An Environment for Incremental Development of Distributed Extensible Asynchronous Real-time Systems

Charles K. Ames, Scott Burleigh, Hugh C. Briggs
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

Abstract

Incremental, parallel development of distributed real-time systems is difficult. Architectural techniques and software tools developed at the Jet Propulsion Laboratory’s (JPL’s) Flight System Testbed (FST) make feasible the integration of complex systems in various stages of development. In particular, two techniques are used: a strict layering architecture for organization of independent subsystems, and a distributed, low overhead, asynchronous messaging system. These techniques were developed in a few user-months and have proven their usefulness in a spacecraft integration test and simulation environment.

introduction

A goal of the Flight System Testbed (FST) is to generalize and optimize system-level spacecraft interfaces in support of rapid prototyping and integration testing. Using software to simulate the spacecraft and environment at arbitrary levels of abstraction is the theme of this paper.

A snapshot of a spacecraft under development includes a combination of hardware engineering (ICs, breadboards, brass boards, etc.) and software simulations. The FST development environment includes techniques for the smooth replacement of software simulations with hardware or flight software as it becomes available. This facilitates complex, hardware-in-the-loop simulations.

In particular, an architecture was developed to support rapid prototyping of planetary spacecraft systems. This involves encapsulating core on-board services required of any spacecraft (pointing, command handling, telemetry, storage, etc.) and simulating a flight-like environment. Simulating the motion of a spacecraft, or the output of a camera in real-time is a substantial task. Our approach is to collect and integrate simulation systems that model the in-flight environment.

Our simulation architecture is a layered ball's-eyes, or onion pattern reminiscent of traditional uniprocessor operating system design [Tan87]. The center is the system under test. The first layer out is the interface driver layer. This shaded layer presents services to the core system that will remain constant as outer layer simulations are replaced with real devices. For example, in Figure 1 the core cockpit test might be a spacecraft attitude control system, and the first layer out might provide the core system with a gyro_get_state() function that is initially implemented as a remote procedure call (RPC) to a real-time dynamics model simulating spacecraft motion. This function might later be implemented to send a message across a bus to spacecraft rate sensors.

![Figure 1. The layered model.](image)

The outermost layers are virtual subsystems and environment models being used to fool the coresystem. For example, during spacecraft system test, the spacecraft at the core sends thruster commands to turn ("torque") itself, and reads sensors to determine the current attitude for comparison to its desired attitude. It then issues another set of thruster commands to correct remaining error. Thruster commands are "executed" by the dynamics model and sensor outputs are produced by the dynamics model. Because of the inner layers insulating the core of this architecture, the flight software cannot
tell that it is in a test environment and not on interplanetary cruise.

Software development within this architecture is supported by an asynchronous messaging system developed at JPL. This system, Tramel (Task Remote Asynchronous Message Exchange Layer), is based on the techniques underlying Remote objects Message Exchange [ROME1]. Tramel provides application software written in C with a simple and highly portable, platform-independent abstraction for communication among UNIX processes, VxWorks tasks, and Posix threads. In effect, each Tramel-literate process/task/socket (called a "zone") attaches itself to an abstract network of processes (an "application universe") that insulates the application from details of the actual communication network such as processor architecture, operating system, and communication protocol. In addition, Tramel implements a publish/subscribe communication model that further shields application code from having to understand the configuration or state of the distributed application at any time.

All, and only, those zones within a given Tramel universe can use Tramel to exchange messages, and no zone can be in two universes at the same time. To guarantee that no dependencies on virtual subsystems and environment models (which would compromise fidelity) are built into core software, and vice versa, we partition the FST into two application universes. The core software elements inhabit a "flight software universe" and use Tramel only to exchange data among themselves. The virtual subsystems and environment models inhabit a separate "support equipment universe." That is, we build a firewall between the core and the outer layers of the FST architectural bull's eye by mapping them into different Tramel universes and committing to use Tramel for interprocess communication. All communication between the core and the outer layers uses non-Tramel techniques implemented in the interface driver layer of the architecture.

Organization of the Paper

This paper is organized as follows: the next section describes the architecture of the integration and test environment. Within that section are subsections on the primary subsystems. The penultimate section is a discussion of the software support for the architecture previously described. The paper concludes with a summary of lessons learned and topics for future work.

System Simulation Architecture

The spacecraft avionics are surrounded by simulation support equipment, ground data system software, and consoles. Standard network interfaces and bases, commercial real-time operating systems and widely-used languages (C, C++, and 1 abView) mean that the FST environment consists primarily of off-the-shelf commercial products or JPL-developed software using industry-standard techniques.

To illustrate the plug-and-play nature of the architecture, the initial communication medium was ethernet. This was replaced with a MIL-STD-1553 bus without affecting spacecraft or simulation subsystems.

The emphasis on modularity and the low cost of network-ready embedded processors has resulted in a distributed, multi-processor FST. The avionics and support simulation processes themselves typically use multiple processors.

The FST builds on JPL's efforts since 1992 to build prototype spacecraft. A specific example is the Asteroid Comet Moon 1 explorer (ACME) spacecraft studies performed in 1993. ACME is a small, rigid body spacecraft stabilized in three axis using six reaction control system (RCS) thrusters. ACME differs from more complex spacecraft in that hardware redundancy is minimized, for example, attitude control, command processing, and data handling reside on a single processor. This is representative of the smaller spacecraft in JPL's future.

The FST simulation environment provides software and test-equipment support for attitude control, command and data communication, power, telecommunication, instrument, and data recording subsystems. Realistic functional interfaces have been defined and implemented to allow subsystems to be replaced. This results in a testbed with low inter-subystem coupling and high intra-subsystem cohesion. Subsystem simulators are replaced by breadboards, engineering models, and flight hardware as they become available.

Attitude Control Subsystem (ACS)

The design of the attitude control subsystem was based on the Cassini spacecraft's software object architecture. The resulting architecture is a collection of disjoint objects communicating via standard data paths [AB95]. This results in an extensible system in which changes resulting from the incorporation of new technology or growth in capabilities are localized in a few objects. For example, switching from one type of gyroscope to another impacts only the Gyro Manager module.
Figure 2. Attitude Control Subsystem Test

Figure 2 is an instantiation of the onion-layer architecture of the previous figure. In the center is the spacecraft prototype. The enclosing box represents the interface drivers. Simulation subsystems and test equipment are the outer layer of the diagram. The emboldened arcs represent communication triggered by a "point" command initiated from the ground data system.

Evolutionary bus implementations are being evaluated. A MIL-STD-1553 bus has been used as the ACS engineering bus and will be replaced by MIL-STD-1773 (fiber) without impacting the ACS software. The Controller Area Network (CAN) developed by the automotive industry, is being planned as an alternative communication medium. Insertion of these new media was facilitated by the connection-oriented interface drivers layer chosen to hide the details of the underlying network hardware.

The ACS provides core services such as "point and bold" and main engine burn. These services are the foundation for higher level maneuver sequences or data acquisition.

Command and Data Handling Subsystem (C&DI)

The C&DI subsystem consists of a telemetry management system and a command sequence manager for execution of sequences of commands stored on-board. These sequences can be either time or event-driven. Sequences can be parameterized, allowing the implementation of higher level spacecraft functions as macros. This is particularly effective in a prototyping environment where mission scenarios can be accomplished by pasting together core services using scripts.

Dynamics Subsystem.

The spacecraft dynamics simulation is based on the Dynamics Algorithms for Real-Time Simulations (DARTS) software developed at JPL. DARTS provides a library of sensor and actuator modules that emulate the physics of a given sensor or actuator (such as thrust rise times and gyroscope tick counts). The DARTS Shell (Dshell) accepts thrust commands as input, calls the appropriate actuator modules to produce a resultant force vector, calls DARTS to advance the collations of motion by one time step, then calls the appropriate sensor modules to produce sensor readings based on the new orientation of the spacecraft.

In addition to physical models of sensors and actuators, a hardware simulation models the electronic interfaces to specific sensors and actuators, e.g., valve drive electronics, inertial reference units or a MIL-STD-1553 remote terminal. These hardware simulations make it possible to substitute various bus implementations with little or no impact on the rest of the system. For example, TCP/IP Sockets over Ethernet can be transparently substituted for bus hardware during development.

Instrument Simulation.

It is common that after the power, data, and command interfaces to a scientific instrument are determined, development of the instrument proceeds in parallel with spacecraft development. To decouple the development of the instrument and spacecraft, the FST provides a two-part instrument simulation. spacecraft simulation and test can proceed without the instrument present. Figure 3 is the onion-layer model with emphasis on communication between subsystems during test of a simulated or prototype instrument.

![Instrument Simulation](image)

Figure 3. Instrument Simulation

The two parts of the simulation are an instrument simulation and a simulated environment. These two parts are elaborated in Figure 4. The instrument, possibly simulated in software, communicates through the same command-data interface and produces the same power load as the physical instrument will. In addition, the simulated environment takes observational parameters output from the instrument and creates plausible data (images, in some cases) delivered back to the instrument subsystem. The simulated environment can be simple or complex, the latter incorporating position and attitude data from the DARTS simulation, rendered images, backgrounds, derived from star catalogs, typical scene blur provided by an optics model, and simulated detector noise.
Software Support for the FST Environment

The FST software development environment provides several mechanisms that support incremental development, smooth integration, and extensibility. The common foundation of these mechanisms is Tramel, the media-independent message passing system described briefly above. Tramel enables subsystem modules to produce output by publishing messages without knowing which other modules will receive them, and to consume input by subscribing to messages without knowing which other modules will produce them. Each message has a subject, which is an application-selected integer that functions somewhat like a method selector in object-oriented programming language such as Smalltalk, and may also optionally have content, an arbitrarily long string of bytes. A task joins an application universe by registering (basically, declaring some ASCII string to be its name) and after having registered may subscribe to any number of message subjects; different message handlers (callback functions) may be declared for each subject. A task publishes a message by specifying to Tramel its subject, content, and content length; Tramel handles delivery of the message to every subscriber, whether on the same processor or on other processors, using sockets, messages queues, pipes (FIFOs), or whatever other communication channels are available; the publishing task is never aware of the location of the recipient(s) or the transport mechanism(s) used. In this way, the implementation of one module is wholly decoupled from that of any other. The Appendix contains a sample C program that uses Tramel to publish analarm message every sixty four seconds.

One helpful extension of Tramel is Tel Tramel, a Tel [Osu84] application programming interface to Tramel functionality. This library provides Tel commands that record subscriptions and unsubscriptions, and publish Tramel messages. Subscribing to a given subject from within a Tel script automatically links that subject to a callback function that passes the content of each message to a Tel interpreter. This enables applications written in Tel can participate fully in a Tramel application universe.

"fstshell", which is built on Tel tramel, is another useful mechanism for encapsulating modules. By linking with fstshell and invoking its telstart() function, FST application code automatically acquires the ability to interact with other modules in the same application universe and also to be commandable via Tel. Figure 5 contains an example of a high-level FST shell command sequence. Subsystems can exchange commands by publishing messages containing Tel commands and subscribing to the commands published by other subsystems. This makes integration of new, higher level functionality simple.

For example, an optical pointing module can be added simply by having it subscribe to image messages from the camera and publish "point" commands. Encapsulating subsystems behind FSTshell facilitates distribution of functionality over processors -- an instrument pointing module can be transparently moved to another processor.

[Conclusion]

The FST employs several mechanisms which facilitate spacecraft subsystem integration and test and also provides an environment for demonstration of new technology in an end-to-end system context. The FST provides core services under a layer of higher level functions that enable integration of new technologies.

These technologies might include new hardware as well as more abstract functionality implemented in software. Hardware such as instrument interfaces, data compression engines, or ACS devices can be quickly integrated using commercial hardware while boards are being implemented in programmable gate arrays. Complex software entities, such as Tel-based logic engines, can be layered quickly behind the reusable subsystem interfaces and demonstrated interacting with other software components.

Future work includes formal specification of the Tramel messaging functions. Reverse engineering of formal specifications from existing spacecraft code has been demonstrated [CA93], and distributed asynchronous messaging schemes similar to Tramel have been formally specified [AK86].

The techniques described in this paper were developed...
by an eight person team in about six months. The resulting architecture is used as demonstration of spacecraft technology in a flight-like environment. Experience with extending the architecture and using it for integration of new technology support the conclusion that these techniques are robust and suited for incremental development of large, complex, distributed real-time systems.

Acknowledgments

The work described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Dr. Auernheimer's work was supported through a NASA/A SII Summer Faculty Research Fellowship and a sabbatical provided by California State University, Fresno.

Bibliography


Dear Author,

Congratulations on the acceptance of your paper for publication in the proceedings of the Workshop on Parallel and Distributed Real-Time Systems, being held April 15-16, 1996, in Honolulu, Hawaii. PLEASE READ THE FOLLOWING COMPLETELY BEFORE FORMATTING YOUR PAPER - THIS LETTER CONTAINS IMPORTANT INFORMATION!

Attached to this message is a set of electronic instructions for formatting your paper. If you have any formatting/publication questions, please feel free to contact me (numbers below). For registration, program, or other questions, please contact either Lonnie Welch at Ph: + Intl. 540-653-2193 or e-mail: welch@homer.njit.edu; or David Andrcws at Ph: + Intl. 501-575-5090 or e-mail: dla@engr.uark.edu.

- GUIDELINES/FORMATTING - These instructions have been updated. Please see the attachment: INSTRUCT.TXT. You should be able to format your paper successfully if you use the measurements in the attachment.

- PAGE LIMIT - Short papers: 4 pages. Extra pages are allowed, at a cost of $100.00 US EACH. Please make your check payable to: WPDRTS 96 and include it with your paper. [Do NOT make it out to any other name. The conference "does not accept credit card payments and extra pages not paid for will not be published.] If you need a receipt, complete the receipt request form and include it with your check and the original manuscript that you send to me. See attachment: RECEIPT.TXT

- COPYRIGHT release - print out the form, fill it out COMPLETELY -- BE SURE TO INCLUDE THE TITLE OF YOUR PAPER, THE AUTHORS' NAMES, etc. -- Just signing it is NOT enough! Send it in with your paper -- we cannot publish your paper without it. See attachment: COPYRIGHT.TXT

- MANUSCRIPT - I[ease submit an **ORIGINAL** of your paper. We must have the best quality possible. We will accept PostScript copies of your paper with the provision that they are not encapsulated, that you send them to me with the subject referenced as WPDRTS paper, and that you send them in promptly. If you send a PostScript paper, you still **MUST** send in your copyright form (fax it to me first, then send the original to my attention), plus your check for any extra pages. If you use LaTeX formatting macros, we have an updated set available and ask that you use it and not any older versions you may have. Please request it by sending "me an e-mail.

- ELECTRONIC ABSTRACT of your paper - Only E-mailed abstracts will be accepted. Send your abstract to: WPD96-abs@computer.org.