Hardware Emulation Strategies for Concurrent Microsatellite Hardware and Software Development

G. James Wells * Robert E. Zee ** Christopher J. Damaren *

Abstract

In small-satellite projects on short schedules, there is often insufficient time to develop new hardware and subsequently write software once the hardware is tested and ready. In some cases, it is not possible to build a satellite and develop new hardware for that same satellite. However, if the target functionality and performance of the new hardware elements are known together with their interfaces to other parts of the system, then emulating these elements may be useful if the effort involved in doing so is kept to a minimum. Ideally, the proper hardware interfaces should be implemented, and the purpose of the emulation should be to act as a substitute for the missing hardware so that flight code can be developed concurrently with the hardware. The use of the real-time development system RT-Lab™ (RT-Lab is a trademark of Opal-RT), which combines both software-based emulation and customized hardware interfaces, provides a flexible environment to develop embedded software early in the development cycle of a small satellite. As hardware elements become available, they can be interfaced with the real-time system, seamlessly replacing the software modules previously simulating their performance. This paper investigates the degree to which hardware can be emulated using the development of the attitude control system for the MOST microsatellite as an example. A trade study is presented that indicates when the cost of programming the emulator outweighs the benefits, and a law of diminishing returns applies. A level of hardware emulation is recommended that facilitates the early development of flight code, but beyond which only the actual hardware should be used.

continued on page 88

* Institute for Aerospace Studies
University of Toronto
4925 Dufferin Street
Toronto, ON M3H 5T6, Canada
E-mail: wellsj@ecf.utoronto.ca

** Space Flight Laboratory
Institute for Aerospace Studies
University of Toronto

Received 15 February 2002.

1. Introduction

Microsatellite projects tend to have small budgets and short schedules. This places constraints on how much work can be done in the early stage of development. At this stage, some hardware for the microsatellite might not be available because it has yet to be developed. The time spent creating this hardware will delay the development of the flight code that requires the presence of this hardware. If the functionality of the hardware can be efficiently emulated using software, then it would be possible to use a computer simulation system to replace the missing hardware. Along with a space-environment software model, this would allow the development of flight code while the hardware is being developed. The simulator should be one such that once the hardware is available, it can be inserted into the simulation, replacing its software emulation. The simulation system can then be used to test the interaction between flight code and hardware while working in a simulated space environment. The simulator can also provide operations support for the microsatellite after it is launched and be used to validate upgrades to flight code before they are uploaded to the orbiting microsatellite.

The use of a hardware-in-the-loop simulator involving the emulation of hardware is not new in small-satellite development. Past work has been done at Los Alamos National Laboratory (Ruud et al., 1997), Utah State University (Fullmer and Sevilla, 1997), and the Harbin Institute of Technology in China (Sun et al., 2000). All three institutes used computer simulator systems that combined both commercial off-the-shelf (COTS) technology with in-house-developed systems, all three involved hardware-in-the-loop, and all three used their simulator to design and test small-satellite systems. If hardware emulation is going to be frequently used when designing small satellites, it will be necessary to identify strategies that can be employed so that work can be done in an expedient manner. Being able to get a simulator system working quickly and being able to emulate missing hardware with little effort is critical to develop flight code at an early stage. The goal is to minimize any "throw-away" work: work that cannot be used either on the microsatellite or by the simulation system when it is used as an engineering model for testing the hardware once it becomes available.

suite de la page 87

RÉSUMÉ

Lors du développement de projets de petits satellites réalisés à court échéancier, le temps accordé à la mise au point du matériel est insuffisant. En conséquence, les concepteurs élaborent les logiciels une fois le matériel testé et prêt à être utilisé. Dans certains cas, il est impossible de construire tout d'abord le satellite et de concevoir ensuite les logiciels qui lui sont destinés. Toutefois, si le rendement et la fonctionnalité de la nouvelle pièce d'équipement et de son interface sont connus des autres éléments du système, il peut alors être utile d'émuler le fonctionnement de ces éléments en prenant soin de le faire avec modération. Idéalement, il faudrait intégrer les interfaces matériels appropriés et le but de l'émulation devrait être de servir de méthode d'analyse auxiliaire pour le matériel manquant de manière à ce que les logiciels de bord puissent être développés parallèlement au matériel. L'utilisation du système de développement en temps-réel RT-Lab™ (RT-Lab est une marque de commerce déposée d'Opal-RT), qui permet à la fois d'effectuer des émulations logicielles et la création d'interfaces matérielles personnalisées, fournit un environnement flexible pour la conception de logiciels intégrés tôt dans le cycle de développement d'un petit satellite. Au fur et à mesure que les éléments matériels deviennent disponibles, ils peuvent être couplés aux système d'exploitation en temps-réel, remplaçant directement les modules logiciels qui simulaient antérieurement leur performance. Ce mémoire détermine jusqu'à quel point les éléments matériels peuvent être émulés. Le développement du système de commande d'attitude du microsatellite MOST sert d'ailleurs d'exemple. Le mémoire présente également une étude déterminant le moment où la programmation de l'émulateur n'est plus rentable et l'application de la loi du rendement non proportionnel. Il est recommandé d'avoir recours à l'émulation du matériel dans la mesure où elle facilite le développement des logiciels de vol, mais au-delà de laquelle seul le véritable équipement devrait être utilisé.

Objectives

1. Develop a simulator system that combines both real-time hardware-in-the-loop simulation with easy-to-use software so that emulations of missing hardware can be made with as little effort and coding as possible.
2. Develop a model of the Attitude Control System (ACS) (processor and peripherals) of the MOST microsatellite that can be executed on the simulator system. Assume a maximum development time of 10 months, approximately how long it will take to develop the actual ACS processor. Keep track of the amount of work, in terms of time spent, that goes into emulating missing hardware systems and developing flight code that can run on the ACS processor once it is ready. Any code written in under 10 months is time that the simulator saved in code development after the ACS processor is available.
3. Using the experience gained from this development, create a methodology that can be used when doing work on the simulation system so that throw-away work is minimized. Based on this methodology, a trade study will be done on the work performed on the MOST ACS simulation to determine any relationship between the efficiency of the work done for each ACS subsystem and the complexity of the subsystem.
4. Based on the results of the trade study, determine the types of flight code that can be written early in the life of a microsatellite project vs. the flight code that should not be developed until the hardware is available. The trade study can also be used to determine which hardware systems can be added with little difficulty to the simulation system once it is being used as an engineering model test system.

MOST Background

The Microvariability and Oscillation of STars (MOST) microsatellite (**Figure 1**), being built in part at Dynacon Enterprises Limited, the University of Toronto Institute for Aerospace Studies Space Flight Laboratory, and the University of British Columbia, will be Canada's first space telescope. Being developed under the Canadian Space Agency's Small Payloads Program, it is scheduled for launch in 2002 and will conduct long-duration photometry of nearby stars. MOST requires an accurate three-axis attitude control system to successfully complete its mission.

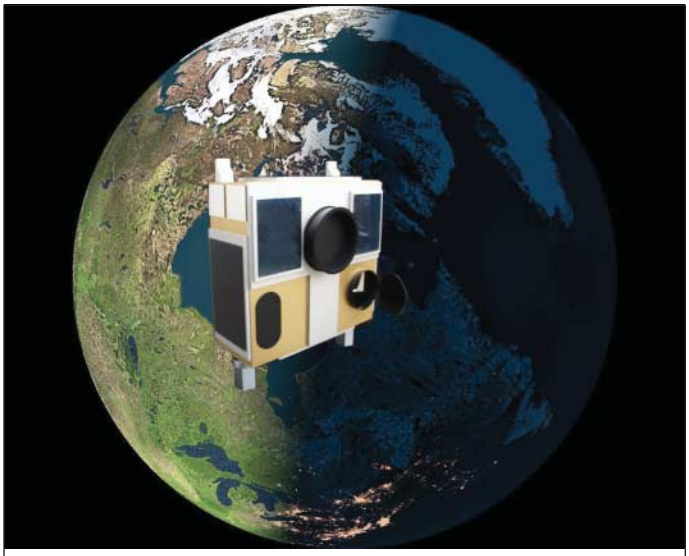

Figure 1. MOST Microsatellite.

2. Simulator Configuration

Hardware System

A simulator system, known as RT-Lab, made up of COTS hardware and software components is used to minimize the work needed to develop the system. The system has easy-to-use software for hardware emulation.

The RT-Lab system is a multiprocessor platform that enables real-time simulation of complex models. The system also includes software that creates, executes, and controls the real-time simulation. The system used here consists of two Pentium II 400 MHz computers and its configuration is shown in **Figure 2**.

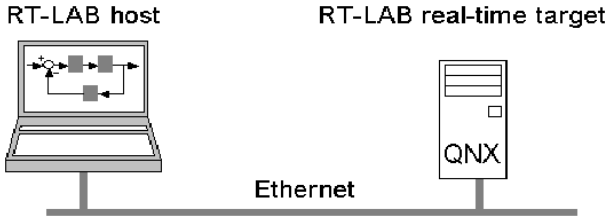

Figure 2. RT-Lab Computer C onfiguration.

The first computer is the host computer of the RT-Lab system. It has Windows NT as its operating system and runs the RT-Lab software. It is from this machine that the user creates the model that will be simulated in real-time. The host machine is also used to display and store data collected during the simulation run. The user can also interact with the model on the host machine by giving it input either before the simulation is started or while the simulation is running.

The software used to create the simulation model is MatrixX/SystemBuild™.[1] SystemBuild is a control block mathematical program; using built-in mathematical function blocks and user-designed code blocks written in C, the user can design a state model of a system. Past aerospace-related model development on SystemBuild include aircraft, spacecraft, and robotic systems. On the RT-Lab system, model work can also be done using Matlab/Simulink™,[2] a program very similar to SystemBuild.

The second computer in the RT-Lab distributed system is the target computer. This machine runs the QNX operating system. QNX is a version of UNIX that specializes in real-time computation. After a model is designed in SystemBuild on the NT host, it is simulated on the QNX machine to take advantage of the real-time kernel, timers, and interrupts that are available. These real-time tools make for an accurate simulation test bed. The target computer is also used to link hardware with the simulation. Using the motherboard slots on the target computer, PCI cards can be connected to provide a variety of data communication interfaces. These interfaces are used to connect hardware systems to the simulator. This hardware reacts to the model in every way, providing both input and reacting to the simulation as necessary. The simulator used here has interfaces for serial communication–RS-232 and RS-422/485 formats, and digital/analog IO. For very complex models, the RT-Lab system can include multiple target computers, each one handling one aspect of the model being simulated. For the purposes of this simulation, one target computer is enough.

The RT-Lab software provides an easy to use interface that can be used to perform all the necessary functions to run the simulation. Once a SystemBuild model is designed on the NT computer, RT-Lab then uses AutoCode, another ISI software program, to convert the model into C code. This code is then transferred by RT-Lab to the QNX computer via an ethernet connection, where it is then compiled and readied for execution. RT-Lab can then be used to control the speed of execution, as well as to store data continuously or upon being triggered by certain events.

Software/Model Design Philosophy

In SystemBuild, a block that contains many other types of system blocks is called a SuperBlock. The top level SuperBlock (**Figure 3**) contains two other SuperBlocks: the Master block and the Console block. The Master block contains all the model blocks pertaining to the actual simulation. Everything that was to be executed on the QNX target computer is placed here. The Console block contains all of the tools used to send and display simulation data on the host computer.

A constraint on the model design is that these two blocks must be included so that the model can work with the RT-Lab system. When the RT-Lab software converts the model into C code and sends it to the target node, it looks for the Master block to know which blocks are to be used in the simulation. It also creates a tool using the Console block information that can be used by the user on the Host computer to interact with the simulation running on the target computer.

When designing a model, the strategy is to maximize the use of built-in SystemBuild mathematical blocks whenever

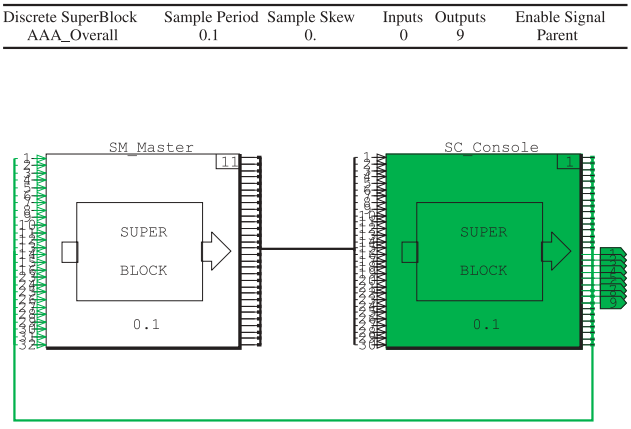

Figure 3. Simulation Top-Level SuperBlock.

[1] MatrixX/SystemBuild is a trademark of Integrated Systems Inc. (ISI).
[2] Matlab/Simulink is a trademark of MathWorks.

possible for sections of the simulation that are emulating missing hardware (e.g., sensors, actuators). At the same time, the use of C-code user blocks is maximized for systems requiring flight code development (e.g., ACS processor). If developed with the right interfaces, the code written in these user-code blocks can be used, with minor modifications, on the microsatellite itself. The ultimate goal is to reduce the amount of time spent emulating missing hardware while still generating usable flight code early in a microsatellite project.

3. Simulation Design

The simulation model designed on the RT-Lab system is based on some of the systems of the MOST microsatellite. **Figure 4** is a system diagram of MOST. The shaded boxes indicate which systems are part of the simulation model. The simulation model focused on the primary ACS processor and its peripherals. The systems modeled include a full set of actuators (three reaction wheels and three magnetorquers) and a full sensor package (a three-axis magnetometer, three rate sensors, and a two-axis Sun sensor). The model also includes an orbital environment simulation, complete with both a dynamic and attitude models along with a model of the Earth's magnetic field. The important model parameters used in the simulation are detailed in **Table 1**. The simple propagator error model was used initially so that the simulator could be up and running as quickly as possible. More complicated error models can be inserted later using SystemBuild blocks. The sensor noise is additive with discrete-time covariance.

As the model progressed from its initial configuration to its current state, ACS flight code was developed. All of the ACS flight code was placed in one SuperBlock. The ACS code written covered the following functionality:

- Serial Communication With Reaction Wheels
- Actuator Torque Estimation
- State Estimator/Kalman Filter
- Detumbling Control Law (Pastena and Grassi, 1998)
- Coarse-Pointing Control Law
- Momentum Desaturation Control Law (Chen et al., 1999)

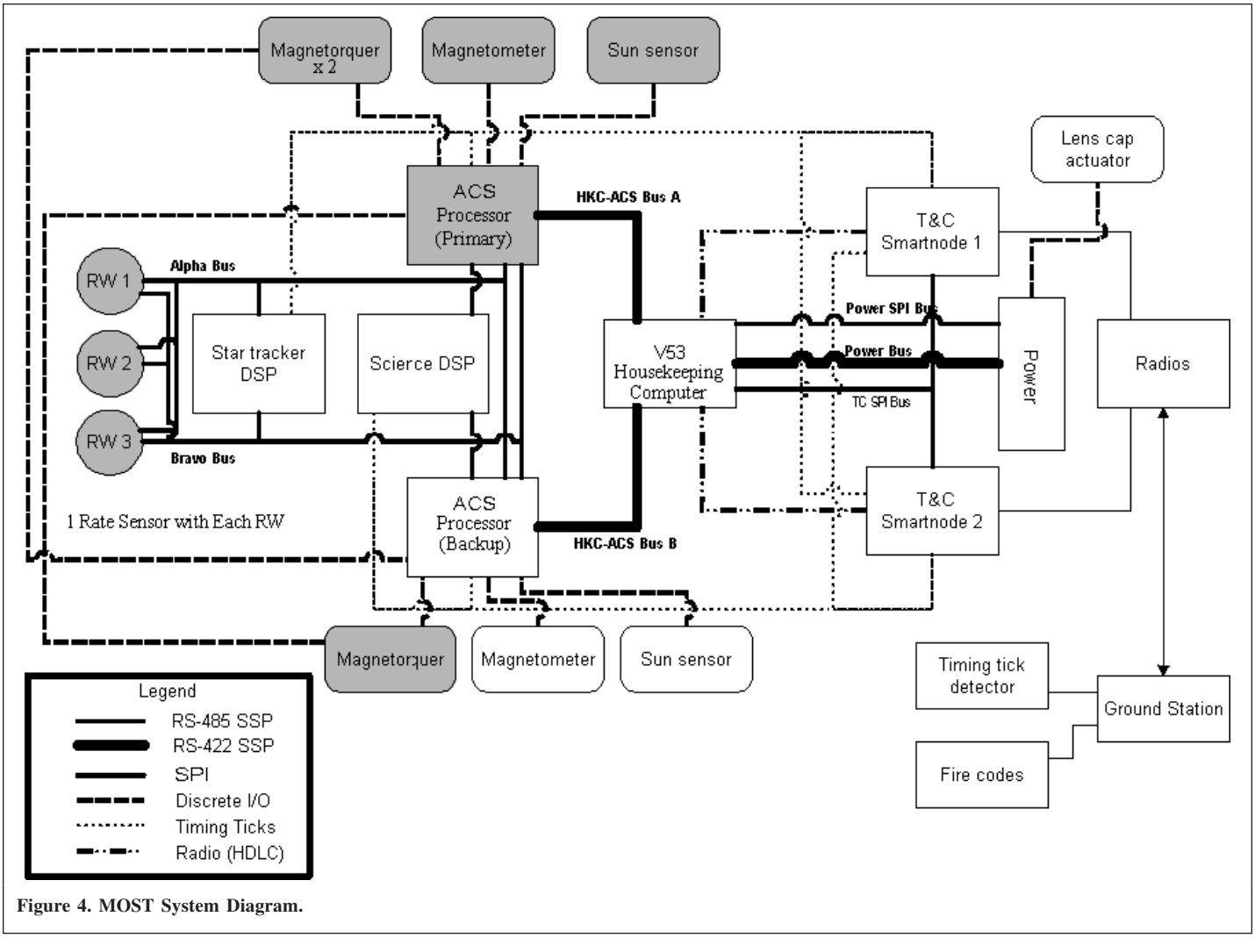

Figure 4. MOST System Diagram.

Table 1. Simulation parameters.

Orbit	Sun synchronous, 06:00–18:00 nodes, Alt. = 785 km
Principal moments of inertia (kg m^2)	$I_1 = 1.0$, $I_2 = 1.2$, $I_3 = 0.2$
Reaction wheel moment of inertia (kg m^2)	0.0001685
Max. magnetorquer magnetic moments (A m^2)	$m_1 = 5.3$, $m_2 = 5.3$, $m_3 = 4.48$
Sensor noise (σ: random number between 0 and 1 with uniform distribution)	Magnetometer: $\sqrt{12\,(2 \times 10^{-7})}\left[\sigma - \frac{1}{2}\right]T$ Rate sensors: $\sqrt{0.016}\left[\sigma - \frac{1}{2}\right]\text{deg/sec}$ Sun sensor: $\sqrt{0.14}\left[\sigma - \frac{1}{2}\right]\text{deg}$
On-board orbit propagator for ACS estimator	Uses simulation orbit model with a 5–10% error introduced
Simulation step period	0.1 s
Actuator alignment	The 3 magnetorquers and 3 reaction wheels are aligned along the principal axis frame of the microsatellite

The emulation strategies described in the previous section were employed and sped up the development of flight code. **Table 2** lists by system the number of SystemBuild mathematical blocks and lines of C code that were used to model each system. The C code listed for the ACS model includes the flight code that was written. The majority of systems could be quickly and easily emulated using only SystemBuild blocks, which saved much time. It took only three months to develop all of the flight code and environment code, including the time required to develop the entire model and test the functionality of the flight code. **Table 2** is ordered from top to bottom by the complexity of the emulation required to model each system, which reflects how long it took to create the emulation. The sensors were easiest to emulate, while the ACS model was the most difficult to create.

The quick development time was also made possible because testing and debugging the simulation and flight code was done on a block-model simulator system. The graphical aspect of the system made it simpler to spot errors and the software Console interface to the model made it easy to control the simulation and create different scenarios to test the ACS code and the fidelity of the actuator and sensor emulations.

Approximately one-sixth of the three month development time was spent working with SystemBuild mathematical blocks while the rest of the time was spent writing, debugging, and testing C code. Based on that time line, the approximate work time required to emulate each system is also listed in **Table 2**.

One of the three software reaction wheels was replaced with an actual reaction wheel.[3] While connected to the simulation via an RS-422/485 port on the target computer and communicating at 19.2 kbaud, it was very easy to replace the emulation blocks with blocks used to communicate with the wheel over the serial connection. The hardware reaction wheel reacted just like the software emulations; there were some slight differences in terms of maintaining very low wheel speeds when the actual wheel plant was compared with the software wheel plant, so the software emulations were modified to better match the hardware.

4. Model Development Methodology

Using the simulator system made it possible to develop important ACS flight code in three months, even without the presence of the ACS processor hardware. Assuming a typical processor development time of around 8 to 10 months, this allows concurrent hardware and software and hardware development, which shortens the amount of time that will be spent developing software after the ACS processor is built.

Once the ACS processor hardware is available, it would be beneficial if the simulator could still be used to work on the

Table 2. System modelling breakdown.

	No. blocks required	Lines of code required	Approx. total work (days)
Magnetometer emulation	6	0	1.7
Rate sensor emulations (×3)	6	0	1.7
Sun sensor emulation	6	0	1.7
Magnetorquer emulations (×3)	13	0	3.8
Reaction wheel emulations (×3)	12	155	12.4
Environment emulation	6	360	22.4
ACS emulation	3	790	46.3
Total	52	1305	90

[3] A Dynacon Enterprises Ltd. Miniature Reaction Wheel or "Microwheel".

microsatellite. From the experience gained in using the simulator to write ACS flight code, a methodology was developed to help write flight code and prepare the simulator once the ACS processor was ready. This methodology helps reduce the amount of wasted work: work that cannot be used either as flight code or as part of simulator code once the ACS processor is connected as hardware-in-the-loop. Though the methodology is focused on simulating the ACS processor, it can be applied to any microsatellite processor with peripheral systems, e.g., Star Tracker processor connected to a CCD camera.

1. Using empty SuperBlocks, do a basic modeling of the ACS system (processor, sensors, actuators, and all the links between the systems) on the simulator.
2. Start creating software emulations of the peripherals, starting with those that can be done using only SystemBuild blocks. Continue with the models that require some C code to develop. Prioritize writing any code that will be used as flight code on the peripherals (e.g., control code on a reaction wheel). Link these emulations to an environment model so that actuators will affect the attitude of the satellite and sensors will observe the environment.
3. Once the peripheral emulations are complete, start writing code for the ACS SuperBlock. The code should focus on functions that interact with the peripherals, which in the case of the ACS SuperBlock involves attitude control code and all the software-to-software interfaces to the peripherals. Test the code using the simulation. Debugging and testing will be an easier process because of the use of a block-model simulator system.
4. If any peripheral hardware becomes available before the ACS processor is completed, insert the hardware into the simulation and compare its behaviour to its software emulation. Update the emulation if there are any significant differences. Remove the hardware from the simulation.
5. Repeat Step 3 if the ACS code has to be updated due to any changes to the peripheral emulations. Repeat Step 4 if any more peripheral hardware becomes available.
6. Once the actual ACS processor is ready, move all of the ACS SuperBlock code to the processor and connect it to the simulation system. The processor is now interacting with the software emulation of all its peripherals. Now that the connection between the ACS processor and the peripherals is a hardware–software connection, the interfaces to the peripheral emulations will have to be replaced.
7. This system is now the basis for a microsatellite engineering model. As other processors become available (e.g., Housekeeping, Science), they can be connected to the ACS processor. The functionality of the entire microsatellite system can now be tested, with the simulation system taking the place of the environment, sensors, and actuators.

The actual ACS processor had not yet been interfaced with the RT-Lab system by the time of this writing. However, an analysis can be made on the ratio of throw-away work to useful work for the various software emulations when the processor is eventually connected to the simulation system. All the work done on the simulator system can be divided into two types: flight-code development and simulator-specific development. All flight code is useful work while simulator-specific work can either be useful or throw-away work depending on if it can be used on the engineering model described in Step 7. **Figure 5** explains this breakdown.

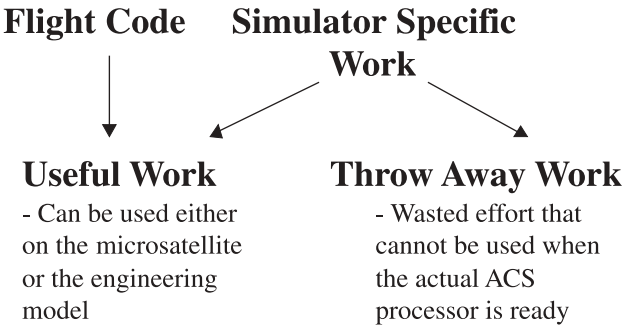

Figure 5. Simulator Work Breakdown.

Applying these definitions to the work done on RT-Lab, a trade study on work efficiency was done. **Table 3** details by system how much of the work for each emulation was useful, in terms of the number of SystemBuild blocks, lines of code, and work time. The work inefficiency ratio is also given for each system, where

$$\text{Work Inefficiency Ratio} = \frac{\text{Throw - Away Work}}{\text{Useful Work}}$$

By focusing the model development on developing only ACS code that interfaces with the peripherals (Step 3), the amount of throw-away work is limited to blocks and (or) code that interface the software ACS SuperBlock to the peripheral emulations. These interfaces will have to be changed if the ACS code on the actual processor is to be linked to the peripheral emulations. **Figure 6** shows the work inefficiency ratio as a function of the complexity (i.e., required work time to develop) of each system emulation, which is in order from the top of **Table 3** to the bottom. The three sensor emulations, being essentially the same in complexity, were placed in the plot as one data point. A trend is developed from the data points.

The goal of efficient flight code and simulation development is to focus on work that is within the bottom end of this trend, work with a work inefficiency ratio much less than 1.0.

Table 3. Useful work breakdown by system.

	Useful blocks	Useful code	Useful work (days)	Work inefficiency ratio
Magnetometer emulation	3	0	0.9	1.000
Rate sensor emulations (×3)	3	0	0.9	1.000
Sun sensor emulation	3	0	0.9	1.000
Magnetorquer emulations (×3)	10	0	2.9	0.300
Reaction wheel emulations (×3)	12	115	10.1	0.228
Environment emulation	6	360	22.4	0.000
ACS Emulation	0	690	39.7	0.167
Total	37	1165	77.6	

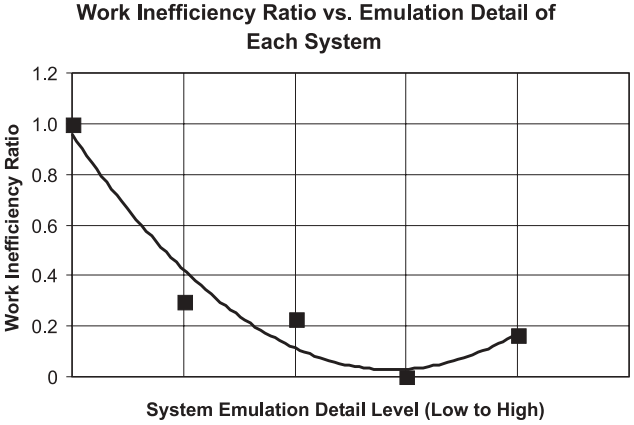

Figure 6. Work Inefficiency Plot based on MOST Simulation.

5. Simulator Limitations

Flight-Code Development Limitations

With this efficiency curve developed, the next step was to study the possibility of developing more flight code requiring more detailed software emulation of the ACS system. Would the curve remain below 1.0, or would it rise above that level? Two types of high-detail flight-code development were studied: intra-processor code and inter-processor code.

Intra-Processor Code: This flight code includes reading telemetry sensors placed on the processor board (temperature, power, voltage), memory access and storage, and some low-level software driver development. The amount of flight code needed to perform these functions tends to be small, around 10 lines of code each, for a total of around 30 to 50. However, to do any useful development work, it will require a low-level emulation of the ACS processor and its linkages to other devices on the processor board, such as the memory devices and telemetry sensors. Such processor simulations, based on experience, tend to require at least 100 to 200 lines of throw-away C code and would also need a few SystemBuild blocks to emulate the sensors. This results in a work inefficiency ratio of 2 at best, 6.666 at worse. Given the high work inefficiency ratio, this type of flight code should not be written until the processor hardware is available.

Inter-Processor Code: This flight code includes all of the serial software drivers needed to communicate between the model ACS processor and a model of the Housekeeping (HK) processor. It also includes the application program interfaces (APIs) needed to create and decode serial packets and the code that uses the APIs to send commands and receive telemetry over the serial bus. The most important aspect of inter-processor communications that can be checked using the simulator is the timing of packet transmissions: response acknowledgements to commands and the handling of commands that time-out. An attempt was made to write code for inter-processor communication using the simulator since the serial packet APIs had been previously been written by other members of the MOST team, but the attempt was eventually abandoned. It was proving too difficult to simulate the timers for each processor, and without an accurate simulation, any of the application code written using the APIs would be suspect when used on the actual processors; it could all end up being throw-away work, which would give a very high work inefficiency ratio. Since the APIs can be written without the processor hardware or the simulator, there is no point in developing inter-processor communication code until the hardware is available.

As model complexity continues to increase, it is found by extrapolation that the work inefficiency ratio also increases, as is shown in **Figure 7**. It is important to keep all flight code and simulator development within the minimum of the curve to get work efficiently done early in the life of the microsatellite and to have a good simulator system ready when the hardware is ready so that an engineering-model test system can be easily created.

Simulation Development Limitations

The philosophy of simulator development as shown in **Figure 7** can be applied to the simulator after the ACS processor has been interfaced as hardware-in-the-loop. It can help determine what other peripherals or processors can be inserted efficiently as hardware-in-the-loop. It is a bit more difficult to quantify a work inefficiency ratio at this point because all work done at this point is useful both for developing the engineering model as well as refining the systems and flight code of the microsatellite. However, it can be shown that

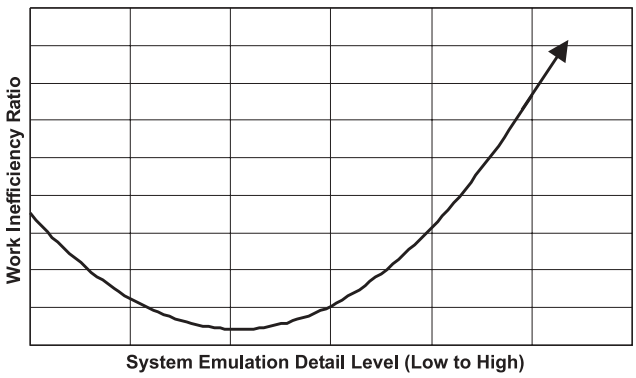

Figure 7. Work Inefficiency Extrapolation.

adding certain hardware systems to the simulator is more difficult as compared with other systems.

Processor Cards: Interfacing any of the other processors (HK, Science, Star Tracker) to the simulator system should be relatively simple because it only requires a hardware–hardware connection to the ACS processor. Since the other processors never need to communicate with the peripheral software emulations nor with the environment model directly, no interfaces need to be made between these processors and the simulator. With the HK processor now part of the simulation system, serial packet communication tests with the ACS processor can be done, and the ACS processor can react with realistic data because it is connected to the environmental model and emulations of all the sensors and actuators. The inclusion of the HK processor also makes it possible to add the Telemetry, Tracking and Control (TT&C) system, and radios to the engineering model.

Sensors and Actuator Hardware: Interfacing the majority of the sensors and actuators to the simulator system so that work beyond simply characterizing their performance, which can be done without the ACS processor as described in Step 3 of the methodology, would be difficult. It would require the use of ground-support equipment, such as an air-bearing table, magnetic isolation chamber, and a bright-light source. The magnetic chamber and light source, by replacing the Sun and magnetic software models, must also be interfaced directly with the simulation system so that the equipment knows where the microsatellite is positioned in its simulated orbit so that the correct environmental conditions can be reproduced. The simplest of these systems to interface to the simulator would be the reaction wheels and rate sensors; the reaction wheel is already designed to feedback its speed and torque telemetry and the air bearing table does not need to be interfaced with the simulation system. However, the other systems will prove to be very difficult to interface properly.

It is interesting to note that before the ACS processor is available, it is better to work on simulating the actuators and sensors than on simulating processor interaction. Once the ACS processor is interfaced with the simulator, other processor hardware can be added to the simulator system with little difficulty while one should be cautious in adding sensor and actuator hardware to the system if testing time is at a premium. This result helps to clearly define what missing hardware systems can be efficiently emulated early on in the life of a microsatellite, and what software work should wait until the hardware is available.

6. Conclusions

Using the RT-Lab real-time simulator, a simulation of the ACS system of MOST was created without need of the actual ACS processor. Using the experience gained from doing this, a simulation design methodology was developed to help minimize wasted work, to maximize the amount of flight code that can be developed early, and maximize any simulation work that could be used as part of a future microsatellite simulator for command software verification once the ACS processor is available. Using a work efficiency trade study based on the result of the simulation, it was determined what flight code can be developed early, and what flight code should be delayed until the processor hardware is ready (**Table 4**). The same trade study was also used to determine what hardware could be added to the microsatellite simulator once the ACS processor is available, and what hardware should be emulated (**Table 5**).

Table 4. Flight code development conclusions.

Early	When ACS processor is available
ACS Comm. with reaction wheels	Software drivers
Actuator torque estimation	Telemetry processing
State estimator/Kalman filter	Memory access and storage
Attitude control laws	ACS Comm. with other processors
Any peripheral system code	

Table 5. Microsatellite simulator conclusions.

Hardware-in-the-loop	Software emulations
ACS processor	Magnetorquers
HK processor – TT&C, radios	Magnetometer
Science processor	Sun sensor
Star tracker processor	Reaction wheels*
Reaction wheels*	Rate sensors*
Rate sensors*	

*Depends on availability of air-bearing table.

By having these lists of what work should be done when using the simulator, early flight-code development for the microsatellite should prove to be efficient. Beyond that, a law of diminishing returns comes into play and work inefficiency increases. At that point, flight-code development should wait until the hardware is available. The savings in time that will result by minimizing work inefficiency are invaluable for a small-satellite project with a short development schedule.

ACKNOWLEDGEMENTS

The first author would like to acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Canadian Space Agency (CSA) for its generous financial support of his graduate studies. He would also like to acknowledge the MOST engineering team for its help and support.

The UTIAS Space Flight Laboratory would like to acknowledge the support of Dynacon Enterprises Limited, the Ontario Research and Development Challenge Fund, the Center for Research in Earth and Space Technologies (CRESTech), the University of Toronto, and the Natural Sciences and Engineering Research Council of Canada (NSERC) for financially supporting the MOST project and the installation of lab facilities.

The authors would also like to acknowledge the following sponsors who have donated software or equipment to UTIAS/SFL, and whose generosity has helped to create a world-class university microsatellite laboratory:

Analytical Graphics Incorporated
Agilent Technologies
Structural Dynamics Research Corp (SDRC)
Integrated Systems Incorporated
Autodesk
National Instruments
Cadence
Raymond RMC
Micrografx
Tasking Incorporated
Altera
ATI Technologies
ENCAD
Rogers Microwave Materials Division
JDL Productions

REFERENCES

Chen, X., Steyn, W.H., Hodgart, S, and Hashida, Y. (1999). "Optimal Combined Reaction-Wheel Momentum Management for Earth-Pointing Satellites". *J. Guid. Control Dyn*. Vol. 22, No. 4, July–August 1999, pp. 543–550.

Fullmer, R.R., and Sevilla, P. (1997). "An Integrated Development System for Small Satellite Attitude Control Systems". *Proceedings of the Workshop on Control of Small Spacecraft*, Breckenridge, Colorado. 5 February 1997.

Pastena, M., and Grassi, M. (1998). "SMART Attitude Acquisition and Control". *J. Astronaut. Sci*. Vol. 46, No. 4, October–December 1998, pp. 379–393.

Ruud, K.K., Murray, H.S., and Moore, T.K. (1997). "FORTE Hardware-in-Loop Simulation". *Proceedings of the 11th Annual AIAA/USU Conference on Small Satellites*, Logan, Utah. 15–17 September 1997, Session II.

Sun, Z., Xu, G., Lin, X, and Cao, X. (2000). "The Integrated System for Design, Analysis, System Simulation and Evaluation of the Small Satellite". *Adv. Eng. Software*. Vol. 31, No. 7, July 2000, pp. 437–443.

