Development of Software-only Simulation Test Beds (SoST) for Spacecraft and SmallSats

Scott Alan Zemerick
szemeric@mix.wvu.edu

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
Zemerick, Scott Alan, "Development of Software-only Simulation Test Beds (SoST) for Spacecraft and SmallSats" (2020). Graduate Theses, Dissertations, and Problem Reports. 7888.
https://researchrepository.wvu.edu/etd/7888

This Dissertation is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Dissertation has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.
Development of Software-only Simulation Test Beds (SoST) for Spacecraft and SmallSats

Scott Alan Zemerick

Dissertation Submitted to the Benjamin M. Statler College of Engineering and Mineral Resources at West Virginia University

In partial fulfillment of the requirements for the degree of Doctor of Philosophy

In Computer Engineering

Powsiri Klinkhachorn, Ph.D., Chair
Thirimachos Bourlai, Ph.D.
Jeremy Dawson, Ph.D.
Jason Gross, Ph.D.
Roy Nutter, Ph.D.

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2020

Keywords: spacecraft, flight software, V&V, CPU instruction set simulators, avionics

Copyright © 2020 Scott Alan Zemerick
ABSTRACT

Development of Software-only-Simulation Test Beds (SoST) for Spacecraft and SmallSats

Scott Alan Zemerick

Software-only-Simulation Test Beds (SoST) are beginning to become more popular among aircraft, spacecraft, and smallsat embedded system developers due to the high cost of duplicating hardware test beds. Traditionally, large Department of Defense and NASA projects duplicate rooms of expensive “one-of-a-kind” Engineering Test Units (ETUs) for unit testing and integration testing prior to operational deployment. The problems with this approach are many; most notably, hardware is limited, which creates a “scarce resource” problem that causes testing, specifically software (integration) testing, to suffer.

SoSTs provide a software-only, or virtual test bed, that creates a “digital twin” that contains software models of the ETUs and often includes modeled components such as flight computers, busses (e.g., MIL-STD-1553, SPI, I2C), compact PCI (cPCI) backplane cards, sensors, and actuators. The ultimate goal of a SoST is for it to run the native system software compiled-binary on its native CPU architecture (e.g., PowerPC, LEON3/4, ARM) on a standard X86 personal computer/laptop without needing to recompile for X86. This methodology maintains the “Test-As-You-Fly” approach and is a powerful capability when used in tandem with hardware ETU test beds.

SoST integration into large projects is beginning to increase, with benefits being immediately realized. When implemented correctly, both technically and managerially, SoSTs can help solve the hardware “scarce resource” problem by providing a nearly unlimited test resource that can be utilized for earlier software unit testing, integration testing, and operator training, which results in schedule relief and improved quality and mission assurance. However, the processes for scoping, designing, implementing, and deploying SoSTs are immature and unknown to engineers and their management, which can often lead to misconceptions, confusion, and project failure.

This dissertation focuses on developing the first SoST process guide using several past spacecraft missions and smallsat missions as examples. Also, West Virginia’s first spacecraft, Simulation-To-Flight-1 (STF-1), and its accompanying SoST, among other missions, will be utilized as case-studies and will cover the modeling of flight computers, avionics, dynamic simulators, and ground system components. This research includes technical engineering detail as well as new approaches to modeling specific hardware components. Analysis and conclusions are provided based upon measurable results. Benefits of SoSTs will be shared along with specific examples of increased spacecraft mission assurance.
Acknowledgements

My last written acknowledgement was in 2002 for my M.S.E.E. thesis. Then, I expressed special gratitude to my girlfriend for her perpetual encouragement. Eighteen years later, Cindy is now my wife, and we have two wonderful daughters. Without her continued encouragement, this would not have been possible.

To both Sarah and Kayleigh, may this serve as a real example that you are very capable of doing anything that you desire with diligent and persistent work. Completing this has been one of the hardest things I have ever accomplished.

I want to express my sincere thanks to Dr. Klinkhachorn for his precious guidance and suggestions throughout my career. Enjoy retirement, Dr. Klink! Also, I sincerely appreciate those serving on my committee.

This thesis is dedicated to my parents because if it were not for their support and encouragement my entire life, none of this would exist.

Finally, I want to express my special thanks to the Katherine Johnson NASA Independent Verification and Validation (IV&V) Independent Test Capability (ITC) team. You are the most capable team that I have ever had the privilege of working with. We have singlehandedly made an enormous impact on the NASA agency.

Ad astra per aspera.
Contents

Acknowledgements ........................................................................................................................................ iii

1 Introduction ............................................................................................................................................... 1
  1.1 Background ......................................................................................................................................... 1
  1.2 SoST Definition and Functional Modeling of Embedded Systems ......................................................... 3
  1.3 Problem Statements and Motivations ..................................................................................................... 7
  1.4 Requirements of a Software-only Simulation Test Bed (SoST) to Address Problem Statements .......... 11

2 Spacecraft Background, Literature Review, and SoST Advantages ....................................................... 17
  2.1 Space System Segments ..................................................................................................................... 17
  2.2 Literature Review and Prior Work ....................................................................................................... 19
  2.3 SoST Invention Development and Maturation ..................................................................................... 29

3 Design... of a Software-only Simulation Test Bed (SoST) for Flagship Spacecraft Missions ........ 33
  3.1 Introduction ......................................................................................................................................... 33
  3.2 SoST Design Process .......................................................................................................................... 33
  3.3 Technical Design and Modeled Components ......................................................................................... 36
  3.4 Implementation and Integration ........................................................................................................... 53
  3.5 Deployment and Usage ........................................................................................................................ 69

4 SoSTs for SmallSats and CubeSats ...................................................................................................... 77
  4.1 Introduction ......................................................................................................................................... 77
  4.2 Considerations for Designing a SoST for a SmallSat Mission .............................................................. 78
  4.3 SoST: The NASA Operational Simulator for SmallSats (NOS3) .......................................................... 80

5 Engineering Roadmap for Scoping, Designing, and Implementing a SoST .......................................... 90
  5.1 Beginning SoST Design and Implementation ....................................................................................... 91
  5.2 SoST Hardware Modeling Best Practices ............................................................................................ 93

6 SoST Verification Analysis, Problem Statement Review, and Benefits Realized ................................. 95
  6.1 Methods of SoST Verification and Analysis ......................................................................................... 95
  6.2 Problem Statement Review and Benefits Realized ............................................................................. 104
  6.3 Direct Benefits: Metrics for Flight Software Issues Identified During SoST Dynamic Analysis ........ 111

7 Conclusions and Future Work Areas .................................................................................................. 113
  7.1 Conclusions ....................................................................................................................................... 113
  7.2 Future Work Areas ............................................................................................................................ 114

8 Bibliography ........................................................................................................................................... 121

9 Publications, Citations, Awards, and Presentations ........................................................................... 125
Table of Figures

Figure 1. Traditional Electrical Function Model (EFM) ................................................................. 5  
Figure 2. Electrical Functional Model (EFM): SoSTs Implement in Software ..................................... 6  
Figure 3. Three Segments of Space Systems ....................................................................................... 7  
Figure 4. cFS “Lollipop Chart” Showing the cFS Architecture and Apps ............................................ 18  
Figure 5. SoST featured in Wind River Simics Book Citation .............................................................. 25  
Figure 6. Workflow for Combining Static and Dynamic Analysis ......................................................... 29  
Figure 7. SoST Custom Design Process ............................................................................................. 34  
Figure 8. SoST Main Architecture ...................................................................................................... 37  
Figure 9. Example Ground Software System ....................................................................................... 38  
Figure 10. Publicly Available JIST Configuration Showing Virtual Machine Configuration .................. 39  
Figure 11. Example Control Loop for Spacecraft Attitude ................................................................. 40  
Figure 12. NOS Engine SoST Architecture ......................................................................................... 42  
Figure 13. Bus and Node Topology ..................................................................................................... 43  
Figure 14: Time Distribution .............................................................................................................. 44  
Figure 15. Phase 1: Publicly Available JWST Architecture: C&DH Starting Point ............................... 50  
Figure 16. Phase 2: Architecture Decomposition for SoST Design: ACS and Comms ....................... 51  
Figure 17. Additional Chassis and Flight Computer in SoST Scope .................................................... 52  
Figure 18. Publicly Available JIST Architecture .................................................................................. 52  
Figure 19. BAE RAD750 Flight Computer ........................................................................................... 54  
Figure 20. Publicly Available BAE RAD750 Architecture for Initial Modeling ................................. 54  
Figure 21. Software Objects Hierarchy of Components and Devices .................................................. 55  
Figure 22. Publicly Available RAD750 Architecture Showing Device Models ................................. 56  
Figure 23. Python Snippet Showing Devices Added to a Flight Computer Component ....................... 56  
Figure 24. Mapping Devices to Memory Layout .................................................................................. 57  
Figure 25. Star Tracker Model Part 1 .................................................................................................. 59  
Figure 26. Star Tracker Model Part 2 .................................................................................................. 60  
Figure 27. Star Tracker Model Part 3 .................................................................................................. 60  
Figure 28. NOS Engine Usage for Time Synchronization ..................................................................... 62  
Figure 29. GPS Model Adding a Callback for Received Time Tick ...................................................... 62  
Figure 30. GPS Model send_periodic_data() Function ..................................................................... 63  
Figure 31. DMA Engine Architecture and SoST Equivalent ............................................................... 63  
Figure 32. Representative DMA Register Definitions ......................................................................... 64  
Figure 33. Typical MIL-STD-1553 Architecture: Public Summit Example ........................................... 65  
Figure 34. 1553 Architecture with Unneeded Modeling Components Shaded ....................................... 66  
Figure 35. Summit SoST Modeling Architecture ................................................................................ 67  
Figure 36. RT communication through the NOS Engine MIL-STD-1553 Hub ....................................... 68  
Figure 37. NOS3 Deployment Example - Part I .................................................................................. 72  
Figure 38. NOS3 Deployment Example - Part II ................................................................................ 73  
Figure 39. Command Terminal .......................................................................................................... 74  
Figure 40. Run Installer Script ........................................................................................................... 74  
Figure 41. Disclaimer ......................................................................................................................... 74  
Figure 42. Level of Install .................................................................................................................... 75  
Figure 43. Complete Configuration ..................................................................................................... 75
Figure 44. Installation Complete
Figure 45. STF-1 CubeSat Components
Figure 46. NOS3 architecture illustrating its Flexible Flight & Sim Configuration
Figure 47. NOS3 component connections between ground (COSMOS), FSW (cFS), & Dynamics (42)
Figure 48. NOS³ Simulator Architecture
Figure 49. NOS3 Architecture Overview
Figure 50. FSW Compiler Diagram
Figure 51. JWST undergoing testing and integration. Image Courtesy Northrop Grumman
Figure 52. Visualization of the JWST Spacecraft During Verification Tests
Figure 53. IRU Rate B Component Comparison Between FlatSat and JIST
Figure 54. IRU Rate A Component Comparison Between FlatSat and JIST
Figure 55. Sun Sensor 1 Y Component FlatSat and JIST Comparison
Figure 56. Comparison of FlatSat and JIST Sun Pitch Angle
Figure 57. Hardware Checkout Tests for FPGAs
Figure 58. JIST Increases Test Resources
Figure 59. JIST Increases Access to Test Resources
Figure 60. JIST Utilization by Organization
Figure 61. JIST Deployment Time Compared with Hardware Deployment Time
Figure 62. Linux FSW with Real Instruments
Figure 63. Linux FSW with Real Instruments Variation
Figure 64. NanoMind FSW with Simulated Instruments
1 Introduction

1.1 Background

In 1993, the NASA IV&V Facility in Fairmont, West Virginia, was established due to the National Research Council (NRC) recommendations following the Presidential Commission's Report on the Space Shuttle Challenger Accident [1]. The IV&V Program specializes in performing flight software’ verification and validation (V&V) to increase the mission assurance of NASA’s highest-profile missions, such as the James Webb Space Telescope (JWST), Space-Launch-System (SLS), and the Nancy Grace Roman Space Telescope (RST). Software for high profile missions is often considered safety-critical, which is a classification given for systems that could cause 1) loss of life, 2) loss of mission, or 3) harm to the environment if a software error would occur.

The IV&V Facility is managed by NASA’s Goddard Space Flight Center (GSFC) but is independently funded by NASA’s Office of Safety and Mission Assurance (OSMA). In 2019, the IV&V Facility was dedicated and renamed to the Katherine Johnson IV&V Facility. During this renaming ceremony, this author was awarded the 2019 NASA IV&V Engineer of the Year Award, which was presented by the NASA Agency Administrator, Mr. Jim Bridenstine, in Fairmont, WV.

The IV&V Program employs a Follow-The-Risk (FTR) [2] approach that states that flight software’s riskiest aspects are given the most V&V attention and analysis. FTR is the process by which the IV&V Program understands, identifies, and prioritizes risk areas so that priorities can be assigned. IV&V services are treated as a scarce resource [3], and their activities need to be managed effectively such that the critical mission assurance activities can be applied to the riskiest aspects of missions. For each mission that the IV&V Program supports, it asks three questions [4] that ultimately drive its V&V approach of NASA’s critical flight software:

* Flight software refers to the software that operates on-orbit, e.g., onboard spacecraft or launch vehicles and is responsible for their mission operations and safety. Generally, flight software does NOT include ground software that assists with ground procedures and/or launch operations, although critical ground software sometimes falls within IV&V scope.
1. Is the system doing what it is supposed to do?

2. Is it NOT doing what it is NOT supposed to do?

3. Is the system going to behave appropriately under adverse conditions?

FTR and Software Risks Analysis are relatively new techniques for NASA and flight software V&V [5]. However, the IV&V Program has pioneered this effort by focusing on V&V activities that have been based upon software quality assurance static methods that follow the flight software development lifecycle. Static activities such as requirements analysis, requirements tracing, test plan review, test result review, and manual/static source code analysis were typically performed at the appropriate flight software development stages. These activities have one thing in common: they are considered static because they do NOT execute the flight software in an operational or simulated-operational environment.

In contrast, another method of V&V is to perform dynamic analysis. Dynamic Analysis is an industry-standard definition and approach, which simply means that the flight software is being executed in an operational environment. In recent years, there has been a push to augment the IV&V Program’s service portfolio with a dynamic testing capability to increase software mission assurance for safety-critical systems. Using flight (test) hardware to meet this capability is not practical for the IV&V Program due to the extremely high cost and low availability of space hardware.

Circa 2009, Software-only Simulation Test Beds (SoSTs) (so named by this author) were invented for the purpose of performing advanced V&V activities that focused on Dynamic Testing capabilities. For a SoST to be invented and be successful, several software technologies also needed to be invented and integrated in order to execute a mission’s unmodified flight software binary on X86 commodity computers. For example, spacecraft often utilize a LEON3/4 or PowerPC CPU architecture for their embedded CPU platform, and their executable software binaries cannot natively execute on X86 computing platforms.

Historically, Dynamic Analysis is challenging to perform due to the very specialized and limited-quantity hardware that is needed to execute the flight software. For example, flight
computers typically utilize a radiation-hardened ("Rad-Hard") PowerPC or LEON3/4 processor, which is not considered a commodity. For example, BAE’s standard radiation-hardened flight computer is the RAD750 [6], which has been utilized on many NASA and Department of Defense (DoD) missions for the past ten years due to its performance and heritage. Also, almost all of NASA’s current missions contain customized FPGAs that define registers, DMAs, and I/O busses specific for each mission. Without these flight computers and FPGAs physically available, the flight software will not execute and is therefore untestable by dynamic analysis. SoSTs solve this problem by modeling the hardware, and thus providing a realistic environment for testing the flight software.

This dissertation describes all aspects of a SoST, emphasizing its design, implementation, analysis, and results. This work encompasses nearly nine years of effort and has successfully led the way for functional simulation, dynamic analysis, and virtual-flatsats at NASA.

Throughout this dissertation, several spacecrafts are utilized as examples of SoST design, implementation, analysis, results, and benefits. Each spacecraft represents a single SoST with a shared common design and implementation. Flagship spacecrafts that are utilized include the Global Precipitation Measurement (GPM) spacecraft, the James Webb Space Telescope (JWST), and Space Launch System (SLS), among others. Also included in more detail is the Simulation-To-Flight 1 SmallSat/CubeSat. No export-controlled nor ITAR information is shared in this dissertation. Each spacecraft diagram in this dissertation is available publicly, and several diagrams/figures have already gone through the NASA release process and were originally this author’s creation.

1.2 SoST Definition and Functional Modeling of Embedded Systems

Traditionally, and still common today, embedded computer systems and their software are tested by constructing a hardware test bed that includes the physical computer, the physical system, and test drivers to exercise the software [7]. This method was manageable in the past, but as embedded systems have become more complex, V&V has become more complex and costly to implement, and commodity computing resources have increased in capabilities, newer methods to perform more cost-effective V&V have
emerged. These methods involve software-only abstraction and modeling of hardware components, which is key for SoSTs.

The modeling and simulation (M&S) systems engineering field is extremely diverse, and more detail is provided to inform the reader of the type of modeling and simulation utilized in SoSTs. The reader must keep in mind that SoSTs are NOT related to Model-Based Software Engineering (MBSwE) tools [8], processes, nor models and are generally not related to the MBSwE field. MBSwE has been written and discussed extensively in literature with respect to V&V and follows an abstraction paradigm, whereas the systems’ software functions are modeled at a higher level than SoSTs to verify and validate system design behavior. SoSTs are also not directly related to Unified Modeling Language (UML) tools or techniques, but SoSTs could be used to augment UML methods. Because the complete transformation of UML design to C source code remains an emerging field for safety-critical systems [9], SoSTs and their functional modeling techniques discussed here are more suited for direct V&V of embedded systems.

Also worth mentioning is SystemC [10], which is often associated with hardware modeling and is an ANSI C++ class library definition for system and hardware designers who have to implement models that are either hybrid between software/hardware or models that must be integrated into other models/simulators. SystemC can be utilized for model development in SoSTs, but it is merely an implementation method and does not represent the entire SoST. This author’s experience is that SystemC models are often defined at a much higher abstraction level than SoST models and thus are not usable for drop-in-integration into a SoST, because they are not capable of executing the native flight software. Typical SoST models are written in C, Python, or Device Modeling Language (DML) because of integration with other software, as discussed in Chapter 3. SystemC is a model implementation choice, and its models could be integrated into a SoST the same as other models. Also, one drawback of SystemC is that it lacks functional coverage [11]; functional coverage is obtained in an SoST by executing ground system tests against the flight software and measuring the code coverage per the SoST instruction set simulator.
In contrast with MBSwE and SystemC, SoSTs are considered a complete functional emulation of spacecraft hardware, thus enabling a software-only simulation system for flight software V&V where the flight binary executable can execute “believing” that is in space on a spacecraft.

As discussed in [12], SoSTs build upon the Electrical Functional Model (EFM) and include high-fidelity (to be discussed later) software-only models for the onboard computer and other aspects of the spacecraft. Traditionally, hardware model stand-ins are utilized in EFMs, but SoSTs are unique in that the entire simulator is implemented in software. Each item shown in Figure 1 is software in a SoST. Note the complications of transitioning from a hardware environment to an entirely software environment when considering time synchronization.

Figure 1. Traditional Electrical Function Model (EFM)
Functional emulation refers to the practice of only modeling the functionality that the flight software requires of its hardware and no more. For example, if the flight software initially only requires thirty-two MIL-STD-1553 [13] I/O registers to exist on a standard compact Peripheral Component Interconnect (cPCI) [14] card for reading/writing, then only those registers should be initially modeled. Other technical aspects such as the communications delay on the MIL-STD-1553 bus should not be initially modeled and only considered later once the SoST timing and synchronicity are established. More information on these guidelines for iterative development and model fidelity is found in Chapter 5.

SoSTs model the hardware in software to a fidelity such that the unmodified flight software binaries will execute unaware that their host environment is software-only. This dynamic simulation gives both IV&V analysts and mission systems a flight software test environment that is functionally equivalent to the spacecraft. A key advantage to SoSTs is that they can be shared an unlimited number of times across the Internet, enabling more users who are not bound by location nor limited hardware resources.

SoSTs have enabled the IV&V Program to perform more advanced V&V. IV&V analysis teams can gain an increased understanding of the software execution and behaviors,
exercise the system under adverse conditions, and inject hardware faults into the system to gain insight into how the software responds to adverse conditions.

1.3 Problem Statements and Motivations

The following problem statements and motivations are presented to more explicitly describe the problems that SoST systems-as-a-whole solve. These problems were considered normal operations and represented the status quo prior to SoSTs being invented. These problem statements also serve as a description of spacecraft V&V test methods before SoSTs and are further described in Chapter 2.

A lack of a SoST for a spacecraft mission will exacerbate these issues and raise the mission risk, specifically regarding its flight software assurance and testing. Each issue is described in more detail with an explanation of its impact on spacecraft mission assurance and mission success.

As an example, to help set the stage of this dissertation, a list of actual risks is presented in Table 1 from a recent high-profile mission. Note that these risks are discussed in the SoST motivations; SoSTs directly benefit missions’ assurance by helping to mitigate these risks:

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Risk of Insufficient Hardware Test Bed Shifts</td>
</tr>
<tr>
<td>2.0</td>
<td>Overutilization of Hardware Test Bed</td>
</tr>
<tr>
<td>3.0</td>
<td>Limited Hardware Availability</td>
</tr>
<tr>
<td>4.0</td>
<td>Lack of Hardware Interface Simulator/Emulator</td>
</tr>
</tbody>
</table>
1.3.1 Problem Statement #1: Limited Hardware Test Bed Resources Due to Custom Hardware Results in a Lack of V&V

Missions rely on engineering test units (ETU) that are hardware devices that either emulate the flight hardware or are prototype hardware units with similar functionality as to their flight version counterparts. This hardware is utilized for software testing but is often in limited quantities, and the Hardware Test Beds are often overscheduled and over-utilized.

Spacecraft do not utilize off-the-shelf components; for the large spacecraft missions, custom hardware solutions are engineered in extremely low quantities. This results in limited hardware test beds, limited test opportunities, and limited testing in general. SoST’s primary purpose was to develop software-only simulators that could help augment traditional spacecraft flight software testing.

FlatSats, synonymous with hardware test beds, is a definition given to the physical connection of spacecraft components, both flight, and non-flight units, in a testing area, not integrated and physically spread out on a flat surface area. “A FlatSat is an engineering unit of the spacecraft or SmallSat that includes all of the components, except the structure. Typically, the components are mounted on some sort of flat board, hence the term FlatSat. Developers can use the FlatSat to test and troubleshoot the CubeSat’s systems without integrating everything onto the structure.” [15]

FlatSats are types of hardware-in-the-loop test beds where the flight software, hardware integration, and input/output interfaces are tested before spacecraft assembly. FlatSats strive to be as close to the actual flight spacecraft hardware and software configuration as possible to ensure an accurate and reliable test bed. FlatSats are in limited quantity and suffer from overutilization, scheduling conflicts, and downtime.

1.3.2 Problem Statement #2: Limited Fault Injection Capabilities for Thorough Flight Software Testing

Hardware Test Beds are precious assets and cannot fulfill the role of fault injection testing because they 1) do not support fault injection testing capabilities, and 2) cannot be failed/broken due to them being a precious commodity. For example, on one mission,
engineers had to result to flight software source code *inspection* to verify requirements because the flight hardware under test could not be failed. There was no way to actually test the requirements to ensure that they were implemented correctly.

1.3.3 Problem Statement #3: Limited / Nonexistent Flight Software V&V Methods for SmallSats and CubeSats

The last ten years have witnessed the revolution and exponential growth of SmallSats / CubeSats in the commercial and government sectors. However, according to the 2019 NASA Ames Research Center study titled, Small-Satellite Mission Failure Rates [16] “…41.3% of all small satellites launched experienced total or partial mission failure. Of these, 6.1% were launch vehicle failures, 11% were partial mission failures, and 24.2% were total mission failures.” Also, as noted in this report, “Another reason [for the failures] could be that as the small satellite software complexity has increased, the methods used to perform verification and validation of the small satellite software has not increased commensurately.”

According to [17], SmallSat *Soft Faults*, also known as software bugs, transients, or electromagnetic interference, are caused by inherent software issues, cosmic ray intensity, or impending hardware failure. Soft Faults can significantly contribute to a SmallSat’s decreased on-orbit operations, partial mission failure, or complete mission loss. For example, the Goddard Space Flight Center (GSFC) Dellingr SmallSat [18] had a near-mission ending uncontrolled rapid spin-rate that occurred due to a software fault.

Further discussing the lack of SmallSat testing, [19] surveyed approximately two-hundred (200) SmallSat organizations and found that “adequate software testing is a concern everywhere.” Organizations (both industry and academia) focus on hardware-based flatsats for their systems and integration testing, but there was no software-only simulation usage as virtual-flatsats, because (presumably), organizations did not have the expertise nor funding to implement.
1.3.4 Problem Statement #4: COVID-19 Situation Limits Hardware Testing Causing Schedule Delays

During the current COVID-19 quarantine and situation, hardware test beds have been off-limits for NASA personnel per this author’s first-hand information and are just now (after six months) starting to operate at significantly reduced capacity. COVID-19 is delaying spacecraft hardware development, test and integration, and flight software development and test schedules.

1.3.5 Problem Statement #5: Limited Testing Venues of Embedded Non-X86 Flight Software

Spacecraft flight computers are expensive and in short supply due to their low manufactured quantities and specialized parts. In an ideal world, each flight software developer and tester would each have their own flight computer to perform development and testing. This is not realistic and is further exacerbated by additional custom hardware that needs to be included for the flight software to boot to an operational state successfully. This results in only one to two testing venues that are capable of executing the flight binary in a test-as-you-fly configuration.

1.3.6 Probably Statement #6: Lack of Virtualization Technologies

Virtualization technologies were slow to be adopted in the government, with computing systems being installed and operated as a single host computer, with no virtualization, no guest operating systems, and no cloning or migration capabilities. An ancillary benefit to SoSTs is the utilization of virtualization to generate ready-to-run test environments on the fly. Virtualization is also the first step to enabling industry-standard practices such as Continuous Integration (CI) and Continuous Testing (CT) [20].

1.3.7 Problem Statement #7: Flight Hardware Driver and Software Interfaces are Tightly Coupled

Traditionally, the hardware’s software driver and software interface layers are tightly coupled and not meant to be “separated.” This causes inflexible software that only executes on the very specific and custom flight hardware, thus exacerbating the limited hardware test bed issue. In the past, the software would only be intended to be executed on the specific hardware, not commodity X86 computing platforms.
1.4 Requirements of a Software-only Simulation Test Bed (SoST) to Address Problem Statements

1.4.1 Dissertation and Technical Approach

This dissertation’s technical approach describes the SoST invention, design, implementation, and roadmap. The entire dissertation and the SoST invention are designed to address the Problem Statements in various ways, usually in a holistic manner. When implemented, SoSTs are considered a complete system; it is challenging to show individual components’ benefits without considering the entire integrated SoST. In fact, integration of the various components is one unique feature of SoSTs, and often the most challenging part of the process, requiring other integrated technologies.

The SoST modeling processes can be applied to multiple missions, with the resulting product being a specific SoST containing models of a single spacecraft’s hardware.

1.4.2 Main Technical Objective

The main technical objective is to execute the flight binary executable on commodity X86 computing hardware, thus providing an inexpensive platform for performing flight software V&V and fault injection testing.

1.4.3 Level-0 Requirements and Goals

The Level-0 initial requirements for a SoST are listed below, along with a description and their rationale. Level-0 represents the high-level requirements that drive the design and implementation and are derived from the aforementioned Problem Statements.

These requirements are assumed to be SHALL statements and begin with the prefix “The SoST SHALL...”. The requirements' text has been shortened for readability.

<table>
<thead>
<tr>
<th>Shall Requirements</th>
<th>Description and Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 ...Include CPU Instruction Cycle Accuracy</td>
<td>Provides as close as possible results to hardware test beds</td>
</tr>
<tr>
<td>2.0 ...Support the Execution of the Flight Binary</td>
<td>Maintains the Test-As-You-Fly Mantra</td>
</tr>
<tr>
<td>SHALL Requirements</td>
<td>Description and Rationale</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><strong>3.0</strong>...Support a Ready-To-Run, Distributable Virtual Environment</td>
<td>Quickly created and shared environment to lower the activation energy of use, allowing more time for testing and development.</td>
</tr>
<tr>
<td><strong>4.0</strong>...Provide Synchronous System with the ability to Pause, Stop, or Resume Execution</td>
<td>The ability to pause, stop, or resume execution is very powerful for debugging and testing and is a capability that is not present in hardware test beds</td>
</tr>
<tr>
<td><strong>5.0</strong>...Include Fault Injection Capabilities</td>
<td>The ability to inject faults and fail emulated hardware is critical for flight software V&amp;V edge testing.</td>
</tr>
<tr>
<td><strong>6.0</strong>...Utilize the mission’s ground software system as the main Graphical User Interface (GUI)</td>
<td>The mission’s operational ground software system serves as the primary user interface because it supports 1) spacecraft commanding, 2) spacecraft telemetry and 3) scripting of spacecraft commands. Also, using the operational ground software system provides significant familiarity to the SoST users, as they already are experienced with the software.</td>
</tr>
<tr>
<td><strong>7.0</strong>...Be a Software-only Simulation with no hardware-in-the-loop</td>
<td>SoSTs abstract the hardware interface away and replace it with a software-only interface that is synchronous.</td>
</tr>
<tr>
<td><strong>8.0</strong>...Be scalable for both large spacecraft missions and for SmallSats / CubeSats</td>
<td>SoSTs originally were applied to large NASA missions, but the explosion in popularity of SmallSats / CubeSats and their low-reliability rates required additional V&amp;V and risk reduction methodologies to be applied in cost-constrained environments.</td>
</tr>
<tr>
<td><strong>9.0</strong>...support analysis and verification of models and benefits</td>
<td>SoSTs need to be verified themselves, with measurable results and metrics in order to provide unquestioning assurance and findings.</td>
</tr>
</tbody>
</table>
1.4.4 Dissertation Organization to Address Problem Statements

This dissertation follows the standard outline, whereas:

- Chapter 1, this chapter presents the topic, problem statements, and dissertation organization.
- Chapter 2 describes the Background, Literature Review, and History. This chapter provides context for the status quo before when SoSTs were invented.
- Chapter 3 describes the Design of a Software-only Simulation Test Bed (SoST) for Flagship Spacecraft Missions. This section covers topics such as flight computer modeling, synchronous compact PCI (cPCI) modeling and operations, functional modeling of registers, memory banks, and DMA, and synchronous modeling of input/output (I/O) interfaces. Specific spacecraft examples will be used as design references and examples.
- Chapter 4 describes the specific application of SoSTs to SmallSats/Cubesats, including design, implementation abstraction, and utilization on the STF-1 CubeSat.
- Chapter 5 presents an Engineering Roadmap that provides a unique process for scoping, designing, and implementing a SoST.
- Chapter 6 discusses the analysis, verification, and results of SoST systems and models. Realized benefits are also discussed.
- Chapter 7 presents the conclusion/contributions and describes future work areas for SoSTs.

Table 3 outlines the Problem Statements’ mappings to Requirements, and then where the Requirements are addressed in the dissertation. This mapping shows how each Problem Statement relates to the aspect of a SoST system. The dissertation’s body provides technical details as to the design, implementation, analysis, and results of SoSTs.
<table>
<thead>
<tr>
<th>Problem Statement</th>
<th>SoST Level-0 Requirement(s)</th>
<th>Addressed in Dissertation</th>
</tr>
</thead>
</table>
| 1.3.1 Limited Hardware Test Bed Resources Due to Custom Hardware Results in a Lack of V&V | 2.0 Support the Execution of the Flight Binary | 3.3.2 Unmodified Flight Software Binary  
3.3.4 Spacecraft Segment: Modeled Spacecraft Hardware – Command and Data Handling (C&DH)  
3.4.1 Flight Computer and Component Modeling |
| | 3.0. Ready-To-Run, Distributable Virtual Environment | 3.5 Deployment and Usage  
3.5.1 Ready-To-Run Virtualization Implementation  
4.3.5 Ready-To-Run Deployment and Operations |
| | 7.0. Software-only Simulation with no hardware-in-the-loop | 3.3.5 Spacecraft Segment Integration: NASA Operational Simulator (NOS) Engine Middleware  
3.4.2 Advanced SoST Modeling Techniques  
3.4.3 MIL-STD-1553: Synchronous Modeling of Input / Output (I/O) Interfaces |
| 1.3.2 Limited Fault Injection Capabilities for Thorough Flight Software Testing | 4.0 Provide Synchronous System with the ability to Pause, Stop, or Resume Execution | 3.3.5 Spacecraft Segment Integration: NASA Operational Simulator (NOS) Engine Middleware  
3.4.2 Advanced SoST Modeling Techniques  
3.4.3 MIL-STD-1553: Synchronous Modeling of Input / Output (I/O) Interfaces |
| | 5.0 Fault Injection Capabilities | 3.3.5 Spacecraft Segment Integration: NASA Operational Simulator (NOS) Engine Middleware |
| | 7.0. Software-only Simulation with no hardware-in-the-loop | 3.3.5 Spacecraft Segment Integration: NASA Operational Simulator (NOS) Engine Middleware  
3.4.2 Advanced SoST Modeling Techniques  
3.4.3 MIL-STD-1553: Synchronous Modeling of Input / Output (I/O) Interfaces |
| 1.3.3 Limited / Nonexistent Flight Software V&V Methods for | 8.0 Scalable for both large spacecraft missions and SmallSats / CubeSats | 4 SoST for SmallSats and CubeSats  
5 Engineering Roadmap for Scoping, Designing, and Implementing a SoST |
<table>
<thead>
<tr>
<th>Problem Statement</th>
<th>SoST Level-0 Requirement(s)</th>
<th>Addressed in Dissertation</th>
</tr>
</thead>
</table>
| SmallSats and CubeSats | 9.0. Support analysis and verification of models and benefits | 6 Methods of SoST Verification and Analysis  
6.1 Method 1: Utilization of Hardware Test Procedures as Verification Methods  
6.2 Method 2: Utilization of Hardware Checkout Tests as Verification Methods  
6.3 Method 3: SmallSat SoST Model Verification Examples for NOS3 |
| 1.3.4 COVID-19 Situation Limits Hardware Testing Causing Schedule Delays | 2.0 Execution of the Flight Software Binary | 7.2 Direct Benefits: Flight Software Issues Identified Due to SoST V&V |
| 1.3.5 Limited Testing Venues of Embedded Non-X86 Flight Software | 1.0 CPU Instruction Cycle Accuracy  
2.0 Execution of the Flight Software Binary | 3.3.2 Unmodified Flight Software Binary  
3.3.4 Spacecraft Segment: Modeled Spacecraft Hardware – Command and Data Handling (C&DH)  
3.4.1 Flight Computer and Component Modeling  
3.0. Ready-To-Run, Distributable Virtual Environment | 3.3 Deployment and Usage  
3.5.1 Ready-To-Run Virtualization Implementation  
4.3.5 Ready-To-Run Deployment and Operations  
6.0 Utilize the mission’s ground software system as the main Graphical User Interface (GUI) | 5 Engineering Roadmap for Scoping, Designing, and Implementing a SoST |
| 1.3.6 Lack of Virtualization Technologies | 3.0. Ready-To-Run, Distributable Virtual Environment | 3.5.1 Ready-To-Run Virtualization Implementation  
4 Ready-To-Run Deployment for SmallSats and CubeSats  
4.3.5 Ready-To-Run Deployment and Operations  
6.0 Utilize the mission’s ground software system as the main Graphical User Interface (GUI) |
<table>
<thead>
<tr>
<th>Problem Statement</th>
<th>SoST Level-0 Requirement(s)</th>
<th>Addressed in Dissertation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.7 Flight Hardware Driver and Software Interfaces are Tightly Coupled.</td>
<td>7.0. Software-only Simulation with no hardware-in-the-loop</td>
<td>5.1 How to Begin SoST Design and Implementation 5.2 Choosing Appropriate Levels of Fidelity 5.3 Scoping Modeling Efforts Based Upon Abstraction 5.4 Hardware Modeling Best Practices</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 Conclusion and Benefits Realized</td>
</tr>
</tbody>
</table>
2 Spacecraft Background, Literature Review, and SoST Advantages

This section describes several aspects of space systems and the status quo before the invention of SoSTs. This background information is needed for the reader to understand the SoST context within larger space systems. This chapter’s subsections describe the building blocks that SoSTs are built upon.

SoSTs remain a unique invention, with their advancements in software/systems integration, synchronous/asynchronous accurate timing, and virtualization.

2.1 Space System Segments

Spacecraft systems are divided into three segments, and a basic understanding is needed to understand the corresponding SoST use-cases. The three segments include 1) Space Segment, 2) Ground Segment, and 3) User Segment. These segments are described briefly below, along with how they relate to SoSTs. Figure 3 shows the segments; however, SoSTs are mostly focused on spacecraft flight software; thus, the software aspect of the segments is discussed in detail.

2.1.1 Space Segment and Flight Software Overview

The Space Segment represents the spacecraft and is the most critical segment of the SoST. SoST engineers must understand the complete spacecraft system, its hardware components, and its software. Furthermore, an intimate knowledge of spacecraft flight software is needed because the unmodified flight software binary needs to execute “inside” the SoST so that it is unaware that it is executing in a software-only environment.
NASA spacecraft, SmallSats, and launch vehicles utilize various flight software frameworks, with the choice dependent upon the NASA center that is leading the effort and past flight software heritage. Within the last several years, NASA Goddard Space Flight Center’s (GSFC) Core Flight System (cFS) [21] framework has increased in popularity and is now utilized at the Ames Research Center (ARC), Johnson Space Center (JSC), Goddard Space Flight Center (GSFC), and the Johns Hopkins University Applied Physics Laboratory (APL).

“The core Flight System (cFS) is a platform and project independent reusable software framework and set of reusable software applications. There are three key aspects to the cFS architecture: a dynamic run-time environment, layered software, and a component-based design. It is the combination of these key aspects that makes it suitable for reuse on any number of NASA flight projects and/or embedded software systems at a significant cost savings.” [22]

cFS has been released as NASA open-source software [23] and is available to anyone wanting to utilize a C framework for developing flight (or embedded) software. cFS is a modular publish/subscribe (“pub/sub”) architecture that supports the concept of applications (or “apps”), which are separately and dynamically loaded modules (or components) that contain specific mission functionality.

2.1.2 Core Flight Software (cFS) Flight Software Architecture

The cFS architecture is a publish/subscribe message bus implemented by the cFS Executive Services, shown in the green box in Figure 4. Hanging off the message bus box are applications (or “apps”) that are considered mission-specific or user-specific applications that address and implement specific mission functionality.

![Figure 4. cFS “Lollipop Chart” Showing the cFS Architecture and Apps](image)
requirements. Out-of-the-box, cFS provides several applications that are common across most NASA spacecraft missions. cFS is able to execute on x86 Linux computers in addition to flight target hardware such as the Xilinx Zynq, BAE’s RAD750, and Atmel’s AVR32 CPUs.

2.1.3 Ground Segment and Ground Software Considerations

Every spacecraft mission, including major spacecraft and smallsats, requires a Ground Software System (sometimes referred to as Mission Operations) that is utilized for sending commands to the spacecraft and receiving data (often called telemetry) from the spacecraft. These commanding and telemetry functionalities are requirements for each spacecraft. The Ground Segment is not the main focus of a SoST in this dissertation, but it should be noted that SoSTs integrate the spacecraft mission’s ground software system into a virtualized container. For most SoSTs, their main graphical user interface (GUI) is the ground software system. Thus, the Ground Segment is critical to SoST usage, but it does not represent the most challenging engineering problems with SoSTs. Specifically, Mission Operations can significantly benefit from a high-fidelity spacecraft simulator, such as a SoST. As discussed in [24], the SoST can be utilized to verify and validate flight procedures earlier in the spacecraft lifecycle before actual launch and on-orbit operations.

2.1.4 User Segment

The User Segment represents the end-user of the spacecraft, which is sometimes also the user of the SoST and responsible for the flight software verification and validation (V&V) results. The User is most often considered the spacecraft operator and responsible for sending commands to the spacecraft and receiving/interpreting health and engineering telemetry from the spacecraft. During normal spacecraft operations, the User interacts with the Ground Segment to control the spacecraft.

2.2 Literature Review and Prior Work

2.2.1 Embedded Systems, Flight Software Testing, and V&V Methods

Flight Software is a type of embedded system, and thus, V&V testing methods that apply to embedded systems can also apply to spacecraft flight software testing and V&V. Flight software testing involves loading the target flight software binary onto the flight computer
and then performing a series of unit, integration, and formal acceptance tests; these tests form the framework for the entire V&V process for an embedded system.

As noted in [25], verification is a significant challenge for embedded systems, especially those that are categorized as safety-critical. Verification costs are estimated to consume 50-70% or more of development time, and automated verification methods suffer from state-space explosion, thus limiting automation’s effectiveness. Software errors are still discovered late in the development lifecycle. SoSTs, however, approach safety-critical verification differently in that automation is utilized, but not attempted for state-space coverage. Instead, automation can be performed by having Risk-Based Scenario Tests execute automatically via the ground station software. This type of verification focuses on the highest risk software areas. Also discussed in [25] is the usage of abstraction, as it has been touted as being effective; however, in practical applicability, it is difficult to transition from a high-abstraction model, implemented via SystemC or Transaction Level Modeling (TLM), to the actual source code. SoSTs avoid a high degree of abstraction by focusing on functional and behavioral modeling paradigm, that enables actual execution of the flight software, maintaining the test-as-you-fly mantra.

As described in [26], NASA has traditionally performed a standard V&V process composed of hardware-in-the-loop “simulation” that relies on the actual flight hardware or flight hardware stand-ins to perform flight software testing. The use of the term “simulation” in this context is not similar to SoSTs. In the context as described in [26], “simulation” is referring to a complete hardware-in-the-loop test bed setup that simulates the actual aircraft or spacecraft, as evidenced by the quote:

“Without detailed modeling of the aircraft’s computers and systems, it is impossible to evaluate the interaction of their embedded systems. Because timing is an important and sometimes overriding consideration in complex systems, the use of existing flight computers in the simulation is essential.” [26]

This method of V&V is still common today in NASA due to its effectiveness, albeit costly. The goal of SoSTs is not to replace hardware-in-the-loop testing but rather to augment it and be utilized for different purposes throughout the flight software development lifecycle, thus helping to increase a mission’s assurance.
There are many examples of NASA utilizing hardware-in-the-loop test beds (often referred to as “FlatSats”) for performing various testing methodologies. A more recent (public) example is the testing of the Frontier radio utilized by the Applied Physics Laboratory (APL) Parker Solar Probe spacecraft and planned to be utilized for the Europa Clipper spacecraft. Testing of the Frontier radio, described in [27], illustrates how the radio flight unit was tested in a FlatSat configuration. Because no accurate CPU simulator nor instruction set simulators were available, the team was required to utilize actual FPGAs and a MIPS CPU to execute the radio’s flight software under test.

2.2.2 Static Software Source Code Analysis

One method of traditional V&V is to perform static source code analysis utilizing commercial off-the-shelf tools (e.g., Klockwork) designed to analyze the software source code and identify bugs, errors, memory leaks, and security weaknesses, all without executing the software. These tools will perform a light-weight compilation process and then “walk” each software branch, attempting to identify risky code areas. According to [28], the main advantages of static analysis are that it can find weaknesses in the code earlier in the software lifecycle, detect unreachable code, uncalled functions, and security issues that may escape a human’s review.

Limitations of static analysis include a considerable effort to eliminate false positives and try to review false negatives. These tools can produce thousands of results that an engineer must manually review, which remains error-prone. More importantly, software behavior during execution cannot be tested nor verified using only static analysis. This is a severe limitation of static analysis and should be considered when planning and scoping test activities. Static analysis, while necessary for its purpose, cannot answer nor prove the three V&V questions presented in Chapter 1:

1. Is the system doing what it is supposed to do?

2. Is the system NOT doing what it is NOT supposed to do?

3. Is the system going to behave appropriately under adverse conditions?

The NASA IV&V Facility has traditionally excelled in static source code analysis and is often utilized by NASA missions to provide an independent static analysis service. The
IV&V Facility receives a specific flight software version from the mission center (i.e., GSFC, MSFC, JPL), perform static analysis utilizing several COTS tools, manually review the results, then report the findings back to the project. When received, the project addresses the issues, deciding upon either fixing them or, in some cases, not fixing the issues. This process repeats for the entire IV&V lifecycle.

2.2.3 Dynamic Software Source Code Analysis

Dynamic software code analysis is a relatively new definition that is directly related to the SoS invention. In the past, dynamic software analysis for safety-critical systems, such as presented in [29], often meant that the system’s states were dynamically determined, and then analysis was performed to ensure that the software states were tested entirely and analyzed. As related to SoSTs, the term dynamic refers to the execution of the embedded flight software in typically a software-only environment. Please note that Dynamic Analysis, when discussed with SoSTs, is referring to the test-as-you-fly embedded flight software where the flight software is executed while being under-test. Dynamic Analysis should always execute the exact flight software binary (with no recompile) that will be executed on board the spacecraft.

Dynamic Analysis is usually not performed on hardware test beds, but instead is used as a method that complements static code analysis in simulated/emulated environments, such as SoSTs. Dynamic Analysis for SoSTs was mentioned at the Safe and Secure Systems and Software Symposium (S5) conference in 2014 [30] with respect to verifying software behavior. Also, the Charter of the NASA IV&V Independent Test Capability (ITC) Team is to “acquire, develop, and manage adaptable test environments that enable the dynamic analysis of software behaviors. Dynamic Analysis is performed on flight software to verify software behavior” [31].

Dynamic Analysis provides several advantages to typical testing methods, whether related to hardware test beds or static code analysis. The following table outlines known and author-experienced advantages, along with a specific example from this author's experience. Dynamic Analysis cannot replace other testing/V&V strategies, but the assurance of the safety critical system increases significantly when used in combination.
Table 4. Advantages and Examples of Dynamic Analysis

<table>
<thead>
<tr>
<th>Dynamic Analysis Advantages</th>
<th>Specific Author Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Identifies possible bugs/issues in a flight software execution environment. For one mission, specific issues were encountered with the Board Support Package (BSP), that contained the flight computer software drivers. The BSP is often one of the least-tested parts of the embedded flight software.</td>
</tr>
<tr>
<td>2.0</td>
<td>Allows for flight software analysis in which you do not have access to the actual source code. Once a SoST is mature, the source code is not needed and seldom used unless there is an issue requiring an engineer’s review. For many missions, the source code is never utilized in the SoST, only the embedded flight software binary. More information on how SoSTs are utilized is found in later chapters of this dissertation.</td>
</tr>
<tr>
<td>3.0</td>
<td>Permits the validation of static code analysis findings, including false negatives. On several occasions, a static code analysis finding has resulted in a more detailed analysis needing to be performed. Dynamic analysis was utilized and targeted in order to answer the specific question and verify the static code analysis finding. In one specific example, an Object-Oriented initialization routine was not evident due to C++ abstraction and inheritance, and actually executing the code in a dynamic method was needed to definitively verify the finding as a flight software error.</td>
</tr>
</tbody>
</table>

2.2.3.1 Usage of Instruction Set Simulators (ISS) in Dynamic Analysis

Instruction Set Simulators (ISS) represent a core SoST component. ISSs are software that is capable of translating one “guest” CPU’s instruction set to the host-computer’s native instruction set so that the guest’s software binary can execute on the host computer. As described in [32], “an ISS generalizes various processors’ instructions, by the means of translating various processors’ instructions into virtual instructions, which then provides a virtual CPU” that can execute on typical X86 commodity hardware.

For example, the SoST invention has grown the use-case in space applications of executing PowerPC instructions on a commodity X86 computer for software verification. Each PowerPC instruction is mapped to an X86 instruction (or group of instructions), so that as the program counter (PC) is advancing in the virtual PowerPC CPU, the PowerPC instructions are translated to X86 instructions in its own Execution Context. The Execution Context is similar to virtualization of the PowerPC computer system, in that the system components such as RAM, I/O busses, etc. are mimicked as the PowerPC software executes, it can successfully manipulate its “memory” and “I/O” that belong to its Execution Context.
Traditionally, ISSs have been standalone and utilized for educational purposes for teaching a CPU’s specific instruction set. As shown in [33], they provide an excellent learning environment for students to write assembly instructions in the native CPU instruction set and execute their code on commodity X86 computers without needing specific hardware for execution and learning. Although not common at the time, early efforts [34] have utilized ISSs for real-time software timing analysis, as the ISS can better estimate the virtual CPU’s cycle instruction time, thus providing a much more accurate calculation of CPU cycle delay in hard real-time systems, whereas previously only offline analysis was performed. This concept represents an essential building block of SoSTs, whereas SoSTs represent a much larger and complete virtual spacecraft environment.

More recently, with the evolution of mobile computing developments, ISSs have started to be utilized for developing mobile applications in standard software development environments (e.g., Microsoft Visual Studio, Microsoft Visual Studio Code), thus enabling software engineers to develop and test their software without having to deploy it to a mobile device. These technologies have matured open-source ISSs in recent years.

The creation of specific ISSs are outside the scope of this dissertation; however, their integration into SoSTs is a crucial step and is unique in the spacecraft market. This author has experience with several ISSs, both open-source and commercial/proprietary. Significant effort is needed to integrate an ISS into a complete spacecraft simulator such that it is capable of executing the flight software. This software integration is discussed in detail in this dissertation; PowerPC and LEON3/4 CPU architectures are common flight computers due to their performance specifications [35], utilized in spacecraft, and are the main ISSs discussed by this author.
There are several ISSs available on the commercial and open-source markets. This author has conducted market surveys to identify competing or complementary technologies available for hardware modeling and integration into SoST, with emphasis on PowerPC and LEON3/4 CPU architectures. The research also attempted to identify a standard modeling language or tool that could be used across various ISS choices. It is vital to choose the appropriate ISS for a SoST; otherwise, the software integration efforts will most likely fail. Characteristics of ISSs that make them appropriate for SoST integrations are presented in Chapter 5 Engineering Roadmap for Scoping, Designing, and Implementing a SoST.

Wind River Simics [36], [37] is one example of a Commercial Off the Shelf (COTS) ISS. Figure 5 describes a Wind River article about the Global Precipitation Management (GPM) SoST from their perspective but references this dissertation work. Although Simics provides a robust ecosystem for computer system model development, including the availability of common hardware models and a powerful debug environment, these tools lock the developer organization into a single vendor. Simics has advantages in that it provides some software modeling infrastructure support out-of-the-box. For example, Simics has the notion of a virtual cPCI bus that modeled cPCI cards are “plugged into.”

Another popular ISS is the widely used Quick Emulator (QEMU) [35], which is a generic and open source machine emulator and virtualizer. It includes emulation capabilities for various platforms and architectures. QEMU is a flexible, software-only emulation platform capable of emulating many hardware families by dynamically translating code written for the target platform into native code. It can emulate CPUs or complete hardware platforms.
based upon those CPUs. It can also be used as a virtualization platform for certain hardware families, running at near-native speed. QEMU has grown in popularity due to it being open-source and flexible. QEMU supports a large number of CPU targets and maintains a fast execution performance profile due to its dynamic recompilation techniques [38].

When used, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, and S390 guests. QEMU’s hardware emulation and host integration are complete enough to run, such as Linux live CDs for the SPARC and PowerPC platform, complete with networking and graphics.

When SoSTs were becoming mature (~2013 to ~2015), others were just starting to utilize ISSs for Soft Error Injection for testing embedded software. As described in [39], QEMU was utilized for error injection of a system that utilized the RTEMS RTOS [40], which is a popular RTOS with the European Space Agency (ESA) and is growing in usage in the United States. However, this work focused on error injection of CPU registers, which are natively supported by QEMU. SoSTs offer a more complete platform for fault error injection tests since it includes spacecraft and bus hardware models.

One disadvantage of utilizing QEMU as a SoST ISS is that although custom hardware models can be developed, QEMU was not developed with a modeling infrastructure in mind, so extensibility and custom modeling present a large initial effort, and a plugin style architecture does not exist, so re-compilation is required to add peripheral models. QEMU is written in C and is tightly coupled to its QEMU libraries, so there is no commonality with other emulators. However, QEMU does have SystemC integration utilities if a common hardware modeling language is desired [41]; however, this author’s experience is that QEMU and SystemC integration remain an academic exercise at this time.

Both Simics and QEMU have been utilized in SoSTs, so they are presented in Table 5 and compared. SoSTs have been designed to be mostly ISS agnostic, with integration implementation details for each ISS discussed in later chapters.
Table 5: Comparison of Wind River’s Simics and QEMU Instruction Set Simulators

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Simics</th>
<th>QEMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC CPU Support</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>LEON3/4 Support</td>
<td>YES (Recently)</td>
<td>YES</td>
</tr>
<tr>
<td>Designed for Device Modeling; Has Software Infrastructure</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Open-Source License</td>
<td>NO (Commercial)</td>
<td>YES (GPL V2)</td>
</tr>
<tr>
<td>Support for SystemC</td>
<td>YES (Limited)</td>
<td>YES (Limited)</td>
</tr>
<tr>
<td>Languages Supported</td>
<td>C, Device Modeling Language (DML), Python</td>
<td>C, (Others when using language bindings)</td>
</tr>
</tbody>
</table>
2.2.4 Combining Static and Dynamic Analysis Methods

Combining Software Static Analysis with Dynamic Analysis is a new paradigm that has been pioneered at the NASA IV&V Facility and by this author in recent years. Since SoSTs provide an additional assurance activity to augment static analysis, it is advantageous to combine the two efforts providing a significantly powerful tool for helping determine software assurance and FTR. Generally, this topic is outside the scope of this dissertation but is mentioned here to provide context into the importance of SoSTs for providing mission assurance. Combining static and dynamic analysis techniques, which could be a published future SoST topic, focuses on first utilizing static analysis results to pinpoint and identify “risky” parts of the flight software that may require additional testing. Once the “risky” parts of flight software are identified, dynamic test cases can then be constructed to further test the flight software by utilizing the SoST. The tests can be executed on the SoST to ensure that the flight software meets the three V&V Objective Questions as discussed earlier. This concept was explored in [42], but SoSTs offer a different and more holistic perspective in that after the static analysis issues are identified, then actual spacecraft V&V tests can be (created and) executed to verify complete spacecraft behavior, thus better assessing the impact of the issue to the entire spacecraft, its risk, chances of occurring, and conditions of occurrence.

Figure 6 illustrates the workflow for combining static and dynamic analysis. The static analysis is performed, reviewed by an engineer, and any software issues are identified. Those software issues are then converted into dynamic tests to be executed within the SoST. The output of the workflow is increased assurance that the flight software will perform as expected.
2.3 SoST Invention Development and Maturation

This author has anecdotal evidence from years of NASA experience and is in a unique position to discuss NASA SoST development, use, and maturation during the past eight years. Prior to SoST development efforts, very little of NASA systems engineering included simulator/emulator development in parallel with flight software development and testing. Instead, simulators were considered one-off stop-gap development efforts that were meant to solve particular problems or operate in niche areas. For example, suppose that sample/example data needed to be produced to simulate a science instrument so that flight software development could continue and maintain its schedule while the science instrument’s development continues in parallel. This science-instrument-simulator would then be developed for its specific purpose, with it never being meant to be integrated into a bigger simulation system, such as a SoST. In the few cases where NASA developed SoST-like systems, the SoSTs were not of the maturation nor the fidelity of the SoSTs discussed in this dissertation.

To help establish the timeline of the SoST invention, usage, and its results, a timeline has been produced to give the reader more information on the SoST evolution over eight years. During this time, this author has been completely involved with the concepts, design, implementation, testing, and usage of SoSTs. Certain aspects of each of these missions appears in this dissertation to “paint-the-picture” of a SoST system, its design, implementation, and results.
Global Precipitation Measurement (GPM) Spacecraft: 2011

The Global Precipitation Measurement (GPM) Core Observatory spacecraft launched on February 27, 2014. It is the anchor to the GPM international satellite mission to unify and advance precipitation measurements from a constellation of research and operational sensors to provide “next-generation” precipitation products. [43]

SoST Efforts Include:

- Initial Concept
- First spacecraft mission that was modeled as a SoST
- One Compact PCI chassis was modeled, with one flight computer model
- SoST named GPM Operational Simulator (GO-SIM) was utilized for fault injection and dynamic analysis
- GO-SIM was the 2016 Honorable Mention for NASA’s Software of the Year Award

James Webb Space Telescope (JWST): 2015

The James Webb Space Telescope (JWST) is the infrared successor to the Hubble Space Telescope. It is a cryogenic infrared space observatory with a 25 m² aperture (6 m class) telescope yielding diffraction-limited angular resolution at a wavelength of 2 um. [44]

SoST Efforts Include:

- Most complex spacecraft modeled at the time
- Two redundant Compact PCI chassis were modeled, with four flight computer models
- SoST named JWST Integrated Simulation and Test (JIST) was utilized for static analysis verification and various dynamic testing that focused on fault injection
- JIST was the 2016 Honorable Mention for NASA’s Software of the Year Award
Space Launch System (SLS): 2018

The flexible system, which can be configured for Orion, cargo, or Orion with co-manifested payload missions, offers high escape velocities to send more mass to deep space destinations. [45]

SoST Efforts Include:

- Modeled triple-redundancy flight computers
- Modeled custom-Boeing hardware and Compact PCI chassis
- First time modeling a launch vehicle instead of a spacecraft

Simulation-to-Flight 1 (STF-1) 2017

The Simulation-to-Flight 1 (STF-1) CubeSat mission demonstrates how legacy simulation technologies may be adapted for flexible and effective use on missions using the CubeSat platform. These technologies, named NASA Operational Simulator (NOS), have demonstrated significant value on several missions such as James Webb Space Telescope, Global Precipitation Measurement, Juno, and Deep Space Climate Observatory in the areas of software development, mission operations/training, verification and validation (V&V), test procedure development and software systems check-out. [46]

SoST Efforts Include:

- Goal was to apply SoSTs to cubesats/smallsats
- SoST invented for SmallSats and named the NASA Operational Simulator for SmallSats (NOS3)
- NOS3 won the 2019 Runner Up for NASA’s Software of the Year Award
Psyche Spacecraft: 2019

The Psyche mission concept is enabled by electric propulsion and would use SPT-140 Hall thrusters to rendezvous and orbit (16) Psyche, the largest metal asteroid in the solar system. [47]

SoST Efforts Include:

- Begin modeling of JPL's avionics and Compact PCI chassis
- First JPL mission
- Currently a Work in Progress
3 Design, Implementation, and Deployment of a Software-only Simulation Test Bed (SoST) for Flagship Spacecraft Missions

3.1 Introduction

This chapter discusses the design process, implementation considerations/methods, and deployment of SoSTs for flagship spacecraft missions. The usage of the term “flagship” refers to the large and ambitious Class-A/B missions that are high in public visibility, usually hundreds of millions of dollars (plus) budgets, and multiple organizations involved; examples include the James Webb Space Telescope (JWST), the Space Launch System (SLS), and the Europa Clipper spacecraft among others. SoSTs are considered high fidelity for flagship missions, meaning that the exact flight software binary is executed in a test-as-you-flight configuration. Flagship missions do NOT include SmallSats nor Cubesats, which are the subject of Chapter 4.

In this Chapter, the JWST spacecraft is utilized as an example. Existing public information, some of which was created by this author, is leveraged. The specific JWST SoST is named the James Webb Space Telescope (JWST) Integrated Simulation and Test (JIST) environment. JIST won Runner-Up in the 2012 NASA Software of the Year Award [48].

3.2 SoST Design Process

The SoST Design Process is a custom process that has been created for SoST design and implementation. This process is based upon an iterative software development process but is significantly modified to incorporate additional SoST design activities such as Test System Scoping, Planning and Analysis, and V&V Planning. This process is followed for each unique mission SoST and is described in more detail below.
• **Test System Scoping Activity**

As shown in Figure 7 above, the process [30] begins with the *Test System Scoping Activity* that receives inputs such as the SoST user requirements and special user requests. The Scoping Activity for SoSTs is unique in that it also includes mission artifacts as inputs. For example, the mission’s test system assets, flight software source code, V&V tests, specifications, and any existing simulators/emulators are considered, scoped, and researched, as these artifacts could be extremely valuable during SoST design and implementation. The outputs (shown in red) of this activity include the Test System Project Plan and the V&V Plan. The Test System Project Plan is the first SoST design and implementation plan, and its purpose is to drive the schedule and SoST scoping. The V&V Plan is a detailed plan of the V&V performed on the SoST to ensure that it is accurate once that it is completed. Ultimately, this activity is designed to be a gate review that
requires buy-in from all stakeholders before the SoST development efforts continue for
the specific mission.

- **Planning and Analysis Activity**
  The *Planning and Analysis Activity* is the first step of the iterative design/implementation
  process. This activity involves a mission architecture review, deep dive of mission
  artifacts, determination of the flight software architecture and real-time operating system
  (RTOS), and identification of crucial hardware spacecraft components and integration
  points. This activity begins the process of identifying the correct CPU instruction set
  simulator and SoST architecture. The outputs from this activity include a SoST Road
  Map document that describes the SoST release cycle, major functionality for each
  release, and a baseline set of high-level SHALL requirements that describe the specific
  SoST.

- **Build, Test, and Verify**
  The *Build, Test and Verify Activity* is the main activity where SoST development, testing,
  and initial verification is performed. For each iteration, a core set of features is developed
  and tested along with the flight software functionality; thus, a parallel track to the flight
  software development is established. This is the most labor-intensive activity and can be
  considered three individual sub-phases: 1) Build, 2) Test and 3) Verify. Build activities
  are discussed more in this chapter, whereas Test and Verify activities are discussed in
  Chapter 6. The output from this activity is a SoST System Design Document that outlines
  the architecture, functionality, and initial testing methods.

- **Validate**
  The *Validate Activity* tests the SoST against known test cases to ensure that the outputs
  match hardware test beds’ outputs. The goal of this validation is to show that the SoST
  is mature with accurate models, and that it is ready to begin flight software V&V activities.
  More information on SoST testing and validation is discussed in Chapter 6. The output
  from this activity is executed Validation Tests with disposition, which are tracked in an
  issue tracker such as Gitlab or Atlassian’s JIRA tools.
• **Release and Operations**

The *Release* Activity is for performing a SoST release and entering an operational phase. A SoST release follows the same industry-standard software engineering practices, which includes the creation of virtual machines, closing and disposition of open issues, and deployment. The output of this activity is the SoST itself (in the form of encapsulated virtual machines), including a User’s Manual and Version Description Document (VDD), which are standard NASA 7150.2C [49] standard deliverable documents. The VDD is utilized to document the versions of all custom and integrated software in the SoST as a baseline for future iterations, it lists new functionality in the release, any caveats or known issues, and resolved bugs. Once the *Release Activity* completes, the process returns to the *Planning and Analysis* Activity and repeats with incremental development until a future release is performed.

### 3.3 Technical Design and Modeled Components

This section titled *Technical Design and Modeled Components* introduces the SoST components and their technical design. Integration and implementation of the components are discussed in the next section. A SoST design is presented generically, and then specific mission examples are utilized to give the reader more detailed design information and context.

A SoST technical design is composed of several pieces, which are depicted in the following Figure 8. This figure is referenced several times during this dissertation because it shows the individual components and shows the relationship of the components in the entire integrated SoST. Each component, its creation, and design are presented in the following subsections; also, and most importantly, the technical methods of performing the component integration is discussed. In the figure, items in blue represent external entities that are either integrated or “dropped-into” the SoST; blue entities such as “Unmodified Flight Software Binary” (Spacecraft Segment) and “The Operational Ground System” (Ground Segment) are discussed first. The purple items represent inventions and creations for the SoST and are discussed in excruciating detail in this section and the next.
3.3.1 Ground Segment: Operational Ground System Software

The Operational Ground System represents the software that is utilized to command the spacecraft and receive data (henceforth referred to as “telemetry”) from the spacecraft. Obviously, the Ground System represents the “Ground Segment” (see Chapter 2), and the software is also capable of executing scripts (series of multiple commands) using basic logic constructs. A standard ground system scripting language is Systems Test and Operations Language (STOL)[50]. With respect to SoSTs, the operational ground system is the primary user interface and provides a graphical user interface (GUI) front-end to the spacecraft operations, as shown in Figure 9. This workflow matches actual spacecraft operations that take place in the Mission Operations Center (MOC) during spacecraft operations.
In SoSTs, the Ground System software integration provides the full spacecraft “day-in-the-life” capabilities. This allows for testing and verification of on-orbit commanding and telemetry response. With the inclusion of a ground system, the SoST is also utilized for ground operator training. For example, SoSTs have been utilized for testing stored command sequences and ground commands before they are uplinked to an on-orbit spacecraft. The SoST can also provide value later in the lifecycle than normal spacecraft development and I&T phases by serving as a rapid-prototyping and virtualized test environment.

Note that the Ground Station software is highlighted blue in Figure 8, indicating an Integrated Software Component. Integrated Software Components are considered third-party software components that are treated as black boxes during SoST integration, often integrated into the SoST by utilizing socket communications. For this integration to occur, research on the ground software must be performed to determine socket ports, server/client configurations, and the protocols being utilized. Figure 10 [51] shows how the ground software, ECLIPSE [52], is an integrated part of the JIST SoST; ECLIPSE
executes in its own Windows Server 2003 virtual machine, interacts with the go-between Telemetry Command Test Set (TCTS) virtual machine which emulates uplink and downlink communications, and the spacecraft virtual machine which contains several emulators for the spacecraft.

Spacecraft operators interact with the SoST via the Ground Station software. Operators are responsible for safely commanding the spacecraft, while testers/analysts are responsible for testing the spacecraft by developing “What-If” test scenarios that are designed to test and perform V&V on the spacecraft flight software.

3.3.2 Spacecraft Segment: Unmodified Flight Software Binary

Also highlighted in blue in Figure 8 is the Unmodified Flight Software Binary. This is the binary executable that represents the software-under-test (SUT) in the SoST. The binary executable is the actual flight software binary compiled for the flight architecture (e.g., PowerPC) and is meant to be a “drop-in” input to a SoST provided by the User. As the flight software binary executes in the SoST, it “believes” that it is executing on actual spacecraft hardware. No changes to the flight software binary are made, hence the use of the term “unmodified” when describing it. Often, the unmodified flight software binary is composed of 1) a real-time operating system (RTOS), 2) board-support-package (BSP), 3) software/hardware drivers, and 4) the actual flight software that is responsible for operating the spacecraft. As discussed in Chapter 2, in NASA, the actual flight software is often based upon the CFS architecture.

3.3.3 Spacecraft Segment: Spacecraft Dynamics and Systems

The Spacecraft Dynamics component is shown as a separate entity in the Figure 8 diagram because it is often modeled and integrated separately from the main spacecraft chassis. The dynamics include gyroscopes, reaction wheels, solar panels, data
recorders, etc. Spacecraft Dynamics represent the sensors and actuators responsible for the control loops that make the flight software believe that it is operating on-orbit in an actual space environment. Figure 11 shows the basic control loop for a spacecraft attitude [53]; this example is provided to give the reader an understanding of the types of hardware that is modeled as part of the Spacecraft Dynamics component.

Spacecraft control is outside the scope of this dissertation and SoSTs, but its integration is a vital component of a SoST because it represents a fully modeled spacecraft; Spacecraft Dynamics must be included in the SoST, or the flight software is not able to close its control loops because key inputs and outputs are missing. Without the Dynamics System adequately modeled, the flight software still executes in the SoST, but it will not be executing in an operational on-orbit “test-as-you-fly” configuration.

The internal timing synchronization of Spacecraft Dynamic hardware models and their implementation/integration are discussed more in the next chapter; however, the design of the hardware models will be discussed here. The process for modeling hardware such as gyroscopes and reaction wheels follow this process:

1) Determine spacecraft architecture and hardware that needs modeling; create a list of the hardware that composes the dynamics system.
2) Determine the software to use for the environmental inputs; a software solution is needed to model the space environment, such as magnetic fields, solar wind, possible GPS for low-earth orbit (LEO), star positions, etc.
3) Gather hardware documentation, specifically the User’s Manual and any available specification documentation for the sensors and actuators. This documentation is invaluable for modeling the specific hardware components.
3.3.4 Spacecraft Segment: Modeled Spacecraft Hardware – Command and Data Handling (C&DH)

Just as in the actual spacecraft, the C&DH system is the core component of the SoST and often is the starting place for the SoST modeling efforts. The C&DH contains the flight computer(s), compact PCI (cPCI) chassis, flight software, and utilizes registers, memory, and interrupts to communicate with other cPCI cards. The cPCI backplane has I/O modules that may include MIL-STD-1553, SpaceWire, analog to digital, data recorders, and numerous others. This “system” must be fully recreated in software inorder for the flight software to execute nominally inside the SoST.

The process of modeling flight hardware begins with an instruction set simulator choice for the target processor on the flight single-board computer (SBC) hardware. This simulator executes the assembly instructions found in the text segment of the flight software binary. Example tools used to perform this task are Wind River’s Simics and the open-source QEMU project. Memory visible to the CPU is laid out as it would appear in the hardware. Segments such as RAM, EEPROM, NVRAM, and PCI mapped memory are created in the virtual environment.

As a space flight system may contain multiple CPUs and other computing components with independent clock sources, the timing, scheduling, and synchronization of each component must be carefully considered in the SoST system integration.

The flight software interfaces with the hardware via registers, mapped memory, and interrupts; these components must be modeled precisely as described in their hardware specifications and interface control documents. A given hardware component, typically an FPGA or ASIC, contains numerous (sometimes hundreds of) registers exposed to the flight software’s device drivers. These registers control I/O devices such as MIL-STD-1553, SpaceWire, and RS-232. The registers are also responsible for controlling the state and reporting status data on the target SBC and other attached components. The hardware model requires recreation of these register behaviors in software, provide interrupts based on internal and external events, and share data between the FSW and the hardware via virtual mapped memory. Often, accurate hardware specifications are not available for various reasons. In these cases, reverse engineering from the flight
software is necessary to recreate the hardware behaviors but should be considered only as a last resort.

3.3.5 **Spacecraft Segment Integration: NASA Operational Simulator (NOS) Engine Middleware**

SoST components must be integrated for data flow to occur. Similar to the actual spacecraft hardware where data busses (e.g., I2C, SPI, MIL-STD-1553, SpaceWire) connect devices to the flight SBC, a SoST also needs a data bus for information to flow through the virtual spacecraft models to the flight software running under the instruction set simulator. In software-only systems, it is common for a software middleware developed on top of TCP/IP sockets to provide the hardware models' data transport mechanism. While this is true for simple software systems, there are many additional software mechanisms that need to be developed for SoSTs to function correctly, with the most important being synchronicity and timing of message data passing. Timing considerations are discussed more in the next chapter, but an introduction into the SoST middleware design and timing is provided below.

The NASA Operational Simulator (NOS) Engine (named and designed by this author) middleware was developed for the sole purpose of SoST integration of various components, emphasizing time synchronization, as depicted in Figure 12. It can be considered the “glue” of the system and is responsible for passing data between the various components. NOS Engine provides a modular communications architecture and a consistent and correct data passing mechanism for simulation components and provides the user with a software application programmer’s interface (API) for different types of synchronous communications. NOS Engine consists of a base layer that implements the NOS Engine core, and all layers (e.g., MIL-STD-1553, SpaceWire) are built upon the base layer. NOS
Engine allows multi-part systems the opportunity to send messages back and forth despite incompatible software. This type of integration, paired with advanced features like time synchronization, data manipulation, and fault injection, allows NOS Engine to provide a fast, flexible, and reusable system for connecting and testing various pieces of a SoST.

NOS Engine is built upon a conceptual model based on two fundamental types of objects: nodes and buses. A node is any type of endpoint in the system capable of sending and receiving messages. Any node in the system has to belong to a group, formally referred to as a bus. A bus can have an arbitrary number of nodes, and each node within the bus must have a unique name to all other member nodes. The nodes of a bus operate in a sandbox; a node can communicate with another node on the same bus but cannot talk to nodes that are members of a different bus. Figure 13 depicts the bus and node relationship.

![Bus and Node Topology](image)

NOS Engine is designed to operate in a client/server architecture for message distribution instead of a full “mesh” distribution system. The server is classified as a centralized entity through which all communication must pass. A client is defined as any entity that wishes to interact with the system that is not the server. As a result, no client can directly communicate with another client without first passing through the server. The server acts as the arbiter of message delivery and processing in the system.

A client/server architecture was chosen over a mesh architecture for several reasons.

1. The development and design are more straightforward with a client/server architecture. Mesh systems are of non-trivial complexity with regard to both design and implementation.
2. Due to the requirement that a single bus may contain multiple transports, each client would need to maintain an active connection to every other endpoint that it may communicate with. As the number of available transports increases, the message passing logic increases in complexity exponentially.

3. The system’s ability to provide interception and fault injection capabilities becomes significantly easier if a centralized arbiter exists. Clients do not need to be aware of an ever-changing list of interceptors associated with it. Also, the interceptor delivery order does not need to be negotiated by each client.

4. A centralized server provides administrators an easy way to view a snapshot of the system’s state due to all information being stored in one place.

NOS Engine provides a critical time distribution feature. The real power of time distribution is not just broadcasting a time value, but rather the ability to synchronize SoST activity.

Time distribution is handled by two different node types: TimeSender and TimeClient. On any given bus, there can be a single TimeSender and unlimited TimeClients. The TimeSender has the sole responsibility of sending a time value. Each time a new time value is sent, it is distributed to each TimeClient on the same bus. Synchronization is possible since the TimeSender waits until each TimeClient has received the tick and performed any activity. This allows activity to be synchronized between multiple clients when each may take a different amount of “wall-time” to complete. Figure 14 illustrates the behavior between the TimeSender and TimeClients when a time value is sent.

Figure 14: Time Distribution
Note that \textit{TimeSender} and \textit{TimeClient} nodes are not registered and used directly in the same way that a \textit{DataNode} is. Instead, sending and receiving time is designed and implemented directly into the \textit{Client::Bus} class.

\subsection{Setting SoST Bus Time}

\textit{TimeSender} nodes are not directly registered or used; instead, time sending is directly implemented into \textit{Client::Bus}. Time sending is enabled by calling \textit{Bus::enable_set_time}; and of course, there can only be one \textit{Bus} instance enabled for setting time.

Once enabled, the \textit{Bus} object is used to distribute time using the \textit{Bus::set_time(...) call. The first parameter is SimTime (currently defined as a 64-bit signed integer). Note that both \textit{enable_set_time} and \textit{set_time} have a timeout parameter. If a timeout is specified and a response isn’t received from the server within the desired time, the request fails, and an error is thrown.

The following example shows how to enable time and send a time message.

```cpp
#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Utility/Error/Error.hpp>

int main(int, char**) {
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");
    NosEngine::Client::Bus bus_1("copy://example:0", "Bus_1");
    // ERROR: set time is not enabled
    try {
        bus_1.set_time(0);
    } catch (NosEngine::Utility::Error::Error &e) {
        std::cout << e << std::endl;
    }
    // Success
    bus_1.enable_set_time();
    // Send time 0 to all clients, blocks until all clients are done
    bus_1.set_time(0);
}
```
3.3.5.2 Receiving Bus Time

*TimeClient* nodes are also not directly registered or used; receiving of time is also implemented directly into *Client::Bus*. To receive time, a ‘tick’ callback must be added to the *Bus*. This callback is executed each time a time-value is received. Any work that should be performed during the ‘frame’ should be completed during the callback (either directly within or using condition variables to control other parts of the application). Once the callback returns, the *TimeSender* has permission to send the next tick.

The following example shows how to receive the ‘tick’ within a callback function.

```cpp
#include <iostream>
#include <Server/Server.hpp>
#include <Client/Bus.hpp>
using namespace NosEngine;
int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");
    Client::Bus bus_1("copy://example:0", "Bus_1");
    bus_1.enable_set_time();
    bus_1.set_time(0);
    bus_1.add_time_tick_callback([](Common::SimTime time) {
        // perform any work as necessary
        std::cout << time << std::endl;
        //...
        // return when work is complete
    });
    // send time tick for every 10 units of time
    for (Common::SimTime time = 10; time < 100; time += 10)
    {
        bus_1.set_time(time);
    }
}
```

3.3.5.3 Retrieving Last Bus Time Received

The time for the last tick received can also be retrieved directly from a *Client::Bus* or *Client::Node* instance via the *get_time* method. This method can be called from inside or outside of a ‘tick’ callback, in fact this method will work even if no ‘tick’ callbacks have been added to the *Bus*.
```cpp
#include <iostream>
#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
using namespace NosEngine;

int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");
    Client::Bus bus_1("copy://example:0", "Bus_1");
    bus_1.enable_set_time();
    NosEngine::Client::DataNode *node_1 = bus_1.get_or_create_data_node("Node_1");
    bus_1.set_time(0);
    std::cout << "Time: "
              << "bus=" << bus_1.get_time() << " "
              << "node=" << node_1->get_time()
              << std::endl;
    bus_1.set_time(100);
    std::cout << "Time: "
              << "bus=" << bus_1.get_time() << " "
              << "node=" << node_1->get_time()
              << std::endl;
}
```

### 3.3.5.4 Timers

Sometimes in a SoST it is not desirable to have your ‘tick’ callback executed for every time an update is received; this is where timers can be utilized. The `Client::Bus` class has the following two methods, which allow for adding a timer which is executed once (and only once).

```cpp
void set_timer_absolute(Common::SimTime time, BusTimerCallback callback);
void set_timer_relative(Common::SimTime time, BusTimerCallback callback);
```

The `BusTimerCallback` has the following signature.

```cpp
void timer_callback(Common::SimTime time, Common::SimTime requested_time)
```
Note that for both absolute and relative timers, if the time provided is not a multiple of the time sender’s period; then the callback is executed for the tick which occurs immediately after the provided time. If the period is 1000 ms (simulated time), a wait for 150 ms (delta) or 1150 ms (absolute) will return about 850 ms (simulated time) later than expected. It is recommended to wait for multiples of the period.

An example of how to create timers is shown below:

```cpp
#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Client/BusGroup.hpp>
using namespace NosEngine;
typedef Common::SimTime SimTime;
int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");
    Client::Bus bus_1("copy://example:0", "Bus_1");
    bus_1.enable_set_time();
    bus_1.set_time(0);
    // add a timer to be executed 10 units of time from current time
    bus_1.set_timer_relative(10, [&bus_1](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;
        // add a timer to be executed 5 units of time from current time
        bus_1.set_timer_relative(5, [&bus_1](SimTime time, SimTime requested_time) {
            std::cout << time << " " << requested_time << std::endl;
        });
    });
    // add a timer to be executed at time 50
    bus_1.set_timer_absolute(50, [](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;
    });
    // add a timer to be executed at time 52
    bus_1.set_timer_absolute(52, [](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;
    });
}
```
3.3.6 James Webb Space Telescope Integrated Simulation and Test (JIST) SoST Design Walk-Through

This section is meant to build upon the previous design sections in this chapter and provide a walk-through of the design process and design decisions that are needed before SoST implementation. This design process is specific to SoSTs and represents a considerable advancement in the field of spacecraft simulation. This process fits into the design process described in Section 3.2.

Figure 15 [54] is a publicly available JWST spacecraft architecture. When scoping and planning a SoST, one must first thoroughly understand the spacecraft architecture, its hardware components, flight software architecture, subsystems, and interfaces. The SoST design approach is an iterative approach that focuses on refinement to reduce the development risk.

First, focus on the Command and Data Handling (C&DH) subsystem (highlighted in blue), as this contains the flight computer/CPU and interfaces to the other systems. Instrument systems should be deferred until later in the development process as they often represent secondary goals.
Next, after the C&DH, focus on the communications and Attitude Control System (ACS) subsystems, highlighted in yellow below. These subsystems contain custom hardware that needs modeled in the SoST because they interface directly with the flight software. Notice how the SoST architecture begins with the C&DH and then branches outward in an iterative process.

Finally, as shown in Figure 16, the remaining interfaces that “touch” the C&DH system need to be modeled. In this case, the science compact PCI chassis (highlighted in orange) needs to be modeled because it contains bus I/O and its own flight computer.
At this point, the critical components of the SoST architecture have been identified (Figure 17): C&DH, ACS, Communications, Power, and Science Instruments have been identified as the hardware components that need scoped and understood in order for modeling implementation to begin. The remainder of the spacecraft hardware includes instrument hardware, detailed power system components, and ancillary communications hardware. The method to determine the hardware models to develop/include in the SoST and what hardware models to defer can be answered by considering the following question:

“What hardware does the flight software know about, and does the flight software interact directly with it?”

Only modeling the hardware that the flight software directly interfaces with is an essential tip for designing a SoST. It is too easy to want to model each hardware component or spacecraft subsystem, but this is not feasible when considering time, budget, and labor.
For a specific example, the James Webb Space Telescope Integrated Simulation and Test (JIST) SoST architecture is shown in Figure 18 in the publicly available architecture diagram [30], for which this author was a major contributor.
Note that items in blue are custom hardware models developed specifically for this SoST, items in green are integrated software components, and the virtual MIL-STD-1553 and SpaceWire busses tie the software components together. A SoST design architecture will very closely match the actual spacecraft hardware architecture.

Please keep in mind that there is no hardware in a SoST; these components are software-only, and implementation requires hardware abstraction and integration touchpoints that are discussed in more detail later in this chapter.

The red and orange busses represent NOS Engine busses, and hardware components are Clients on the NOS Engine bus. The details of each modeled component are considered sensitive and are not provided in this text. However, their integration design and implementation are presented.

3.4 Implementation and Integration

This section presents the implementation and integration of SoST components that have been previously described in the design section. SoST integration is the most challenging part of the iterative process, and thus, it requires an incredibly mature design. This section also describes component implementations that utilize different technologies. The technologies are presented along with their specific components in order to fully describe the SoST integration. Detailed examples that use both Wind River’s Simics ISS and the open-source QEMU ISS are presented.

3.4.1 Flight Computer and Component Modeling

The Flight Computer model is the most critical component and is crucial to the SoST. The accuracy of the Flight Computer model must be nearly perfect in order for the flight software to execute correctly. In this stage, accuracy refers to how closely the model matches the hardware so that the flight software cannot distinguish between the two. For the initial implementation, an ISS must be chosen for the specific CPU computing architecture. Currently, the PowerPC, LEON4, and ARM CPU architectures dominate the space domain, and a compatible ISS needs to be chosen. Throughout this section, the popular BAE RAD750 (PowerPC 750) flight computer, shown in Figure 19, is used as an example reference SoST implementation. **Only referenced public RAD750**
information is utilized. In some cases, the publicly available PowerPC e600 CPU board is used as a RAD750 stand-in example when discussing SoST implementation. The PowerPC e600 is a similar RISC big-endian, memory-mapped I/O, System on a Chip (SoC) CPU.

Note, the ISS design and implementation are not discussed in detail in this dissertation; the integration of the ISS is the focus. For the initial discussion, the Wind River Simics instruction set simulator is utilized.

At its core, the ISS can translate flight CPU architecture (e.g., PowerPC) machine instructions. Once the ISS is chosen, it needs to integrate with modeled peripherals (e.g., UART, discrete I/O, PCI bus, etc.). Detailed Flight Computer documentation is needed to completely model the board and describe its peripherals, interrupts, RAM, EEPROM, cPCI BAR registers, and memory maps. Having this documentation on-hand is critical to begin modeling the flight computer.
The publicly available Figure 20 above [55] describes the internal architecture of the RAD750 flight computer. Notice that the flight computer comprises various peripheral-components such as SpaceWire, RAM, UART, EEPROM, and MIL-STD-1553. This is a ubiquitous architecture for both custom flight boards and Systems-On-Chip (SoC), whereas the CPU is “surrounded” by peripheral-components (also known as “devices”), often tied together via an internal I/O bus or Compact PCI (cPCI) bus. If a specific version of flight software utilizes any of these devices, then the modeled flight computer must include corresponding modeled virtual devices. This technique is useful for determining what functionality and how much functionality to model. For example, if the flight software does not use SpaceWire or MIL-STD-1553, this functionality should NOT be modeled on initial iterations. Only model the needed behavior and functionality; refer to Chapter 5.2 for more information.

3.4.1.1 Components and Memory Mappings

The flight computer model is considered a Component object in software, and the peripherals are considered “Device Objects,” as shown in Figure 21. The higher-level component CONTAINS devices, and the component is responsible for aggregating the devices into a functional flight computer model. Notice the similarity between actual hardware system architectures and the software-only modeling architecture. The architectures are meant to be similar, with the differences being the software-layer that connects components-to-components and devices-to-components, providing an abstraction layer.
Components can be represented by any computer language, with C and Python being the most common for SoSTs. If Python is utilized, then an object-oriented methodology is used, which naturally follows the component architecture depicted in Figure 21. Software objects are added to Components to form a flight computer that contains devices. For example, Figure 22 shows how a specific PowerPC e600 flight computer model in Python CONTAINS a UART/serial port, disk controllers, timers, and a real-time clock (RTC). The critical part of this concept is to understand that the component is made up of many objects, with those objects being devices that need further definition and models developed in either Python or C. The component ties the devices together into a working flight computer board. Figure 23

```python
# Serial
uart = self.add_pre_obj('uart[%d]' % self.serial_ports.val, 'NS16550')
for i, irq in enumerate(uart):
    uart.irq[i] = [pic, 'irq[%d]' % (uart_irq_base + i)]
reset_devs += uart

# Disk Controllers
diskCtrls = self.add_pre_obj('diskCtrls[%d]' % self.disk_controllers.val,
                            [DiskController])
for i, diskCtrl in enumerate(diskCtrls):
    diskCtrl.irq_target = [pic, 'irq[%d]' % (diskCtrl.irq_base + i)]
    diskCtrl.phys_mem = phys_mem
    reset_devs += diskCtrl

# Timers
timers = self.add_pre_obj('timer[%d]' % self.timers.val,
                          [Timer])
for i, timer in enumerate(timers):
    timer.irq_target = [pic, 'irq[%d]' % (timer.irq_base + i)]
    timer.freq_Hz = 1800000
    reset_devs += timer

# RTC
rtc = self.add_pre_obj('rtc', 'rtc')
reset_devs += [rtc]
```

Figure 23. Python Snippet Showing Devices Added to a Flight Computer Component

![Figure 22. Publicly Available RAD750 Architecture Showing Device Models](image)
illustrates the concept of “device-models” by showing the RAD750 hardware architecture overlayed in blue with the corresponding software device models.

There is a one-to-one mapping so that the SoST architecture matches the hardware architecture. The mapping of devices to memory addresses also occurs in the Component. For example, in Figure 24, the Python code shows that the devices’ base addresses define the flight computer’s physical memory map. The actual devices are then mapped into the SoST’s flight computer model’s virtual memory by using the device’s offset address. This mechanism in the SoST is similar to the actual hardware method for mapping devices into memory address space for PowerPC CPUs. Up to this point, how the components manage the device objects has not been fully explained; in order to better understand the implementation, the device models need a detailed explanation.

3.4.1.2 Device Modeling

Device Modeling is performed on individual devices then aggregated together to build larger component models, such as the flight computer board. Device Models can be implemented in nearly any language, specifically Python, System-C, C, Device Modeling Language (DML), or C++ in the SoST. If using the QEMU ISS, then C or C++ will be the language of choice; if using Wind River’s Simics ISS, several choices are available, notably DML, a C-like language is auto-coded to C. Once developed, the models are integrated using a composition object-oriented

```python
# Physical RAM Layout and Device Mapping
phys_mem.map = [
    [ram_base, ram, 0, 0, ram_image.size],
    [flash_base, flash, 0, 0, flash_image.size, flash],
    [pic_base, [pic, 'regs'], 0, 0, 0x1000],
    [sysregs_base, [sysregs, 'regs'], 0, 0, 0x1000],
    [rtc_base, [rtc, 'regs'], 0, 0, 0x1000]
]

for i in range (self.serial_ports.val):
    phys_mem.map += [
        [uart_base+i*0x1000, [uart[i], 'regs'], 0, 0, 0x800]
    ]

for i in range (self.network_ports.val):
    phys_mem.map += [
        [mac_base+i*0x1000, [mac[i], 'regs'], 0, 0, 0x1000]
    ]

for i in range (self.timers.val):
    phys_mem.map += [
        [timer_base+i*0x1000, [timers[i], 'regs'], 0, 0, 0x1000]
    ]

for i in range (self.disk_controllers.val):
    phys_mem.map += [
        [disk_ctrl_base+i*0x1000, [disk_ctrls[i], 'regs'], 0, 0, 0x1000]
    ]

for i in range (self.led_controllers.val):
    phys_mem.map += [
        [led_base+i*0x1000, [leds[i], 'regs'], 0, 0, 0x1000]
    ]
```

*Figure 24. Mapping Devices to Memory Layout*
(OO) paradigm, where each device represents a software object that forms the entire flight computer board model.

Memory devices are the simplest to model, as they represent a memory area on the host computer that is addressable in a specific range in the SoST. For example, if a PowerPC e600 has an addressable RAM range from 0x00 to 0x10000 and an addressable EEPROM range from 0x20000 to 0x40000, then heap memory objects (allocated with `malloc()` or `new()` in the SoST) can be created on the host computer and utilized in the SoST for the memory objects. The ISS, via its various software mechanisms, can map the heap memory into the SoST virtual-CPU address space so that the virtual-CPU now has virtual RAM and EEPROM address ranges. In SoSTs, the electrical characteristics of the RAM/EEPROMS are NOT modeled because these details are usually not required from the flight software’s perspective. Refer to Chapter 5.2 for details on applying the correct level of modeling fidelity.

Other than memory devices, the device/peripherals require significant modeling that includes I/O address space mapping, register mapping, interrupts, direct-memory-access (DMA), First-In-First-Out (FIFOs), and their respective behavior. For example, a reference star tracker cPCI device is described below (in Figures 25 – 27) and annotated in DML for conceptual readability. Each aspect of the Star Tracker’s model is discussed, but some ancillary details have been removed for clarity.
Declare the device and its methods. Methods are any general processing activities that the model needs to perform.

```plaintext
parameter documentation = "Representative Star Tracker Device Model";

$load_star_catalog = map.bin // star map to load into star tracker model

method init() {
    fread($load_star_catalog); // load the map to the Star Tracker memory
}

method hard_reset() {
    call $clear_registers();
    call $clear_interrups();
}

method soft_reset() {
    call $clear_registers();
    call $clear_interrups();
}

method do_centroid() default {
    // perform centroid by utilizing x,y,z register values
    // converge on a probable star
}

call $clear_registers()
{
    x_attr = 0;
    y_attr = 0;
    z_attr = 0;
}

call $clear_interrupts()
{
    $interrupt_done.lower();
}
```

Figure 25. Star Tracker Model Part 1
For PowerPC architectures, devices are memory-mapped, meaning that their registers for status/control and behavior are mapped to specific memory locations. Then, the flight software accesses those registers via its virtual memory addressing. The device model must execute in the virtual-time context (See Chapter 3.4.3) and also handle the appropriate interrupt mappings to the virtual PowerPC CPU ISS. There is a single
interrupt for the PowerPC CPU whose input is often multiplexed from various peripheral
devices, including cPCI. The entirety of these virtual-hardware interactions occurs in the
software-only context. For example, registers are constructs of the model, and interrupts
are “virtual signals” in software that are connected via the component model to devices.

3.4.2 Advanced SoST Modeling Techniques

The majority of SoSTs developed to date have utilized a Compact PCI architecture as the
system bus. In the actual hardware system, the flight computer is a cPCI “card,” and the
missions develop custom flight cPCI cards with various FPGAs providing register banks,
FIFOs, DMAs, and interfacing to external peripherals. Flight software interacts with
hardware via the cPCI bus by utilizing PCI Plug-N-Play (PnP) and initializing the Base
Address Registers (BARs) in cPCI Configuration Space. Flight software writes to the
cPCI BAR registers, which maps the cPCI card/device into the CPU’s address space.
These activities are standard PCI transactions; for more information on the PCI bus, refer
to its standard. Modeling PCI transactions in a SoST, and their synchronicity, are the
subject of this section.

3.4.2.1 Modeling of Synchronous Transactions

Synchronicity is a fundamental feature of SoSTs and is needed to ensure consistency
and accuracy when executing the flight software. A synchronous system is implemented
by utilizing a cycle-accurate ISS and converting real-world hardware events into
synchronous events in the software-only model.

For contrasting real-world systems with software-only (or virtualized) systems, real-world
(especially real-time systems) operate on a processing schedule, usually with polling
hardware drivers that are not interrupt-driven. Wall-clock time continues to move forward
in real-world systems, regardless of the CPU execution; often, a synchronization method
is utilized to sync the wall-clock time with the real-time system clock.

In contrast, software-only systems, such as SoSTs, utilize a virtual-CPU (e.g., ISS) that
executes in a simulation time context. Simulation time is joined with the ISS execution
and does not correlate to wall-clock time. The ISS becomes the time master of the SoST
and is responsible for driving the virtual “time-tick” to the entire SoST. The “time-tick”
drives the simulation and can be stopped, paused, or restarted. Often, the “time-tick”
corresponds to an $X$ number of CPU cycles; thus, for each step forward in virtual time, the ISS will perform $X$ CPU cycles in addition to allowing input/output to also execute. Stopping or pausing the “time-tick” will stop/pause the entire SoST execution in a known state, which is a significant advantage for SoSTs over their hardware equivalents. The stopping/pausing its state is a unique property of SoSTs; hardware test beds are not capable of pausing complete CPU execution.

Referring to its design, Section 3.3.5 and in Figure 28, NOS Engine is utilized for time distribution between the SoST components. Each component is linked to the SoST by implementing the standard NOS Engine API interface; thus, it is able to consume ticks and operate as a synchronous member of the system.

For example, the following two Figures demonstrate a Novatel GPS model using NOS Engine’s time synchronization to provide periodic Position-Velocity-Time (PVT) to the other SoST components. Figure 29 shows that the GPS Model [56] uses the `add_time_tick_callback()` function to define a function pointer to the `send_periodic_data()` function (Figure 30). Thus, on every received time-tick from the NOS Engine Time Sender, the GPS model will execute the `send_periodic_data()` function and emulate the PVT for the model. This data is then written over the virtual UART interface so that the SoST’s virtual flight computer believes that it is communicating with a GPS on its UART interface in real-wall-clock-time. All transactions are synchronous, meaning that the flight computer model is not aware that an arbitrary wall-clock time has elapsed.
Memory Banks and Register modeling was covered previously, but a natural extension of the two is Direct Memory Access (DMA) Engine modeling, with its architecture shown in Figure 31. DMA is often used by the custom cPCI hardware to transfer data to peripherals, store incoming data from peripherals/busses, and transfer data between the flight computer and the cPCI peripheral efficiently. Because DMA is essential to all spacecraft architectures, its modeling is described here in detail.

From the SoST hardware perspective, DMA can be considered a moving-of-bytes-from-one-address location to another address location. To facilitate this transfer, registers that control the DMA must first be defined and implemented in the model so that the flight
software can utilize them for DMA initialization and operation. For example, a standard DMA would have registers similar to the ones defined in Figure 32.

The flight computer writes to the modeled registers to setup the DMA Engine per its specification. Behind the scenes, the DMA Engine model needs to initialize the engine and prepare the RAM for future operations. Once the DMA is setup, its model will begin operation by writing to a particular register typically named “ENABLED” or “START_EN.” When the model sees a “1” written to this register by flight software, it will then proceed to execute its function (such as a MIL-STD-1553 transaction) and copy bytes into a ready-buffer in RAM on the cPCI I/O card. Once the DMA operation is complete, the model needs to inform the flight computer by writing a “1” to a DMA_COMPLETE register, which will either be polled by the flight software or serviced by an interrupt service routine.

Note that because the SoST is a synchronous system, this DMA transaction by default will execute in what appears to be instantaneous time to the flight computer’s perspective. In some cases, this is not ideal because the flight computer expects DMA transactions to occur within a pre-determined time that is a characteristic of the hardware. This time delay will most likely need modeled and allow the SoST to run forward in virtual-time so as to emulate the DMA transaction wall-clock time. Failure to do this can cause the flight computer software to miss the DMA_COMPLETE state if it is not polling for it yet in its processing schedule. This is a common SoST model development bug that needs careful design and attention to avoid.

```c
// dma registers

// uplink dma control registers
register dma_start_A @ 0x84 "Start Address A"
    { // implementation details here }
register dma_end_A @ 0x88 "End Address A"
    { // implementation details here }
register dma_pointer_A @ 0x8c is [read_only] "Pointer Address A"
    { // implementation details here }
register dma_start_B @ 0x90 "Start Address B"
    { // implementation details here }
register dma_end_B @ 0x94 "End Address B"
    { // implementation details here }
register dma_pointer_B @ 0x98 is [read_only] "Pointer Address B"
    { // implementation details here }
register dma_status_control @ 0x9c "Uplink DMA Status/Control Register"
    { // implementation details here }

Figure 32. Representative DMA Register Definitions
```
3.4.3 MIL-STD-1553: Synchronous Modeling of Input / Output (I/O) Interfaces

Up to this point the modeling of registers, memory banks, and DMA have been discussed in a synchronous context. Data busses and I/O interfaces represent the last set of SoST components to model in order to perform complete system integration.

Standard spacecraft interfaces include MIL-STD-1553, SpaceWire, custom serial protocols utilizing RS232/RS485, and Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI). Each of these data bus interfaces can be modeled to various fidelity levels depending upon the specific functionality needed. For example, higher fidelity SoSTs model the MIL-STD-1553 to the transaction level, even going so far as modeling the transaction time and intermessage-gap time in microseconds. Otherwise, the data busses can be abstracted to a higher software mechanism that performs synchronous data transfer and does not model protocols nor electrical characteristics. The desired level of fidelity is left to the designer to choose based upon the SoST requirements and planned use-cases. Data bus hardware maps itself into the flight computer’s address space and is accessed via memory-mapped I/O operations, DMA, or control/status registers.

The SoST modeling architecture for interfaces will closely match the actual hardware architecture. For example, the MIL-STD-1553 publicly available Summit architecture [57] is shown in Figure 33 and Figure 34; this architecture is useful for describing the SoST 1553 model design and implementation. The 1553 Engine is usually embedded on a cPCI card or auxiliary device that can communicate with the flight software via a DMA interface.

![Figure 33. Typical MIL-STD-1553 Architecture: Public Summit Example](image)
The Summit architecture demonstrates two main parts: 1) The Engine (labeled “Summit”) in the figure and 2) the transceivers. When modeling 1553 and other data busses, only the “forward-facing” hardware that interacts with the flight software/flight computer should be considered for high fidelity modeling. As shown in Figure 34, the transceivers, JTAG debug, and AUTO-INIT Bus do not need to be modeled. These components are superfluous to its operation and are not directly “seen” by the flight software.

Figure 35 highlights in green the interfaces that require modeling, along with the transceivers being replaced with a software abstraction layer using NOS Engine. Most often, the Summit Engine is a 1553 Bus Controller (BC) that utilizes DMA to transfer data between the Remote Terminals (RTs) and the flight software. Internally, the BC contains registers and interrupts that need modeled using previously discussed methods. NOS Engine’s implementation for synchronous MIL-STD-1553 is described below.
3.4.3.1 NOS Engine Client and MIL-STD-1553 Plugin

If a SoST wishes to contain both a virtual NOS Engine Bus Controller and Remote Terminal hub, two unique connections must be made to the NOS Engine server. The Bus Controller ("BC") is the object on the network used to initiate all bus traffic. Currently, the creation of the BC object on the bus will result in the creation of a data node called "bc."

Remote Terminals (RTs) are not directly created by a user on the bus. RTs are instead attached to a user created Remote Terminal Hub. The hub is responsible for distributing 1553 messages to user-designated callbacks in device models as needed. Each hub may represent multiple RTs at any given time (See Figure 36). They may be added and removed from the SoST system at runtime as needed. SoST designers are then provided with a representation of an RT to use to respond to the message.
All messages are received for a given RT synchronously through the provided callback during RT registration. The callback is used to represent the entirety of an RT, meaning that both the receiving and transmission of all data for all subaddresses are expected to be handled. In the event of a broadcast message on the bus, all provided callbacks will be called. Each callback is required to acknowledge the broadcast message before bus communication can proceed, thus maintaining synchronicity.

The NOS Engine MIL-STD-1553 plugin library provides designers with several convenient methods for inserting or extracting protocol information from a given message. “Userspace” protocol elements of data messages are relatively simple. The following table describes the components of the data protocol.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>BC -&gt; RT Message</th>
<th>RT -&gt; BC Response</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC send data to RT (Receive Request)</td>
<td>[Command Word] [Word Count] [Data]</td>
<td>[Error Code] [Status Word]</td>
<td></td>
</tr>
<tr>
<td>BC request data from RT (Transmit Request)</td>
<td>[Command Word]</td>
<td>[Error Code] [Status Word] [Word Count] [Data]</td>
<td></td>
</tr>
<tr>
<td>RT Ignores transaction (Either receive or transmit)</td>
<td>[Command Word] [Optional Data]</td>
<td>[Error Code] [Status Word]</td>
<td>An automatic Status Word is sent for the server to interpret the message. It is ignored by the BC</td>
</tr>
<tr>
<td>BC Broadcast Data</td>
<td>[Command Word] [Optional Data]</td>
<td>[Error Code] [Status Word]</td>
<td>The Error Code portion of the message is always returned as “No Error.”</td>
</tr>
</tbody>
</table>
The requirements for 1553 communications within the NOS Engine server are very minimal. This is mostly due to the fact that the majority of work performed on a 1553 bus is simply data passing. Interactions between nodes are not complicated, and therefore, the Base Layer of NOS Engine can mostly be relied upon to perform the work.

The server-side implementation of the plugin provides a specialized 1553 bus class and router for use. The work performed by these classes is minimal and only performed in a small number of cases. The most common of these special operations is the association of an RT address with an existing data node. A select protocol command is processed to report the association. The server-side work is mostly directing messages to a given RT on an associated node. This allows the 1553 layer to be more dynamic than the traditional Base Layer. For example, any time an RT is successfully associated with a data node, all interceptors for that RT are added to the data node.

3.5 Deployment and Usage

Eight years ago, when SoSTs’ design and implementation began, there were no software tools available to assist with virtual machine creation or distribution. Virtual Machine (VM) technologies existed and were sufficiently mature at the time, and thus SoSTs have always supported and promoted virtualization for their deployments. Virtual Machines are convenient for SoSTs but do inherently involve maintenance activities that need to be addressed and scheduled. Most notably, VMs often require significant storage space, with sizes quickly reaching 50+ Gigabytes (GB) per VM configuration; this size can limit deployment options and decrease deployment time, as VMs need to be copied and shared. Also, and more importantly, once SoST VMs are copied, deployed, and utilized, then configuration management (CM) policies must be implemented for the SoST VMs. CM, on such large files coupled with their configuration and deployment, can be complicated and become unwieldy. For example, if User Y wants a flight software version change for a SoST VM, then CM questions arise such as:

- Will the change be made for just User Y?
- Will the change be made for the entire SoST Baseline for future users?
- Will the change be made for all users immediately?
• How will the change be distributed to the user(s)? It is not feasible to deploy 50+ GB SoSTs to a large number of users.

3.5.1 Ready-To-Run Virtualization Implementation

The solution for this is to implement VM provisioner technologies for SoSTs, thus eliminating the CM challenges of managing large-sized VMs. Provisioners allow VMs to be created “on-the-fly” from a set of source code files that can remain under CM version control. Instead of distributing large SoST VMs, only the VM’s “generation scripts” need to be distributed so that users can generate their own VM on their own VM infrastructure. The “generation scripts” can be maintained in a standard source control CM repository and modified as needed to support many users.

3.5.2 Implementation Details for Auto-Deployment of a SoST

This section describes the details of implementing auto-deployment for the NASA Operational Simulator for SmallSats (NOS3) SoST. The theory and implementation details presented here for NOS3 apply to all SoSTs. Because NOS3 is now NASA open-source, its auto-deployment is very important for its massive user base.

The NOS3 collection of software components are conveniently packaged as a ready-to-run virtual machine, reducing the overhead associated with installing and configuring each software component. NOS3 can be distributed as an Oracle VirtualBox virtual machine image or as a collection of command scripts that are used to recreate and modify a virtual machine image. This allows users to have a common development and testing environment, further reducing risk to the mission. The standard guest operating system utilized by NOS3 is Ubuntu Linux, but the virtual machine can execute using Oracle VirtualBox on Windows, Mac, or Linux computers.

To get started with NOS3, a user only needs to install Oracle VirtualBox and Vagrant on their host computer. Both of these software packages are open source and can be run on various operating systems, including Microsoft Windows, Apple OS X, and Linux. In addition to those software packages, NOS3 comprises a collection of files stored in a git repository. To get started with NOS3, the user receives a copy of those files and places them on their computer. These files include a Vagrantfile, which is a file that is used by the Vagrant software package to create an Ubuntu Linux Virtual Machine where all of
NOS3 is executed. During the Ubuntu Linux Virtual Machine creation, various software packages are installed, including COSMOS, 42, and the NOS Engine libraries and NOS Standalone Server. An alternative to starting with Vagrant is to receive an already generated VirtualBox Virtual Machine with the various packages installed or utilize the same provisioner script on an Ubuntu 14.04 virtual machine/host; however, to build and run the core flight software, simulators, and so on, the source code will still need to be present as described below.

Finally, source code for various simulators is present on the virtual machine through synced folders, which allow access to the same files on the host computer and the virtual machine computer. Build tools can be used on the virtual machine to build and install these simulators, such as a GPS simulator, a magnetometer simulator, an antenna simulator, and more. Also, two special software tools are built and installed as part of the simulators. The first is a NOS time driver, which provides time ticks to drive time for the various simulators, 42, and the flight software. The second is a simple terminal program that the operator can use to command and control other simulators using a separate NOS engine command bus that all of the simulators can be nodes on.

In addition, the cFS flight software source code is also present on the virtual machine through synced folders. Build tools can be used to build and install the generic flight software also. This flight software includes hardware libraries that can interface as nodes on NOS Engine busses in place of the real hardware node and bus connections.

The following two figures, Figure 37 and Figure 38, show an example auto-VM creation file that is maintained under CM and utilized to generate ready-to-run NOS3 SoST VMs.
Multiple host OSs are supported

VM characteristics are always defined for consistency

Figure 37. NOS3 Deployment Example - Part I
During VM generation invoke shell provisioner and execute several shell scripts (SH) for installing specific software.

Exit the provisioner, reboot the VM, then enjoy the fresh NOS3 SoST VM.

Figure 38. NOS3 Deployment Example - Part II
The following steps are shown in Figures 39 through 44 to illustrate the typical SoST ease of installation.

1. Open a command terminal and navigate to the directory where NOS3 was unzipped.

![Command Terminal](image)

*Figure 39. Command Terminal*

2. Execute “windows_nos3_installer.bat”, “linux_nos3_installer.sh”, “pi_nos3_installer.sh”. Note if using linux or pi, may need to dos2unix the file before running. Example: ‘dos2unix linux_nos3_installer.sh’

![Command Prompt](image)

*Figure 40. Run Installer Script*

3. The command prompt will be renamed with the disclaimer displayed.

![NASA Operational Simulator for Small Satellites (NOS3)](image)

*Figure 41. Disclaimer*
4. Confirm you have read the disclaimer by pressing ‘Y.’

![Figure 42. Level of Install](image)

5. The level of install by entering 1, 2, or 3.
   a. Each level builds upon the previous one with minimal, or ‘1’, being the base

![Figure 43. Complete Configuration](image)
6. Wait for the installation process to complete - Completion is certain once the script exits as displayed below:

```
$> default: --- cleanup ---
$> default:
$> default: Reading package lists...
$> default: Building dependency tree...
$> default: Reading state information...
$> default: Upgrading 0 packages, 0 newly installed, 0 to remove and 0 not upgraded.
$> default: Success!
$> default: Exiting shell provisioner ---
$> default:
$> default: Configuring cache buckets...
$> default: Running provisioner: shell...
$> default: Running inline script
$> default: stdin is not a tty
$> default: Run `vagrant reload` to start the graphical environment if necessary
$> default: Configuring cache buckets...
$> default: Attempting graceful shutdown of VM...
$> default: Checking if box 'ubuntu/trusty32' is up to date...
$> default: A newer version of the box 'ubuntu/trusty32' is available! You currently
$> default: have version '20160827.0.1'. The latest is version '20160820.0.0'. Run
$> default: `vagrant box update` to update.
$> default: Clearing any previously set forwarded ports...
$> default: Clearing any previously set network interfaces...
$> default: Preparing network interfaces based on configuration...
$> default: Adapter 1: nat
$> default: Forwarding ports...
  $> default: 22 (guest) => 2222 (host) (adapter 1)
$> default: Running 'pre-boot' VM customizations...
$> default: Booting VM...
$> default: Waiting for machine to boot. This may take a few minutes...
  $> default: SSH address: 127.0.0.1:2222
$> default: SSH username: vagrant
$> default: SSH auth method: private key
$> default: Machine booted and ready!
$> default: Checking for guest additions in VM...
$> default: The guest additions on this VM do not match the installed version of
$> default: VirtualBox! In most cases this is fine, but in rare cases it can
$> default: prevent things such as shared folders from working properly. If you see
$> default: shared folder errors, please make sure the guest additions within the
$> default: virtual machine match the version of VirtualBox you have installed on
$> default: your host and reload your VM.
$> default: Guest Additions Version: 4.3.36
$> default: VirtualBox Version: 5.2
$> default: Mounting shared folders...
  $> default: /vagrant => C:/Users/JPL/NOS3_v1.02.00/nos3/support
  $> default: /vagrant_parent => C:/Users/JPL/NOS3_v1.02.00/nos3
  $> default: /tmp/vagrant-cache => C:/Users/JPL/.vagrant.d/cachзуubuntu/trusty32
$> default: Configuring cache buckets...
$> default: Machine already provisioned. Run `vagrant provision` or use the `-provision`
$> default: Flag to force provisioning. Provisioners marked to run always will still run.

Exiting the NOS3 installer now! NOS3 is ready for use!
```

Figure 44. Installation Complete
4 SoSTs for SmallSats and CubeSats

4.1 Introduction

The last ten years have witnessed the revolution and exponential growth of SmallSats / CubeSats in the commercial and government sectors. However, according to the 2019 NASA Ames Research Center study titled, Small-Satellite Mission Failure Rates [16]

“…41.3% of all small satellites launched experienced total or partial mission failure. Of these, 6.1% were launch vehicle failures, 11% were partial mission failures, and 24.2% were total mission failures.”

Also of significant importance to this research, as noted in this report, “Another reason [for the failures] could be that as the small satellite software complexity has increased, the methods used to perform verification and validation of the small satellite software has not increased commensurately.”

SmallSats that have a primary science focus (excluding Department of Defense SmallSats to some degree) continue to evolve into complicated and constrained missions, with a historical lack of mission assurance that needs to be addressed throughout the industry. Future SmallSats will be interplanetary constellations that will require significant testing, lower failure rates, and increased software robustness and reliability to successfully meet the stringent mission objectives and goals but with lower costs.

The goal of this chapter is to discuss the application of SoSTs to SmallSats and CubeSats. (The term “SmallSats” is used interchangeably with CubeSats.) Flagship missions (as discussed previously) have different (i.e., more formalized) development lifecycles than SmallSats, and thus, SoSTs for SmallSats also need to be developed and applied in different ways in order to be effective. Part of this research and dissertation is identifying and defining the process of successfully applying SoST technologies to SmallSats in ways that make sense and have a considerable return-on-investment that significantly improves their mission assurance and flight software V&V. The goal is to determine if SoSTs can be innovative technologies that can help SmallSats succeed without significant cost increases. Or stated differently, “Can SoSTs be an invaluable tool for SmallSat development?”
4.2 Considerations for Designing a SoST for a SmallSat Mission

A spinoff of applying SoSTs to flagship missions is the new application of SmallSats. SmallSat missions are different from flagship missions in multiple ways and thus requires a different SoST design approach. Table 7 outlines the main differences between flagship and SmallSat missions and describes the SoST design and implementation impact. Essentially, individual SoSTs need to scale up or down in design and implementation in order to serve as an effective development and V&V platform.

Table 7. Differences Between Flagship and SmallSat Missions

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>SoST Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flagship mission budgets are exponentially larger than SmallSats</td>
<td>SoST budgets for engineering and implementation will often track the mission budget. For smaller SmallSat missions, SoST budgets will also be considerably smaller.</td>
</tr>
<tr>
<td>Flagship mission schedules and timelines are orders of magnitude longer than those of SmallSats</td>
<td>The point of SmallSats is to engineer and launch a spacecraft in a short timeframe (~2 years), as opposed to flagship missions such as JWST, which have been in development for 10+ years. SoSTs have to have a quicker development schedule and thus be less complex for SmallSats.</td>
</tr>
<tr>
<td>Flagship missions have more personnel and relatively fewer test resources</td>
<td>SmallSats often have a small core test team. Flagship missions have several test teams, thus requiring additional test hardware resources. This can make SoSTs in higher demand for Flagship missions.</td>
</tr>
<tr>
<td>Flagship missions are more complex, often in the Attitude Control System (ACS) requirements</td>
<td>ACS requires significant testing, and flagship missions require additional V&amp;V, which SoSTs can assist with. SmallSats can take advantage of SoSTs in the spacecraft development phases.</td>
</tr>
<tr>
<td>Flagship missions often operate in deep space, thus requiring more specialized hardware</td>
<td>Specialized hardware results in fewer FlatSats and hardware test beds. The trend for more specialized hardware continues to become more complex and functional for SmallSats as well. SoSTs, now and in the near future, will require more specialized hardware modeling, particularly in the cases of Compact PCI and bus I/O.</td>
</tr>
</tbody>
</table>

Applying SoSTs to SmallSats requires a “Risk Reduction Mindset.” Instead of supporting formal development and V&V phases as is done with flagship missions, SoSTs should be infused into the SmallSat development lifecycle at critical points where targeted
mission risk reduction can be performed. An adopted “Risk Reduction Mindset” throughout the entire development, and testing lifecycle can drive and prioritize the test program activities. For example, for the smaller SmallSat teams where the engineers often have multiple roles of both developers and testers, the engineers can continuously ask themselves, “How can I reduce the component risk of on-orbit failure?” Ingrowing this agile thought process into the entire lifecycle aligns with a SoST test-driven approach. It empowers the engineers to take ownership of the risk reduction, testing, and development phases, thus providing “risk ownership” and an overall lowered risk profile for the entire SmallSat mission.

Consideration also has to be given to the fidelity of the SmallSat SoST so that it is best tailored for the mission. Even within SmallSat missions, their requirements can be diverse and very different. If flagship missions require a high-to-medium fidelity SoST, then SmallSat missions require a medium-to-low fidelity SoST. The definitions of fidelity are loose and do not follow strict definitions in this dissertation because each mission is a different scope with different goals. Choosing the correct level of fidelity is discussed more in Chapter 5.2.

Table 8 shows the typical design differences between FlagShip Mission SoSTs and SmallSat SoSTs. These design differences are driven by the previous table that outlines the major differences between SmallSats and flagship missions. There is more flexibility in the design of SoSTs for SmallSats; for example, instruction set simulators may not be required, and executing the test-as-you-fly binary may also not be required. However, SmallSat SoSTs will still utilize the NOS Engine middleware to integrate various components, the ground software system will be included and integrated, and hardware models will be developed and integrated.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Flagship</th>
<th>SmallSat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilizes NOS Engine</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Utilizes an instruction set simulator</td>
<td>YES</td>
<td>Sometimes, Not Always. Depends upon desired fidelity</td>
</tr>
<tr>
<td>Complete Test-As-You-Fly configuration</td>
<td>YES</td>
<td>Sometimes. Depends upon desired fidelity</td>
</tr>
<tr>
<td>Contains hardware component models</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Includes Ground Software System (e.g., COSMOS)</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Contains actual hardware-in-the-loop</td>
<td>NO, but is a future feature</td>
<td>Sometimes</td>
</tr>
</tbody>
</table>
4.3 SoST: The NASA Operational Simulator for SmallSats (NOS3)

4.3.1 STF-1 CubeSat Introduction

As a result of the demonstrated successes of SoSTs for flagship missions and the opportunity to launch a spacecraft to demonstrate and test technologies that benefit SmallSat missions, the NASA IV&V Program and West Virginia University (WVU) collaborated to develop a 3U CubeSat mission, named Simulation-to-Flight 1 [46]. This author was one of three primary authors of the STF-1 proposal. The primary purpose of STF-1 was to determine and demonstrate the value of developing, utilizing, and maintaining a SoST during the project lifecycle. However, a diverse set of science experiments provided by WVU allowed the project to expand the mission’s overall objective. The instruments included a cluster of Micro-Electro-Mechanical Systems (MEMS) Inertial Measurement Units (IMU) to produce attitude knowledge; a space-weather experiment including a Geiger counter and Langmuir probe; a III-V Nitride-based materials optoelectronics experiment; and a Novatel OEM615 GPS coupled with advanced algorithms for more precise orbit determination. The science experiments enhanced the mission capabilities and provided a diverse set of instruments to assess how the simulator would support science instrument development. Figure 45 illustrates the various components and subsystems of the STF-1 CubeSat mission.

4.3.2 NOS3 Architecture Design

While STF-1 is not the focus of this dissertation, the STF-1 mission did result in the development of the first SmallSat SoST named the NASA Operational Simulator for Small Satellites (NOS3). The goal of NOS3 is to enhance small satellite software development, testing, and training. NOS3 provides the flight software with representative real-world
simulated data inputs that it would expect during nominal on-orbit operations. Some of the NOS3 features include:

- Enabling multiple developers to build and test flight software with simulated hardware models
- Serving as an interface simulator for science instrument/payload teams to communicate with before hardware integration
- Supporting software development activities
- Enabling hardware integration in parallel with software development activities
- Providing an automated testing framework
- Increasing available test resources
- Enabling operation of the simulated spacecraft using the ground software command and telemetry databases.

The flexible configuration of the NOS3 simulation architecture as compared to a typical flight system is illustrated in Figure 46. The Flight Configuration column provides a typical small satellite flight configuration for the flight software: i.e., flight applications, flight libraries, drivers, and flight hardware. The flight software may use flight libraries that provide common functionality. The flight software and libraries utilize hardware drivers, defined as software components communicating directly with the hardware. This is usually accomplished via reading and writing hardware registers and often using a bus protocol such as Universal Asynchronous Receive and Transmit (UART), Inter-Integrated Circuit (I2C), and Serial Peripheral Interface (SPI), or merely applying general-purpose input/output (GPIO) signals. Note that the Simulation Configuration (right column of Figure 46 below) is identical to the Flight Configuration (left column), with the exception being that NOS Engine replaces the Flight Hardware. This is important because the Drivers, Flight Libraries, and Flight Applications remain the same regardless of the Flight Configuration or the Simulation Configuration is being utilized.
A typical SmallSat has numerous hardware interfaces controlled through an on-board computer. These may include hardware interfaces with electrical power systems, radio frequency communication systems, science experiment payloads, orbit, attitude sensing and control systems. The goal of NOS3 is to substitute simulations in place of some or all of these hardware components.

The Simulation Configuration column demonstrates how NOS3 can be utilized in place of the actual hardware. It should be noted that the NOS3 architecture provides users with the flexibility to execute flight software with some or all of the hardware components replaced by a software simulation. This substitution occurs at the functional call interface. Performing the substitution is as simple as linking the flight software against a NOS3 library to replace the hardware driver library. NOS3 utilizes a client-server architecture, and as such, a standalone NOS3 server manages the communications between flight software and various hardware components. The stand-alone server maintains the components, referred to as NOS Engine nodes, that are attached to each hardware bus, the communications protocol used, etc. Additionally, NOS3 includes a logging mechanism so that communications between simulation components can be monitored in real-time or in post-analysis to ensure that the data is passed correctly.
The hardware components that are being substituted with software simulations can be modeled at the fidelity required for the tests being performed. Some of the simulators written for STF-1 simply implemented pre-packaged data responses to commands from the on-board computer, while others required knowledge of the environment or other hardware components. For example, a GPS simulator needs to know the spacecraft position in orbit; therefore, this data must be generated dynamically. Simulators requiring this type of dynamic data utilize a connection to the dynamics software (e.g., “42” software) to collect the necessary data and then proceed to package the response in the proper hardware format. The simulated components can be manipulated by the user, allowing fault testing that typically is not possible or too dangerous to attempt in a hardware-only test.

NOS3 integrates a set of existing open-source software components and custom software components to create a fully functional SoST. Figure 47 illustrates how these software components are interconnected within NOS3. The following subsections examine each of these software components and their individual purpose: i.e., NOS Engine, COSMOS, cFS Flight Software, “42” Dynamics Simulator, Hardware Simulations.

Figure 47. NOS3 component connections between ground (COSMOS), FSW (cFS), & Dynamics (42)
One of the primary software components of the NOS3 simulator is the NOS Engine simulation middleware (previously discussed in Chapter 3.3.5) that abstracts the hardware and connects the flight software with the simulated dynamics. NOS Engine, as previously discussed, is an in-house developed software suite that provides a library of functions to simulate the hardware communication protocols that are utilized by the flight software. As discussed in the previous sections, the hardware driver libraries are replaced with NOS Engine libraries utilizing identical signature function calls. NOS Engine also supports various underlying protocols such as TCP/IP, inter-process communication protocol (IPC), and shared memory to transport software bus messages that represent the actual hardware bus communication. This functionality provides a number of unique advantages: high-speed communications; shared memory on a single computer running the flight software and the software simulators; and distributed processing such as TCP/IP on multiple computers, one running the flight software and others running the software simulators or interfacing with flight hardware, and other configurations based on development and testing requirements.

One of the challenges of simulated communications protocols (e.g., UART, I2C, SPI) is representing their hardware time synchronization clocks within a software-only environment. Time synchronization clocks are used on small satellites to coordinate spacecraft time with ground time, coordinate time between various spacecraft components such as the on-board computer software and the radio frequency communication component, and provide timing signals for clocks that coordinate communication using protocols such as I2C and SPI. To overcome such a challenge, the NOS Engine library contains methods to manipulate and distribute time between various components that are connected via software busses in place of what would typically be hardware busses. For example, within NOS3, NOS Engine is utilized to control epochs and periodic clock signals.

**4.3.3 Ground Segment: COSMOS**

COSMOS [58] is an open-source command and control software package, and it was integrated into NOS3 to allow end-to-end testing of STF-1 and to enable the “test as we fly, fly as we test” philosophy. COSMOS provides a sophisticated framework for command and control of satellites and other embedded systems. COSMOS was
integrated into NOS3 using a collection of text configuration files. A single text file provides the TCP/IP socket configuration information, while additional text files are auto-generated to define the byte patterns representing telemetry and command data sent from the spacecraft to the ground and vice versa.

NOS3 includes several COSMOS enhancements to automatically generate and keep the data descriptions in the embedded code synchronized with the COSMOS command and telemetry files' data descriptions. Data analysis mechanisms, in addition to what is provided with COSMOS, were required for the STF-1 mission and have been built as Ruby language extensions to COSMOS. These extensions are also available in NOS3 and provide some of the post-processing data reduction for STF-1. Despite COSMOS being already integrated into the NOS3 framework, it should be noted that it is not architecturally required and could be replaced by a similar command and control software that supports UDP connection.

4.3.4 Spacecraft Segment: Flight Software: core Flight System (CFS)

The NASA-developed core Flight System [22] is an open-source solution for spacecraft flight software (first discussed in Chapter 2.1.2), with flight heritage on numerous large and small NASA missions such as the Global Precipitation Measurement (GPM) and the Lunar Atmosphere Dust and Environment Explorer (LADEE). The cFS application layer includes a set of reusable software applications to support flight software development. The reusable applications are tailored to the mission requirements using tables, while new applications can also be developed for any mission-specific requirement that is not directly provided by cFS. The software supports table-driven applications, allowing applications to be tuned or changed during development and at runtime by merely changing the tables’ values without changing the codebase. Another cFS component is a set of common services named the Core Flight Executive layer, that is typically needed by satellite systems such as timekeeping and timers, executive services for applications, software bus messaging, and event reporting services. cFS is run on top of a lower level operating system framework called the Operating System Abstraction Layer, which isolates embedded software from the real-time operating system by providing users with an Application Program Interface (API). OSAL libraries are available for a range of operating systems, including Linux, which allows NOS3 libraries to be substituted at build
time without any changes to the other cFS’s layers. The Platform Support Package (PSP) is the cFS component that provides the interface to the hardware drivers for a specific on-board computer. NOS3 is capable of substituting PSP libraries, thus allowing the cFS to use standard function calls for various protocols (e.g., UART, I2C) to effectively communicate with the software simulations. The STF-1 mission and, therefore, NOS3 made use of cFS not just for its flight heritage reliability but also for this ability to substitute libraries that share a common API used by the flight software. It should be noted that it is architecturally possible to use NOS3 without using cFS and OSAL. If cFS is not used, an interface library would need to be written to utilize the NOS Engine API.

4.3.5 Flight Dynamics: “42”

A fundamental consideration in developing a small satellite SoST is how to provide realistic hardware signals reacting to the dynamically changing spacecraft environment. Specifically, as the spacecraft travels, variables such as its position, velocity, orientation, solar radiation direction and intensity, magnetic field direction, and intensity change over time. While the actual hardware signals corresponding to dynamic inputs can be determined from hardware data sheets and user’s manuals, the dynamic inputs must also be identified for a correct simulator development.

To provide a complete framework for spacecraft simulation, including the specific hardware simulations needed for the STF-1 project, a comprehensive analysis was performed of different dynamic environmental data providers within NOS3. After a thorough evaluation of numerous external solutions and the possibility of in-house development options, we chose the “42” software [59] – a general-purpose, multi-body, multi-spacecraft simulation that provides dynamic environmental data. “42” is an open-source software solution that provides the ability to propagate and predict the orbit and orientation of spacecraft by computing the forces affecting these orbital parameters, secondary gravitational effects, aerodynamic drag, solar radiation pressure, magnetic field interaction, and others.

Several simulators have been developed for the hardware components utilized on STF-1, such as the GPS receiver, the antenna deployment system, and the electrical power system. While these simulators have features specific to the hardware components used on STF-1, they also present several elements useful to other satellite developers. For
instance, they provide detailed, practical examples showing how simulators can be written for hardware components, how to use the NOS Engine communication busses, and how to receive dynamic data from “42”. Furthermore, NOS3 supplies a common simulation development framework for adding custom mission simulators; it includes functionalities for logging and text file configuration of simulators, it facilitates integrating custom mission capabilities and integrates environmental data providers such as “42”. The framework also allows the user to create software simulators of a hardware component early in the mission lifecycle to support flight software development and testing. These simulators can be written by referencing hardware interface control documents (ICDs) or datasheets and further augmented with characteristic data from the hardware when available.

4.3.6 NOS3 Hardware Simulator Architecture Design

The three-piece architecture described below is a generic way to design the individual simulators utilized in NOS3. Having a common architecture across NOS3 simulators assists with future simulator integration and maintenance. The architecture of the NOS3 core software components dictates that a simulator only connects to the LibA3200NOS library, which is a C library containing specific board drivers. The NOS3 architecture uses a single NOS Engine Server. This server creates a single bus and provides a transport for each node to connect to. The FSW is represented by a single node and this connection is made by the LibA3200NOS library. Each simulator is then another node on the single bus and is only responsible for receiving messages with data that is the bytes that the flight software would send to the hardware and for sending messages with data that is the bytes that the hardware would send to the FSW.

A simulator can be architected in three pieces. The first piece, the “Simulator” block below in Figure 48, represents the transfer of data to and from NOS engine. The last piece, “Environment Data Provider,” represents the actual physical world that is being simulated and can respond to a request for data at a particular real-world time with data that the sensor would return at that real-world time. The middle piece, “Hardware Model,” represents any hardware-specific processing of the data: the unpacking and parsing of the data from the flight software, the translation of that data into commands and state changes to the hardware, the request to the real physical world simulation at a particular
time and receipt of the response, and the packing of any hardware state or physical world data for the transmission of data to the flight software via NOS Engine.

![Figure 48. NOS3 Simulator Architecture](image)

### 4.3.7 NOS3 and Flight Software Architecture Design

The flight software interacts with the simulators through the NOS3 architecture. This architecture is embedded into the Hardware Library (HWLib), as shown in Figure 49. NOS Engine is utilized to simulate different communications protocols and buses to allow communication between the flight software and the simulated or actual hardware. This middle layer, composed of the HWLib and `libA3200/libGomspace` (shown with the red border), allows for the logging, monitoring, and manipulation of the data, which enables additional tests to be performed on the flight software and the rerouting of the data to the simulators.

![Figure 49. NOS3 Architecture Overview](image)
The HWLib connects all of the flight software CFS applications with the hardware or the simulator depending upon if it is linked with the LibA3200 or LibA3200NOS library. This allows for quick swapping between a simulator environment and a real hardware environment. Also, the simulator environment can be a SoST environment or a hardware-in-the-loop simulator environment. As mentioned in previous sections, the flight software is not changed, allowing the “test as you fly and fly as you test” mantra to remain valid. The library is simply changed to redirect hardware calls to the middleware. This logic can be visualized in Figure 50 Error! Reference source not found.. Additional connectors for other platforms can be written to allow for communication through NOS3.

![Figure 50. FSW Compiler Diagram](image)
4.3.8 Ready-To-Run Deployment and Operations

The NOS3 collection of software components are conveniently packaged as a ready-to-run virtual machine, reducing the overhead associated with installing and configuring each software component. NOS3 can be distributed as an Oracle VirtualBox virtual machine image or as a collection of command scripts that are used to recreate and modify a virtual machine image. As discussed in Chapter 3.5, users have a common development and testing environment, further reducing risk to the mission. The standard guest operating system utilized by NOS3 is Ubuntu Linux, but the virtual machine can run using Oracle VirtualBox on Windows, Mac, or Linux computers.

5 Engineering Roadmap for Scoping, Designing, and Implementing a SoST

To date, the processes for scoping, designing, implementing, and deploying SoSTs are immature and unknown to engineers and their management, which can often lead to misconceptions, confusion, and project failure, as personally witnessed at other organizations by this author. SoSTs do not necessarily follow typical software engineering project lifecycles, and their development can become over budget with a late schedule, resulting in project failures. Also, anecdotal evidence witnessed by this author over the past several years suggests that when budgets shrink and schedules slip, the result is less organizational testing and less organizational support for simulator and emulator development, thus moving back to the traditional paradigm of testing on the “scarce space hardware resources” because it is on the critical path. This research provides a roadmap and implementation tips for both engineers and managers to develop and deploy a SoST successfully. Please note that SoST development is an extremely challenging process that often involves source-level debugging, specific knowledge of hardware systems, and reverse engineering. Having the correctly skilled staff of engineers is extremely important.

The following subsections list commonly encountered pitfalls and technical implementation advice on specific SoST development topics:
5.1 Beginning SoST Design and Implementation

5.1.1 For newly developed custom hardware, do not begin SoST implementation until the flight software has executed on a hardware test bed

For spacecraft with newly developed custom hardware, beginning SoST implementation before a hardware test bed is available is not recommended. It is perfectly acceptable to perform the SoST scoping and design, but the implementation should be deferred until there is at least baseline flight software functionality that has been successfully executed on the hardware test bed in order to help pinpoint modeling inaccuracies. For “product-line” hardware with minimal changes, SoST development can begin early in the lifecycle process.

Implementing the SoST before it has been at least executed once on a test bed is a tricky situation that can significantly hinder SoST development and must be managed carefully. There is a tendency to begin SoST develop and follow the flight software development lifecycle in parallel. On the surface, this would appear to be the correct approach; however, the better approach is to carefully follow the hardware test bed development schedule. Because the SoST models hardware and the flight software needs to execute on hardware, then the SoST development should follow the hardware test bed lifecycle. Hardware documentation often lacks the detail needed to complete the hardware models, and the flight software is required to understand better how the hardware behaves. This approach’s problem is that the flight software may not be mature enough to execute on the hardware, thus leaving the modeling efforts in an ambiguous state. The result is incomplete hardware models that leave the developer to determine if the flight software is the problem or the hardware models are the problem, with usually both being incorrect, thus causing complicated (i.e., time-consuming) issues to resolve.

5.1.2 Treat the Ground Segment system software as a black box for SoST integration

The Ground Software System, regardless of which software is utilized, should be treated mostly as a black box by the SoST designer. The Ground Software System needs to be integrated, most likely by TCP/IP socket communications, but otherwise should be treated as a black box installed in its own virtual machine. Dedicate an engineer at from 50% to 100% of their time to learning and becoming an “expert user” of the ground system.
software. Often, this software is complicated, legacy, and it is not readily apparent what pieces of the software are essential to the SoST operations.

5.1.3 The Modeling Process Itself is a New Type of V&V and Should Be Embraced
Often overlooked, the SoST modeling process itself is a new type of V&V that should be considered by the organization as part of the V&V formalized process. For example, during the modeling development phases, the flight software is executed in the SoST against an incomplete model. The incomplete model also represents a hardware component that is malfunctioning or failing and causes the flight software to execute its fault-logic paths that are seldom (if ever) tested and categorized as flight software Fault Detection and Response functionality. Flight software errors can be uncovered during this process, thus demonstrating the amount of robustness that has been engineered into the flight software.

5.1.4 Determine the Source of the Spacecraft Dynamics Component Early
The Spacecraft Dynamics component is one of the most important and challenging components of a SoST and should be scoped and designed early in the development process. Development of the Dynamics component can take the longest time. Thus, it needs to determine whether the Dynamics component will be developed by the same SoST development team, contracted out to another organization, or provided by another organization for integration. Often in large missions, a Dynamic’s component does exist but needs considerable effort to convert it from a typical hardware-in-the-loop component to a software-only component. This process can involve significant software development, often at lower-level interfaces, in order to properly abstract the hardware interfaces.

5.1.5 Follow the Iterative Development Processes to Reduce Risk
Chapter 3.1 and Chapter 3.2 introduced and discussed the iterative development process to follow for SoSTs, and how it differs from traditional software development processes. Iterative development processes should be utilized for SoSTs, and customers’ expectations need carefully managed, and each iteration’s new functionality and features need carefully communicated. Usually, the SoST development follows (or slightly lags) the flight software development schedule.
5.1.6 Choose the Instruction Set Simulator Carefully and Early

Before beginning SoST design, considerable thought needs to be given to which Instruction Set Simulator to utilize, which often requires a market survey. The obvious starting point is first to choose ISSs that support the target architecture; the next step is to evaluate each ISS while asking the following questions:

- Does the ISS support speed caching or instruction pattern matching? ISSs can often speed up execution by matching instructions to other host instructions to speed up performance. The details are specific to each architecture, but one example is that PowerPC NoOps (no-operational-instructions) can be skipped by the ISS to speed up execution in the synchronous SoST.

- Does the ISS support software integration? While not easily determined, an analysis must be performed to determine if the ISS supports third party software development and integration. Some ISSs simply offer CPU virtualization for executing simple binaries; however, for SoSTs (and their complexity), an ISS that supports the integration of other software components (such as NOS Engine) is a requirement. Evaluate the ISS’s application programmer’s interface (API) or software development kit (SDK).

- The ease of model development of hardware components is crucial for ISS consideration. Some ISSs support model development, while others do not. Similar to the previous bullet, an analysis of the ISS’s capabilities is needed.

5.2 SoST Hardware Modeling Best Practices

5.2.1 Choosing Appropriate Levels of Fidelity – Only Model What You Need

It is important to be able to choose the appropriate level of SoST fidelity and then only model the functionality that is required. While this is an obvious statement, the details of choosing the fidelity and modeling only needed functionality can be challenging to ascertain. Let the flight software and its functionality guide you in the SoST’s iterative design of functionality per release.
5.2.2 Document SoST Model Limitations and Capabilities for Each Release

For each SoST release, plan to utilize an issue tracking system to record the model limitations. Ensuring that model limitations and functionality is documented is extremely helpful for internal scoping and planning while also helping to communicate to users the current capabilities of the SoST. Users need to be aware of any SoST limitations so that they can plan their testing/V&V activities accordingly.

5.2.3 SoST Verification Is Time Consuming and Should Be Planned for Appropriately

SoST development managers need to recognize that the verification activity for each iteration will be extremely time consuming and must be managed carefully to avoid a pitfall of technical churn. The SoST verification process will vary for each specific mission’s SoST, but overall, a detailed plan needs to be generated and followed closely while avoiding activities and problem-solving that could degenerate into compounded issues. Please note that the tests to certify/verify a SoST are NOT simple unit tests with a quickly determined PASS/FAIL criteria; instead, these are considered integration tests by the mission and are often complex functional scenarios that even by themselves can be difficult to configure and execute. Inevitably, results from the mission’s tests and the SoST’s tests will not match, and significant engineering skills are required to determine which platform has the correct results given a direct comparison of two platforms. Original time estimates for this activity should be at least doubled to account for issues that arise.

5.2.4 Avoid Periodic Event Modeling to Significantly Increase Performance

Periodic events are defined as hardware events that occur regularly at a specific period. Often, periodic events are real-time interrupts that are driven by an oscillator or timed events that occur on a schedule. When modeling, the periodicity should not be modeled due to performance; instead, their effects should be modeled based upon their CPU time when visible to the flight software. For example, if a spacecraft time register records elapsed time in seconds and nanoseconds, then do not model the register like its real-world incrementing behavior. The registers should be modeled such that whenever the flight software reads the registers then, and only then are the register values calculated to return the correct values to the flight software. Please note that because SoSTs are synchronous, it is acceptable to perform calculations and other operations when registers
are read. There is no issue with the flight software having to “wait” for the register to be ready-for-reading. This on-demand modeling technique considerably increases SoST performance and helps it execute faster than real-time.

5.2.5 Considerations for Simulation Speed and Operating Faster Than Real-Time

To help the SoST execute as fast as possible, including faster than real-time, the models should react immediately to events and data and ignore the real-world time it would take to fulfill the transaction. For example, MIL-STD-1553 transactions can take 10 microseconds to 16 microseconds to complete in the real world; instead of having the model wait this exact time, batch complete several MIL-STD-1553 transactions. This technique helps to ensure that the models execute faster than real-time. Occasionally, the flight software requires some delay as it is not designed to operate in “instantaneous time”. When this occurs, insert the minimum delay necessary into the models.

6 SoST Verification Analysis, Problem Statement Review, and Benefits Realized

6.1 Methods of SoST Verification and Analysis

“How do you verify your models?” is the most commonly asked question this author receives at all SoST presentations, demonstrations, and user meetings. The question is so common that a presentation slide is always prepared in advance with the explanation and results provided to the audience. The explanation and results have several aspects to them, which are described in detail below, sans the PowerPoint slides. The verification methods are similar between Flagship mission SoSTs and SmallSat SoSTs. There are several minor differences in approaches, and their nuances are explained. Specific mission SoST verification approaches, analysis, and results are presented.

The goal of SoST verification is to ensure that the developed models accurately represent their hardware counterparts and that their integration and timing considerations also accurately allow the flight software to execute in a flight-like environment.
### 6.1.1 Method 1: Utilization of Hardware Test Procedures as Verification Methods

This primary verification method serves as the “Zemerick Gold Rule” for SoST verification [60]: *Execute all original hardware test procedures and obtain the same results as the physical system.* The only exception to this rule is that settling time for signals may need to be extended if the SoST is executing slower than real-time. An example is provided below.

This verification method utilizes the missions’ test and Run-For-Record procedures executed on the hardware test beds to verify the flight software. The SoST verification approach is to execute the same tests on the SoST and then analyze the results to verify that the SoST behavior matches the hardware test bed behavior. The exact flight software version is utilized for both sets of tests to ensure a common baseline of behavior and allow for a direct comparison.

Each SoST has its own unique set of hardware models because each mission has its own set of unique hardware. For this reason, there is no single set of SoST verification acceptance tests; instead, each SoST must be verified using unique mission test procedures that represent a significant mission testing milestone, such as Run-For-Record. Also noteworthy, these test procedures are often very complex and require hours and sometimes days to execute, resulting in significant labor needed for SoST verification, especially when test results show differences. Integration of the Ground Segment/Software System (as previously discussed in Chapter 2.1) is critical for SoST verification due to the re-use and “borrowing” of the mission’s hardware tests. These tests are designed to run in one single way, using a very specific ground station software configuration. Since the SoST includes the Ground Segment, the tests are ready-to-run and do not need to be rewritten for another configuration, platform, or architecture.

Exhaustive lists of specific SoST test cases are not necessary to detail here because each mission is different; instead, one test case with various for the specific JIST SoST is described to provide the reader with the process and context needed to understand SoST verification.

Please note that the Predicted and Raw FlatSat values from the following comparison plots were obtained from an integration lab similar to that shown in
Figure 51. The integration lab (i.e., FlatSat) contains flight hardware that is being tested for final integration.

![Image of JWST undergoing testing and integration.](image)

Image Courtesy Northrop Grumman

A specific Attitude Control System (ACS) test (named ACS10) is a crucial pointing-requirements test for the JWST observatory. The test's goal is to ensure that the JWST observatory can perform a slew maneuver (large) and issue a fine guidance (small pointing) maneuver, all while ensuring that the solar panels and instruments are always pointed away from the sun. ACS10 is a complex test that requires approximately sixty minutes to execute and requires the JWST ground station software to execute. ACS10 is a long series of scripts that contain commands, waits, telemetry checks, and flight software upload actions. Figure 52 shows an experimental visualization of the spacecraft that enables a SoST user to visualize the spacecraft’s maneuvers as the ACS10 test...

![Figure 52. Visualization of the JWST Spacecraft During Verification Tests](image)
case executes. A user can visually watch the spacecraft for jitter, pointing direction, and maneuver speeds; even though visual verification cannot be performed alone without data analysis, it gives the user important information about the spacecraft state in real-time, remarkably similar to the actual spacecraft operations.

The order of events for running ACS10 on the JIST SoST is to:

1) Verify that the correct flight software-under-test is loaded onto the JIST SoST, then
2) run JIST such that the flight software and spacecraft is in a nominal state
3) Load the test files/scripts to the ground station software and
4) Begin the test execution.

The ACS10 test “believes” that it operates in a hardware test lab with a FlatSat configuration when in reality, it is executing in a completely software-only environment. Once a test is finished executing, various telemetry points are compared between the SoST test run and the Hardware Test Bed run. Many telemetry points are usually compared in various ways, such as importing data into Microsoft Excel for data analysis and comparison.

Figures 53 and 54 (below) show one such comparison for the Gyro A/B components of the Inertial Rate Unit (IRU) for the JWST spacecraft. The IRU is critical for JWST observatory pointing, and it must perform accurately in a closed-loop control scheme. The left side of Figure 53 and Figure 54 show the raw value response of the gyro, while the graphs on the right show the JIST-generated raw value response of the gyro. The responses are a match indicating that the JIST gyro behavior matches the behavior of the Hardware Test Bed during ACS10 execution. Upon closer inspection, the JIST plot indicates that the settling of the gyro took longer (referring to the x-axis), even though the response behavior was identical. This was due to the JIST instruction-set-simulator running slightly slower than real-time, whereas the ground software system runs the test and logs data in real-wall-clock-time.
Figure 53. IRU Rate B Component Comparison Between FlatSat and JIST

Figure 54. IRU Rate A Component Comparison Between FlatSat and JIST
Figure 55 is a different plot showing a comparison between the FlatSat (left plot) and JIST (right plot) of the Fine Sensor 1 Y-Coordinates. Notice that the response for the Sun Sensor matches between the FlatSat and JIST with the exception that the JIST graph finishes sooner at approximately 2700 seconds while the FlatSat graph finishes at 3000+ seconds. This difference results from JIST running faster than real-time and thus able to complete the test in faster wall-clock time; this is a unique characteristic of SoSTs.

![Figure 55. Sun Sensor 1 Y Component FlatSat and JIST Comparison](image)

Figure 56 (below) illustrates the results of a test showing the comparison of the sun sensor pitch angle response with the JIST results shown on the right. Also, note that JIST completes approximately 250 seconds sooner, thus indicating that JIST is running faster than real-time, which is a significant advantage for SoSTs.

pg. 100
Inevitably, the tests will not match at some point during the process, and a detailed analysis will need to be performed to determine the root cause of the problem. When this occurs, there is no silver-bullet-quick solution; a detailed engineering analysis needs to be performed that will touch aspects of the spacecraft control engineering, test engineering, flight software, and the hardware models. It is often beneficial to create a small team to collaborate on the issue, as various engineering skills will be required. The chances are good that the SoST models are incorrect or different from the hardware components, but identifying the differences, which are often related to timing, can be challenging.
6.1.2 Method 2: Utilization of Hardware Checkout Tests as Verification Methods

This Method #2 of verifying a SoST is more subtle and often overlooked, because missions have separate teams that develop the flight software and the flight hardware. The two teams naturally have integration points, but in general, the hardware team develops the functionality, and the flight software team is responsible for using the hardware correctly. However, the hardware team will often have simple checkout software tests, sometimes referred to as Built in Self Tests (BISTs), that execute functional unit tests against the hardware to ensure functionality is operational and that logic in the FPGAs is implemented correctly. For example, as shown in Figure 57, most custom space hardware has FPGAs with well-defined behavior for its registers, DMA, etc. These checkout software tests provide the inputs to the FPGAs and ensure that the logic output is expected compared with the hardware FPGA specification. These tests can become valuable verification tools for SoSTs because they can execute on the flight computer model against the SoST hardware models. If the tests do not pass or the test results do not match the hardware test results, then there are apparent model discrepancies that need to be resolved. This Method #2 can be a more efficient verification method than #1, because the test software in Method #2 is more straightforward with only input/outputs being executed and tested. This is analogous to unit testing in standard software engineering practices.

This method was utilized on JIST for the instrument RAD750 flight computer and a custom JWST cPCI card, but the tests and results cannot be included here. The tests uncovered several minor hardware modeling bugs that were resolved quickly in the custom cPCI card model.

6.1.3 Method 3: SmallSat SoST Model Verification Examples for NOS3

SmallSat SoST verification methods can be similar to Method #1 and Method #2 previously discussed. However, since SmallSats are inherently different mission-types
with different development processes, lower-cost and less-custom (more COTS) hardware, and shorter schedules, SoST verification can sometimes be more targeted with an immediate focus on the hardware models and simulators and a direct comparison with the actual hardware.

For all the models listed below, the SmallSat hardware itself was utilized for verification. In most cases, the simulator was developed from documentation well in advance of the hardware delivery. Once the hardware arrived, the models’ behavior was compared with the model to ensure accuracy.

In Table 9 are hardware models, the specific NOS3 simulator name, the verification method utilized, and relevant results.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>NOS3 Simulator</th>
<th>Verification Method</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOTON GPS</td>
<td>GPS Sim</td>
<td>Serial data dumps recorded from the FOTON hardware were read through the CFS application for GPS to ensure proper decoding and data integrity. The results from the hardware and simulation were compared to validate the GPSSim.</td>
<td>Sim matches FOTON GPS. No issue swapping between simulator and hardware.</td>
</tr>
<tr>
<td>Honeywell HMC5843 Magnetometer</td>
<td>MAG Sim</td>
<td>Because 42 is the source of the Earth’s magnetic field model for NOS3’s MagSim, validation of this sim is basically a validation of the model used in 42. The plan is to validate this simulator by obtaining on-orbit magnetic field data and comparing it with the MagSim data. A rough comparison will be possible by comparing the hardware readings with a cell phone magnetometer and then comparing that with 42’s model. This will also validate the translation of the data from values in nanoteslas to the format and units that FSW expects.</td>
<td>Still in progress. A more detailed analysis of on-orbit magnetometer data needs to be performed.</td>
</tr>
<tr>
<td>Clydespace Gen III Electrical Power System</td>
<td>EPS Sim</td>
<td>The EPS hardware was used for telemetry, and switch commanding, and its behavior will be compared with the EPS Sim, which was developed well in advance of the hardware delivery.</td>
<td>EPS Sim allowed flight software development and testing to remain on schedule even while the EPS hardware was continually delayed.</td>
</tr>
<tr>
<td>Arducam OV2640 Camera</td>
<td>CAM Sim</td>
<td>The CAM Sim was updated to provide similar performance (namely picture acquisition time) and similar image size as to the actual OV2640 hardware.</td>
<td>CAM Sim was utilized when few cameras were available. The CAM Sim was utilized for flight software testing and development, specifically in support of performance testing.</td>
</tr>
<tr>
<td>Hardware</td>
<td>NOS3 Simulator</td>
<td>Verification Method</td>
<td>Results</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>InvenSense MPU-3200 3-Axis Gyroscope</td>
<td>Gyro Sim</td>
<td>Similar to the magnetometer, the output ranges of the gyro can be compared with the output ranges of the simulator. Full-scale ranges were verified and compared.</td>
<td>Still in progress. A more detailed analysis of on-orbit gyro data needs to be performed.</td>
</tr>
<tr>
<td>Cadet Radio</td>
<td>Cadet Sim</td>
<td>Testing with the actual Cadet radio was needed to better understand its behavior in order to properly model the Cadet radio.</td>
<td>Cadet Sim behavior was matched with the actual radio. CadetSim was extremely valuable because the radio is a “scarce resource.”</td>
</tr>
<tr>
<td>Temperature Sensors</td>
<td>Temp Sim</td>
<td>Temperature simulators were validated by ensuring that the simulator provided the correct modeled resolution, accuracy, and scale ranges as the temperature sensors used on STF-1.</td>
<td>Simple simulator was verified with the actual temperature sensors.</td>
</tr>
<tr>
<td>Sun Sensors</td>
<td>Sun Sim</td>
<td>A sun sensor model was validated by testing the characteristics of hardware light sensors and then making appropriate sensor model modifications.</td>
<td>Simple simulator was verified with the actual light sensors.</td>
</tr>
</tbody>
</table>

6.2 Problem Statement Review and Benefits Realized

This chapter aims to review the Problem Statements and provide additional technical and nontechnical supporting evidence as to the resolution of the problem statements by utilizing SoSTs. This chapter focuses on the benefits of developing, deploying and utilizing SoST for spacecraft and SmallSat missions. This chapter does not redefine the Problem Statements; for information on the Problem Statements refer to Chapter 1.

6.2.1 Problem Statement 1: Limited Hardware Test Bed Resources Due to Custom Hardware

Results in a Lack of V&V

The JIST SoST is the software-only equivalent of a considerable amount of hardware in a physical test bed laboratory. Each JWST test bed costs at least $1,000,000 (see Table 10 below for component prices excluding labor) and is in limited quantity and impossible for smaller organizations to procure. This limitation of hardware causes extremely tight V&V schedules due to laboratory overutilization. For a cost comparison, each SoST deployment costs approximately $10,400 in software licensing costs (excluding labor development SoST costs), making SoSTs extremely affordable and deployable. For each hardware component in the table, the JIST SoST must emulate and abstract the hardware by utilizing models. Also, note that this hardware includes custom cPCI cards that require significant lead times and are manufactured in small quantities.
Table 10. SoST Compared with Hardware Equivalent Test Bed

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Unit Cost</th>
<th>Total Cost</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>6U RAD750 SBC with SpaceWire and 1553 (Prototype Board)</td>
<td>4</td>
<td>$193,000</td>
<td>$772,000</td>
<td>BAE Quote 12-049</td>
</tr>
<tr>
<td>JWST Peripheral Interface Module (JPIM) Card</td>
<td>2</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Contractor Proprietary Card</td>
</tr>
<tr>
<td>Custom cPCI Cards (BIC, FPAPs, HK, CMM-S, CMM-Ka, SIM)</td>
<td>12</td>
<td>$7,500</td>
<td>$90,000.00</td>
<td>Estimated average cost of custom cPCI cards</td>
</tr>
<tr>
<td>VDS Hardware (dSPACE)</td>
<td>1</td>
<td>$50,856</td>
<td>$50,826.00</td>
<td>dSPACE Quote: QUI-0938105</td>
</tr>
<tr>
<td>VDS Software (dSpace)</td>
<td>1</td>
<td>$32,880</td>
<td>$32,880.00</td>
<td>dSPACE Quote: QUI-0938104</td>
</tr>
<tr>
<td>TCTS and MTTS Support Hardware</td>
<td>1</td>
<td>$40,000</td>
<td>$40,000.00</td>
<td>Two hardware racks, including RTLogic T501 Front End Processor Cost, etc.</td>
</tr>
<tr>
<td>SpaceWire Test Set (SWTS)</td>
<td>1</td>
<td>$17,639.00</td>
<td>$17,639.00</td>
<td>Microtel LLC: 071211-1</td>
</tr>
<tr>
<td>SpaceWire Cables</td>
<td>2</td>
<td>$535.00</td>
<td>$1,070.00</td>
<td>Microtel LLC: 071211-1</td>
</tr>
<tr>
<td>MIL-STD-1553 Cables</td>
<td>4</td>
<td>$58.20</td>
<td>$232.80</td>
<td>MilesTek 1553 Cables</td>
</tr>
<tr>
<td>cPCI Chassis</td>
<td>4</td>
<td>$3,610.00</td>
<td>$14,440.00</td>
<td>Tracewell Quote 012131-00</td>
</tr>
<tr>
<td><strong>JIST HARDWARE-EQUIVALENT COST</strong></td>
<td></td>
<td></td>
<td><strong>$1,019,087.80</strong></td>
<td></td>
</tr>
</tbody>
</table>

6.2.2 Problem Statement 2: Limited Fault Injection Capabilities for Thorough Flight Software Testing

SoSTs have enormous fault injection capabilities, particularly in data bus transfers where values can be overwritten dynamically during execution. For missions with limited hardware test beds, the ability to deploy SoSTs directly increases the amount of V&V and fault injection testing that can be performed.

As shown in Figure 58, for JIST, the JWST mission had four hardware test

![JIST Increases Test Resources](image)

*Figure 58. JIST Increases Test Resources*
beds, but deploying JIST in strategic areas increased the deployments to sixteen, thus resulting in a significant increase in testing environments and fault-injection V&V opportunities. Similarly, as shown in Figure 59, this had a significant increase in the number of testers, increasing from twenty-four users to ninety-six users once JIST was deployed. More test resources and more testers directly benefit and increase the mission’s software assurance.

6.2.3 Problem Statement 3: Limited / Nonexistent Flight Software V&V Methods for SmallSats and CubeSats

- Results: NOS3 SoST Impact on STF-1 Flight Software Development

As a metric to assess the overall software complexity, the Source-Lines-of-Code (SLOC) utility (SLOCCount) was executed against the source code. SLOCCount estimates that the STF-1 applications take 8.25 person-months for development. However, this metric does not consider unit testing, integration testing, and access to flight hardware for testing, which are typically the bottlenecks for small satellites and space missions.

Operationally, the STF-1 flight software is not trivial due to its semi-autonomous on-orbit functionalities that are needed to perform science experiments, record science data, and transmit the data to the ground station during downlink periods of just a few minutes long. The flight software must be able to simultaneously provide the following core functionalities: 1) operate without interaction/commanding from the ground station; 2) it must be aware of its power level status for executing time-lapse science experiments; 3) it must start, stop, and pause experiments; 4) it is responsible for communicating with various STF-1 hardware components such as sensors, radio, camera, and the deployable
antenna. This flight software complexity results in increased mission-risk with respect to
development and testing schedule.

Table 11. STF-1 Flight Software SLOC Counts of the Newly Developed Software.

<table>
<thead>
<tr>
<th>Software Component</th>
<th>Description</th>
<th>Source Lines of Code (SLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Flight System (CFS) + Platform Support Package (PSP)</td>
<td>GSFC reusable flight software framework</td>
<td>50 K + 7 K</td>
</tr>
<tr>
<td>Operating System Abstraction Layer (OSAL)</td>
<td>GSFC reusable operating system abstraction layer API</td>
<td>41K</td>
</tr>
<tr>
<td>STF-1 Mission Specific Applications</td>
<td>Newly developed flight software</td>
<td>34K</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td><strong>132K</strong></td>
</tr>
</tbody>
</table>

- **Reduced Hardware Reliance**

The NOS3 SoST enabled multiple STF-1 developers to work in parallel without monopolizing either a single simulator, engineering test unit, or spacecraft flight computer, thus reducing the STF-1 mission’s reliance on hardware. For example, while one engineer was developing the electrical power system software, another engineer developed the communications software. Neither engineer needed to utilize the hardware for their development and initial testing.

NOS3 was utilized extensively by the STF-1 software development team for all aspects of flight software development and testing. Over the course of three person-months in which most of STF-1 software development was accomplished, each team member maintained their own NOS3 virtual environment. The virtual environment provided realistic inputs and feedback to the flight software while under development.

Additionally, NOS3 provided a suitable test environment to support STF-1 flight software integration testing. Similar to many other small satellite missions, the STF-1 mission hardware was expensive, limited in supply with few spares, and it needed to be configured and setup quickly to support testing. NOS3 provided the ability to develop and test most flight software functionality without requiring a hardware-in-the-loop test configuration. Hardware is still needed to test specific performance and timing requirements. Without NOS3, STF-1 developers would not have been able to develop and test software.
applications in parallel to these activities. As a result, it would have been challenging to maintain the flight software development and test schedule.

- **Reduced Risk and Provided a Living Training Package**

  The effortless deployment process of the NOS3 software allowed us to set up and configure a large number of medium-fidelity simulation environments to cross-train personnel and support risk reduction testing during the STF-1 software development. For example, NOS3 was provided to multiple interns during the summer months to support mission understanding, static analyses, and additional software testing of custom STF-1 software applications. The additional simulation resources allowed the team to test how the various STF-1 software applications would respond to adverse conditions, thus ensuring STF-1 software robustness. One of the most critical STF-1 software applications, the *Manager Application*, which is responsible for semi-automating the spacecraft operations, was exhaustively tested using NOS3. NOS3 also allows the tester to introduce fault conditions that are too dangerous or expensive to test using hardware, which further reduced mission risk and raised confidence in the flight software.

- **Improved the Software Development Schedule**

  NOS3 was able to increase the STF-1 development team’s control of the software development schedule and demonstrate how future software development effort schedules can be shifted ahead of hardware components’ receipt. Table 12 reports the lead times associated with the major STF-1 flight components as compared with the associated development time for the NOS3 hardware simulator. It is evident that the level of effort required to develop a hardware-equivalent simulator for the STF-1 mission with NOS3 was rather minimal. Furthermore, a NOS3 hardware simulator can be scoped, planned (effort, simulator fidelity, etc.), and efforted, whereas hardware lead times from vendors change and slip regularly. NOS3 allowed STF-1 software development to begin as scheduled versus when the hardware arrived.
Table 12. STF-1 component lead time compared to NOS3 software simulator development time. By reducing lead time, flight software development can start earlier in the mission.

<table>
<thead>
<tr>
<th>Hardware Component</th>
<th>STF-1 Hardware Procurement Lead Time</th>
<th>NOS3 Simulation Development Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antenna Deployment System</td>
<td>6 Months</td>
<td>2 Weeks</td>
</tr>
<tr>
<td>Electrical Power System (EPS)</td>
<td>10 Months</td>
<td>3 Weeks</td>
</tr>
<tr>
<td>GPS Receiver</td>
<td>2 Weeks</td>
<td>2 Weeks</td>
</tr>
<tr>
<td>Magnetometer</td>
<td>6 Months</td>
<td>1 Week</td>
</tr>
<tr>
<td>UHF Radio</td>
<td>7 Months</td>
<td>1 Month</td>
</tr>
<tr>
<td>Experimental Payload</td>
<td>12+ Months</td>
<td>1 Week</td>
</tr>
</tbody>
</table>

- SoST Application to CubeSat/SmallSat Problem Statements

The primary purpose of the STF-1 CubeSat mission was to develop a software-only simulation framework and supporting tools that would support STF-1 as well as support future small satellite missions. The resulting byproduct of STF-1 is an open-source software environment named NOS3. The NOS3 architecture was designed to be flexible and allow multiple configuration and deployment options. NOS3 conveniently packages a set of open-source tools (cFS, COSMOS, and “42” dynamics simulator) and a set of STF-1 specific hardware simulators to provide a virtual spacecraft environment that is easy to configure and deploy to end-users. NOS3 has demonstrated its extreme value to the STF-1 mission by reducing hardware reliance, increasing available test resources, serving as a training and risk reduction platform, enabling parallel software development activities that shorten cycles, reducing developer costs, and alleviating schedule pressures due to slips in hardware component deliveries.

6.2.4 Problem Statement #4: COVID-19 Situation Limits Hardware Testing Causing Schedule Delays

Although somewhat difficult to quantify at this time, the COVID-19 pandemic (ongoing) has caused significant schedule delays across many organizations, especially those requiring hands-on laboratory integration efforts. In many cases, SoSTs are the only available option for engineers for flight software development, V&V and testing, and deployment. The COVID-19 pandemic has increased the demand for SoSTs due to their flexibility and availability. COVID-19 has provided the opportunity to demonstrate that
SoSTs are an essential tool that can be effectively and efficiently deployed alongside hardware FlatSat test beds.

6.2.5 Problem Statement #5: Limited Testing Venues of Embedded Non-X86 Flight Software

As shown in Figure 60, utilization of the JIST SoST unexpectedly increased significantly outside of the NASA IV&V Program to other venues. For example, 44% of JIST utilization was for JWST Mission Operations, and 33% of the usage was from the JWST Spacecraft Mission contractor-developer. The Mission Operations and Spacecraft Developer utilization was not expected; utilization by JIST expanded to other use-cases once organizations realized the benefits of the JIST SoST for developer, mission operations, and also cybersecurity.

6.2.6 Problem Statement #6: Lack of Virtualization Technologies

The SoST collection of software components is conveniently packaged as an automated ready-to-run virtual machine, reducing the overhead associated with installing and configuring each software component. Each SoST can be distributed as an Oracle VirtualBox virtual machine image or as a collection of command scripts that are used to recreate and modify a virtual machine image. This allows users to have a common development and testing environment, further reducing risk to the mission. Figure 61 compares the JIST deployment time with that of its equivalent hardware.

![Figure 60. JIST Utilization by Organization](image1)

![Figure 61. JIST Deployment Time Compared with Hardware Deployment Time](image2)
6.2.7 Problem Statement #7: Flight Hardware Driver and Software Interfaces are Tightly Coupled

Having highly coupled software modules is a well-documented software engineering criticism. Low coupling and high cohesion between software modules are always desired. However, the layer between the flight software (application layer) and its hardware drivers are often tightly coupled. The reason for this is that the flight software must at some point in its design, interact with the real-world, and that means directly controlling hardware or receiving its telemetry over a standard bus interface. SoSTs address this problem in their design in two ways:

1. A hybrid architecture of software-only and hardware-in-the-loop can be utilized to help break the coupling, ensuring a more flexible flight software build system. Hybrid SoSTs that include hardware-in-the-loop are discussed more in Chapter 7.2.1.

2. Advanced SoSTs address this problem now by designing a software module known as a Mock Hardware Interface (MHI) (author-invented definition) that sits between the flight software (application layer) and the driver interface. The MHI is responsible for duplicating the hardware driver interface in its design but also being able to interface to other SoST components. For this to operate correctly, the MHI must be aware of the SoSTs virtual time synchronization.

6.3 Direct Benefits: Metrics for Flight Software Issues Identified During SoST Dynamic Analysis

The IV&V Program maintains a database of flight software issues identified, their severity, and their disposition once reported to the mission’s flight software development team. To date, eighty-one (81) flight software issues have been identified across twelve (12) NASA projects. Of those 81 issues, 5 are considered a Severity Level 1, indicating that if they were to occur, they would result in loss-of-mission. Please note that these flight software issues are not simple bugs; SoSTs' usage is designed to find the “big problems.” To provide context below is explanation text from two Severity Level 1 issues:
• Violation of Red Safety Limit if IMU Outage Occurs During Autonomous Momentum Dump
• Software Crash During Flight Phases Does Not Issue a Critical Software Abort

The primary goal of SoSTs is to enable advanced V&V activities so that flight software issues and defects can be identified earlier in the software development lifecycle and to catch the mission-ending flight software issues. Below are descriptions of actual high-severity flight software issues identified by utilizing an SoST. The list below does not include all issues identified; instead, the list describes the highest severity issues that if they were to occur, would result in possible mission loss. Details are changed/removed because of both mission sensitivities and ITAR/export control considerations.

6.3.1 Defect Identified 1: Mission-Ending Issue That Escaped Initial Flight Software Verification Tests

During SoST development for a specific mission it was discovered that a software data validity flag for a specific hardware component was set incorrectly in software to “INVALID” per documentation. However, the flight software was processing the data as if it were “VALID” and would continue utilizing the data from the hardware component, not realizing that the data was “NOT VALID.” The problem was traced to a mission simulator used for the flight software requirements, design, source code development, and tests instead of using the documentation as the correct reference. Had this issue not been discovered before launch, it would have resulted in a mission-ending software fault.

6.3.2 Defect Identified 2: Issues Identified in the Board Support Package (BSP) Hardware Driver Software Due to Simulated Hardware Failures

During SoST development for a different mission, several significant issues were identified in the board support package (BSP), which contains the hardware driver software. All of the issues were related to interrupt and timing and could only be exposed in a software-only environment where fault-injection can be performed, and timing is analyzed in lock-step detail. The issues could not be tested on actual hardware due to the various faults needed to be injected into the hardware, which is not feasible. The root causes of the issues were centered around invalid software states that would occur if hardware failures were to happen.
BSPs and hardware drivers represent pieces of software that are the most difficult to test. Often, BSPs are considered vendor-supplied (e.g., COTS) and do not undergo extensive verification and validation. SoSTs are the perfect tool to helping ensure that BSPs and hardware drivers will perform correctly after the launch of a spacecraft.

### 6.3.3 Defect Identified 3: Radio Configuration Not Set Correctly Due to Invalid Register Writes

On a different mission than those described in Defect #1 and Defect #2, it was discovered during SoST execution that flight software was incorrectly writing to a register field that controls the radio configuration. The cause of this issue was due to a flight software error resulting in the wrong bitmask being utilized in the source code. The radio configuration was not being set correctly. This issue would have gone unnoticed until much later in the software development and testing lifecycle and probably would not have been detected until hardware-in-the-loop testing occurred with an actual radio performing radio frequency (RF) testing in a hardware laboratory.

### 6.3.4 Defect Identified 4: Safe Mode Needs More Functionality After Fatal Fault

During V&V testing of a mission utilizing a SoST, it was found that a flight software error was present in the Safe Mode software configuration. Once identified, the Safe Mode source code was re-worked so that if a Fatal Fault would occur and the spacecraft transitioned to Safe Mode, then the identified issue would not occur.

### 7 Conclusions and Future Work Areas

#### 7.1 Conclusions

This dissertation covers many aspects of software-only simulation, spacecraft, SmallSats, and focuses on the design, development, and operation of Software-only Simulation Test Beds (SoSTs). SoSTs require significant systems engineering knowledge that encompasses spacecraft hardware, embedded software, attitude control, virtualization, and modeling/simulation. The broad engineering knowledge required is a significant challenge for successfully implementing a SoST and has required years of research and development to get the established maturity witnessed today. This dissertation is the culmination of years of advanced research, development, and engineering across many NASA missions and several SmallSat missions.
The Conclusion Statements below are meant to summarize this dissertation by highlighting the major contributions and accomplishments to the field. The entire SoST invention, including its processes and design, is unique and novel; however, the conclusion statements focus on the SoST technical accomplishments critical to SoST successful development.

7.1.1 Conclusion/Contribution Statement 1
SoSTs complemented hardware V&V test laboratories by providing more resources and enabling fault injection testing. During the COVID-19 pandemic, hardware test laboratories have been closed, and SoST usage has increased due to their ease of deployment.

7.1.2 Conclusion/Contribution Statement 2
SoSTs have provided measurable assurance for both NASA and SmallSat missions, resulting in several Severity 1 "Mission Ending" issues being identified and resolved.

7.1.3 Conclusion/Contribution Statement 3
SoSTs can be applied to SmallSats to increase their mission’s assurance. SoSTs were typically applied to large NASA missions, but their application to SmallSats is similar and proven and assists with the entire SmallSat flight software development and test lifecycles.

7.1.4 Conclusion/Contribution Statement 4
Time synchronization of modeled hardware components is critical and is accomplished by utilizing the invented NOS Engine middleware. Synchronous operations allow for all models to be paused simultaneously during any CPU transaction, allowing for powerful V&V analysis.

7.2 Future Work Areas
This research has seen the birth and maturation of SoSTs for both large spacecraft missions and SmallSat missions. While the SoST invention has had significant impacts on the NASA IV&V’s processes and results, including the NASA agency in general, there is additional future work that needs to be researched and implemented to push state of
the art forward another leap. The following subsections describe the next leap in SoST capabilities.

7.2.1 Hybrid SoST Architecture with Hardware-In-The-Loop Capabilities

The subject of this dissertation is **Software-only** Simulation Test Bed (SoST), with emphasis on the “software-only” where hardware models are developed, and hardware systems are abstracted so as to provide a software-only digital representation. While there are many advantages to an SoST (most of which have been noted in this dissertation), there are specific use-cases when a hybrid SoST and hardware-in-the-loop configuration is required. Over several years this author has been asked about such configurations on occasion, and the answer is always related to timing design and implementation between the software-only components and the hardware-in-the-loop components. For example, the software-only components execute in a synchronous virtual time, whereas the hardware-in-the-loop components execute in a real-world wall-clock time context. A synchronization method must be provided between the two, which can become extremely tricky, requiring significantly more research, design, and implementation efforts.

The NOS3 Chapter 4 alludes to a hybrid hardware/software in the loop configuration but does not dive into the detail. NOS3 provides the most straightforward path toward a software/hardware hybrid configuration due mainly to NOS3 not always needing an instruction set simulator. The usage of an instruction set simulator requires significant development for timing synchronization. Without an instruction set simulator several configuration options are possible and need to be investigated further. These configurations are described briefly below.

The hybrid hardware/software architecture could demonstrate a closed-loop system encompassing a flight software controller commanding a simulated actuator and receiving feedback from the affected simulated sensors.

- Possible Configuration 1: Linux Flight Software and Real Interfaces/Instruments

This hybrid configuration is ideal for the flight software development related to interacting with hardware. Since access to the on-board flight computer is limited, the ability to
validate software design and implementation using a portable Linux flight software is advantageous. This would allow teams to develop software in parallel without coordinating access to the flight computer.

One possible configuration involves connecting the flight software on Linux to the real instruments using IO adapter cards, as shown in Figure 62. In this configuration, a mix of simulated instruments and real instruments can be combined to provide a complete simulation. The simulated instruments interface with the flight software using NOS Engine is done in the typical SoST. A time synchronization (TimeSync) application within the flight software receives a command from the scheduler app at 1 Hz. When the TimeSync app receives the command, it sends out a time value using NOS Engine Time Distribution, which is bridged through the STF-1 HWLIB and LIBA3200NOS libraries. This will enable all software simulations to run at pace with the flight software, which takes care of the time synchronization.

To interface the real instruments, IO hardware adapters are needed to bridge the interfaces to the computer (likely over USB). Additional NOS Engine client side functionality will need to be developed to enable the various IO adapters to be used. There will be no time synchronization between the flight software and the real instruments; each will run open-loop at normal speed. This is not considered a problem, as the goal of testing in this configuration is to verify the hardware interface code, not the ‘day-in-the-life’ operational logic.

This design provides flexibility. By using NOS Engine as a common interface in either situation, interception capabilities are available and compile-time flags can be kept minimal. If no software simulations are present, the TimeSync application can be removed from the CFE startup script and the command message removed from the scheduler table, disabling the app without the need to recompile the entire FSW.
Another possible configuration is shown in Figure 63. This is similar to the previous figure, except that the GPS hardware has been replaced with a second UART adapter, which connects to another computer hosting the GPS Sim. This would allow testing of the performance of the hardware interfaces without the real instrument. This would also provide the ability to synchronize the GPS Sim data with the flight software clock. This setup can be used to identify any potential issues related to the timing or performance of the IO busses. The caveat is that the interfaces are not identical to the actual flight computer, so the accuracy of the results may be questionable.
• Possible Configuration #2: Target FSW and Simulated Instruments

This hybrid simulation is suited for verifying the flight software in its unmodified, flight-ready configuration. The use case for this simulation is to ensure the FSW responds to different stimuli as expected. Figure 64 shows the configuration.

In this setup, the unmodified flight software runs on the NanoMind flight computer – the LIBA3200 library is now used, providing access to the built-in IO interfaces. The instrument simulations connect to the NanoMind board using IO cards. NOS Engine, with the additional IO-card functionality, is used as the software interface for the simulations. This retains the interception capability and reduces code refactoring within the Sim applications. The devices (flight computer and instrument simulations) can run open loop based on their own internal clocks.
7.2.2 Better Integration of SoST Development with Mission Development Lifecycles

Developing a V&V environment (e.g., hardware test beds, SoSTs, etc.) is currently considered a separate and later activity from designing the spacecraft’s computing/avionics platform and hardware architecture. Typically, V&V environments consist of additional hardware, system simulations, and possibly cross-compiling the flight software for a different hardware target. These activities add additional time and cost to the project and may not be completely representative of the spacecraft’s actual architecture, thus breaking the test-as-you-fly mantra. For example, it is cost prohibitive for V&V teams to utilize actual flight versions of the RAD750s; instead, they often rely on the much cheaper Freescale MPC750 which is a similar PowerPC, but not the same flight architecture as the RAD750. This end-of-life CPU provides a method to test software, but cross-compiling is often necessary because the RAD750s and MPC750s are not
identical hardware platforms. It would be extremely beneficial to the spacecraft project if both the developer, testing, and integration teams would utilize the same hardware and SoSTs so that the “test-as-you-fly” and “fly-as-you-test” principles could be upheld.

7.2.3 Spacecraft Computing Architecture with Accompanying SoST

The goal of this task is to identify the computing components that would facilitate an architecture that would produce both a hardware design as well as an SoST without any additional effort. This is a novel concept in that the flight hardware architecture is driven by both mission requirements and SoST V&V environment requirements. Having these concepts drive the hardware flight architecture, and flight computer hardware would significantly improve the V&V and mission assurance capabilities.

For example, an investigation into this advanced topic area could begin with experimentations on a Xilinx Virtex-5 FPGA development kit. The Virtex-5 is a radiation-tolerant FPGA that supports significant software-core configuration. This particular development kit is popular for implementing the LEON processor SPARC v8 architecture [61], which is an architecture also popular for SoSTs. Research needs performed on specific spacecraft missions that utilize the LEON processor so that current practices can be well understood. The VHDL that is used to program the FPGAs can be fully simulated in an SoST allowing for a “digital spacecraft twin” to be quickly evaluated [62] and executed.

7.2.4 SoSTs for CyberSecurity Applications

The demand from SoSTs during the past two years has increased significantly due to cybersecurity and penetration testing needs for spacecraft. Organizations, both within and outside NASA, have realized that having access to high-fidelity spacecraft simulators and their incredibly realistic data can be utilized for numerous purposes.

The spacecraft cybersecurity topics are related to 1) Command & Telemetry Verification, 2) Protocol Configuration Verification, 3) Vulnerability Testing, and 4) Incident Response Training. More work needs to be performed to further define cybersecurity requirements and implement “hooks” into the SoST to further develop cybersecurity capabilities.
8 Bibliography


9 Publications, Citations, Awards, and Presentations

The following publications, citations, and presentations are directly related to this dissertation. In all cases, I was a primary author and contributor. The research and development (R&D) described in this dissertation was performed for several years while being a full-time NASA contractor and part-time Ph.D. student.

Publications

- Simulation-To-Flight 1 (STF-1): Automating the Planning, Scheduling, Assessment and Data Processing/Reduction for a Small Satellite, Mark D. Suder, Justin Morris, Scott Zemerick, Matthew D. Grubb, John Lucas


• Off-the-Shelf Real-Time Operating System (RTOS) Space-Qualification for NASA: An RTEMS Case-Study, Mr. Scott A. Zemerick, Mr. Isaac H. Lambert, NASA Independent Verification and Validation (IV&V), Independent Test Capability (ITC) Team, Jon McBride Software Testing & Research (JSTAR) Laboratory

• ED24B-05 Results from the Space Physics Experiments on West Virginia’s STF-1 Cubesat Mission Dimitris Vassiliadis, Justin R. Morris, Matthew D. Grubb, Scott Zemerick, John Lucas, Majid Jaridi, Earl S. Scime, Gregory D. Lusk, Timothy Cameron, Mark D. Suder, Matthew Lazo, Michael E. Moran and Ollie M. Lehki, AGU 100, Fall Meeting, 2019, San Francisco.

Citations


Awards

• NASA Agency Group Achievement Award, NASA Operational Simulator for SmallSats (NOS3) and STF-1 (2020)

• NASA IV&V Engineer of the Year Award, 2019

• Runner Up, NASA Software of the Year Award, NASA Operational Simulator for SmallSats (NOS3), 2019
• NASA Agency Group Achievement Award, 2015, JIST Development and Validation Team, for demonstrating innovative and dedication to the development of a JWST software-only simulator for the NASA Independent Verification and Validation (IV&V) Program
• NASA IV&V Excellence in Values Award, 2016, ITC Team Support of JWST Project
• NASA IV&V Excellence in Values Award, 2014, Leadership in Providing JSTAR Deliverables

Public Presentations
• WV Legislature Joint Committee on Technology, Tuesday, January 8, 2019, WV State Capitol, Charleston, WV, Presentation Link
• SOST, NOS3, and STF-1 Press/Public Stories: http://www.stf1.com/articles.php