
May 19-21, Long Beach, CA

Summary

The purpose of this paper is to provide an overview of the current trends and practices in small-satellite software verification and validation. This document is not intended to promote a specific software assurance method. Rather, it seeks to present an unbiased survey of software assurance methods used to verify and validate small satellite software and to make mention of the benefits and value of each approach. These methods include simulation and testing, verification and validation with model-based design, formal methods, and fault-tolerant software design with run-time monitoring. Although the literature reveals that simulation and testing has by far the longest legacy, model-based design methods are proving to be useful for software verification and validation. Some work in formal methods, though not widely used for any satellites, may offer new ways to improve small satellite software verification and validation. These methods need to be further advanced to deal with the state explosion problem and to make them more usable by small-satellite software engineers to be regularly applied to software verification. Last, it is explained how run-time monitoring, combined with fault-tolerant software design methods, provides an important means to detect and correct software errors that escape the verification process or those errors that are produced after launch through the effects of ionizing radiation.

Introduction

While the space industry has developed very good methods for verifying and validating software for large communication satellites over the last 50 years, such methods are also very expensive and require large development budgets. However, for a large satellite costing hundreds of millions of dollars, a large budget is available. This is not true for small satellites. Although small satellite programs have smaller development budgets, the cost of verifying and validating software is not smaller. For this reason, the cost of verifying and validating the software could easily dominate the small satellite budget. As a consequence, small satellites typically do not receive the same level of software assurance as their larger counterparts.

The software assurance of small satellite software, particularly nanosatellite and CubeSat software, has received little attention. One pragmatic reason for this is that in order for small satellites to be developed quickly and cheaply, elaborate small satellite software verification and validation programs cannot be pursued. Moreover, if small satellites are produced cheaply and quickly, then mission failures caused by software problems can be tolerated. In fact, simultaneous or sequential launches of small satellites are sometimes planned assuming that software failures and other failures (like separation failures from launch vehicle) may happen. Whatever data is lost from one small satellite due to a software glitch may easily be recovered from another small satellite.[11] This approach is just an alternate form of redundancy which works better for hardware than for software. Whereas hardware usually fails due to wear or damage, software failures are generally the result of defective software design or programming errors.

The number of small satellites placed into orbit has grown significantly in just the last few years. Industry, academia, and government organizations are launching small satellites at an aggregate rate approaching more than a dozen every month. The meaning of “small” generally refers to the low mass of the satellite.

Table 1 presents the mass ranges for a number of small satellite types. Across the world, there is some variability in this terminology. For example, what one organization may call a picosatellite, another may refer to as a nanosatellite. It is also true that when some organizations use the word "small", they mean a satellite that can be designed quickly and produced cheaply. Cheap may be anything from less than a million dollars to 50 million dollars, and quick may mean two years or much less.[1, 2]

The most common small satellite type is the nanosatellite, having a mass of 1-10 kg (2.2-22 lbs.). CubeSats generally fall into this category. These satellites are usually small enough to fit in the empty space of the separation rings of rockets carrying larger spacecraft into orbit. They ride along as excess mass and are spring-ejected into space after stage separation. This has dramatically reduced the small satellite launch cost and has led to an equally dramatic increase in the number of small satellites placed into orbit. Figure 1 shows the number of successful small satellite missions from 1957 to 2013 as compiled from data presented by Swartwout and Jansen.[3, 4]

Table 1: Small Satellite Mass Ranges

Type	Mass Range
Femtosatellite	Less than 100 g
Picosatellite	100 g to 1 kg
Nanosatellite	1 kg to 10 kg
Microsatellite	10 kg to 100 kg
Minisatellite	100 kg to 500 kg

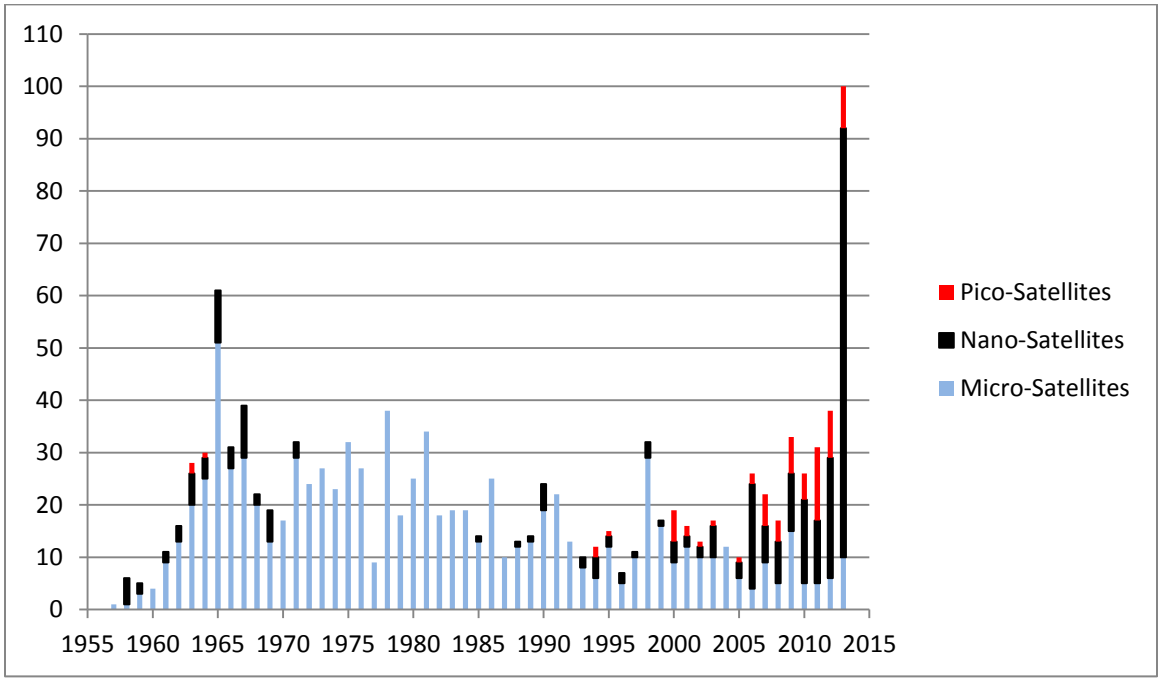

Figure 1: Number of small satellite placed into orbit between 1957 and 2013.

What is Software Verification and Validation?

Software verification is the testing done to make sure the software code performs the functions it was designed to perform. Validation is the testing performed to prove that the verified code meets all of its requirements in the target system or platform. These definitions are consistent with the RTCA DO-178C "Software Considerations in Airborne Systems and Equipment Certification" document which the Federal Aviation Administration uses to guide the development of certifiable flight software.[5] According to that document, software validation does not need to be considered (and is not considered by DO-178C) if the software requirements are complete and correct. If they are, then in theory the properly verified software will also validate. Unfortunately, in practice it often happens that the verified code does exactly what it was designed to do, but what it does is not the right thing.

Software verification and validation are commonly described in connection with the V-shaped software lifecycle diagram shown in Fig. 2. The software development process begins on the top, left side which indicates the first step is a thorough development of the software requirements. The software functional requirements are decomposed into a software architecture, from which the detailed software design and the software code are ultimately produced. The sequential decomposition of requirements to code is sometimes referred to as the waterfall software development model, because each step flows down from the next. Ritter provides a good description of the waterfall development approach used by NASA Marshall Space Flight Center for development of ground software supporting the International Space Station.[6] Software verification is the process of proving that the progressive decomposition of requirements to the software code is correct. The blue arrows represent the verification testing done to prove the correct decomposition at each step.

The testing done to prove that the code meets the requirements is called validation and is associated with the right side of Fig. 2. The red arrows represent the validation testing at each level. At the lowest level, each software unit is tested in a process called unit testing. If errors are found in unit testing, the code is corrected and then re-tested. When all the software units have been successfully tested, they are integrated into the full code. The integrated code is tested in computer simulation to validate that it satisfies the software architecture requirements. If errors are found at this point, then either the code or the architecture must be corrected. If faulty architecture is to blame, then the software architecture and the detailed design must be repeated and new software is coded. Then both unit testing and integration testing need to be repeated.

Once the integrated software is validated to be correct, the last step of validation (top right box in Fig. 2) is to check that the software operates correctly on the target platform. This testing is usually performed using high-fidelity computer simulations first, then using hardware-in-the-loop (HIL) simulations, and finally the actual spacecraft computer system. If the software fails to operate as intended on the target system computer, the problem is almost always that the software requirements were not correct or were not complete. It is also sometimes discovered that assumptions made about the software interfaces is incorrect. If it is found that new requirements are needed, then the whole software development process must be repeated, which is adds greatly to the software development cost. Quick fixes to the software architecture may result in the creation of a fractured architecture which could produce complex validation problems. The validation process ends when it is proved that the code does what it has been designed to do on the target platform.

Obviously, it is very important to make sure that the software requirements are complete and correct prior to coding the software. To make sure this happens, the traditional approach to software development is to have numerous human reviews of the requirements and the software architecture. As pointed out by Ambrosi et. al., these meetings include the System Requirements Review, the Preliminary Design Review, the Critical Design Review, the Detailed Design Review, the Qualification Review, and the Acceptance Review.[7] These are also indicated in Fig. 2.

The problem is that although it is easy to specify software requirements for a given subsystem in isolation (e.g., communication subsystem), it is more difficult for software engineers to make sure that the software requirements meet the overall system requirements (communication with all other systems working on the satellite and on the ground, plus all fail-safe requirements, etc.). Ideally, software requirements should be

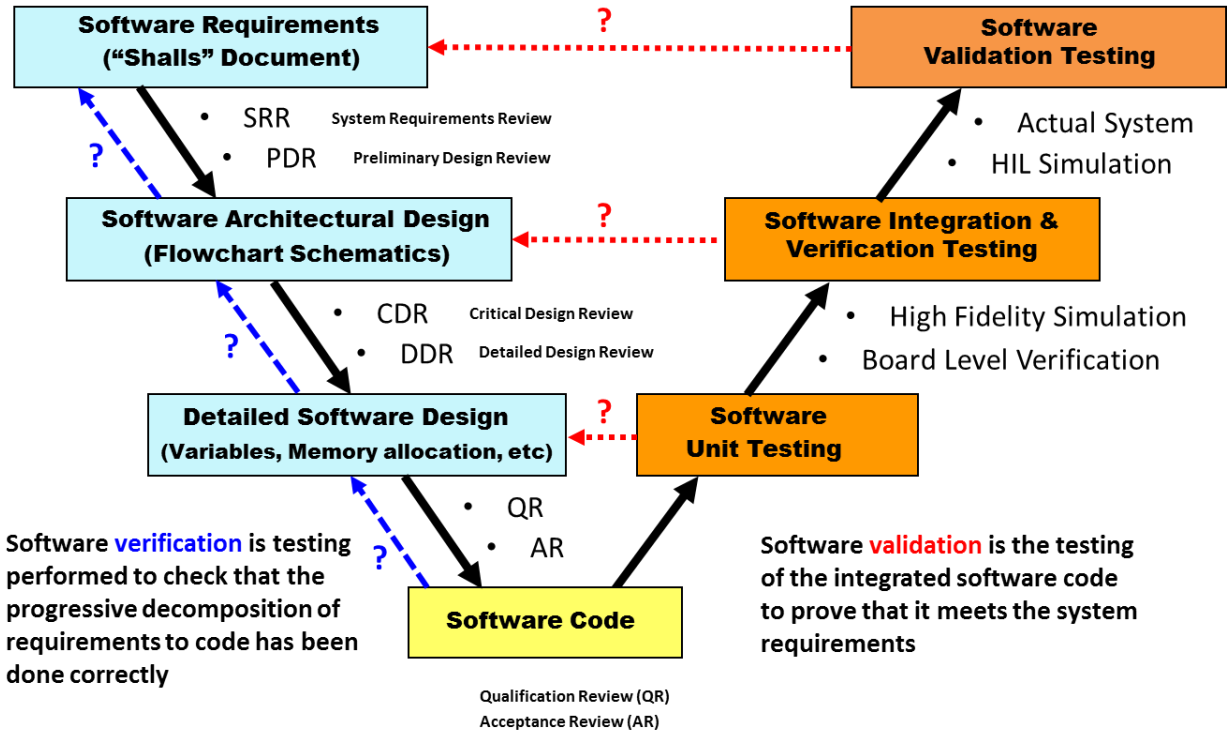

Figure 2: Waterfall software development lifecycle, verification, and validation.

distilled or flow down from the system (satellite + ground) performance and system safety requirements. If all of the requirements are complete and correct prior to code development, the software re-design and re-verification costs can be avoided.

However, to get the ball rolling, software creators frequently make assumptions regarding the software requirements. Some software developers refer to this approach as the spiral software development method. The idea is to develop software prototypes early based on incomplete requirements so that the software users can then review the software to see what other requirements are needed. This approach probably better models what actually happens in most software development programs. The problem with this approach is that it does not in any way soften the impact of incomplete requirements being discovered late in the program. Software problems discovered late in the validation phase are not only more costly to fix, but also take more time since all verification testing needs to be repeated after the software is fixed. For this reason, it sometimes happens that late discovered requirements are eliminated (at the cost of reduced spacecraft performance) rather than repeat the verification testing all over again.

Survey of Satellite Verification and Validation Techniques

A survey of satellite software assurance methods used to verify and validate small satellite software was conducted to determine the benefits and value of each approach. This survey was based on a review of more than 165 articles available in the public domain.[8] Aside from the traditional verification method of human review of requirements and software, the following verification and validation methods were found in the satellite literature:

- Simulation and testing
- Model-based software design and verification
- Formal methods
- Fault-tolerant software design and verification by runtime monitoring

Each of these methods will be described separately below. Simulation and testing has by far the longest legacy and remains the most widely used method to verify and validate small (and large) satellite software. However, model-based design methods are becoming more prevalent because they work to accelerate the iterative software design-code-test/verify development cycle. Of most importance, model-based design tools enable the early validation of the software requirements. Verification using formal methods (static analysis, model checking), though sometimes difficult to implement, may offer new ways to improve small satellite software verification and validation. Static analysis is relatively easy to apply and can catch some types of programming errors, but is limited in scope. Although verification using model checking is much more comprehensive, in order for it to be more usable by small-satellite software engineers, methods to deal with the state explosion problem need to be found. Last, fault-tolerant software design methods can be used to implement run-time monitoring as a means of software verification either before or after launch. Run-time monitoring, combined with fault-tolerant software design, can effectively deal with either software errors that escape the verification process prior to launch or those created after launch, such as those produced by the effects of ionizing radiation.

Simulation and Testing of Satellite Software

Extensive testing of satellite software function and performance on complex simulations remains the most common (nearly universal) method of providing satellite software assurance, especially for the massive telecommunications and military satellites. The method of verifying and validating satellite software by simulation and testing has been used by companies such as Boeing Space Systems for over 50 years to create reliable software for large communication satellites.[9] In this approach, after the software requirements and code development reviews have been completed, the satellite software as well as the space environment are simulated by high-fidelity computer simulations. In these simulations, the satellite control software and space environment are simulated in varying degrees of complexity. Figure 3 shows a simplified model that might be used early in the test program to verify a satellite attitude control system on a desktop computer or work station. As the verification effort progresses, more detail is added to the simulations.

To test the software against every possible failure mode, the fidelity of the simulations must be quite detailed and complex. Simulations for large satellites typically include sensor models (e.g., star trackers, sun sensors, earth sensors), actuators models (e.g., reaction wheels, thrusters, magnetic coils), and functional models of the power system management, telemetry control functions, the thermal management system, the fault detection and isolation system, and the propulsion system, if applicable.

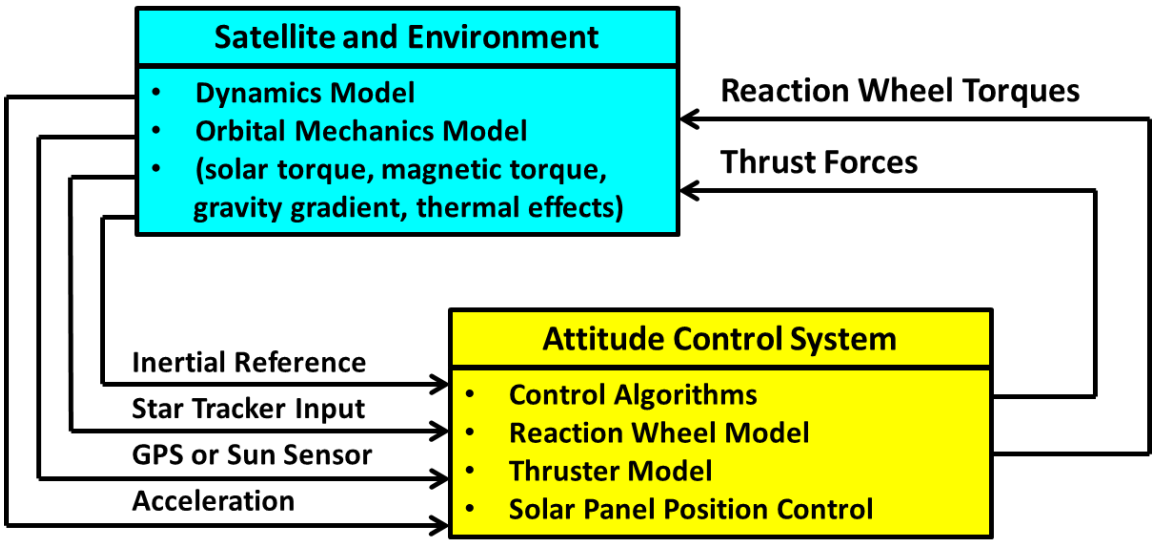

Figure 3: Early simulation model for satellite attitude control system software.

The simulations must also have sufficient fidelity to model the many data and control buses used to connect all the microprocessors, memory, and devices. These simulations verify that all component functional interfaces, all harnesses, all spacecraft functional operations, and all commands work as expected. The simulations may incorporate high-fidelity spacecraft dynamics and orbital dynamics models. In addition to the spacecraft, the space environment is also carefully modeled by simulating the effects of the earth's magnetic field, the orbital gravity gradient, the solar radiance, the thermal gradients, and the effects of solar disturbances. NASTRAN models of the satellite and antennae may also be included in the simulation to model the non-linear, flexible body dynamics.

During the software validation phase, simulations are performed again, but with as much of the actual hardware as possible instead of computer models. During the verification phase, the functional behavior of the satellite on-board computer may have been simulated by executing the control algorithms to show proper software function. During the validation phase, however, the actual satellite motherboard with actual flight code will be used instead of a simulated motherboard. This validates the actual implementation of the software on the real satellite hardware. Likewise, the sensor and actuator models will also be replaced with real hardware during the validation phase. Sensor data with exact signal wave-shape, type, and frequency may be fed into onboard control electronics to emulate the space environment.

Firms in the business of developing large satellites invest considerable resources in the development of complex simulation models to ensure that the software used in large satellites performs as expected. Although it is expensive to develop hi-fidelity simulations, the cost of doing so can be spread out over follow-on satellite programs. With the coming and going of each satellite program, the fidelity of the simulation models is improved to incorporate greater detail.

Small satellite developers also use simulation and testing to verify and validate software, but at a much reduced level of investment. For small satellites, a large development budget is seldom available. Indeed, the low-cost aspect of small satellite development is a key advantage of small satellites, thereby placing the building and launching of a satellite within the reach of university students, small firms, and small government agencies around the world. The development of an expensive software verification and validation process is understandably infeasible.

However, some aspects of the simulation and testing done for large satellites has been implemented at a reduced level of effort for small satellites. The testing of satellite software is often first performed by simulating the control algorithms on a desktop computer. As a next step, the small satellite motherboard may be directly connected to a desktop computer as shown in Fig. 4. This type of simulation and testing can help verify timing margins, identify interface problems, test sun and earth sensors, test gyro operation, check reaction wheel operation, identify errors in data bus transmission, test the real-time clock circuit operation, and test the telemetry input-output capability. However, most small satellite developers do not generally test the operation of the software using high-fidelity space environment and satellite dynamics models. Small satellite validation testing may be performed using an ad hoc collection of tools and software.

Some small satellite developers use multiple desktop computers to better simulate asynchronous inputs to the satellite software as shown in Fig. 5. One laptop or desktop computer may be used to simulate satellite sensor input signals while another might simulate the ground station, and a third could simulate an intelligent device such as a camera. By simulating these entities using independent computers, it is possible to independently vary inputs and outputs to verify that the software works for many simulated events.

One of the most common means of software assurance for small satellites (typically CubeSats, but other embedded system boards too) is the use of testing software and hardware provided by the makers of small satellite parts and software. For example, Pumpkin, Inc., the makers of the CubeSat Kits, markets a CubeSat Kit Flight Motherboard.[10] This board is not intended to fly, but it is electrically identical to the actual CubeSat control motherboard, so that when it is outfitted with the same processor pluggable

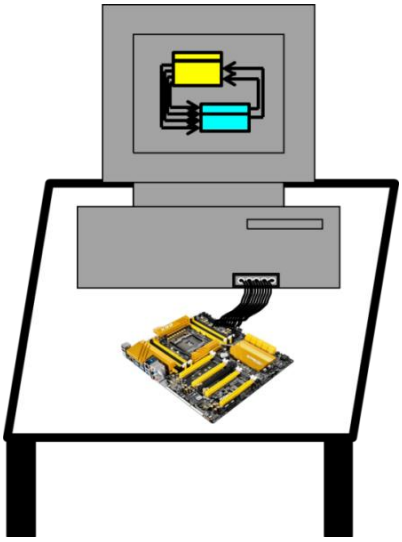

Figure 4: Satellite motherboard connected to desk top computer.

module (PPM) as the real motherboard, it is completely code compatible with the board supplied in the real CubeSat. The firmware developed for the CubeSat will run on the Motherboard without rebuilding. Because the motherboard is physically larger than the CubeSat flight board, it provides easier laboratory probing of all CubeSat electrical signals. Pumpkin also sells pre-written code for small satellites that can perform various housekeeping functions on CubeSats. This provides the users with a jump start on code development. Combined with the motherboard, it's possible to incrementally develop more complex code and note how signals change when a software build suddenly has problems relative to the last code build.

A closely related but somewhat more sophisticated approach for small satellite software and hardware verification is used at the NASA Ames Research Center. Ames uses intermediate hardware development and testing platforms referred to as DevSats, FlatSats, and EDUs. The use of these platforms in development of the EDSN satellites is described by Chartres, Sanchez, and Hanson in Ref. 11. A DevSat (development satellite board) consists of the processor and other chips that are hand-wired together on consumer grade development breadboards used for prototyping integrated circuits. These boards allow the early prototyping, development, and testing of the flight software and interfaces. After the actual satellite boards are developed, a special printed circuit board (PCB) is made that has the same data buses (backplane) as the actual satellite, but allows the boards to be connected along the its edges as indicated in Fig. 6, as opposed to the stacked geometry of the real satellite. The flat geometry of the PCB and the satellite boards is called a FlatSat. This configuration exposes the surface of every board in

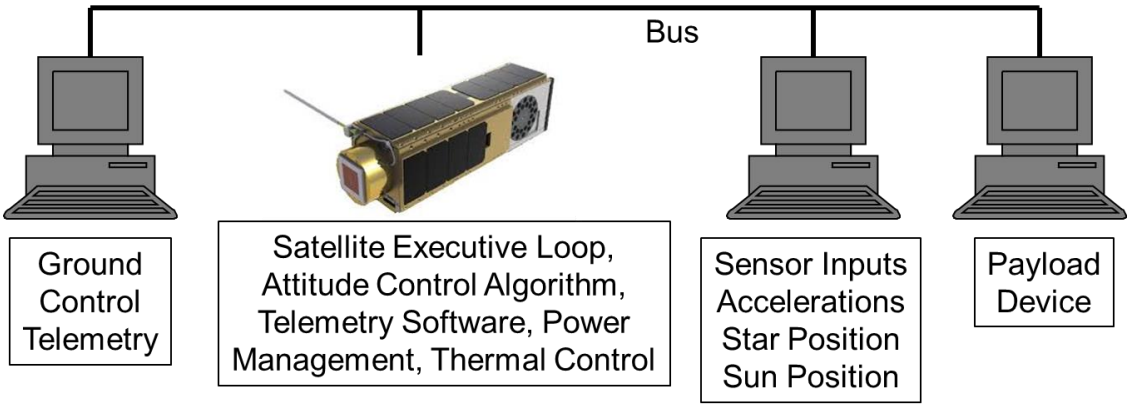

Figure 5: Simulation of sensor inputs and space environment using network of lab computers.

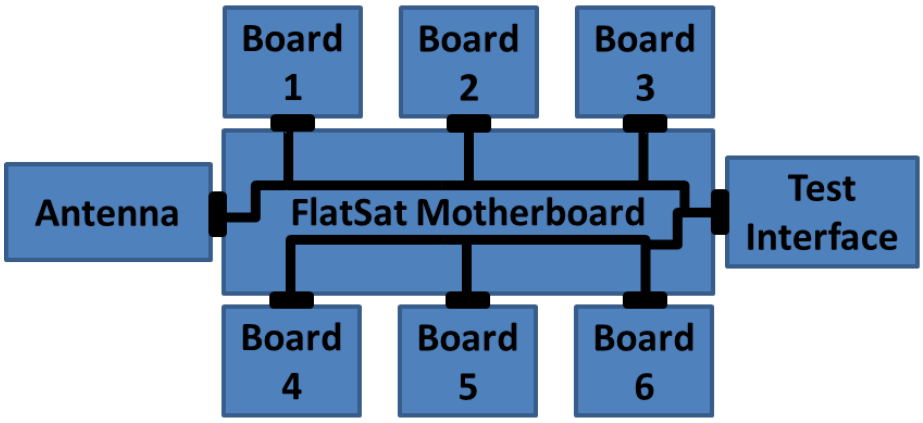

Figure 6: Configuration of small satellite boards into a FlatSat by interfacing to a custom motherboard.

the satellite to enable probing of the data lines and interfaces that are inaccessible when the boards are densely packed in the actual configuration of the small satellite. EDUs (Engineering Development Units) are nearly identical to the final small satellites, but are used for fit check of components, early system testing, ground station interface testing, and mission simulation. These satellites undergo extensive testing and may have plastic cases for safe handling if taken to remote testing sites at partner agencies.

The Ames approach to satellite software testing includes Workstation Simulation (WSIM), Processor-In-the-Loop (PIL) simulation, and Hardware-In-the-Loop (HIL) simulation testing. Gundy-Burlet describes these processes in the context of the Lunar Atmosphere Dust Environment Explorer (LADEE) spacecraft mission in Ref. 12. Although LADEE is not a small satellite, the methods used to verify and validate the software are beginning to be regularly applied to small satellites developed at Ames. PILs are useful for testing the software integration, timing, and communications. The WSIM testing uses a medium-fidelity model of the spacecraft and environment sufficient to test the flight software, but not to model every aspect of vehicle behavior. The model includeds limited spacecraft dynamics, a thermal model, and models of the electrical power and propulsion systems. The software is tested by executing small scripts that verify that the software requirements are satisfied. This process is facilitated by expressing all requirements in a manner to enable testing. The test scripts are used to explore module behavior and grow in number as the software development effort progresses. The test scripts are run on a weekly basis (typically over the weekend) to detect the introduction of software error as early as possible. The HIL testing performed on the EDUs represents the highest level of V&V and is used to assess the integrated performance and resource utilization of the software. The EDUs are taken to remote locations to test the integration with hardware and software being developed by partners at other sites.

Model-Based Software Design and Verification

The small satellite literature reveals that an emerging trend is to use automation to create detailed software designs from high-level graphical inputs, and then use automatic code generation to create the code. This process is called Model-Based Design (MBD) or Model Design-Driven (MDD) software development. Model-based design tools work to accelerate the iterative software design-code-test/verify development cycle. On the surface, it might seem like MBD tools have more to do with model development and code creation than verification and test. However, this is not true. Persons new to MBD tools generally find it easier to design code without MBD tools, and also easier to merge hand-written device drivers if MBD tools are not used. The reason for this is that there is a learning curve associated with learning how to model software using the MBD tools. For those who are willing to make this investment to learn, however, MBD tools offer substantial savings in the time needed to verify and validate the software.

Model-based design tools offer no help with getting the software requirements right, but they do help alleviate the much of the burden to make sure the software meets the requirements. The model-based design approach begins with a careful delineation of the software requirements, just as for the waterfall development process. However, after the textual requirements are complete, the MBD approach creates the high-level software architecture in the form of graphical models of the code that shows connections between program modules and describes the software behavior using statecharts, sequence diagrams, and/or timing charts. It takes time to learn how to model software using these constructs in a model-based development environment. A requirement of the model-based design approach is that the top-level software architecture must encapsulate software function within individual modules as depicted in Fig. 7. This requirement is imposed to guarantee that each module can function without dependence on other modules. This simplifies the automated creation of the software from the detailed design, and also enables direct re-use of software modules in other programs. The modules interface to each other so that control signals and data are passed between the modules. This graphical modeling of the software at a high level is usually more understandable to other design engineers trying to understand the function of the code, and hence promotes sharing and code re-use. It's much easier for a new user to apprehend the function of the code from a block diagram than reviewing commented code.

The model-based software development approach is shown in Fig. 8. Once the software architecture is specified in the model notation, the model-based design tools perform the detailed software design without the user having to worry about declaring variables, allocating memory, or anything else normally associated with writing a Java, C, or C++ program. The detailed software design created by the model-based design tools may be the actual code, but it can also be a metamodel that can be executed to verify that the software design meets the software requirements before the actual code is generated. This allows verification that the expected high-level behaviors of the software are actually triggered by high-level mechanisms. It can be executed to verify that all states of the system are reachable and that all communication interfaces work properly. The ability to easily and quickly execute the metamodel eliminates the need to perform unit testing. If problems using the metamodel or actual code are discovered, all debugging is done at the graphical model level (diagrams, statecharts).

Model-based design tools use the process of automatic code generation to keep the model in synch with the code. The code is automatically generated from the detailed software design or metamodel. All debugging to fix problems with the code occurs at the model level. Hand coded changes are not allowed because the introduction of hand-coded "fixes" would cause the code and model to get out of synch. Automatic code generation thereby ensures coherency between the model and the code which is important for software engineers desiring to re-use model elements in new programs. Generally, however, hand-coded device drivers must usually be integrated into the complete software at build time (unless the drivers are also produced using model-based design methods).

Model-based design tools, while taking some time to implement up-front, have the potential to save a lot of software development time and cost. These tools allow the automation to automatically do low-level

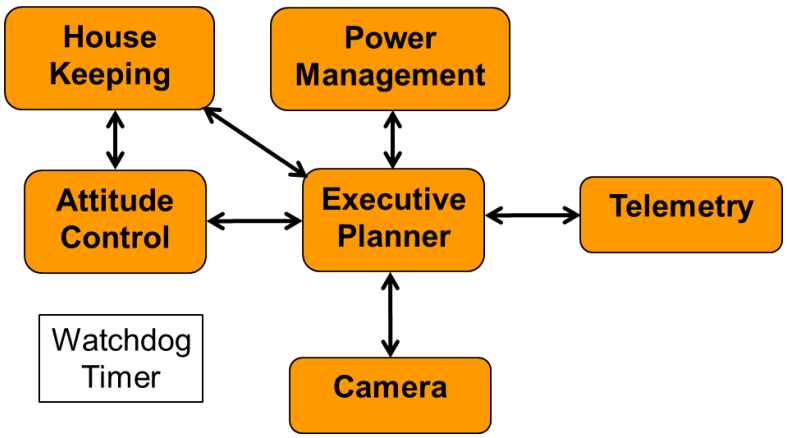

Figure 7: In model-based design, software function is encapsulated in modules.

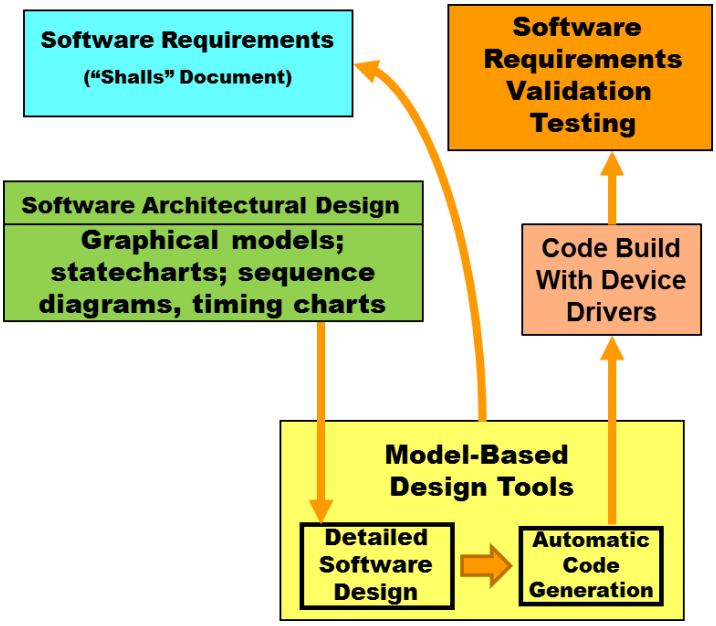

Figure 8: Model-based software development process.

software design, coding, unit testing, and automatic code generation. This allows the cyclic steps of software design, test, and validation to be iterated very efficiently and at low cost. The use of automatic code generators also enables another nice feature: being able to generate software in multiple languages, or for different platforms, without additional software design work. Several model-based design tools have the capability to generate code in different execution languages (e. g., C, C++, Java), thereby allowing easy migration of software between small satellites having different operating systems or different microprocessors.

There have been some reports in the literature that it is sometimes very difficult to use model-based design tools to generate code having the same flexibility and fidelity as that of hand-written code.[13] Fidelity can become an issue if the autocoder generates too much extraneous code or cannot accommodate external drivers well. Joining model-generated code with un-modeled driver code is a potential source of problems. For this reason, autocoding of the low-level drivers might offer a big payoff, but it would certainly be difficult.

There are many model-based design tools available. However, the model-based design tools that most stand out in the small satellite literature are the Unified Modelling Language™ and the MathWorks' MATLAB®/Simulink® model-based design tools. These are discussed separately below, but only briefly since a full discussion is well beyond the scope of this paper.

Model-Based Design Using the Unified Modelling Language™

The Unified Modelling Language™ is perhaps the most widely known and used model-based design and verification tool. The current specification of this modelling software is called UML2® and it is available for purchase from the Object Management Group (OMG).[14] UML® was used by NASA, CSA, and all other subcontractors to design the software architecture and all software components for the James Webb Space telescope (200,000 lines of C++ code).[15] UML® was used by the University of Alcala to design a small satellite real-time operating system.[16] CNES used UML to create autonomous software for the AGATA (gamma ray observatory) satellite.[13] The Spanish National Institute of Aerospace Technology used UML in the design of the Nanosat-1B and UPMSat-2 micro-satellites.[17]

UML® is primarily used to model object-oriented programs, but it can also be used to model procedural languages (e.g., FORTRAN). In UML®, the components are only allowed to communicate using message passing through interfaces. The components reactive behavior is defined by a Harel-type statechart and

the messages trigger the transition between the states. UML2® can be used to describe the interaction of software components using behavior diagrams, component diagrams, activity diagrams, sequence diagrams, class diagrams, use case diagrams, and communication diagrams, and may ultimately be used to generate code using one of several commercially available tools.

Most practitioners of UML2® do not purchase it directly from OMG, but rather use one of the comprehensive model-based design tools based on the UML2® specification. Some of the tools are available without charge. For example, EDROOM is a free tool developed by the Computer Engineering Department of the University of Alcala, Madrid, Spain for the development of real-time small satellite code from UML2 diagrams.[16] EDROOM is an acronym for Event Driven Real Time Object Oriented Modelling. EDROOM generates embedded C++ code from UML2. Most UML tools are not free but are widely available and quite powerful. Topcased by Astrium can support UML models having up to 250 segments, 1,000 classes (including interfaces), 4,000 attributes, 4,000 operations, 200 class diagrams, and 30 state diagrams.[18] Topcased UML editor can be used with the Acceleo tool to automatically generate code. Other UML2 design and code generation tools are No Magic's MagicDraw, Sparx System's Enterprise Architecture, Oracle's JDeveloper, and IBM's Rational Software Architect (RSA) and Rhapsody tools.

Model-Based Design With MATLAB®/Simulink®

MathWorks' MATLAB®/Simulink® has also been used to facilitate model-based design of small satellite software as embedded systems.[19] The Aerospace Corporation has designed and developed software for several cubesat satellites using MATLAB®/Simulink as embedded programs in MATLAB® and then auto-generated to C code using MATLAB® Embedded Coder. MATLAB® has been used by Lalbhai Dalpatbhai College of Engineering to develop a customized toolbox for implementing small satellite communications.[20] OHB AG used MATLAB® MBD tools to model and simulate code for the guidance, navigation, and autonomous control of the PRISMA satellite.[21] NASA Ames Research Center used MATLAB®/Simulink® to model, autogenerate code, and test the resulting software for the Lunar Atmosphere Dust Environment Explorer (LADEE) satellite.[12]

Like UML®, creating programs using the MATLAB® block diagram approach is much faster than coding in a traditional language (e.g., C, C++) because it is unnecessary to declare variables, specify data types, or allocate memory. MATLAB® excels in describing vector and matrix operations and provides a wealth of pre-written libraries supporting linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration, and ordinary differential equations. In MATLAB®/Simulink®, the components of the system are modeled as a series of interconnected boxes and the connection lines are data paths. MATLAB® has tools to connect MATLAB® based algorithms to external drivers written in languages such as C, Java, .NET, and Microsoft® Excel®. To aid in the development of model-based design, the Mathworks offers the StateFlow tool to generate state machine diagrams. In a state machine diagram, boxes in the diagram represent operational modes and are connected by arrows that are labeled with the conditions that result in transition from one state to another.

The Mathworks does not offer a special library or toolbox for creating small satellite software, but it does have many pre-written functions that can be used directly to facilitate small satellite software model-based design. For example, within the description of the MathWorks Communications Toolbox there is provided a model for a satellite RF communication link.[22] This model is designed using blocks (functions) taken from the Communications Toolbox. In addition, several third-party vendors do offer MATLAB® toolboxes that support model-based satellite software design. GPSoft LLC and L3NAV Systems offer MATLAB® toolboxes for global positioning systems (GPS) software design. GPSoft LLC also offers a toolbox to support the development and simulation of inertial navigation system software. Princeton Satellite Systems offers the Spacecraft Control Toolbox as a collection of over two thousand MATLAB® functions for satellite orbital dynamics, control systems, propulsion systems, formation flying, and spin axis attitude determination.

The MathWorks offers several options to create auto-generated code.[23] MATLAB® Coder™ generates standalone C and C++ code from MATLAB® code. Simulink® Coder™ generates C and C++ code from Simulink® diagrams, Stateflow® charts, or MATLAB® functions. Embedded Coder® generates C and C++ code for use on embedded processors, such as small satellite applications. Using Embedded Coder®, it

is possible to link to non-MATLAB® code in the build process to produce an executable code for an embedded system. Embedded Coder® also provides traceability from requirements to models, test cases, and generated code using Simulink® Verification and Validation™. Using this tool, requirements entered through software like IBM's Rational DOORS™ can be linked to models and to code.

Software Assurance Using Formal Methods

Formal methods use mathematical techniques to prove that the coding logic used in a program is correct. Unlike traditional software testing which attempts to find bugs in the program, formal methods seek to prove the absence of error. There are two types of formal methods. One type is called static analysis, the other type is called dynamic analysis and includes model checking methods and run-time monitoring.

Static Analysis Verification Methods

Static analysis methods do not execute the code. They are an outgrowth of compiler technology and many modern compilers offer some form of static analysis. Compared to dynamic methods, static analysis tools are easy to run and can quickly find violations of coding standards and naming conventions. Static analysis methods may also be used to find lexical, syntactic, and semantic mistakes, as well as to identify code that may produce overflow, divide-by-zero, or out-of-bounds array access errors.

There are many vendors of static analysis tools. One example is the MathWorks Polyspace Code Prover™. This tool can find conditions under which variables exceed their specified range limits and can detect code that produces overflow, divide-by-zero, out-of-bounds array access, and other run-time errors in C and C++ source code. Other static analysis tools that do similar analyses include Coverity's SAVE, Klocwork's Inspect, Grammatech's CodeSonar, Parasoft C/C++test, Parasoft Jtest, Parasoft Insure++, LDRA's Testbed, GNU's Cppcheck, and NASA's IKOS. Static analysis tools have been designed for programs written in Ada, C, C++, C#, COBOL, visual basic, Pearl, Python, XML, .NET, PHP, Java, and JavaScript.[24] Some static analysis tools (e.g. Cppcheck) are free.

No examples of static analysis tool usage is seen in the literature for small satellites, most likely because using these programs is relatively easy and it is considered good programming practice to use them. They can find common programming mistakes referenced above, but cannot find logic errors.

The most common problem encountered using static analysis programs is that they have a tendency to produce too many false alerts and warnings. Although these can be ignored, the problem is that once potential problems are identified, software developers tend to take time to check that each one really is a false alert. This can discourage usage. An active area of research is to find static analysis methods that produce fewer false warnings.

Verification Using Model-Checking Methods

Dynamic formal methods execute the software to find undesired execution paths, deadlocks, and race conditions. Depending on the size of the code being verified, model checking can be a useful method to verify program state and mode logic, complex switching functions, Boolean mode logic, and solving constraint problems. Examples of model checkers are NASA's JavaPathfinder, Cadence SMV, Carnegie Mellon's NuSMV, Bell Lab's SPIN, The Mathworks CodeProver, University of Oxford's PRISM, Uppsala University's UPPAAL and the MathWorks Simulink® Design Verifier™ tool. Model checkers can be used to detect dead logic, integer overflow, division by zero, and violations of design properties and assertions. Model checkers vary somewhat in the way they represent software, and may be referred to as explicit state model checkers, bounded model checkers, symbolic model checkers, or probabilistic model checkers. All model checkers operate on a discretized model of the software. An informative description of model checking, as well as a comparison of the Simulink® Design Verifier tool and the SPIN model checker, is presented by Leitner.[25]

All model checkers seek to provide proof of the absence of error by performing an exhaustive search of every possible path through the software to verify that no erroneous results are produced. Although

some satellite subsystems have been partially verified by model checking, to date, this author knows of no one who has been able to use model checking to verify the software for a complete satellite. The difficulty in using model checking is caused by the restrictions relating to the desire to exhaustively search every possible path through the software.

First, model checking methods work by representing software as finite state machines. In computer science, a finite state machine is basically an entity that can be in only one of a number of discrete states at any moment in time. So for example, the pressure inside an automobile tire may be allowed to take on state values of too low, normal, or too high, but representing the pressure as a real variable (e. g. 32.5 psi) is not allowed. Model checkers can't use floating point variables to define states because then an infinite number of states would be required, which would then require an infinite amount of time to analyze every state. For small satellite software verification, this means that both the internal state of connected modules and the variables used therein must be discrete. In Fig. 9, the discretization of the input variables is shown by the changes indicated by the red arrows. The real variable describing the angular orientation to the sun can't be expressed as a real value in degrees, but must be reduced to a Boolean variable of sun exposure or not. Similarly, a current is not represented in Amps, but rather if it exceeds a given value or not. Additionally, within the program modules, the execution state or branch points within the software modules must be indicated by variables such as true or false, A or B, or yes or no. The small connected shapes in the orange boxes of Fig. 9 are meant to depict the flowchart of algorithms within the modules. The red diamonds are decision or branch points in the software. Each of these decision points adds to the state of the software. The state of the program as a whole is the n-dimensional vector depicting the state of each input variable, every intermediate variable, and every decision outcome within the program. The problem encountered in model checking is that of the state explosion problem, whereby the number of possible states for a continuous, time-domain system may become very large, even with suitable discretization of continuous variables.

A second requirement before model checking can be used is that both the model and all desired functional properties of the software must be specified or stated in temporal logic, a precise mathematical specification. Once a property can be phrased as a mathematical assertion, then every path through the software model (state machine) can be exhaustively exercised to make certain that every path satisfies the assertion. If a path through the model can be found that does not satisfy the property, that path is recorded as a counterexample which shows the error path. The pitfall with this approach, however, is

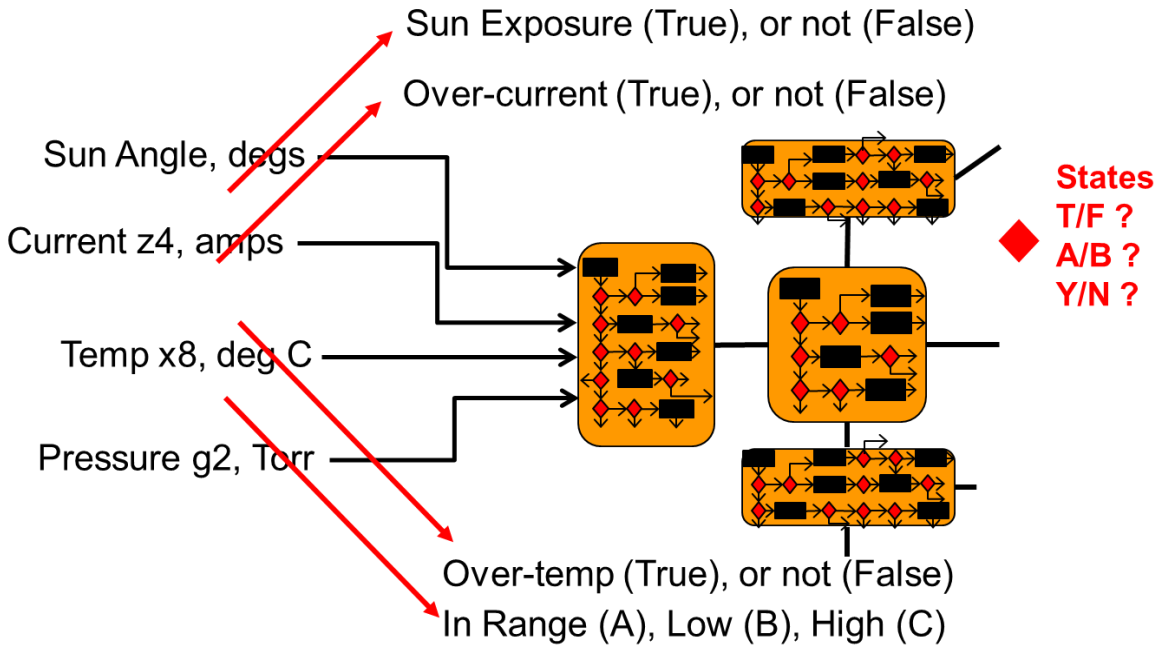

Figure 9: Discretization of model into finite state inputs and machine states for use in model checking.

that it may be difficult to express intuitive, easy-to-understand, satellite operational requirements as mathematical assertions or properties.

To this author's knowledge, the discretization and temporal logic requirements has prevented anyone from using model checking to verify the complete software for any satellite. This is because it's very difficult to model check the interaction of dozens of microprocessors in multiple devices, a large number of sensors using disparate data types, many modes of operation, and many system states. All of these complexities make formal analysis very difficult to perform whether for large or small satellite software. Even small satellites are truly systems-of-systems. (As an aside, one might argue that there is an opportunity to improve small satellite software design by removing floating point values from the software (i.e., have all Boolean operators) and creating operational modes that are easier to express in temporal logic.) Although no one has been able to successfully apply model checking methods to completely verify the correctness of a whole satellite, there have been some partial successes verifying satellite subsystems, like an attitude control system or FDIR system. Described next are two examples to illustrate the difficulty and partial success of applying model checking to satellite software verification.

The Software Systems Engineering Section of the European Space Agency has used the COMPASS toolset in an attempt to perform bounded and symbolic model checking of satellite software based on NuSMV model checker.[26] The authors acknowledged that they could not hope to model the complete software, and so they chose to analyze only the fault detection, isolation, and recovery (FDIR) software of an un-identified satellite project currently in progress. The FDIR system was required to detect failures and implement recovery actions. Once the FDIR software detects an error condition, it can restore the spacecraft to a nominal condition, restore it to a degraded condition, or place it in a safe configuration. (Ref. 27 provides a good discussion of FDIR logic developed in support of the Navy PANSAT micro-satellite.) The FDIR system typically has to handle hundreds to thousands of recovery procedures due to the many operational modes, mission phases, satellite states, and use of redundancy in the hardware.

The application of the COMPASS model checker to the satellite software, written in Architecture Analysis and Design Language, (AADL), proved to be very difficult. First, discretization of the outputs for the sun sensors and current levels was required to avoid a combinatorial explosion of the state space. Hence the value of the sun sensor was not in degrees of sun ray impact, but rather abstracted to a Boolean indicating sun exposure (or not), and for the current, a Boolean indicating an over-current (or not). Second, some measurements (temperature changes or fluid pressure) needed to be incorporated into the model without discretization in order to check compliance of a range of temperature or pressure. To avoid a state explosion problem, simplified models of satellite operation were required to express valid ranges. Third, real-time correctness required the addition of a timer. This requirement happens because it is not sufficient that the correct recovery procedure be chosen; it must be chosen in the required time allowed. Mode invariants were added to force transitions when the invariants became invalid by the passage of time.

The approximations above, while required to implement model checking, also may have introduced pathways for error. The use of simplified states and operational modes strikes against the thoroughness of using formal methods in the first place. At a later point, all such approximations must be revisited to consider their effect on the final outcome. Even having made these simplifications, it turned out that most of the requirements were not expressible as properties in temporal logic. From the several thousands of requirements identified, the team was able to only analyze 106 requirements, and ultimately down-selected to 24. Even so, a model was constructed that had nearly 4,000 lines of code and nearly 50 million states.

The analysis was run on two 64-bit, Linux computers; each having a 2.1 GHz AMD Opteron CPU and 192 GB of RAM. Although in some cases the model checker produced results after only a few to tens of seconds, other cases took considerably longer. For the double earth sensor failure condition, the analysis was stopped after 7 days without results. In attempting to determine the reliability of the satellite in the presence of a sensor failure, computer memory was exhausted after nine hours. All told, the team was able to verify 16 of the 24 properties.

As a second example, Ref. 28 reports the use of symbolic model checking to verify satellite attitude and orbit control system (AOCS) software. The executable code for the AOCS software was provided by Space Systems Finland to Aalto University Department of Information and Computer Science for analysis. The requirements of the system behavior were provided in the form of extended state machine diagrams with prioritized transitions. These diagrams were translated to a set of temporal logic properties, allowing the piecewise checking of the system behavior, one extended state machine transition at a time. In order to use symbolic model checking, the Ada implementation of the system was modeled using the input language of the symbolic model checker NuSMV2. Like the previous example, various modelling techniques and abstractions were needed to avoid a state explosion problem due to the handling of timers and the large number of system components controlled by the AOCS. Bounded model checking was required for tractability. In bounded model checking, abstractions are employed to limit the state space to prescribed regions. The authors concluded that even after the abstractions, the AOCS system was too large to be fully automatically verified using symbolic model checking methods. The NuSMV2 model code comprised about 800 lines and had about 80 variables and 25 state labels. The model had $2^{124.483}$ states, of which 84.5 million states were reachable.

This study used a computer cluster having two, 6-core, AMD Opteron CPUs and 32 GB of RAM. Using bounded model checking, the team was able to analyze about half of all the checked properties while for the other properties the approach timed out. Even for these cases, however, the incremental bounded model checking algorithm was thought useful to show the non-existence of short counterexamples for the properties that timed out. Space Systems Finland felt that using the bounded model checking algorithm to verify 12 out of 28 properties was acceptable from an industrial perspective. While this is good news, it must also be tempered by the realization that the AOCS code represented only a part of the total satellite software.

Fault-Tolerant Software Design and Runtime Monitoring

An important consideration of small satellite software design is that the verification and validation effort may be unable to detect all software errors. Small satellite software malfunctions include, but are not limited to failure of the operating system, live-lock of the operating system, failures in one or more threads, data corruption problems, and watchdog computer failures.[24] The lack of an elaborate software V&V program may increase the frequency of these errors in small satellites. However, small satellite software may also suffer anomalies caused by space radiation damage to various memory locations. The disastrous effects of radiation flipping a single bit in memory, referred to as the Single-Event-Upset (SEU) or Single-Event-Latch-up (SEL) failure is well-documented in the literature.[29, 30] As such, it is wise to plan ahead for the detection and correction of software problems in space. This is referred to as fault-tolerant software design.

The most common approach for adding fault-tolerance is the use of redundancy for critical systems. Redundancy may include the use of two on-board computers, two inertial reference units, four gyros in each direction, etc. In order for this approach to be successful, it is necessary to have a failure detection scheme that can direct when switching to backup hardware is required. One approach for implementing this approach is presented by Hunter and Rowe for the PANSAT satellite and has been used many times since.[26] However, as demonstrated in the case of the Ariane 5 accident, redundancy does not always protect against common mode failures where the same software error causes both primary and secondary systems to fail.[31] In that case, a software exception fault code was erroneously sent to both flight computers as valid data, thereby causing the rocket to veer off course. Some software fault-tolerance schemes guard against this by requiring that dissimilar software algorithms be used in redundant hardware. This approach is one method suggested by the Federal Aviation Administration in DO-178C.[5]

The second most common means of adding fault-tolerance, and for implementing real-time software verification, is the use of run-time monitoring. Run-time monitoring is generally used to debug code still in development on the ground, but it can be applied in the space environment. In runtime monitoring, the program is “instrumented” or programmed to provide information on the program status at any desired

point. For example, the software can be programmed to dump register information every time a code segment is executed or a break point can be set in the program that allows register information to be sent to the ground station. The two most common means of run-time monitoring for small satellites are the use of a watchdog processor and the procedure of functional telemetry.

A watchdog program can compare the results of two or more redundant computers or devices to check for agreement and declare an error in the case of disagreement. A watchdog processor can also periodically request each program segment (or thread) to report its status. If no reply is received or a bad reply is received, the watchdog program can raise an error flag and notify the ground station. Threads producing exceptions like divide by zero, overflow, and underflow, may be programmed to report them to the watchdog processor. The watchdog processor can also keep track of the execution time of each thread. Provided that the worst case execution time is known for every thread or process, the watchdog processor can raise a flag to indicate a thread is taking longer than expected to complete and is perhaps live-locked.

Another method of run-time monitoring is implemented through functional telemetry. To use this method, each thread is programmed to compute a known result, test if the expected result is produced, and if not, set an error flag indicating a failure has occurred. In functional telemetry, each thread is required to send telemetry to the ground station as it executes to report the status of these tests.[32] Although the ground station may receive a lot of extraneous information when everything is working right, such information is very valuable to pinpoint the cause of software anomalies when things go wrong.

Of course, once the satellite has been placed into orbit, the value of verifying correct software behavior operation would have little value were there not the means to do something about it. Most large satellites and some small satellites have the capability to upload corrected software. This is enabled by using software that allows the ground station to always be able to communicate with the BIOS of the satellite. From talking to the BIOS, the ground station can first download telemetry and log files after a crash from satellite flash memory to help discover the cause of the software problem. After the problem with the software is discovered and corrected on the ground, the link to the BIOS can be used to direct the satellite to upload new software from the ground station and then to restart. This method of satellite access is called the “back door” into the operating system and it provides a means to recover from software failures after launch.

Concluding Remarks

This paper has attempted to provide an unbiased assessment of the software assurance methods currently used to provide small-satellite software verification and validation. In doing this survey, the verification and validation methods used for both large and small satellites were examined.

The survey found that the most common method of software assurance is simulation and testing. Elaborate simulations and test facilities have been developed by the makers of large satellites over the last 50 years, which has put in place detailed processes, high-fidelity simulations, and test practices that produce a low rate of software anomalies. Although the method of simulation and testing is also performed to some extent for small satellites, the fidelity and rigor of this testing is less comprehensive. The reason for this partly stems from the fact the low-budget, quick turn-around, small satellite programs cannot fund expensive, time-consuming software verification and validation programs. The other reason for this is that since the cost of a small satellite is low, the option to launch a replacement satellite for a failed predecessor is viable. Nevertheless, some aspects of the simulation and testing done for large satellites are being implemented at a reduced level of effort for small satellites. This includes the use of DevSats, FlatSats, and software simulations of varying degrees of fidelity.

Although the literature reveals that V&V by simulation and testing has by far the longest legacy, model-based design methods are becoming more prevalent because of their capability to accelerate the iterative software design-code-test/verify development cycle. Of most importance, model-based design tools

enable the early validation of the software requirements. Model-based design tools use graphical models, statecharts, and other constructs to specify the software architecture. The model-based tools create the software detailed design so that the software engineer need not allocate memory, declare variables, or write iteration loops. The detailed design can be executed as a metamodel before code is generated to ensure that the software model meets all functional requirements. This greatly lowers the cost of the design-code-test/verify cycle of software development. When this development has been done, the model-based design tools auto-generate the software that will run on the target system for validation testing. Should errors be found, all software debugging is done by changing the model and then auto-generating the code. Hand-coded fixes are not allowed in order to ensure coherency of the code to the model. The benefits of model-based software design include reduced verification time and the ability to better share code between software engineers. Automatic code generation also has the advantage of being able to generate code for different platforms or operating systems without needing to create a different model. A drawback of using model-based design tools is that some people have found that the auto-generation tools initially seem not to have the same flexibility as writing code by hand. However, the literature indicates that this appears to be more of a learning curve issue rather than an actual limitation of the model-based design tools.

Verification using formal methods (static analysis, model checking), though sometimes difficult to implement, may offer new ways to improve small satellite software verification and validation. Static analysis is relatively easy to apply and can catch some types of programming errors, but is limited in scope. Static analyses can be used to find coding violations, overflow conditions, and undefined variables. However, static analyses cannot detect logic errors or find problems like live-locks or race conditions in concurrent systems. While model checking methods can find these types of errors, these methods need to be further advanced to deal with the state explosion problem to make them more usable by small-satellite software engineers. In addition, designing the software architecture to be more amenable to formal analysis is a topic for future research. For example, if the software was originally designed using only discrete abstractions and Booleans, instead of using floating point variables, perhaps better software designs would result due to clearer logic.

Last, fault-tolerant software design methods can be used to implement run-time monitoring as a means of software verification either before or after launch. Run-time monitoring, combined with fault-tolerant software design, can effectively deal with either software errors that escape the verification process prior to launch or those that are discovered after launch, perhaps produced by the effects of ionizing radiation. The most prevalent scheme seen in the literature is the use of redundant microprocessors, components, and sensors, with a watchdog processor employed to identify failed components. The second most common means of adding fault-tolerance, and for implementing real-time satellite software verification, is the use of run-time monitoring. When applied in the space environment, the technique of run-time monitoring can be used to detect and report software anomalies to either the watchdog processor or the ground station. This can be used to enable functional telemetry to alert the ground station of software failures. It is possible to correct the software of satellites already placed into operation if the ground station software has been given the capability to communicate with the BIOS of the satellite. With this feature, the ground station can direct the BIOS to accept new software uploaded from the ground station, thereby enabling recovery from software failures discovered after launch.

References

[1] "Optimization Software Improves Small, Low-Cost Satellite Design," Sara Black, Composites World Magazine, Aug. 20, 2013. http://www.compositesworld.com/articles/optimization-software-improves-small-low-cost-satellite-design

[2] http://www.nasa.gov/mission_pages/smallsats/

[3] "Cheaper by the Dozen: The Avalanche of Rideshares in the 21st Century," M. Swartwout, 2013 IEEE Aerospace Conference, 2-9 March 2013. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6497182

[4] "25 Years of Small Satellites," Siegfried W. Janson, 25th Annual AIAA/USU Conference on Small Satellites, 2011. http://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1117&context=smallsat

[5] "RTCA DO-178C, "Software Considerations in Airborne Systems and Equipment Certification," December 2011.

[6] "Software Development and Test Methodology for a Distributed Ground System," George Ritter, SpaceOps 2002 Conference, 2002. http://arc.aiaa.org/doi/pdf/10.2514/6.2002-T2-69

[7] "An Independent Software Verification and Validation Process for Space Applications," Ana Maria Ambrosi, Fatima Mattiello-Francisco, and Eliane Martins, SpaceOps 2008 Conference, 2008. http://arc.aiaa.org/doi/pdf/10.2514/6.2008-3517

[8] "Small-Satellite Software Design, Architecture, Verification and Validation," Stephen Jacklin, NASA TM in progress, date TBD.

[9] "Achieving Software Validation Through Simulation," Loren Slafer, Boeing Satellite Systems, Applied Dynamics Conference, 2001. http://www.adi.com/wp-content/uploads/2012/08/BoeingSatelliteSoftwareValidation.pdf

[10] "CubeSat Kit Development Board," Pumpkin, Inc., web, 2009. http://www.cubesatkit.com/docs/datasheet/DS_CSK_MB_710-00484-D.pdf

[11] "EDSN Development Lessons Learned," James Chartres, Hugo Sanchez, and John Hanson, 28th Annual AIAA/USU Conference on Small Satellites, Aug 2014.

[12] "Validation and Verification of LADEE Models and Software," Karen Gundy-Burlet, 51st Aerospace Sciences Meeting, January 7-10, 2013.

[13] "Model-based specification of the flight software of an autonomous satellite," Jeremie Pouly (CNES) and Sylvain Jouanneau (ALTEN SO), ERTS2 2012 - Embedded Real-Time Software and Systems - European Congress, 2012. http://www.erts2012.org/Site/0P2RUC89/5A-2.pdf Unified Modeling Language™ (UML®) Resource Page, www.uml.org

[14] Unified Modeling Language™ (UML®) Resource Page, www.uml.org

[15] "IBM Code to go Aloft with NASA Space Telescope," CNET Magazine, January 19, 2007. http://news.cnet.com/IBM-code-to-go-aloft-with-NASA-space-telescope/2100-11397_3-6151505.html

[16] "EDROOM: A Free Tool for the UML2 Component Based Design and Automatic Code Generation of Tiny Embedded Real Time System," A.V. Sanchez, O.R. Polo, O.L. Gomez, M.K. Revuelta, S.S. Prieto, and D.M. Luna, Proceedings of the 3rd European Congress on Embedded Real-Time Software (ERTS), 25-27 January 2006. http://www.researchgate.net/publication/247158442_EDROOM_a_free_tool_for_the_UML2_component_based_design_and_automatic_code_generation_of_tiny_embedded_real_time_system

[17] "Component Based Engineering and Multi-Platform Deployment for Nanosatellite On-Board Software," Óscar R. Polo, Pablo Parra, Martín Knoblauch, Ignacio García, Javier Fernández, Sebastián Sánchez, and Manuel Angulo, Proc. 'DASIA 2012 – Data Systems In Aerospace', Dubrovnic, Croatia, 14–16 May 2012. http://www.researchgate.net/publication/247158608_COMPONENT_BASED_ENGINEERING_AND_MULTI-PLATFORM_DEPLOYMENT_FOR_NANOSATELLITE_ON-BOARD_SOFTWARE

[18] "Using TOPCASED UML Editor for Satellite On-Board Software Design," Unknown, Trade conference on TOPCASED, Jan 14, 2011. http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CCgQFjAA&url=http%3A%2F%2Fgforge.enseeiht.fr%2Fdocman%2Fview.php%2F52%2F4282%2FA2-Astrium.pdf&ei=wVI0U6D8Lsi1sASg24CYBw&usg=AFQjCNF6-p9sy36x9aSOL2VO7ygPIUoXiA&bvm=bv.63738703,d.cWc

[19] MathWorks' MATLAB® and Simulink®: http://www.mathworks.com/products/MATLAB®/features.html

[20] "Design of Satellite Communications Toolbox for MATLAB®," Kiram Parmar, et al, International Journal of Electronics and Computer Science Engineering, 2012. http://www.ijecse.org/wp-content/uploads/2012/08/Volume-1Number-2PP-356-363.pdf

[21] "OHB Develops Satellite Guidance, Navigation, and Control Software for Autonomous Formation Flying," Mathworks User Story, 2014. http://www.mathworks.com/tagteam/79238_91872v02_OHB_UserStory_final.pdf

[22] MathWorks' Communications Toolbox http://www.mathworks.com/help/comm/examples/rf-satellite-link.html

[23] MathWorks' Auto-coding tools: http://www.mathworks.com/products/?s_tid=gn_ps

[24] http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

[25] "Evaluation of the Matlab Simulink Design Verifier Versus the Model Checker SPIN," Florian Leitner, Technical Report soft-08-05 (and Thesis), University of Konstanz, Department of Computer and Information Science, 2008. http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-08-05.pdf

[26] "Formal Correctness, Safety, Dependability, and Performance Analysis of a Satellite," Marie-Aude Esteve , Joost-Pieter Katoen, Viet Yen Nguyen, Bart Postma, and Yuri Yushtein, 34th International Conference on Software Engineering, 02 – 09, Jun 2012. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=06227118

[27] "Software Design for a Fault-Tolerant Communications Satellite," G. Ken Hunter and Neil C. Rowe, 2000. http://www.dodccrp.org/events/2000_CCRTS/html/pdf_papers/Track_2/011.pdf

[28] "A Symbolic Model Checking Approach to Verifying Satellite Onboard Software," Xiang Gan, Jori Dubrovin, and Keijo Heljanko, Proceedings of the 11th International Workshop on Automated Verification of Critical Systems, 2011. http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=17&ved=0CFgQFjAGOAo&url=http%3A%2F%2Fjournal.ub.tu-berlin.de%2Feceasst%2Farticle%2Fdownload%2F681%2F699&ei=DVRMU4mpCcK0yAGR4YHYDQ&usg=AFQjCNHykCoXvjjZ71B11FbUOXlFPUMAQg&sig2=pbxTihoNw4EUpuiKCgUi

[29] "Mission Results and Anomaly Investigation of HORYU-II," Yuki Seri, Hirokazu Masui, and Mengu Cho, 27th Annual AIAA/USU Conference on Small Satellites, Aug 12-15, 2013. http://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=2984&context=smallsat

[30] http://en.wikipedia.org/wiki/Single_event_upset

[31] "ARIANE 5 Flight 501 Failure," report by the inquiry board, July 19, 1996 http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[32] “Fault Management on Communications Satellites,” R. D. Coblin, Lockheed Martin Missiles & Space, 1999. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00822707