Flight Software for the Entire Operationally Responsive Space Vision

George J. Cancro, Edward J. Birrane III, Mark W. Reid, J. Douglas Reid, Kevin G. Balon, and Brian A. Bauer

Development of flight software for Operationally Responsive Space (ORS) is not simply the rapid development and testing of software in time schedules as short as 1 week. By examining the requirements from the original vision for tactical satellites and the plan for ORS, one can glean a set of software requirements that describes the needs of ORS in a more expansive manner. The ORS software solution needs to encompass capabilities that enable modification to meet future needs, to support rapid assembly of a system from existing component parts, and to provide the flexibility to add new capabilities to a system without compromising the existing development and testing. This software solution must also cover the entire life cycle from requirements development, to the time the spacecraft goes operational, and finally to the maintenance phase in the event that an on-orbit asset must be modified to meet a new need. A better understanding of the requirements for ORS software has led APL to define a new concept architecture made up of five key properties, which are described in this article. APL is pursuing the development of this architecture across multiple programs. This pursuit is a practical attempt to achieve our new architecture by coordinating multiple achievable steps en route to the ultimate goal of software enabling the entire ORS vision.

INTRODUCTION

Operationally Responsive Space (ORS) encompasses a vision of a new class of satellites that are rapidly developed and directly tasked by the warfighter in theater. The ORS Office, tasked with making this vision a reality, has developed a plan to rapidly develop and deploy space assets. However, the rapid development and deployment of spacecraft for ORS is hampered by the cost and time associated with development, testing, and integration of software.[1]

Efforts at the Air Force Research Laboratory (AFRL) to develop software to support ORS timescales have focused on self-realization of hardware components by the software,[2,3] much like the current PC model when you plug in a USB device. While the AFRL effort enables hardware "plug-and-play," we believe that the software application associated with this effort will continue to increase in size, much like the growth of the number of .DLL files that the PC needs to enable plug-and-play. The end result is a large complex piece of software that may overflow space-qualified processor capabilities, will be very difficult to test, and may contain the potential for many different adverse reactions among different aspects of the code.

Multiple efforts evoking the principles of modularity and standardization have been proposed and developed to combat the issues with hardware development, testing, and integration.[4] Concepts of plug-and-play architectures suggest rapid development through the assembly of existing parts. The question can be asked: Is there an analog for software development?

This article first returns to the original ORS vision and plan for rapidly developed spacecraft and attempts to develop requirements for an ORS software architecture. An analysis is performed by looking critically at the software requirements that are needed to support the desired ORS tiers. Next, a new architecture is outlined in terms of the features needed to meet these requirements. Finally, this article demonstrates how APL is moving toward the ultimate goal of software designed for ORS. By moving out in multiple arenas and with different customers, APL is slowly building and gaining confidence in all elements of the final architecture needed for ORS before bringing it all together. Each step brings a new usable capability and is closer to enabling the elements outlined in the ORS plan.

ORS SOFTWARE REQUIREMENTS AND ANALYSIS

By examining statements and documents from the ORS community, one can glean a set of software requirements that provide a basis for the development of software for ORS.

Admiral Cebrowski, in his statement to the Senate Armed Services Committee,[5] stated: "The time function for responsiveness is then driven by adaptive contingency planning cycles rather than predictive futures or scripted acquisition periods." Cebrowski envisioned a new class of rapidly developed satellites directly tasked by the warfighter in theater. The *Plan for Operationally Responsive Space*[6] divides these planning cycles into three tiers. Tier-1 uses or modifies existing assets to deliver capabilities in minutes to hours. Tier-2 uses field-ready, or already produced, assets to deliver capabilities in days to weeks. Tier-3 develops new capabilities to be delivered for use in months to 1 year.

Requirements from Tiers-1 and -2 suggest that future systems will have a capability to be modified. The ORS plan states: "Tier-1 solutions will not typically involve the design, engineering, or fabrication of new material items." Therefore, to produce new capabilities, the concept of operations of a spacecraft has to change. This must be accounted for either in operations or in software and leads to the following requirements for rapid development.

ORS Software Requirement 1 (OSR-1)

Flight software systems shall be able to be rapidly modified at any time after the deployment to field-ready or flight status.

The ORS plan goes on to describe "mission or system utilization analyses may be needed" such that any modification must be accompanied by rapid analyses to determine the modifications that are required for the existing capability to meet the new need and whether these modifications result in a safe and workable capability. This is due to the fact that changing any aspect of the asset risks the asset itself as a result of the faults that may be induced indirectly by the change.

ORS Software Requirement 2 (OSR-2)

A software system's capability to be modified shall be accompanied by the ability to certify that the change is providing the desired capability and not endangering the existing asset.

The ORS plan describes Tier-2 activities as "achieving responsive . . . capabilities through rapid assembly integration, testing, and deployment of a small, low cost satellite." Given that the time frame for Tier-2 capabilities is days to weeks, there is no opportunity for development or testing of any software. As the ORS plan states, "much of the ORS work will be anticipatory in nature." Therefore, all software development and testing must be accomplished before call-up.

ORS Software Requirement 3 (OSR-3)

Software systems shall enable complete development and testing before integration into a target.

If software can be developed and tested before a project starts, software development is then shifted from coding to integration of preexisting parts, and therefore the software must support rapid development in the form of easy and rapid integration.

ORS Software Requirement 4 (OSR-4)

Software systems shall enable construction through the rapid integration of preexisting software elements without negating any prior testing.

Requirements from Tier-3 suggest that new capabilities must be delivered in less than 1 year. To deliver a spacecraft in 1 year, the software development process must be completed in 2–3 months to allow for system-level design and testing. The duration of the 2- to 3-month development schedule relegates new development to potentially only one or two new functions. The 2- to 3-month schedule, however, does not leave any time to modify and retest the existing code to accommodate these new functions. In a standard spacecraft software system, modifying the existing code would force a series of regression tests of the entire software system, which is extremely time consuming. In an ORS software system, the software must be architected such that the addition of new functionality does not force the modification or retest of existing code to be reused.

ORS Software Requirement 5 (OSR-5)

Software systems shall enable the addition of new software applications without modification of existing software or negation of existing software testing.

Rapid development for ORS in the minds of Cebrowski[5] and Wegner[7] extends past traditional software development and even past the launch to the point at which the checkout phase completes and the spacecraft operational capability is brought online. Cebrowski states that ORS spacecraft must "reach the required orbit without months of state-of-health checks . . . by large squadrons of satellite controllers." Therefore, the software system must be sufficiently autonomous to assist the rapid drive to operational status.

ORS Software Requirement 6 (OSR-6)

Software systems shall be sufficiently autonomous to support rapid on-orbit checkout.

CONCEPT ARCHITECTURE

Considering the requirements described in the previous section, a conceptual software architecture for ORS can be constructed consisting of the following five key properties:

1. Modular components
2. Test-once testing
3. Autonomous checkout and calibration
4. Ability to modify both before and after launch
5. Fault detection and self-healing

These properties are described in detail in the following sections.

Modular Components

The term "modular components" defines a principle that software is developed, stored, tested, and deployed at the functional level rather than at an overall application level. Whereas traditional development of spacecraft flight software follows the model of custom-built, tightly coupled software compiled into a monolithic architecture, the modular-components model advocates the development of flight software into smaller modules at the functional level. By reducing the size of software components and using a modular approach, an organization can maximize the use of common parts and reduce the complexity for the rapid-response workforce to implement a wide variety of missions.

Developing components at the functional level enables a new development process, shown in Fig. 1, in which requirements, the source code, the executable code, and test cases are stored together for each module. The formation of a new project begins with a requirements gap analysis to determine which modules meet project requirements and which requirements may result in new modules. The process ends with a requirements document formed by the summation of module requirements, a test suite formed by the summation of test cases, and a software system formed by the list of executable code files from the modules selected. This new process meets the development portions of OSR-3, OSR-4, and OSR-5.

The modular-components model offers a contrasting strategy to the AFRL Software Data Model concept[2] of self-realization discussed in the introduction to this article. The modular-components model advocates the development of a software application warehouse, as shown in Fig. 2, filled with already tested software modules, very similar to the hardware plug-and-play approach. According to the modular-components model, software is selected from a warehouse on the basis of individual project needs. Applications can also be associated with specific pieces of hardware. Either way, only the necessary applications for each project are

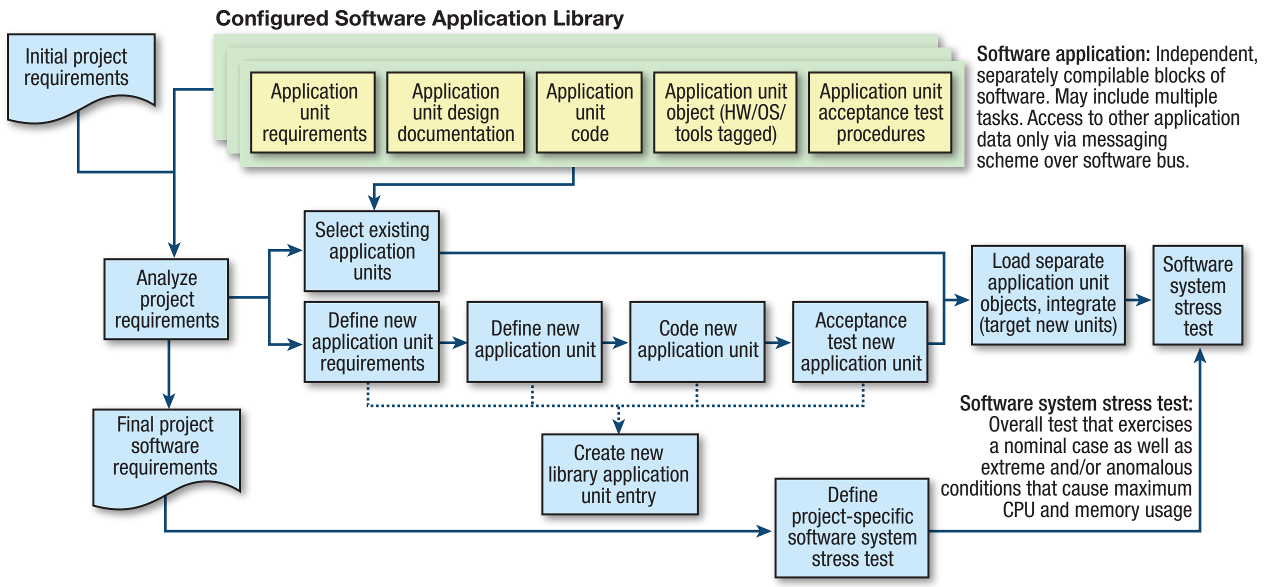

Configured Software Application Library
Initial project requirements
Application unit requirements
Application unit design documentation
Application unit code
Application unit object (HW/OS/ tools tagged)
Application unit acceptance test procedures
Software application: Independent, separately compilable blocks of software. May include multiple tasks. Access to other application data only via messaging scheme over software bus.
Select existing application units
Analyze project requirements
Define new application unit requirements
Define new application unit
Code new application unit
Acceptance test new application unit
Load separate application unit objects, integrate (target new units)
Software system stress test
Final project software requirements
Create new library application unit entry
Software system stress test: Overall test that exercises a nominal case as well as extreme and/or anomalous conditions that cause maximum CPU and memory usage
Define project-specific software system stress test

Figure 1. Modular-component development process. CPU, central processing unit; HW, hardware; OS, operating system.

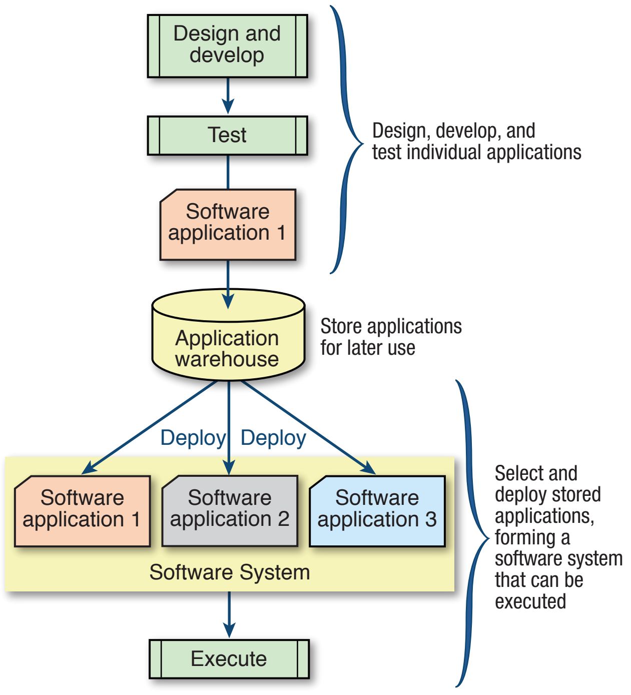

Design and develop
Test
Software application 1
Design, develop, and test individual applications
Application warehouse
Store applications for later use
Deploy
Deploy
Software application 1
Software application 2
Software application 3
Software System
Select and deploy stored applications, forming a software system that can be executed
Execute

Figure 2. Software application warehouse.

integrated together, resulting in a more compact system with improved testing capabilities.

Test-Once Testing

Test-once testing advocates the development of an isolation infrastructure that enables testing and system integration without invalidating any new or existing software component. With traditional highly coupled software, making a change to any heritage code usually results in a repeat of the entire testing program. However, to meet the testing portions of OSR-4 and OSR-5, the integration of modular components must not invalidate the testing of the components or other existing modules. Component tests are invalidated if the component under test is changed to enable the integration or if the component test environment used to validate the component is not the same environment as the integration environment. In addition, to comply with OSR-3, the complete functional testing of a module must be able to be performed before integration.

To accomplish these goals, APL advocates that each application be tested and executed in isolation (i.e., its own "little world" with fixed boundaries in processor time and memory and a single interface to other applications that does not require assumptions about who or what else exists in the system). Figure 3 shows the test and integration environment using the test-once process. The test environment includes a software bus, the application in its memory partition, and another test application in another memory partition providing data on the bus to stimulate the application on the target processor. The remainder of the memory space is empty. The application being tested runs in its own time slice, as does the test application, but the remainder of the processor time map is idle. As long as the memory and time allocations are the same and the same software bus communication interface is used, the test environment and the deployed environment are the same; therefore, the test-once testing is sufficient to establish the correctness of the module for all future deployments.

Autonomous Built-in Checkout and Calibration

Autonomous built-in checkout and calibration defines a capability to rapidly prepare in-flight spacecraft for

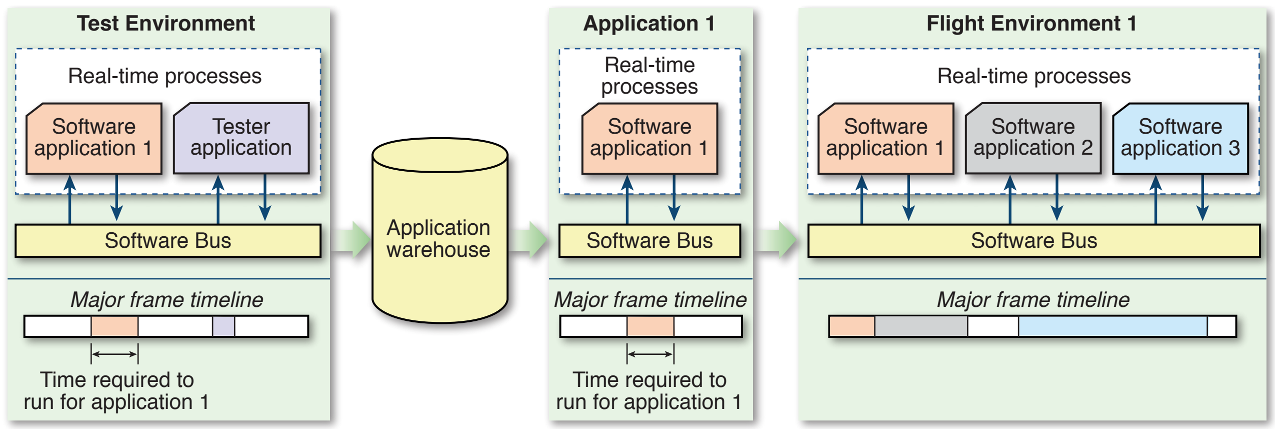


Figure 3. Testing and integration environment.

operations. The principles of ORS are not complete at launch. To support combat commanders and warfighters on the ground, the ORS concept must extend throughout all phases until the spacecraft goes operational in the desired theater. To rapidly go operational, ORS satellites require the capabilities to perform autonomous checkout of the bus and the payloads and to perform calibration of payloads.

These capabilities are essential for meeting OSR-6 for both ORS Tier-2 and Tier-3 missions because all spacecraft and sensors must be analyzed to ensure that the required spacecraft performance survived launch.

To achieve this, APL advocates a fully automated checkout and calibration of payloads through a built-in test (BIT) feature. This feature would be a separate software module existing alongside the standard flight software functions (i.e., built-in) on the flight processor. The BIT module would exist on the flight processor throughout the development, testing, and postlaunch phases, providing a single method of performing checkout for the instrument or spacecraft function throughout the entire life cycle. In this manner, the detail of the checkout or calibration matures with the development and also increases the speed of checkout on the ground and in flight. In addition, we advocate the development of standards for the BIT to enable instrument developers to develop their own BIT modules. This would allow checkout procedures to be paired with instruments, such that the integration of an instrument into an ORS spacecraft bus includes integration of the instrument into the bus and integration of their BIT into the flight software.

Ability to Modify Both Before and After Launch

Modification of onboard executing software usually requires patching activities or the long process of replacing an entire software system on board and then rebooting the processor for the change to take effect. Both processes are time consuming and dangerous to the existing asset. Therefore, currently most late or postlaunch changes are implemented by increasing the burden or requirements on operations staff. Increasing this burden results in an increase in the size of operations teams, which is contrary to the ORS plan.

Because modifications range in size, we advocate development of a capability that enables modification at two levels to comply with the postlaunch modification OSR-1. Level 1 modification would occur at a functional level through independent module loading, and level 2 modification would occur within the function through the use of engine-based designs.

Functional-Level Modifications

A functional-level modification would consist of the addition or replacement of a software module. This is viewed as an extension of the modular-components concept discussed previously in this article except that the new module is added to the existing code after launch.

Engine-Based Designs

In some aspects, modifications may need to be made at a finer level than at the functional-level module. However, making just basic modifications to an application will invalidate the testing and therefore create a new application. In the event that an application will be known to be modified in future missions, we advocate developing that application as an engine-based design.

An engine-based design is one in which the application acts as a generic interpreter for a data set rather than requiring a developer to hard-code the entire software module (Fig. 4). The data set (or memory object) is an expansion of programming parameters. The difference is that the memory object can be modified without having to modify or recompile the module. In this manner, the module does not change and does not invalidate the testing done before integration. The memory object contains the information to enable the interpreter to act correctly in a specific mission. The combination of the

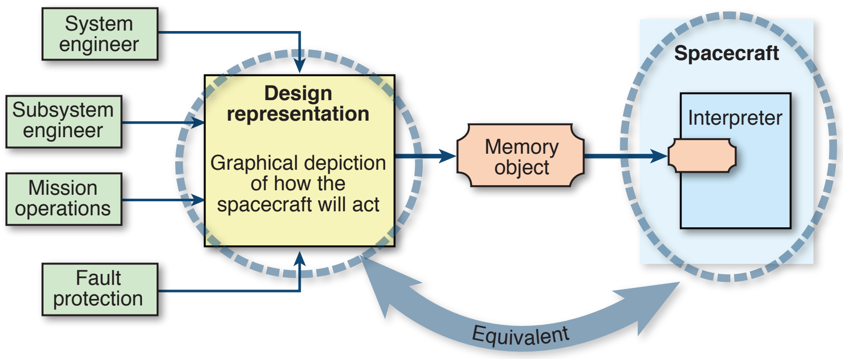


Figure 4. Engine-based design with generic interpreter and data set (memory object).

memory object and the generic interpreter produces the mission-specific functionality.

Analysis must still be performed to determine whether the change has achieved the desired benefit and to confirm that it does not endanger the system as stated in OSR-2. But with engine-based designs, the analysis is performed on the memory object; the benefit is that the analysis of the memory object may be trivial or can be performed offline and parallel to the integration of applications into the final system. In addition, other test techniques, including different automated methods such as model checking, which drastically reduces the time to check the memory object versus the requirements, can be used on the memory objects that could not be employed on the application code.

Fault Detection and Self-Healing

Fault detection and self-healing defines an ability to restart individual software tasks or functions, in the event that a task exception or fault occurs, without affecting the other tasks or rebooting the processor. One method to achieve the ability to modify existing assets without endangering the asset, as stated in OSR-2, is to ensure that a fault occurring in the new software module does not propagate to other existing software modules. In contrast, in traditional systems a fault that occurs in even a low-priority task compromises all software on that processor, and all activity on the central processing unit may need to be restarted through a cold or warm reboot. If this central processing unit were the active one on board, the reset could result in the spacecraft stopping operations and retreating to a "safe" mode, which may require weeks and a large ground staff to recover. The principle of self-healing enables the fault detection and response to faults with individual modules to be localized to that module only. The remaining modules continue as the faulty application is stopped or restarted by reloading the application from persistent storage to random-access memory. The end result is that new software modules can be fault isolated from existing modules, enabling the ability to modify or add to the software of a spacecraft without risking the existing asset to meet ORS Tier-1 objectives.

ACHIEVING ORS SOFTWARE

For APL, enabling a software architecture that enables rapid development for ORS is not only a requirement or conceptual exercise, but also a practical drive to gain the capabilities described in this article. We are moving toward achieving these principles on multiple projects. The following sections detail a sample of projects advocating one or more of the architecture principles described above.

Radiation Belt Storm Probes Software Bus

APL software engineers on the NASA Radiation Belt Storm Probes (RBSP) program[8] are making progress in the area of modular components. Key to the achievement of the modular components concept is the ability to add and remove software components without having to modify components that already exist. The direct fallout from this statement forces communication between software modules to a singular interface that cannot change based on the number of modules in the system. RBSP will use NASA Goddard's Core Flight Executive[9] software bus architecture to implement a single common interface for all software modules. Figure 5 shows the RBSP modular-component software bus architecture. By implementing a bus architecture, RBSP takes the first step toward the modular-component addition necessary for rapid development with ORS software.

RBSP is also advocating a modular-component approach in development of software. Requirements and code will be separated by function, with each function being a separate entity on the software bus. This represents a first step in the software-warehouse concept outlined previously. Future missions that use the same software bus architecture should be able to select RBSP software components and get immediate requirements and code reuse.

Naval Research Laboratory–ORS Modular, Self-Healing Patterns

APL has also been working with the Naval Research Laboratory to develop software technologies for ORS. The architectural concept developed by this effort describes a "buffet-style" software approach to software development whereby missions configure flight software through the assembly of existing software modules followed by integration testing. This

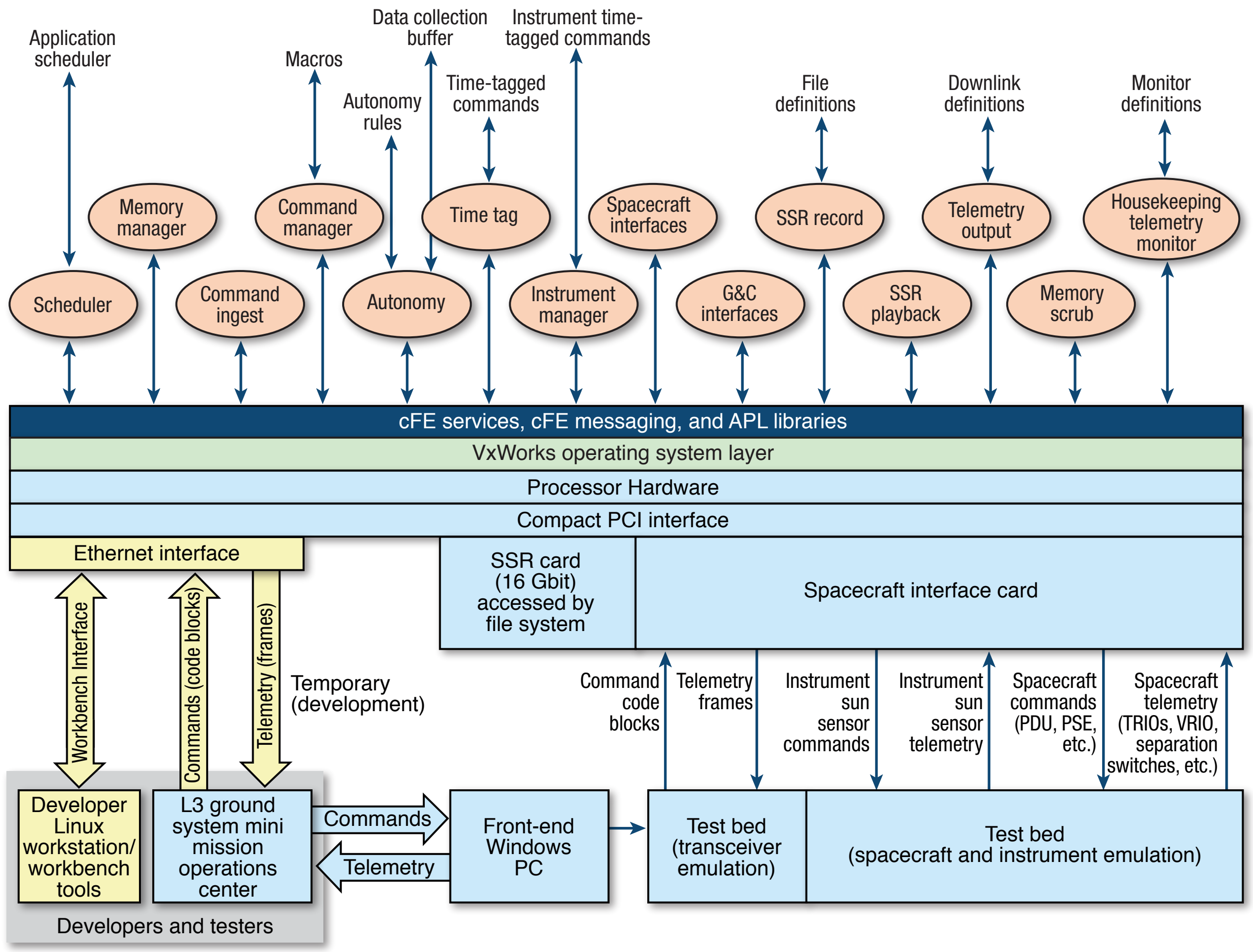


Figure 5. RBSP modular-component software bus architecture. cFe, NASA Goddard's Core Flight Executive; PDU, power distribution unit; PSE, power system electronics; SSR, solid-state recorder; TRIOs, temperature remote input output; VRIO, voltage remote input output.

approach significantly reduces software development schedules and budgets once a set of software building blocks has been created. This architectural concept was broken down into seven software-development patterns that eventually matured into the architecture properties discussed in the previous sections: modular components, fault detection and self-healing, and ability to be modifiable before and after launch.

Software in the Loop

APL, using independent research and development funding, has been working on a concept called SWIL (software-in-the-loop) testing. The concept advocates colocating test software with flight software in the flight processor, as shown in Fig. 6. The benefits of this include reducing the cost of test infrastructure (i.e., SWIL can replace some aspects of hardware-in-the-loop testing) and eliminating bandwidth issues associated with testing high-speed spacecraft guidance and control algorithms.

SWIL also made progress in the area of test-once testing and autonomous BIT. To achieve isolation between the tester and the application under test, SWIL used a memory partition, called a real-time process, in the real-time operating system VxWorks 6.3. Memory partitioning is the first step in achieving isolation between modules described above in test-once testing. In addition, the SWIL concept was a trailblazer for testing the environment outlined above, in which the test application and the application under test can exist in a single target processor without corrupting the test.

Finally, the SWIL concept forms the basis for our development of autonomous BIT. APL is currently investigating how to extend this concept to allow the test software to coexist with the flight software all the way from development through launch and into flight. The in-flight SWIL could be used to perform in-flight checkout and/or in-flight calibration of instruments. Having this feature from development into flight provides a consistent test capability rather than having to add in-flight checkout or calibration late in the development. In this manner, the ground checkout or calibration is the flight checkout or calibration.

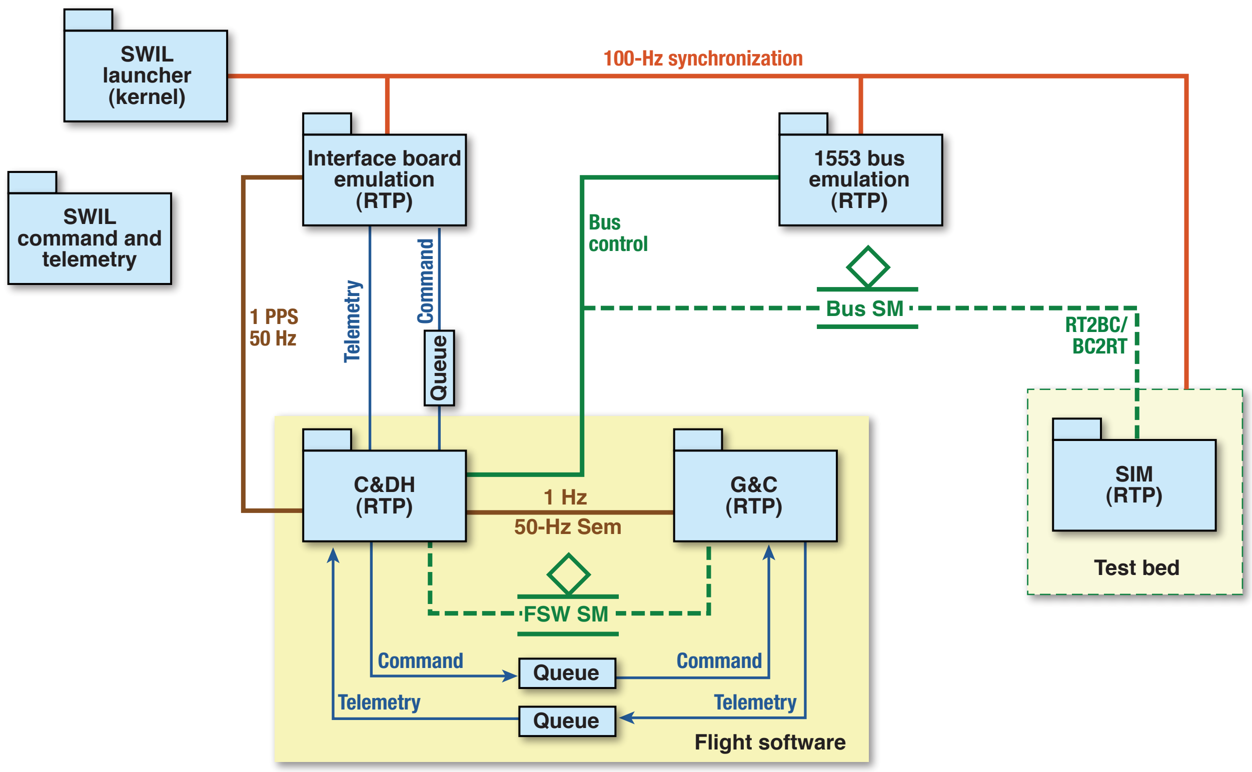


Figure 6. SWIL testing where command and data handling (C&DH) software and guidance and control (G&C) software can be tested against hardware emulations and physics-based simulations all within the flight processor. FSW, flight software; PPS, pulse per second; RTP, real-time process; Sem, semaphore; SIM, simulation (dynamics, environment); SM, shared memory.

SmallSat Software Isolation Architecture

The NASA Small Explorers Office has initiated a study with APL with the purpose of introducing innovative approaches to lowering the cost and improving the performance of 50- to 200-kg spacecraft. The SmallSat study leveraged work done on the RBSP, SWIL, and Naval Research Laboratory initiatives to advance multiple software concepts, including test-once testing, fault isolation and self-healing, and function-level postlaunch modification. Engineers working on this study have demonstrated the combination of memory partitioning with software bus architectures enabling isolated modules to talk to each other without adversely affecting each other. The SmallSat study also demonstrated self-healing, the ability to automatically reboot and restart a single offending application after an error without affecting the rest of the system, through an experiment with a four-module system in which one of four threw an exception every minute for 12 hours. The demonstration proved that the offending module could fail repeatedly without disturbing the other three functions and without growing the memory required for the entire system. Finally, the SmallSat study used the ability to self-heal to load a new application over an existing one, demonstrating a method to achieve function-level postlaunch modification without affecting existing executing applications.

ExecSpec Engine-Based Autonomy

APL has been working on a concept called ExecSpec, which is an engine-based solution to onboard autonomy. In this concept, described in another article by Cancro in this issue, the memory object is a diagram describing how the system should react to faults or operational situations on board. The interpreter is a generic piece of flight code that interprets the diagram on the basis of onboard telemetry and enables a user to load or remove diagrams by command before or after launch. To test memory objects, ExecSpec uses model checking, a test technique that automatically and exhaustively verifies whether the diagram of spacecraft fault responses achieves the written requirements on the system. With model checking, ExecSpec can verify most requirements at a rate of one requirement per second.

ExecSpec provides an example showing that the properties of engine-based design can be used to meet objectives of OSR-1 and OSR-2. APL believes that the engine-based example can be also extended to multiple other applications, including the following:

- Software bus controllers (e.g., a generic 1553 application with a memory object describing all of the bus transactions)
- Downlink data management (e.g., a generic downlink application with a memory object describing rates and quantities of packets to downlink)
- Housekeeping (e.g., generic application that builds housekeeping packets based on memory objects that define the data contents and formats for each packet)
- Observation scheduling (e.g., a generic application that executes time-based schedules based on memory objects that describe observations with types, start times, and stop times)

CONCLUSIONS

APL has returned to the original founding documents of ORS in an effort to understand the real needs of rapid development expressed by the ORS Office. A set of requirements has been gleaned from these documents, and these requirements, in turn, have been used to develop a new software architecture that can achieve the vision expressed in all three tiers of ORS.

APL is pursuing this software architecture across multiple programs. This pursuit is a practical attempt to achieve our rapid development architecture by coordinating multiple achievable steps toward our goal of rapid software development. APL will continue to move forward in this area and hopes that ORS will benefit from the concepts that we have already achieved as well as from the completed architecture in the near future.

REFERENCES

[1]Center, K. Murphy, G., and Strunce, R., "Software as a Tall Poll in Achieving Rapid Configuration and Integration," in *Proc. 3rd Responsive Space Conf.*, Los Angeles, CA, paper RS3-2005-4003 (2005).

[2]Sundberg, K., Cannon, S., Hospodarsky, and Fronterhouse, D., "The Satellite Data Model," in *Proc. International Conf. on Embedded Systems and Applications (ESA'06)*, Las Vegas, NV (2006).

[3]Cannon, S. R., "Responsive Space Plug & Play with the Satellite Data Model," in *Proc. AIAA InfoTech@Aerospace 2007*, Rohnert Park, CA, paper AIAA-2007-2924 (2007).

[4]Lyke, J., "Space-Plug-and-Play Avionics (SPA): A Three-Year Progress Report," in *Proc. AIAA InfoTech@Aerospace 2007*, Rohnert Park, CA, paper AIAA-2007-2928 (2007).

[5]Cebrowski, A., "Statement of the Director of Force Transformation," Statement Before the Subcommittee on Strategic Forces, Armed Services Committee, United States Senate, 25 Mar 2007.

[6]U.S. Department of Defense, *Plan for Operationally Responsive Space: A Report to Congressional Defense Committees*, Department of Defense Report (17 Apr 2007).

[7]Wegner, P. M., and Kiziah, R. R. "Pulling the Pieces Together at AFRL," in *Proc. 4th Responsive Space Conf.*, Los Angeles, CA, paper RS4-2006-4002 (2006).

[8]RBSP website, http://rbsp.jhuapl.edu (accessed 5 May 2010).

[9]Wilmot, J., "Implications of Responsive Space on the Flight Software Architecture," in *Proc. 4th Responsive Space Conf.*, Los Angeles, CA, paper RS4-2006-6003 (2006).

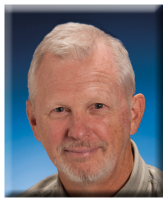
The Authors

George J. Cancro is the Assistant Group Supervisor of the Embedded Applications Group in the Space Department. He holds a B.S. in engineering science from Penn State University and an M.S. in mechanical engineering–astronautics from the George Washington University. Before joining APL in 2002, he worked at the NASA Jet Propulsion Laboratory and NASA Langley Research Center on projects such as Mars Global Surveyor Aerobraking and the Dawn Mission to Vesta and Ceres. Since joining APL, he has worked as a systems engineer on the MESSENGER, New Horizons, and STEREO missions; a project manager for the NASA SmallSat project; and a principal investigator of two research projects in the areas of autonomy and telemetry visualization. He is currently a principal investigator of a research project investigating spacecraft tactical commanding and an advisor to the NASA Constellation program in the area of fault detection, isolation, and recovery. His areas of interest include modular software, hardware/software architectures, fault protection, and spacecraft autonomy. **Edward J. Birrane III** is a Section Supervisor in the Embedded Applications Group in the Space Department. He holds a B.S. in computer science from Loyola College in Maryland (now Loyola University Maryland) and an M.S. in computer science from The Johns Hopkins University (JHU). Before joining APL in 2003, he worked in both the communications satellite and telecommunications sectors. Since joining APL, he has worked as a software engineer and software lead on New Horizons, the Revolutionizing Prosthetics 2009 program, the SMC/SY Integrated Space Situational Awareness (ISSA) program, and NASA efforts to define and implement disruption-tolerant space networks. He is currently pursuing a Ph.D. at the University of Maryland, Baltimore County, in the area of computer networking. His areas of interest include disruption-tolerant networks, real-time operating systems, software reuse and maintenance, and onboard data processing systems. **Mark W. Reid** is a member of the Senior Professional Staff of the Embedded Applications Group in the Space Department. He holds a B.A. in mathematics with a minor in physics from Western Kentucky University and an M.S. in computer science from JHU. Before joining APL in 2005, he worked at the NASA Goddard Space Flight Center and Northrop Grumman Electronic Systems, where he developed flight software for various NASA missions. His experience includes guidance and control, command and data handling, instrument control, power subsystem, and autonomy subsystem software. He has received numerous NASA achievement awards, and his work in software development has been presented at both American Institute of Aeronautics and Astronautics (AIAA) and Institute of Electrical and Electronics Engineers (IEEE) conferences. Mark joined APL in support of the New Horizons mission and is currently the flight software lead engineer on the Radiation Belt Storm Probes mission. He is also a member of the JHU faculty in the Engineering for Professionals Program of the Whiting School of Engineering, where he teaches software engineering management. **J. Douglas Reid** is a member of the Senior Professional Staff in the Embedded Applications Group of APL's Space Department, where he develops and tests guidance and control flight software as well as dynamic and environmental simulation software for missions such as TIMED, CONTOUR, MESSENGER, STEREO, and New Horizons. His interests include astronomy, celestial navigation, arboriculture, entomology, sedimentary geology, and 18th-century history. He received his B.Sc. in applied science from the Royal Military College, Canada, and a master's degree in applied physics from JHU. **Kevin G. Balon** is a member of the Space Department's Senior Professional Staff specializing in embedded flight software engineering. He holds a B.S. in electrical engineering (magna cum laude) from the University of Maryland, College Park, and has pursued additional graduate courses there, completing his coursework toward an M.S.E.E. Before joining APL, as an aircraft avionics flight test engineer he supported development and flight certification of military, commercial, and corporate avionics, including the Enhanced Vision System currently used on Gulfstream aircraft. At APL, he developed C&DH flight software for the MESSENGER mission and is presently the C&DH flight software lead for STEREO. He is currently a principal investigator considering ARINC-653 for spacecraft use. **Brian A. Bauer** is a space systems engineer with the Space Systems Engineering Group. He earned a B.S. and an M.S. in aerospace engineering from Washington University in St. Louis. His area of expertise is in spacecraft fault protection and autonomy. For further information on the work reported here, contact George Cancro. His e-mail address is george.cancro@jhuapl.edu.

George J. Cancro

Edward J. Birrane III

Mark W. Reid


J. Douglas Reid

Kevin G. Balon

Brian A. Bauer