AN ENABLING DEVELOPMENT ENVIRONMENT FOR SPACECRAFT AUTONOMY

JOHN P. ENRIGHT
Research Assistant
Space Systems Laboratory
Massachusetts Institute of Technology
Tel: 617-258-0742
Email: enright@mit.edu

RAYMOND J. SEDWICK, DAVID W. MILLER
Research Scientist, Assistant Professor
Space Systems Laboratory
Massachusetts Institute of Technology
Tel: 617-253-7831/617-253-3288
Email: sedwick/millerd@mit.edu

ABSTRACT

To maximize the profitability of modern satellites, designers must invest their spacecraft with both capability and reliability. On-board, autonomous software holds the promise of greatly enhancing spacecraft abilities, yet software glitches have been directly to blame for recent, highly publicized failures. This paper presents the flight-software development framework created for the Generalized FLight Operations Processing Simulator (GFLOPS) testbed. Based upon a robust, commercial, real-time operating system, the methodology applies principles of object-oriented design in order to separate subsystem software functions into protected, quasi-independent modules. The flight software modules are coupled to simulation modules, which provide high fidelity, real-time, representations of system hardware and dynamics. The GFLOPS approach provides tools and a methodology suitable for rapid flight software development. Since the basic implementation of the framework does not rely on any advanced techniques such as autonomy, it can be used in both conservative and aggressive engineering programs.Focus applications include an MIT-designed formation flying experiment (SPHERES) and a U.S. Air Force-funded distributed satellite mission (TechSat 21).

Keywords: Flight Software, Real-time Simulation, Spacecraft Software Engineering,

1 Introduction

Paradoxically, space engineering is simultaneously one of the most conservative fields and one of the most innovative. Designers are forced in many situations to produce revolutionary systems, driving new technology development. At the same time, one needs to take a very measured attitude when trusting an expensive satellite to the whims of unproven techniques. With satellites routinely costing on the order of at least $100 million (and some even several billion), there is great reluctance to stray very far from proven solutions if they exist. This trend is particularly apparent in the development of spacecraft *flight software (FSW)*.

Researchers in *artificial intelligence* (*AI*) and autonomy have recognized for several decades that certain areas of space engineering can benefit considerably from ‘smarter’, more capable software (Marshall, 1981). Sadly, concerns about reliability have limited the applications of autonomy technology to situations in which traditional methods have proved completely inadequate. Usually this meant critical phases of interplanetary missions where propagation delays prevented traditional ground-based commanding. When employed, autonomy techniques were applied in a minimalist fashion; using only as much sophistication as necessary. Recently, interest is growing in expanding autonomy’s role to include more comprehensive spacecraft operations.

The *Generalized FLight Operations Processing Simulator (GFLOPS)* is designed to provide an environment for the high-fidelity, real-time simulation of distributed spacecraft systems. Intended to be an integral part of space software engineering, GFLOPS provides a convenient migration path from prototypes to deployed software. The simulation design process enforces a distinct separation between simulation code and flight-software. This promotes unbiased performance and more trustworthy results. While thoughtful simulation design is still necessary, the potential for ‘cheating’ (i.e. logic tailored to the simulator and not reality) is reduced.

This paper presents an overview of GFLOPS and an introduction to the development tools and design guidelines featured in the *GFLOPS Rapid Real-time Development Environment* (*GRRDE*). GRRDE represents an ongoing effort to improve the process of engineering advanced FSW from the conceptual definition of a space program, through manufacturing, deployment and continued operation. Three goals are represented in the GRRDE approach:

- Speed. Promote reduction in the time needed to develop FSW
- Reliability. Encourage design and coding practices resistant to systemic failures.

- Innovation. Provide a mechanism for incorporating advanced capabilities into tomorrow's spacecraft.

In order understand the context in which the system has been developed, it is important to identify the unique constraints that characterize space flight software development. The GRRDE framework is then presented with reference to these factors. Both design and implementation techniques are then discussed in reference to the system characteristics. Finally some unresolved issues are examined.

2 Requirements Definition

Development of the GRRDE framework requires an understanding of which factors differentiate space systems software from other software engineering domains. Two particular terms identify the peculiarities of this application: *embedded* and *space*. Developing software for embedded or real-time systems entails a set of concerns and techniques substantially different from those used in creating a program for a personal computer (PC). The application to space systems adds its own requirements too. To reaffirm the commitment to providing advanced software capability, the notion of *autonomy* must also be considered. Each of these terms represents a dimension to the software engineering process that must be accounted for during design, implementation, deployment and operations.

2.1 Embedded Systems

Many factors differentiate embedded software engineering from mainstream development. While space does not permit an exhaustive discussion of the factors here, it is sufficient to give a sense of the issues involved. In broad terms these can be categorized as temporal, correctness and reliability requirements.

Perhaps the most common misconception in the entire field of software engineering is the erroneous assumption that 'real-time' can simply be equated with 'fast'. This thinking can give rise to systems that are only coincidentally real-time. Often overlooked is that real-time code must satisfy certain guarantees regarding the completion of a computation. Lean, efficient (i.e. 'fast') code, is simply a mechanism for making the most of a given processor. It is a waste of money and power to run a lightly loaded processor. In addition to guarantees on *worst-case execution time* (*wcet*), real-time systems will often be concerned about *jitter*, the variance in the execution time. Jitter can disturb proper synchronization between system components and may lead indirectly to systems violating their deadline.

No software engineer intends to create faulty products, but the stakes are generally much higher in an embedded system. Incorrect software in this application is more likely to cause loss of life or money. While the elimination of local errors is important, a greater danger is caused by systemic problems in specifying requirements or conceptualizing the big-picture function of the software. Even though the software does exactly what it is supposed to do, the result is a system failure. A rigorous testing and verification regime is needed for embedded systems before they are put into service. It is even becoming popular to subject critical portions of software systems to formal verification methods (Heitmeyer and Mandrioli, 1996)(Vytopil, 1993). These techniques apply mathematics to an abstract description of system behaviour. The intention is to ensure that no foreseeable execution of the software will lead to a system level failure. While these techniques can be valuable, representing the problem or deciphering the results of such automated methods requires a high degree of skill.

The potential for systemic problems suggests that software does not typically 'fail' in the same statistical manner as more mundane components(Leveson, 1995). That being said, there are circumstances where undetected errors in the software can lead to system failure. Systems can potentially run out of memory, pointers may get scrambled, unexpected values may create math errors such as overflow. While quitting and restarting an application on a personal computer is acceptable the same solution cannot be applied to embedded software. Termination of a process will often results in the loss of critical state information. Consequently real-time systems must be very careful about checking and, where possible, preventing error conditions. Dynamic memory allocation is typically forbidden and many operating systems offer error recovery mechanisms and memory protection. Even if a particular process fails, the entire system may not crash. An embedded system must be designed to operate perpetually and the style of coding must reflect that.

2.2 Space Systems Engineering

Space system applications add an additional layer of considerations to the process of embedded software engineering. The distinctive characteristics of the space industry makes software development even more difficult. These challenges apply to both on-board flight software and ground support software as well. High risks engender very cautious design and operations mentality, punishing environments strain processor capabilities and low volumes necessitate effective testing and certification strategies.

Few other engineering environments are as demanding as space systems. Once launched, a spacecraft is on its own for as much as a decade or more. Except in very specialized circumstances, such as the Hubble telescope, repairs are out of the question. For a commercial satellite, a failure may mean bankruptcy. Consequently, the overall atmosphere is very risk adverse. The benefits of any new technology must

be carefully weighed against the potential for introducing failures. Unless critical for mission success new techniques are unlikely to be adopted. This is as true for software as for any other aspect of systems design.

It is not merely a matter of attitude that restricts coding practices. Harsh radiation makes the orbit environment an unfriendly place for modern electronics. Any device intended for launch must either be protected with heavy shielding, or subject to extensive and costly 'radiation hardening'. Consequently, the state of the art in spacecraft processors is often a decade or more behind their terrestrial counterparts. The dangers of radiation is not limited to the direct degradation of components. High energy particles are known to flip bits in memory or microprocessor registers. Some actions can be taken to detect and correct most of these errors, but undetected problems of this type can be very serious.

While other industries such as airline manufacturing share the same or greater sense of criticality as the spacecraft manufacturing, certain factors further constrain the space industry. All large systems must undergo a period of qualification and testing before deployment, but it is very difficult to test space hardware in the same environment in which it will be employed. Hence, greater reliance is placed on component and subsystem certification. Careful systems analysis must be employed to foresee undesirable interactions between system segments. Furthermore, the relatively low volumes of spacecraft produced, make it difficult to work out all the problems with a common design before it is obsolete.

An understanding of the constraints on embedded systems and the particular characteristics of spacecraft design establish the necessary background for enhancing the process of FSW development. The next few sections build upon this foundation and introduce the particular formalisms and tools introduced in the GRRDE.

2.3 Autonomy

Spacecraft autonomy is a term much bandied about but often ill defined. Rather than examine any particular definition or autonomous system, it is more productive to identify some of the roles that are considered suitable for autonomy in the aerospace field. The following is a list of several of the most popular:

- Control Based Autonomy. Based in control or optimization theory, generally an analytic control law is derived to direct a high level behaviour.
- Fault Identification/Recovery. This class of functionality is derived from diagnostic artificial intelligence.The software attempts to detect and identify faulty components, based on a model of nominal operation
- Planning. Planning can be done either on the spacecraft or on the ground. The system must organize a number of actions or observations in a system that is generally highly-constrained.
- Execution. Execution is a somewhat hybridized AI-based task. Its role is to coordinate the interactions between abstract plans and real-world events.
- Data Mining. Often based in Bayesian reasoning, these systems attempt to decide what observations to make or what data to send to Earth.

Each of these applications promises its own benefits and levies its own set of requirements. Although an elaboration of each of these roles would prove insightful, a discussion of the synthesis between autonomy and space systems is more germane to the goals of the GRRDE.

Attempts to introduce radical reforms in the design and capabilities of spacecraft flight software are often balked by what are termed 'cultural' issues. It is almost certainly true that a good part of this resistance to change comes as a result of habit and tradition rather than technical merit. Yet, resisting the addition of extra capability ('gold-plating') is an essential part of the systems engineering process. If the promised benefits seem marginal and the perceived risks are still substantial, widespread acceptance of spacecraft autonomy will never occur. In consideration, before spacecraft autonomy can become more than an experimental curiosity it must first overcome several hurdles.

The first challenge is to gain a measure of acceptance in the embedded/systems community. There is an adage that says "you can't fly unless you've already flown." Hence, a measured, incremental approach is more likely to gain support than a revolutionary one. Second, systems must meet a specific need. It is not enough merely to be useful. Complex, multi-purpose autonomous systems sound great in theory but are often ill suited to space systems. Consider an analogy between a pocket knife and a common screwdriver. While the screwdriver on the knife is functional, the latter is the better choice for setting many screws. On the other hand, a mountain expedition isn't the best place to carry an entire tool-chest.

Initial development of the GRRDE has focused on addressing some of the synthesis challenges of combining autonomous and space systems. It is the eventual goal of the program to develop autonomy based tools targeting a recurring, yet specific, set of space system issues.

3 GRRDE Approach

GFLOPS is designed to provide a platform for high fidelity real-time simulation of distributed or monolithic space systems. Three key components comprise the system:

- The Physical Tested. This includes the computer hardware, operating system (OS) and programming language.
- The GRRDE Design Guidelines. These rules help to establish effective methods of functional decomposition and software design
- The GRRDE Toolset. This is an Application Programming Interface (API) that accelerates FSW implementation.

Each component reflects certain design choices selected to maintain generality and enable the goals presented in Section 1.

3.1 Hardware, OS and Language

The success of the GFLOPS testbed can be measured in part by its accurate representation of spacecraft computing environment. At the same time it must maintain a certain degree of generality if it is to be useful for representing a wide range of missions. A careful choice of hardware, operating system and programming language ensures that GFLOPS can aggressively adopt new technologies while retaining credibility with the embedded community.

The premise of the GFLOPS simulation approach is to allow the FSW to operate in an environment as close to flight conditions as possible. Embedded single board computers were chosen to represent the on-board spacecraft processors. However, GFLOPS is purely a software simulator. Hardware in the loop simulation can be more accurate, but is both expensive and too closely tied to a particular design for application in early stages of development. Consequently, all the peripherals and sub-systems must be represented virtually. To minimize artificial effects, all simulation is performed on one or more of the support PCs. This ensures that the only code running on the embedded machines is the FSW. The testbed still allows full SW development. The designer can provide code down to the level of detail of writing command bits to an Input/Output register.

A schematic representation of the testbed hardware is shown in Figure 1. Eight, PowerPC-based computers represent a modern family of processors currently being investigated for space applications. Simulation tasks and user interface are the responsibility of several common PCs. These support tasks are partitioned into three categories:

- Payload. All simulation tasks related to the primary mission purpose
- Orbit/Environment. Environmental effects, orbit/attitude propagation as well as spacecraft subsystem simulation
- Ground Terminal. Acts as the user's access to the system. Capabilities similar to operator or end user's workstation.

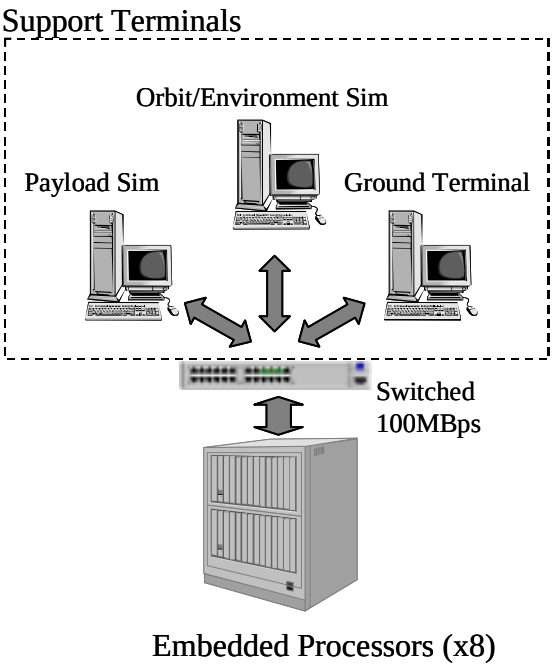

FIG. 1: *The Physical Testbed Architecture*

Lastly, 100MBps Ethernet provides the interconnections between components. The capacity of Ethernet is reasonably representative of both space to ground communication and inter-satellite links. It does not, however, provide a good mechanism for representing transmission errors, antenna tracking or latency. A summary of the hardware selection is given in Table 1.

TABLE 1: GFLOPS Hardware Summary

Embedded Processor	
Manufacturer	Force Computers
Model	PowerCore-6750
Processor	PPC-750 (G3)
Speed	400 MHz
Memory	256 MB
Support PCs	
Processor	Pentium III
Speed	600 MHz
Memory	384+ MB
LAN	
Switch Type	3Com SuperStack 100BaseT

GRRDE is designed to operate with the OSE operating system provided by ENEA Systems. The process model is based upon preemption and static priority assignments. Prioritized processes are design to handle the bulk of system load, but other process types are also supported (i.e. timer/

HW interrupts, background). Flexible memory protection is offered and the designer can group related processes into common address spaces. OSE is a modern real-time OS that supports many features that make it particularly appealing for use in distributed systems. Inter-Process Communication (IPC) in OSE is achieved through message passing (termed *signals* in the OSE parlance). The IPC operates transparently across memory protection boundaries and through network links. The operating system also supports the capacity to load executable code at runtime. This facility is a boon to simulation debugging in which the correction of small errors does not require rebuilding the entire system.

Without a language in which to write code the simulation system is useless. The choice of languages must balance traditional RT concerns of determinism, speed and space efficiency with more modern interests in readability, extensibility and expressiveness. As explained earlier the philosophy of the GRRDE is to take an aggressive approach to adopting contemporary software engineering tools, while retaining credibility with the spacecraft software community.

To achieve these goals the suggested language convention is that of Embedded C++ (EC++). EC++ is a subset of modern C++ that includes many of the important features of Object-Oriented Programming while acknowledging that the eventual target for the code is an embedded application. The restrictions on code content are driven by the following rationale:

- To minimize memory usage and code size.
- To maintain determinism.
- To promote a specification appropriate to the application.

The interested reader is encouraged to refer to the official reference for further information (EC++).

3.2 Object Oriented Design/Programming

Central to the development approach are the complementary notions of *Object-Oriented Design (OOD)* and *Object-Oriented Programming (OOP)*. While often assumed to be the same thing, it is more correct to see that the latter is but one possible implementation of the former. OOD, begins by examining the functional breakdown of a system in terms of compartmentalized entities called objects. An object is then an abstraction of a sub-system function. Functions that share similar behaviours can be further abstracted as more general objects. This process results in placing objects into hierarchical classification according to their function. Moving towards the root of the decomposition tree, one encounters increasingly abstract objects with generic functionality; moving towards the leaves reflects increasing specialization. Particularly important in this stage is the identification of interfaces and dependencies between various system components. The interface definition includes both internal connections between software components and external interfaces to hardware.

Functional decomposition from the systems design process is mirrored in the code development cycle. It is commonly agreed that reusing source-code is a good way to save time, money and aggravation. Embedded systems often run into trouble when trying to employ source code not designed or particularly well suited for reuse. The failure of the maiden voyage of the Ariane 5 can be directly attributed to just such an occurrence (Lions, 1996). The GRRDE approach to software implementation simplifies this problem. Code reuse between two systems should stop at the point at which the functional decomposition diverges. The GRRDE offers two mechanisms that simplify code reuse. First, Embedded C++ supports the notion of classes and inheritance. The second, is best described as a type of layering or macro-inheritance. This approach is tailored specifically to the GFLOPS environment and the streamlining of embedded development

3.3 State Centric Simulation Design

From the design process, specifications can be developed for each functional block. The specification process describes the abstract function of each module (including timing information), its inputs, its outputs, and lastly any external dependencies. A central tenet of the GRRDE design process is to think of these features in terms of *state information* (or simply *state*).

Taken literally, the state is a set of internal variables that reflect a module's perception of some aspect of the system. More importantly, the function of a module can be seen as operations performed on state information. Blocks then become sources or consumers of state information. The level of state abstraction can vary greatly. Thus, "The status register of the star-tracker contains 0xffec0001," represents a possible state as is, "The spacecraft altitude is 500.021 km," or "The +Z reaction wheel is acting erratically." In essence, the operation of a module is concerned with converting one form of state information into another.

Blocks cannot operate completely independently. Inputs and outputs identified in the design process can be expressed in terms of state as well. Inputs represent the state information that the block requires. This interface can either be directly to hardware (i.e. a sensor reading) or to another software module. Outputs are likewise a specification for a certain type of information that a module can provide. The data flow through a system can be charted by identifying the *sources* (i.e. providers) and *sinks* (i.e. consumers) of state information. At first this does not seem to have a remarkable effect on software development. One of the primary services that the GRRDE toolset provides is a

generic mechanism for state transport. Based on a subscription concept, clients can arrange *contracts* for periodic delivery of specific state information. These features decouple the process of operating on state information from the act of distributing it. Thus the logic of a module, in fact the entire block, can be written without reference to the precise origin or destination of the state information.

This state delivery mechanism makes the system inherently more modular. The information paths (i.e. state delivery) can be configured at runtime, eliminating the need to break encapsulation to make the appropriate connections. Functionally identical blocks can be interchanged provided they both provide the same public state outputs. Additional sinks can be introduced without directly affecting the source. The modularity promotes rapid and effective simulation development through the complete development process.

From concept definition through deployment, simulations fill a number of vital roles in spacecraft development. These applications can span bit-level simulation of software/hardware interfaces to rapid prototyping of command schemes (Reinholtz, 1999). One can view this FSW development as progress along two conceptual axes (Figure 2). The first dimension to consider is simulation *fidelity.* At first, only operational abstractions need be captured by the simulation, while later stages require accurate bit-level simulation of the information exchanged between sensors and actuators. The second dimension involved in the process is *functionality.* This can capture the local behaviour of a particular subsystem, a simple 'skeleton' spacecraft or even the full set of interactions and non-idealities of the complete system. Software design progresses along both of these axes towards the point of acceptance testing and full hardware integration. Even after deployment, simulations can be useful in the migration of capabilities from ground to space.

The modularity of the GRRDE system facilitates progress from one part of this design space to another: code can run in a real-time emulator or on embedded processors, simple modules can be replaced by more accurate ones. The common framework of the GRRDE approach is flexible enough to encompass all of these applications and to ease the transitions between them

3.4 The GRRDE Implementation

Employing common software objects in different projects is a direct example of code reuse. This is useful, but in the discussion of embedded systems, it is common to think about software engineering at the level of processes rather than just objects. To attain a second type of code reuse, the GRRDE exploits layering techniques as well.

Sub-functions are assigned to independent OSE blocks whose specifications are derived with reference to the

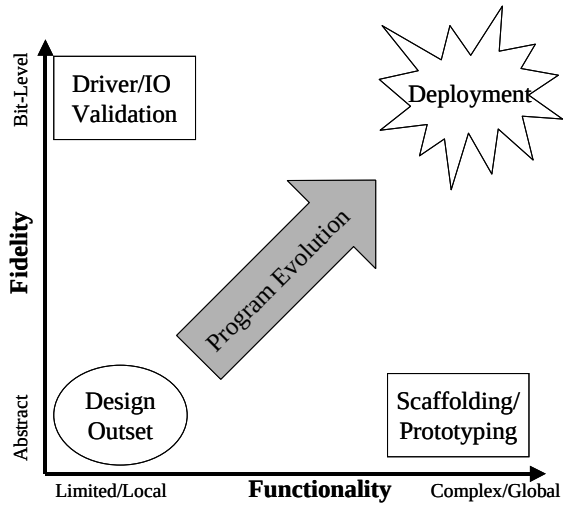

FIG. 2: *The roles of simulations in spacecraft development*

exchange of state. The basic set of services that GRRDE provides, is a generic a*pplication programming interface (API)* tailored to embedded systems. It is important to note that this API contains functions as well as distinct processes. Consider Figure 3. To the user, the operating system provides a number of basic services such as message passing and timing functions. The GRRDE layering process introduces three, progressively specialized layers on top of the operating system.

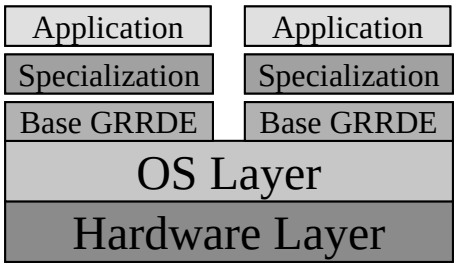

FIG. 3: *The GRRDE layering scheme.*

The *base* layer is common to all applications developed with the GRRDE. It provides a number a synchronization primitives as well as the information interchange system. The *specialization* layer provides the user with the opportunity to develop a library of functions and tools oriented to a specific type of task and yet still maintain a certain degree of generality. For instance, when developing an attitude control system (ACS), certain features and functions will be required regardless of the specific sensors and actuators on a satellite. Tools such as estimators, filters, and reference frame transformations are likely to be employed in all ACSs. The final stage is the *application* layer. This is where

the developer can provide all the customization specific to a given system. Particular sensors and actuators can be accommodated at this level. This layering process is somewhat analogous to adding features to an operating system. Unlike enhancing the operating system, common code may be duplicated in each block. While this does create a certain inefficiency in the use of memory, it does foster module independence and greater robustness.

To date, the GRRDE does not contain specialized code of its own. As the architecture matures, a richer variety of tools will become available. More importantly, the simple system developed thus far establishes a solid foundation for future refinement. If the GRRDE gains some external acceptance, manufacturers can provide standardized libraries of component models (sensors, actuators, etc.) to aid in the development process.

3.5 Comparative Reflections

Some perspective on the utility of the GRRDE can be gained by considering its role in relation to other software engineering efforts. Although the GRRDE attempts to provide some common-sense guidelines for design, implementation, and operations, it does not necessarily preclude the use of other tools. In many cases, the GRRDE can be seen as either an alternative or a complimentary approach.

From a philosophical standpoint, the GRRDE derives some of its inspiration (e.g. State Centrism), from elements of the Mission Data Systems (MDS) program developed at the Jet Propulsion Laboratory (Dvorak, D., Rasmussen, R., Reeves, G., Sacks, A., "Software Architecture Themes in JPL's Mission Data System," 2000 IEEE Aerospace Conference, March 2000, Page(s). 259 - 267). However, the GRRDE attitude towards implementation is substantially different from MDS. As described in the literature, MDS requires substantial 'buy-in' from systems engineers. In contrast, adopting the GRRDE approach represents a more modest step forward. Designers are left with the latitude to perform software task decomposition and implementation as their own development culture mandates. Whereas MDS places an emphasis on exploiting software reuse between missions, the GRRDE focus is on facilitating software reconfiguration and evolution within a particular mission.

Other strategies developed for enhancing FSW development integrate well with the GRRDE. Recently there has been increasing interest in graphical tools for control system development. Block-oriented software design with packages such as Wind River's *MatrixX* or The Mathwork's *Matlab* allow control engineers to generate embedded source code directly. These tools have already been employed in FSW development (Ptak, A., Foundy, K., "Real-Time Spacecraft Simulation and Hadware-in-the-Loop Testing," Fourth IEEE Real-Time Technology and Applications Symposium, 1998., Page(s): 230 -236). Automatic code generation can create the constituent 'logic' processes for the appropriate modules. The state transport mechanism allows inter-operability with other components of the system. In a complementary manner, it is expected that the GRRDE can be subsumed as part of the information infrastructure required by concepts such as the Space Project Mission Operations Control Architecture (SuperMOCA) (Jones, M. K., Carrion, C., Klassen, E. L., "SUPERMOCA: Commercially Derived Standards for Space Mission Monitor and Control", AIAA Defense and Civil Space Programs Conference and Exhibit, Huntsville, AL, Oct. 28-30, 1998(A98-45901 12-66), Page(s) 29-38).

Other terrestrial tools exhibit some of the properties of the GRRDE message transport mechanism. The Common Object Request Broker Architecture (CORBA) has a Real-Time variant (RT-CORBA) (The Object Management Group, The Common Object Request Broker: Architecture and Specification, Rev 2.4.1 November 2000, Page(s) 871-927), customized for embedded applications. Although the origins of GRRDE and RT-CORBA are different, some similarities in function are apparent. The GRRDE team is currently evaluating whether a CORBA interface could be adopted by GRRDE or whether they should remain separate complementary (or even alternative) tools.

4 Ongoing Research

The principles of the GRRDE approach have been carefully developed but it does not yet represent a finished product. A few technical issues regarding the composition of the real-time FSW system await resolution. Also, before suitable autonomy systems can be developed it is necessary to discern the tasks and missions that hold the most promise for enabling emerging missions. Finally, GFLOPS is a simulation testbed, designed to investigate operational issues for a variety of missions. A careful choice of mission simulations will demonstrate the utility of the GRRDE techniques.

4.1 Compositional Scheduling

Modern approaches to real-time computing have tended to move away from strict cyclic scheduling techniques. Approaches such as Rate Monotonic Analysis (RMA) (Lin and Layland, 1973) have gained acceptance, even in the space community (Ksenia, et al, 2000). The principle behind these techniques is to allow a more flexible process model, deferring process scheduling until runtime.

In RMA, each process is described by a period (how often it must run) and an execution time (worst-case). A process is enabled at the beginning of its period and will disable itself once it has completed a cycle. RMA then

assigns relative priorities to the processes based on their periods. At run time, the OS simply allows the highest priority process that is currently enabled to have access to the processor. This simple algorithm is guaranteed to meet all process deadlines provided the system utilization is less than a critical value. This value approaches 0.7 (or 70%) as the number of processes goes to infinity*.

The difficulty encountered with the GRRDE arises from its modularity. To maintain strong encapsulation, it is desirable to assign priorities to a module's constituent processes without reference to the rest of the system. However the effectiveness of RMA depends upon assigning priorities based upon the entire system. It is unclear whether any guarantees can be made about the performance of composed systems with independently assigned priorities. This topic is currently under investigation. It may become necessary to include a configuration pass during system build that will configure each code module and perform global priority assignments.

4.2 Autonomy Trade Study

There is an adage in the field of systems engineering that says "better is the worst enemy of good enough." If autonomous systems try to do too much or provide capabilities not mandated in the system requirements, they are unlikely to be implemented. Part of the aims of GRRDE research is suggest effective ways of adopting autonomy to real-time applications. Before this can be effective, a better understanding of the relation between autonomy and space systems is required. Specifically, the following questions must be answered:

- Who needs to use autonomy? Communications? Military? Science?
- What can autonomy realistically do? What can't it do?
- Where is autonomy most effectively employed? Ground? Space?
- When can autonomy be employed? When will the technology be mature?
- Why do you want to use autonomy at all?
- How is autonomy best implemented?

These questions are currently being carefully examined. It is anticipated that the answers will help further focus the future progress of the GRRDE tools.

* It is important to note that RMA provides only a sufficiency condition. It is a conservative estimate of what load the system can handle

4.3 Selected Simulations

The usefulness of a simulation architecture can only be measured by examining the quality of the simulations it produces. In addition to the work performed in developing the GRRDE approach, the testbed will be employed to study operational issues for a number of selected missions.

Two target applications have currently been identified for study with the GFLOPS testbed. The first is a distributed satellite demonstrator mission called TechSat 21. This experimental radar program uses a cluster of small satellites and interferometry techniques to do highly accurate *ground* and *air moving target indication (GMTI/AMTI)* (Das and Cobb, 1998). The spacecraft must be capable of complex on-board processing and high accuracy formation-flying. The second mission is a small micro-gravity technology demonstration known as SPHERES. Several, soccer-ball sized devices propelled by cold-gas thrusters must coordinate translational and rotational maneuvers. The SPHERES simulation will help establish the effectiveness of applying the GRRDE framework to a clearly specified system.

It is hoped that the results of the autonomy trade study presented in will identify other missions where autonomy will be crucial. Possible examples may include communications or space/earth science. Simulations for these systems will be developed as appropriate.

5 Conclusions

Software developed for real-time systems must account for temporal constraints, correctness, verification and low level reliability. The core of the GRRDE system reflects these interests. The embedded processors and commercial RTOS provide a platform for realistic development. The language choice of EC++ is a middle ground between expressiveness and semantic precision. All GRRDE tools are designed to incur very little overhead and operate deterministically. The development architecture represented in the GRRDE provides an opportunity to create reusable modules for many spacecraft functions. As more and more simulations are selected and developed the tool resources will grow. Lastly, the GRRDE has been designed to meet the needs of advanced flight software. Rather than champion a particular technique, the GRRDE approach represents an enabling technology for the promotion of spacecraft autonomy in general. For the autonomy community the GRRDE provides an object oriented language environment and flexible knowledge manipulation capability. Since it is possible to write FSW in the GRRDE framework with only traditional amounts of independence or cognitive ability, this approach represents a stepping stone towards achieving a spacecraft computing platform more suited for autonomy applications.

Once the basic architecture has gained some heritage, further enhancements are more easily introduced.

Many innovative and profitable missions are currently impossible due to crippling operations costs. Although many autonomous techniques have been studied for reducing the cost of operations, establishing the necessary confidence in these methods is still a difficult task. GRRDE attempts to redress these difficulties by introducing a capable software environment firmly grounded in principles of high reliability real-time computing.

6 Acknowledgments

The authors would like to thank the sponsor of this work: The (U.S.) Air Force Office of Space Research for "The Generalized FLight Operations Processing Simulator (GFLOPS)" under the technical supervision of Capt. Alex Kilpatrick

REFERENCES

[Marshall, 1981] Marshall, M. "Goals for the Air Force Autonomous Spacecraft" *USAF Report-SD-TR-81-72*, JPL Report 7030-1, 1981

[Heitmeyer and Mandrioli, 1996] Heitmeyer, C. Mandrioli, D. (eds.), *Formal Methods for Real-Time Computing*, John Wiley Press, 1996

[Vytopil, 1993] Vytopil, J. (ed.), *Formal Techniques in Real-Time and Fault-Tolerant Software*, Kluwer Academic Publishers, 1993.

[Leveson, 1995] Leveson, N., *Safeware: System Safety and Computers*, (New York: Addison-Wesley 1995), Page(s) 26-33

[EC++] The official EC++ website is: http://www.caravan.net/ec2plus/.

[Lions, 1996] Lions, J.L. (chair) "Ariane 5: Flight 501 Failure", *Report of the Inquiry Board*, Paris, July 19, 1996.

[Reinholtz, 1999] Reinholtz, K., "Applying Simulation to the Development of Spacecraft Flight Software", *IEEE Aerospace Conference*, Aspen CO, Mar. 6-13, 1999, Vol. 1 (A99-43255 12-31), Pages 469-476.

[Dvorak, et al, 2000] Dvorak, D., Rasmussen, R., Reeves, G., Sacks, A., "Software Architecture Themes in JPL's Mission Data System," *2000 IEEE Aerospace Conference*, March 2000, Page(s). 259 - 267

[Ptak and Foundy, 1998] Ptak, A., Foundy, K., "Real-Time Spacecraft Simulation and Hadware-in-the-Loop Testing," *Fourth IEEE Real-Time Technology and Applications Symposium*, 1998., Page(s): 230 -236

[Jones, Carrion and Klassen, 1998] Jones, M. K., Carrion, C., Klassen, E. L., "SUPERMOCA: Commercially Derived Standards for Space Mission Monitor and Control", *AIAA Defense and Civil Space Programs Conference and Exhibit*, Huntsville, AL, Oct. 28-30, 1998(A98-45901 12-66), Page(s) 29-38

[OMG, 2000] The Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Rev 2.4.1 November 2000, Page(s) 871-927

[Lin and Layland, 1973] Liu, C. L., and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *Journal of the Association for Computing Machinery* 20, 46-61, January 1973

[Ksenia, et al, 2000] Kolcio Ksenia O., Hanson, Mark L., Fesq, Lorraine M., Forrest, David J. "Integrating Autonomous Fault Management With Conventional Flight Software: A Case Study", *AAAI Spring Symposium on Real-time Autonomous Systems*, Spring 2000

[Das and Cobb, 1998] Das, A.; Cobb, R.; "TechSat 21: Space Missions Using Collaborating Constellations of Satellites".SSC98-VI-1, *Proc. of the 12th An. AIAA/USU Conf. on Small Sat*. Logan, Utah, Sept 1998