The Impact of Autonomy Technology on Spacecraft Software Architecture: A Case Study

Edward B. Gamble, Jr., California Institute of Technology, Jet Propulsion Laboratory
Reid Simmons, School of Computer Science, Carnegie Mellon University

THE AUTHORS DRAW ON THEIR EXPERIENCE WITH THE AUTONOMY TECHNOLOGY DEMONSTRATION ON NASA'S DEEP SPACE ONE MISSION TO DESCRIBE THE WIDE-RANGING EFFECT AUTONOMY WILL HAVE ON THE DEVELOPMENT OF SPACECRAFT SOFTWARE.

AUTONOMY TECHNOLOGY FOR high-level, closed-loop control of spacecraft offers considerable benefits to space-flight projects. Those benefits can enable whole new classes of missions; however, they are not without cost. In this article, we describe both the impact that autonomy technology has on spacecraft software and the implication for the software architecture that arise from those impacts. Some of the impacts are inherent in the challenging problems generally confronted in the spacecraft domain yet are exacerbated by autonomy technology.

Overview

To increase reliability and enable new missions, spacecraft systems are moving toward more autonomous operations. In particular, today's spacecraft designers need autonomy in cases where long communication time delays make command and monitoring operations by ground-based operators infeasible. By analyzing data and making more decisions on board, spacecraft can handle contingencies quicker and more intelligently, can take actions based on current data, and can react effectively to new opportunities.

Spacecraft designers also need new approaches to spacecraft software design and implementation for handling the added functionality provided by autonomy technologies. For instance, in contrast to traditional spacecraft software, the computation time of many autonomy technologies cannot be predicted precisely, so the process scheduler must be more flexible. Similarly, software that can react to unexpected contingencies and opportunities will likely have many branch points, so brute-force, exhaustive testing is not feasible. These, and many other factors, complicate the design, implementation, and validation of autonomy technologies.

We contend that a well-conceived software architecture can have significant positive impact on the development of autonomy technologies and on the ability to integrate them with traditional spacecraft functions, such as attitude control and telemetry.

While attitudes toward software architectures are sometimes dismissive—"it's just boxes and arrows"—recent research has laid a rigorous foundation for the field of software architectures.[1] The analysis we present here, while not formal, attempts to discuss in depth the constraints that the domain of deep-space missions place on the design and implementation of autonomy technologies and, similarly, describe how the addition of high levels of autonomy affects the overall software architecture.

The term *software architecture* encompasses several different notions. For our purposes, we consider only architectural *style* and *structure*. Architectural style refers to computational concepts that can be uniformly applied throughout the system. For instance, one system might be designed in an asynchronous, publish-subscribe style, while another uses a more synchronous, client-server model. An architecture's structure refers to its decomposition into component parts. This includes specifying the functional behavior of individual components and interfaces between components, to indicate both information and control flow. For hierarchically structured systems, the architecture specifies the levels of abstraction and how the various layers are decomposed.

A well-defined architecture gives the software developer a number of advantages in design, implementation, testing, and maintenance. From a software design perspective, both the style and structure of an architecture are very important. The architectural style often greatly influences how components behave and interact with one another. For instance, a publish-subscribe style has implications for the need to maintain internal state and how a component reacts to unexpected inputs.[2] A well-defined architectural structure lets us develop systems in a modular, distributed fashion, with the expectation that integration will be facilitated by careful detailing of functionality and interfaces.

From an implementation point of view, architectural styles can make it easier to implement a design by providing standardized languages, code libraries, and tools tailored for the particular style. In essence, these languages, libraries, and tools encapsulate patterns of computation that the particular architectural style needs. Common examples include languages and tools for object-oriented programming and code libraries for interprocess communication (message passing). For spacecraft architectures, other types of packages might be used to support telemetry, exception handling, sensor management, and so forth. In addition to simplifying implementation, such tools can facilitate validation and verification, because they themselves are often reliable and well-tested.

Finally, a well-structured architecture can facilitate system maintenance. Because the function of, and interfaces between, components are well-specified, we can replace components without fretting about how the replacements will affect the rest of the system (note, however, that this is less true for tightly coupled systems, which is the case for many spacecraft systems).

In essence, architectures provide constraints on how problems should be approached and solved. In any given architecture, some things are hard to do or express, and some are easy. A good architecture should make it simple to do simple things in the domain, while not precluding one from doing more complex tasks.

Challenges of spacecraft autonomy

In general, building autonomous systems is a challenge. Autonomous systems must create and execute plans of action to achieve high-level goals, while also maintaining the ability to react, in real time, to unexpected contingencies. Often, they must choose between conflicting goals and deal with resource conflicts.

The domain of deep-space missions adds several constraints that make spacecraft autonomy particularly challenging. These constraints relate to the facts that it is very expensive to launch mass into space; that the space environment is harsh, the destinations distant, and the scientific missions exacting; and that spacecraft are highly complex and sophisticated mechanisms.

A GOOD ARCHITECTURE SHOULD MAKE IT SIMPLE TO DO SIMPLE THINGS IN THE DOMAIN, WHILE NOT PRECLUDING ONE FROM DOING MORE COMPLEX TASKS.

Limited resources. Probably the most significant factor in controlling deep-space spacecraft, either manually or autonomously, is that resources are very limited and so must be used with maximum efficiency. For deep-space missions in particular, the maximum available electrical power typically cannot run all the spacecraft's devices simultaneously. Usually, to run one piece of equipment (such as a science instrument, motor, or transmitter), other devices must be turned off. A similar situation occurs for the solid-state recorder (storage) devices, which for deep-space missions generally have insufficient capacity to hold all the relevant science and engineering data simultaneously.

Fuel (for propulsion) is another highly limited spacecraft resource. This case is somewhat different from the power situation in that the fuel resource is finite—after it is used up; thruster or main engine firing is no longer possible. Decisions about when and how to use fuel (such as for course corrections or orbit insertions) must be made in such a way as to guarantee that sufficient reserves will be available for later parts of the mission, including unexpected contingencies.

Finally, computation itself is a limited onboard resource. Spacecraft computers typically lag several generations behind commercial products, both due to power limitations and because onboard CPUs and memories must be radiation hardened, an expensive and time-consuming development process.

Reliability. Given the vast distances involved, deep-space missions tend to be very long. The Pathfinder mission to Mars was a relatively short hop—six months travel. More typical are missions such as Cassini (six *years* to Saturn). During this extended time, the probability of unexpected and unanticipated events is very high. Problems can arise from hardware or software failures, either transient or permanent, caused by design flaws or unexpected environmental conditions.

An autonomous spacecraft system must be able to detect any and all such problems and deal with them, at some level. This can range from the basic response of achieving a *safe* spacecraft state (where resource use is minimized and Earth telecommunications is enabled), up to autonomously dealing with the situation. Recovery strategies must consider how the solutions will affect other spacecraft activities. For instance, if the spacecraft is in the midst of a critical orbit-insertion activity, achieving a safe spacecraft state is not an option.

Another factor complicating recovery from failures is that, to save cost, many spacecraft have limited hardware redundancy. Thus, when problems occur, the only available solution might lead to reduced system functionality. For instance, if a thruster fails, and the spacecraft has no back-up thrusters, the system will have to operate in a degraded mode, perhaps with reduced turn rate and stability.

Spacecraft dynamics. Spacecraft are constantly moving and events are continually occurring that must be handled in a timely fashion. To complicate matters, in many situations we do not completely understand the spacecraft dynamics. This is particularly true for new types of space missions, such as landing on comets or asteroids.

The most critical challenge imposed by spacecraft dynamics, however, is that actions might have irreversible effects. For instance, if a spacecraft misses an orbit insertion, it gets no second chance. Often, spacecraft systems must take likely contingencies into account. For instance, in doing an orbit insertion, the autonomy system should consider the possibility that the main engine will fail and include commands to prepare the backup engine (such as by preheating its components) well in advance. Otherwise, by the time it discovers that the main engine has failed, the system will not have enough time to prepare the backup.[3]

Science mission. Fundamentally, deep-space spacecraft are science-delivery platforms. It does not matter if the autonomy system successfully controls the spacecraft for 90% of the time if it fails to acquire the anticipated science data. Given the nature of science opportunities, a deep-space mission is one of long stretches of relative quiet punctuated by short periods of intense activity. In such situations, there are usually more science opportunities than the mission can accommodate, so spacecraft systems must be able to prioritize actions and use resources effectively in these critical periods.

Also, by its very nature, scientific discovery is unpredictable. Analysis of data leads to new insights, but also leads to new questions to be answered. Unexpected opportunities often arise that are of immense scientific interest (witness the Levy-Shoemaker comet, the discovery of volcances on Io, or the observation of the moon Dactyl orbiting about the asteroid Ida). It is important to scientists that the spacecraft be able to readily adapt to new mission goals.

Sociological issues. Like many complex systems, spacecraft systems are designed, developed, and validated by large teams of technical experts. The teams might be spatially dispersed and certainly have a wide variety of backgrounds. Concepts and terminology must be shared across groups and be easily accessible to all.

Autonomy technologies also affect ground operations. Ground-operations personnel have deep expertise in controlling, monitoring, and diagnosing spacecraft. They are used to being able to predict, to a very high level of detail, how the spacecraft will act. Thus, it is important that the autonomy technologies be seen to have predictable behavior, at least at some level of abstraction. In addition, ground personnel must be able to easily assume control when the autonomy software does not operate as anticipated.

The remote agent architecture

The Remote Agent is an autonomy technology that NASA will demonstrate on the Deep Space One mission. DS1, which will include an asteroid and comet flyby, is the first in a series of technology demonstration missions within NASA's New Millennium Program. Other deep-space missions in the NMP include a Mars surface penetrator (DS2), multispacecraft interferometry (DS3), and a comet lander mission (DS4). Because these NMP missions are designed for technology demonstration, they deemphasize science objectives and so can tolerate significantly more risk. On DS1, for example, there are 12 technologies for demonstration.

The RA demonstration on DS1 occurs over a two-week period during a thrusting-cruise mission phase. On DS1, thrusting cruise requires

- navigation, to measure the relative position and velocity between the spacecraft and the target asteroid,
- attitude control, to stably direct the spacecraft toward the target and orient the spacecraft for ground communication, and
- propulsion, to accelerate the spacecraft along its path to the target.

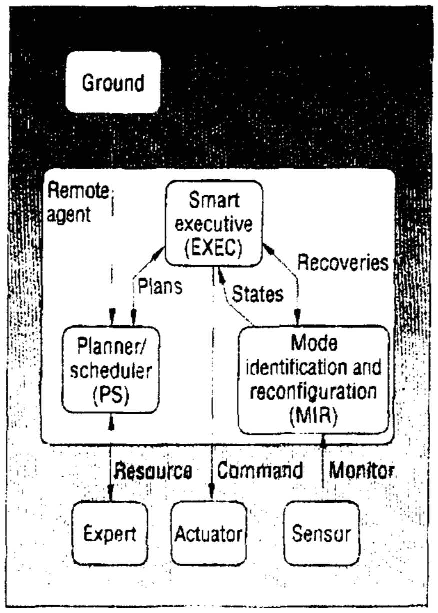

Figure 1. The Remote Agent components.

Within the two-week period, the RA demonstration will run twice. The first 12-hour phase is designed to allow the DS1 ground operations team to gain confidence in the RA technology. In this phase, the spacecraft will have continuous antenna coverage so that were anything to go wrong the ground team could easily reestablish spacecraft safety. Also, planning will not be performed onboard; rather, a fully validated plan will be uplinked to the RA and executed by the EXEC. In the second six-day phase, onboard planning is performed with goals designed to achieve thrusting cruise, as itemized above. During the both phases, faults are artificially injected into the RA so as to demonstrate closed-loop control and the resulting robust execution.

The RA components. The RA consists of three major components.[4] The *Smart Executive* (EXEC) robustly executes plans and fault recovery strategies, monitors constraints and runtime resource usage, and coordinates the top-level commanding loop. The *Planner/Scheduler* (PS) merges ground-supplied mission goals with the current spacecraft state and produces a coordinated set of time-delimited activities that the EXEC performs. The *Mode Identification and Reconfiguration* (MIR) component performs model-based fault diagnosis based on the monitored spacecraft state and, when requested by the EXEC in response to a fault, provides plausible commands to recover to the desired state.

The three RA components are closely coordinated (Figure 1). In the top-level command loop that supports autonomous planning, the EXEC builds plan requests based on the current spacecraft state, issues a plan request to PS (which then merges the state with the mission goals), receives a reply from PS with the completed plan, and robustly performs the activities coordinated by the planner. If a fault occurs, the EXEC builds a recovery request with the desired spacecraft state, requests a recovery from MIR that will restore the desired state, receives a reply from MIR with a command for recovery (if one exists), performs the recovery, and continues with the plan activities.

The RA components interact with actuators, sensors, and planning experts. *Actuators* perform actions based on commands.

The actuators themselves can be as low-level as a single switch or as high-level as the attitude control system (where a typical command would be to orient that spacecraft at a given celestial body). *Sensors* provide data that can be combined in numerous ways to ascertain the state of the spacecraft. *Planning experts* provide estimates of resources (electrical power, thruster propellant, turn duration) required to perform spacecraft actions.

Standard flight software components. Standard *flight software* (FSW) for deep-space missions (those beyond lunar orbit) typically have an onboard, open-loop, temporally-based command sequencer. Given a list of time-tagged commands, the sequencer issues the commands at the specified times without regard to their results. Only after the results have been telemetered to Earth can ground personnel ascertain the sequence's success. Flight software engineers achieve robust execution not with runtime flexibility but rather by detailed hand-crafting of the sequences based on accurate spacecraft-performance models and by extensive tests in suitable test environments.

At a very abstract level, standard FSW consists of drivers, managers, and subsystems (see Figure 2). *Drivers* are essentially the software device drivers that operate at the level of bus transactions (memory accesses, for example). *Managers* encapsulate bus communication (so that devices on different hardware buses use a similar software interface), produce telemetry, utilize nonvolatile memory for persistent parameters, close some hard real-time loops, and generally map to a single device driver. Managers provide an increment in the software's hierarchical structure because there are usually some interdevice interactions. *Subsystems* coordinate activities involving several managers and close higher-level control loops. The flight software does not explicitly represent all subsystems. Subsystems such as navigation, propulsion, and science generally require ground personnel for closed-loop control (although for DS1, one technology scheduled for validation is onboard, autonomous optical navigation). Other subsystems, such as attitude control (ACS) and fault protection (FP), are explicitly represented and do exhibit closed-loop control.

The fault-protection subsystem is charged with ensuring the spacecraft viability. If the spacecraft state indicates an off-nominal, dangerous situation, fault protection issues commands to restore the spacecraft to a safe state where fuel and power resouces are maintained, where thermal constraints are satisfied, and where communication with Earth is established. Many of the commands issued will not be confirmed, so robust execution relies on the detailed crafting of the command sequences.

The attitude-control subsystem must stably point the spacecraft and its devices toward desired celestial targets. For solar-electric powered spacecraft, the solar panels must face the sun (except when alternate power, such as a battery, is available); cameras must point toward planets for science observations; and antennas must aim at the Earth or other communication sites, such as a lander.

INDEED, THE SEVERE RESOURCE CONSTRAINTS ON DEEP-SPACE MISSIONS SUGGEST THAT A SINGLE ARCHITECTURE MIGHT NOT BE SUFFICIENT FOR THE VARIETY OF MISSIONS.

Closed-loop control is possible because stellar-reference units and sun sensors, combined with spacecraft geometry measurements and trajectory projections, provide the relative positions and orientations between spacecraft hardware components and the celestial targets. Rotation rate and acceleration sensors provide the basis for precise, stable pointing.

RA on DS1 architectural issues. The RA on DS1 provides onboard, high-level, closed-loop control of a spacecraft, while also replacing the standard FSW components of fault-protection, command sequencing, and portions of ground-based planning. Some architectural issues arise from both the particulars of the RA and the fundamental nature of closed-loop control. (The discussion of these issues that follows is based on architectural issues that applied prior to March 1997 when the RA was the nominal control subsystem for DS1. The post-March redirection, whereby the RA was rescoped as a two-week experiment, had architectural implications that we do not describe here.)

Weak coupling. The RA architecture is fundamentally weakly coupled. The architecture achieves this weak coupling with publish-subscribe, query-response message passing for all communication within the RA and between the RA and the DS1 flight software. (Communication between the RA and DS1 flight software changed after the March 1997 redirection.) The need for a weakly coupled architectural style arose from a number of goals:

- *Minimally impact FSW components.* Because the RA is functionally similar to standard ground-based command sequencing and onboard fault protection, the FSW components need only provide additional feedback directly to the RA, rather than into the Earth-bound telemetry stream. The distinction between the dynamic command sequencing performed by the RA and the time-tagged commands in standard FSW does not significantly affect FSW command interfaces.
- *Enable concurrent, largely independent development teams.* Interfaces were specified based on little shared code other than the messaging infrastructure. Thus, there is little serialization of the development process while waiting for a large amount of infrastructure to be provided. Instead, the development teams develop their own infrastructure, customize it for the team's own needs, and proceed with the work of providing their component's functionality.
- *Support multiple implementation languages.* While the non-RA flight software is implemented in the C programming language, the RA is implemented in CommonLisp (Harlequin, Inc. supplied the flight version). The choice of Lisp was based both on its natural support for RA technologies and the fact that the RA prototype had been written in Lisp (and the compressed development schedule of DS1 prevented significant rewrites of existing code).

The DS1 development software team originally believed that a weakly coupled architecture supporting these three goals would enable the DS1 RA to be produced on the compressed schedule and within the cost-capped limitations that face DS1. However, not all spacecraft autonomy architectures should be weakly coupled. Rather, the limited resources on spacecraft suggest that strong coupling, while maintaining other architectural features such as modularity, is highly desired.

Interface specification. The interface specifications between the RA and the standard FSW on DS1 is highly. By *recursive*, we mean that two subsystems need to know about the interfaces of the other. For example, the RA commands attitude control and attitude control responds by invoking sensor update functions specified in the RA interface. Avoiding recursive interfaces is important for modularity concerns in light of closed-loop control and can be achieved with simple polling or *callback* patterns. *Nonfunctional requirements.* Nonfunctional requirements are those system requirements not *directly* related to system functionality.[5] These requirements capture the shared context within which development will occur. They include shared models, tools, and code. Mainly as a consequence of the weakly coupled architecture, RA development did not incorporate many of the relevant nonfunctional requirements. For instance, in the area of models, the various RA components used inconsistent state transition diagrams; in the area of tools, different C and Lisp compilers were used; in the area of code, multiple versions of the same functions were implemented. This lack of adhering to nonfunctional requirements significantly affected DS1's development. The impact was felt mainly during the software-integration process, which quickly exposed inconsistent assumptions.

Architectural impacts and implications

The challenging nature of spacecraft and the ambitious goals of autonomy systems significantly affect spacecraft software. Some impacts arise primarily from spacecraft's challenging nature but are exacerbated by autonomy systems. Others are due primarily to autonomy itself. In the following, we describe the impacts and implications that autonomy can have on spacecraft software. We don't intend, however, to provide architectural solutions to all the impacts described. Indeed, the severe resource constraints on deep-space missions suggest that a single architecture might not be sufficient for all types of missions. Highly customizable and reusable architectural frameworks are an active topic in spacecraft software design.

Modularity. Autonomy affects modularity because autonomy systems tend to employ a global view whereas modularity demands that software components can be individually designed, developed, and delivered,[6] which is a highly local view. For example, a high-level autonomy system such as MIR uses inputs from numerous, sensors, models of spacecraft devices and systems, the environment and its physical interactions, and inference engines to infer some aspects of the spacecraft's state. Modularity, on the other hand, dictates that spacecraft devices and systems should contain their own models, and their detailed operations should be opaque to higher-level systems.

Consequently, the key to achieving modularity in light of autonomy is to split the autonomy system into several parts:

- a low-level part that contains models and data—provided by device engineers,
- a high-level part with configurability, commandability, goals, and global models—provided by system engineers, and
- a skew level with the inferencing engine—provided by autonomy software engineers.

Visibility. Autonomy systems affects software visibility by requiring that state and behavior information be both present and accessible in the software for use by the autonomy software's deliberative components. For example, when a planner is involved, it is often not enough to have a software component that runs a state-transition diagram. Rather, the state-transition diagram needs to be explicitly represented and thus accessible to the planner. Visibility is essentially the opposite of encapsulation—increased visibility tends to expand interfaces by making information more accessible through the interface.

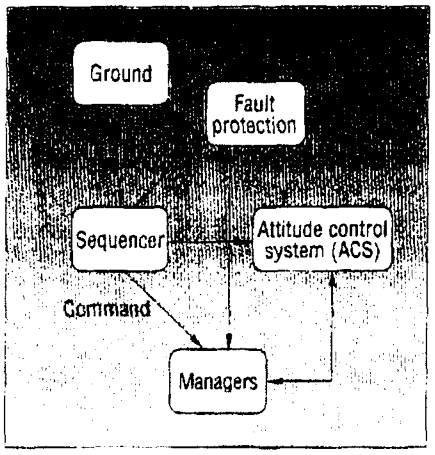

Figure 2. Standard flight software components.

Increased visibility therefore suggests architectures that support model-based, declarative programming. The models both constrain and formalize the visibility (thereby preserving modularity and encapsulation), while declared-model instances maintain the information about state and behavior in an accessible form.

Configurability. For reasons of reliability and resource constraints, spacecraft require high levels of configurability. Three general areas of configurability are considered here: fault protection, resources, and structure. In general, autonomy systems affect configurability because such systems need to have a global view of the lower-level software.

Fault protection. Autonomy systems affect fault-protection configurability because an appropriate exception handler is likely to be selected based on global information and because the number and scope of the provided exception handlers needs to increase to support enhanced robustness. For example, a star tracker might produce bad data because either the tracker itself has failed or because the communication bus (to or from the tracker) has failed. Global information about whether other devices on the same bus had similar problems could resolve the ambiguity and point to the proper recovery strategy. Configurability implies that a particular exception handler need not be statically chosen at design time but can be installed at runtime based on spacecraft state or mission phase.

The software architecture thus needs to support the dynamic association of exceptions with exception handlers. This needs to be done respecting modularity, however, which requires that exceptions and exception handlers be defined locally (only the component can know the context in which an exception happens and which particular handlers apply).

Resources. Autonomy systems affect resource configurability because resources must be explicitly represented to allow optimal runtime allocation and plan-time deliberation. Resource allocation can be of many types—the most common is for the resource to be used exclusively by a specified subsystem. Other resource types include *nego-*

tiated or *prioritized* resources. As with exception handlers, configurability implies that the basis for allocation need not be statically chosen and might be based on both local and global information.

Consequently, the software architecture should explicitly model spacecraft resources and provide resource managers that serve as the brokers for resource-allocation requests and as the protectors against resource oversubscription.

Structural. Autonomy systems affect structural configurability because global spacecraft models can better isolate failed components and because robustness in the face of faults requires that components be readily replaced. Structural configurability allows for different components to be employed in response to different requirements, faults, or resource limitations. An example is to replace a faulty physical gyroscope with an inferred-state, virtual gyroscope.

A wider variety of spacecraft software components thus need to allow their structural subcomponents to change dynamically. Large numbers of plug-compatible components should be provided to facilitate robust operation. This also has implications for visibility, because the different components are likely to have different models, in terms of accuracy, resolution, functionality, etc.

Asynchronousity. Spacecraft, by their nature and in spite of our best engineering efforts, are event-based, asynchronous systems. Faults occur, and those faults are always asynchronous. Sensor states change based on actuation; those changes are asynchronous owing to various sources of indeterminacy. The environment is unpredictably rife with interesting scientific events like volcanic explosions that are always asynchronous.

Therefore, accurate software modeling of this inherent asynchronousity requires that the software use event-based multiprocessing with a preemptive scheduler. Event-based processing ensures that external events map accurately into software events. Multiprocessing is required because events arise independently and simultaneously from different sources. Preemptive scheduling acknowledges that some events are more important than others and that there cannot be one processor for each source of events.

Dynamism. Spacecraft exist in a dynamic environment and are expected to perform different activities, derived from high-level goals, at different times. Examples of deep-space mission activities are launch detumble and checkout, thrusting cruise, and encounter.

The software architecture thus should avoid, where possible, a priori limitations on the computational resources allocated to any particular component. Like other resources, computational resources should be explicitly modeled and allocated dynamically.

Integration. Autonomy systems affect integration because autonomy technology fundamentally provides closed-loop control, which requires lower-level software to provide feedback for the autonomy system. This introduces a necessarily tight coupling between the higher- and lower-level components, making it difficult to develop and test components independently.

SPACECRAFT EXIST IN A DYNAMIC ENVIRONMENT AND ARE EXPECTED TO PERFORM DIFFERENT ACTIVITIES, DERIVED FROM HIGH-LEVEL GOALS, AT DIFFERENT TIMES.

Consequently, the software architecture should forbid recursive interface definitions, and avoid modeling of low-level components within higher-level components. Doing this, and having well-specified interfaces, increases the probability of producing fully functional, modular, tested components for the integration process.

Tools. A good set of tools can make any software system easier to develop, test, and maintain. This is particularly true of autonomy technologies, which systems typically have complex interactions among components. For example, configuration management tools can be used to increase the likelihood that changes in one part of the system will be propagated to other affected parts. Visualization tools can help in understanding system behavior. This can be difficult to do manually, due to large amounts of data and the asynchronous nature of the processing. For instance, for DS1, we developed tools to visualize the message traffic between components and to visualize the execution of plans.[7] These tools let users easily get gestalt views of the overall system behavior and interactively investigate particular problematic aspects.

Interactions between components should thus be made explicit, to delimit the scope of changes. Accessibility to the runtime execution of components is important for visualizing and analyzing system behaviors. Ideally, components should log and timestamp all their state transitions to provide a complete picture of the system execution.

Testing. The expense of spacecraft failure dictates that spacecraft software be thoroughly tested prior to use, which is typically several times more time-consuming and expensive as creating the software initially. The added complexity and functionality of autonomy technologies compounds this problem. There are aspects of the software architecture, however, that can facilitate this task.

Unit testing. As discussed earlier, autonomy systems tend to be highly coupled. Thus, to thoroughly unit-test an autonomy component (such as the EXEC), system designers need to know not only how it functions internally, but also how it responds to the behavior of other system components. Thus, they must often be able to embed the component within the system, which is generally not possible because the various system components are being developed concurrently.

The architecture thus should allow system designers to *stub out* individual components and replace them with functionally equivalent (but simpler) modules. A message-based publish-subscribe architecture, such as is used on DS1, makes it relatively easy to replace one component with another.

Simulation. Because spacecraft hardware is a scarce resource, and is typically not even available during software development, adequate hardware simulations are essential to testing. Different components, however, have different simulator requirements. Some, such as ACS, need to simulate dynamics; for other components, kinematic simulation suffices. In spite of this, all simulations should be based on consistent hardware models.

Therefore, a single, multiresolution simulator should be used, allowing scaling of the simulation's fidelity. The simulator should also support fault injection and the ability to dynamically change the scale of simulated time. The latter is crucial in testing mission-level aspects of the autonomy system, because deep-space missions are typically characterized by long periods of inactivity in which little of interest to higher-level components occurs..

Formal Verification. A spacecraft can encounter a huge number of possible scenarios—much larger than can be tested by trial and error. Formal verification methods can be used to significantly reduce the develop-test-debug cycle for complex spacecraft systems. For instance, in DS1 aspects of the EXEC inference engine were model using temporal logic and verified using model checking.[8] Similarly, a designer can formally represent the domain models created by the spacecraft developers (for example, the models used by PS or MIR) and verify properties of the models (such as that the MIR models will not exhibit false positives or false negatives).

To apply such formal methods, software components must have well-specified semantics and explicit requirements and specifications (so they can be checked automatically). The desire for formal verification also places constraints on the models used by the autonomy technologies – languages that are too expressive may be difficult, if not impossible, to verify automatically.

THE IMPORTANCE OF AUTONOMY technology for present and future generations of spacecraft cannot be overestimated. High-level, closed-loop control based on sophisticated model-based, fault-tolerant, configurable, and dynamic software architectures will let NASA pursue space missions that for technological or financial reasons it could not otherwise attempt. Spacecraft systems that exhibit a significant amount of autonomy have the potential of being both more reliable and more powerful than those based on standard flight software.

As the rescoping of the RA technology demonstration on DS1 painfully suggests, however, autonomy technology has a significant effect on spacecraft software that should not be ignored. While many of the effects exist to a lesser degree in current software architectures, they are exacerbated by autonomy systems. Awareness of these impacts and a willingness to reexamine the shortcomings of both spacecraft software architectures and autonomy implementations will be a welcomed, needed result of the DS1 experience.

Acknowledgment

The research we have described was carried out by the Jet Propulsion Laboratory, California Institute of Technology, Ames Research Center, and Carnegie Mellon University under contracts with the National Aeronautics and Space Administration. Thanks go to the members of the DS1 Remote Agent team, for developing and integrating the autonomy technologies, and to the anonymous reviewers who provided insightful comments on an earlier draft of this article.

References

1. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, New York, 1996.
2. K. Lieberherr and I. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, Vol. 6, No. 5, Sept. 1989, pp. 38–49.
3. B. Pell et al., "An Autonomous Spacecraft Agent Prototype," *Proc. Autonomous Agents '97*, ACM Press, New York, pp. 253–261.
4. D.E. Bernard et al., "Design of the Remote Agent Experiment for Spacecraft Autonomy," *Proc. IEEE Aerospace Conf.*, IEEE Press, Piscataway, N.J., 1998, pp. 259–281.
5. *Developing Object-Oriented Software: An Experience-Based Approach*, Prentice Hall, 1997.
6. G. Booch, *Object-Oriented Analysis and Design*, Benjamin Cummings, Menlo Park, Calif., 1994.
7. R. Simmons and G. Whelan, "Visualization Tools for Validating Software of Autonomous Spacecraft," *Proc. In'tl Symp. AI, Robotics, and Autonomy in Space (i-Sairas)*, JSME Press, Tokyo, Japan, 1997, pp. 39–44.
8. M. Lowry, K. Havelund, and J. Penix, "Verification and Validation of AI Systems that Control Deep-Space Spacecraft," *Proc. 10th Int'l Symp. Methodologies for Intelligent Systems*, Springer-Verlag, New York, 1997, pp. 35–47.

Edward B. Gamble, Jr., is a member of the Advanced Multimission Software Technology group at JPL. He received his bachelor's and master's in electrical engineering from UCLA. His doctorate was awarded in electrical engineering with a specialty in artificial intelligence from MIT. His interests have ranged from laser scattering in fusion plasmas and in critical phenomenon, to computational vision and integration of sensory information, as well as to programming languages and real-time systems. He is currently involved in spacecraft software architectures for reuse and autonomy. For the Deep Space One Remote Agent experiment, he is a deputy project element manager and the team lead for both the flight software interface to the Remote Agent and for the Smart Executive team. He is a member of the IEEE. Contact him at JPL, 4800 Oak Grove Dr., Pasadena, CA 91109-8099; ed.gamble@jpl.nasa.gov.

Reid Simmons is a senior research computer scientist at Carnegie Mellon University's School of Computer Science. He earned his BA from SUNY at Buffalo, and his MS and PhD from MIT in artificial intelligence. His research has focused on developing self-reliant robots that can autonomously operate over extended periods of time in unknown, unstructured environments. The research involves issues of robot navigation, planning and reasoning under uncertainty, and control architectures that combine deliberative and reactive control, selective perception, and robust error detection and recovery. He is a member of the IEEE, AAAI, Phi Beta Kappa, Sigma Xi, the New Millennium Autonomy IPDT, and was program chair of the 1998 International Conference on AI in Planning Systems. Contact him at the School of Computer Science, Carnegie Mellon Univ., 5000 Forbes Avenue, Pittsburgh, PA 15213; reids@cs.cmu.edu; http://www.cs.cmu.edu/~reids.