Trusted Autonomy for Spaceflight Systems

Michael Freed
NASA Ames Research Center

Pete Bonasso
NASA Johnson Space Center / TRACLabs

Michel Ingham
Jet Propulsion Laboratory

David Kortenkamp
NASA Johnson Space Center / TRACLabs

Barney Pell
NASA Ames Research Center

and

John Penix
NASA Ames Research Center

In most cases, intelligent control technologies that could enable spaceflight systems to operate autonomously or with reduced human supervision are seen as unacceptable sources of risk and not flown. This paper examines the reasons for lack of trust in these technologies and overviews efforts to enhance the technology and to define systems engineering practices that improve reliability and earn trust.

I. Introduction

NASA has long supported research on advanced automation technologies that could allow space systems to operate autonomously or with reduced human supervision. Proposed applications include increasing automation in complex critical systems (e.g. life support) that otherwise require vigilant monitoring, intelligent mobile robots that assist or substitute for astronauts and automated control of entire space vehicles. The potential for such technology to extend the kinds of missions that are possible is well understood, as is its potential to reduce costs and improve the robustness, safety and productivity of diverse mission systems.

Despite its acknowledged potential, advanced automation is rarely used in spaceflight systems. There are a few exceptions. For instance, one well-known example is the Remote Agent eXperiment (RAX) [Bernard *et al.*, 1998] flown on the Deep Space One (DS-1) mission. Another is the Advanced Sciencecraft Experiment (ASE) used on the Earth Orbiting 1 satellite [Chien *et al*., 2004]. However, these are only partially exceptions. RAX, originally intended to have full control of the spacecraft for the duration of the mission, was reduced to having partial control for a two-week non-critical period. ASE was allowed onboard only in a post-mission phase – i.e. one the satellite had completed its nominal mission. Nonetheless, these cases are exceptional. In most cases, mission managers consider intelligent control systems an unacceptable source of risk and elect not to fly them. Overall, the technology is not trusted.

From the standpoint of those who need to decide whether to incorporate this technology, lack of trust is easy to understand. Using advanced "intelligent" automation means allowing software to make decisions that are too complex for conventional software. The decision-making behavior of these systems is often hard to understand and inspect, and thus hard to evaluate. Moreover, such software is typically designed and implemented either as a research product or custom-built for a particular mission. In the former case, software quality is unlikely to be adequate for flight qualification and its functionality is typically driven, at least in part, by the need to publish

innovative work. In the latter case, the mission represents the first use of the system, a risky proposition even for relatively simple software.

In general, trust requires confidence in the software artifact itself, the processes and organizations that determine how the software is maintained and improved, and the affordances the software offers to both developers and other mission personnel that allow its behavior to be understood, tested, critiqued and communicated. Perhaps the single most effective source of trust in complex software is a long history of use, preferably in diverse applications by numerous, distinct users. Long deployments provide time to shake out bugs, improve performance and refine the system. They provide opportunity to analyze performance and failure modes, study and try alternative designs and compile documentation of users' experiences, all providing concrete evidence of software reliability. Long deployments also lead to more and more people becoming familiar with the technology, thus providing a source of technical guidance and staffing for new deployments. The paper will overview work addressing the problem of trust in autonomy, focusing on factors peculiar to space systems.

II. The Case for Advanced Automation in Spaceflight Systems

All NASA spacecraft employ automation. *Advanced automation* is distinct in that it allows goal-based commanding of system activities. Goal-based commanding can be understood in contrast to, e.g., timed action-sequence commanding traditionally used to control unmanned spacecraft such as planetary rovers and deep space probes. For example, a sequence for changing spacecraft attitude might include the command "at time 13d:12h:05m:27.2s turn on thrusterA" followed by "at time 13d:12h:05m:29.5s turn off thrusterA." Most deployed spacecraft control systems today operate on the basis of such sequences which provide no support for context-sensitive or conditional activity. In contrast, goal-based commands are expressed in terms of what should be achieved instead of what actions should be taken and when. A goal-based equivalent to the above sequence might be "during the interval in which science-target T is being imaged by camera-1, attitude should be f(T)."

The goal-based approach has important advantages over the traditional time-based sequencing approach. For instance, a goal-based controller can achieve a desired result when conditions differ from what was anticipated when the goal was initially formed. It could thus compensate if, e.g., attitude had drifted from the expected position or had been changed intentionally in response to some unanticipated problem or opportunity. Similarly, such a system could determine that no thruster action is needed if desired attitude had already been achieved as a result of past actions. Using traditional time-based sequencing, discrepancies between expected and actual conditions require abandoning the nominal sequence and "safing" into near-inactivity until a new command sequence is received from Earth. The traditional approach thus depends on accurate predictions of execution conditions. This can be accomplished to good approximation for, e.g., deep space probes during their cruise phase, where most change can be accounted for by the equations of celestial mechanics. In environments containing humans, complex systems and dynamic natural processes that reduce predictability, the adaptability and responsiveness provided by goal-based control becomes a necessity.

At a more general level, the expected benefits of increased spacecraft automation are the same as for virtually every modern industry: lower cost, higher productivity, higher reliability and making it possible to do things that would otherwise be impossible or impractical. In the case of spaceflight, it is helpful to distinguish benefits that are possible if the automation software is located on ground (Earth-based) systems from those that require the software onboard.

Automation software resident on ground systems offers a number of potential advantages in comparison with humans carrying out the same decision-making or control functions. First, machines are better and faster calculators than people. They can make decisions faster, operate on fresher data, operate on more data, consider more scenarios, and perform more computationally intensive calculations with fewer calculation errors. These can translate to faster and higher quality responses to threats and opportunities, reducing risks and increasing mission performance. In addition to direct increases in performance, automation can provide added reliability that makes it feasible to reduce safety margins, benefiting performance indirectly.

Second, automation reduces dependence on people in potentially advantageous ways. Reduced dependence can pay off as reduced staffing and training cost or as a means of freeing people up for different work. In addition, automation doesn't switch jobs or get sick, bored or tired, yielding 24x7 operations and higher utilization. Lastly, fewer human interfaces can reduce vulnerability to diverse forms of human error and to safety and security hazards introduced by people. Third, automation can increase scalability since the software can be replicated, allowing lower amortized cost for new systems and making it possible to deploy these systems in large numbers.

Incorporating automation into onboard systems can provide additional benefits. As compared to ground-based automation, onboard automation can reduce or remove requirements for communication between ground and

onboard systems. This frees up communications infrastructure (e.g. deep space networks, relay satellites, onboard antennas, transmitters and receivers) and may eliminate the need for it entirely. This, in turn, can reduce costs and risks associated with designing, deploying, maintaining and managing complex communication infrastructure.

A second benefit to onboard automation is to remove communication delays and restrictions. For planetary and deep space missions, removing speed-of-light delays permits faster responses to threats and opportunities than would be possible with ground-based automation. Removing communication restrictions resulting from occlusions and potential system failures enables continued operation in these conditions, thereby removing constraints on sphere of operation and enabling response to failure in conditions where human intervention is not possible.

Finally, onboard automation can perform actions that would otherwise be performed by crew. One potential benefit is to free up crew to perform scientific or exploration activities, instead of maintenance and operations, giving increased crew utilization and mission return. Similarly, it can reduce crew size requirements. This has substantial benefits, including reduced costs from carrying people (human weight, size, life support, waste management, and resources). In some cases, the requirement for human crew can be eliminated – a desirable situation for missions that pose excessive risk to astronauts, take place over especially long periods or would be impractical from a cost perspective if human support- and return-infrastructure were required.

III. Software Reliability in Infrequently Used, Hard to Test Systems

The first source of concern in using advanced automation is that it is founded on complex software, a notorious source of space mission failure. Software, unlike hardware, does not wear out. It fails due to design errors not detected during testing. However, testing intelligent control software for spaceflight systems presents significant challenges. In comparable terrestrial systems such as the flight automation used on commercial aircraft, extensive flight testing is conducted prior to operational use. Such testing is infeasible for space systems. Once fielded, additional design flaws in commercial flight automation inevitably emerge. However, this rarely results in catastrophe due to large safety margins, the presence of highly competent human pilots who can take over from the automation at a moment's notice and the low cost of aborting a flight if there are significant problems. Space environments are far less forgiving and space flights less far less frequent, so operational testing cannot play as central a role in engineering space systems.

The technology underlying advanced automation poses additional software reliability challenges. For example, the adaptable nature of some of these systems means that a single test environment may not accurately reflect the system's behavior in the deployed environment. As a result, the range of possible testing scenarios becomes extremely large, reducing the effectiveness of traditional testing [Havelund *et al.,* 2000]. In addition, autonomous systems typically require real-time processing with asynchronous communications between cooperating tasks. This raises particular concerns about race and deadlock conditions that are difficult to discover in traditional testing-based verification and validation (V&V).

A. Verification and Validation

Several large-scale case studies have been conducted to evaluate analytic verification methods applied to intelligent automation. The first of these studies applied a model checking method to a model of the resource management architecture from the Remote Agent Executive for DS-1. This resulted in the detection of several concurrency errors that had not been discovered during testing [Lowry *et al.,* 1997]. In fact, an error of this sort occurred in flight during the DS-1 mission (in a different part of the software), causing the executive component of the controller to deadlock. A follow-on study indicated that increased tool automation would have reduced the cost of applying model checking and very likely enabled this error to have been detected prior to flight [Havelund *et al.*, 2001].

More recently, a study funded by NASA's Intelligent Systems Program examined the relative effectiveness of several V&V technologies on the executive control component used on the K9 rover [Brat *et al*., 2004]. The study consisted of a controlled experiment in which static analysis [Brat and Klemm, 2003], runtime analysis [Havelund and Rosu, 2001] and model checking [Visser *et al.,* 2003] were compared to traditional testing with respect to error detection. The results showed that these tools outperform traditional testing, each in its own way, especially when detecting concurrency errors.

The results of the experiment inspired a novel framework for testing the K9 Executive [Artho *et al.,* 2003]. Runtime analysis, which monitors temporal relationships between software events during testing, was successful in detecting errors when given test cases that provide relatively good coverage of the test input space. Model checking, a process that uses exhaustive execution of software or system models to provide extensive test coverage, could

systematically cover all input plans up to a specific size for the rover. These two approaches were combined in a V&V framework which uses model checking to create all plan structures (up to a specific size) as test inputs and runtime analysis to monitor temporal formulas that describe correct plan execution. This combination has been very effective in automating V&V for the K9 executive.

While it is clear that analytic V&V tools will provide improvements in the reliability of advanced automation, software architecture design can either enhance or impede the effectiveness of these tools. Studies on modular (or compositional) verification techniques, which break the verification tasks into smaller sub-tasks and then properly combine the results, show that software design models can provide early, and therefore more cost-effective, V&V of critical software design properties [Giannakopoulou *et al.*, 2002;]. In addition, the results of the design verification can be used to effectively decompose the code-level V&V into more tractable analysis problems [Giannakopoulou *et al.*, 2004; Cobleigh *et al.,* 2003]. Based on these results, reaserchers have begun to assemble a set of design patterns that increase the verifiability of critical software properties [Mehlitz and Penix, 2003]. An initial experiment showed that the use of an event queue to coordinate K9 executive tasks provided a more verifiable architecture and eliminated several concurrency errors.

Although intelligent automation software presents some new challenges to the software V&V problem, its rigorously architected nature and reliance on domain model representations presents an opportunity for more thorough validation than has been possible using traditional ad-hoc control software designs [Feather *et al*., 2003]. For example, to provide the flexibility and reliability required for reconfigurable autonomous systems, program synthesis (automated code generation) can be combined with V&V technology to reduce software defects. Integrating an advanced automation architecture with real-world robotic systems and hardware involves specification of functional components and associated behavior models. Code generation techniques to synthesize components and models are needed to ease integration and enable on-site reconfiguration. Automatic formal V&V methods can leverage knowledge of the code generator and the autonomous control software framework to ensure the correctness and robustness of new components [Denney *et al., 2004*].

B. Engineering Domain Knowledge

Intelligent automation software typically combines two complementary components. The first provides general-purpose reasoning and control capabilities not tied to a particular space vehicle, system or mission. This component, the "engine," may incorporate, e.g., search algorithms used for planning [Chien *et al.*, 2000; Jonsson *et al.*, 2000], temporal constraint propagation algorithms used for scheduling [Tsamardinos *et al.*, 1998] and model-based reasoning algorithms used for fault diagnosis and recovery [Williams and Nayak, 1996]. The second component is "domain knowledge" specific to the missions and controlled systems for which the automation is designed. This may include, e.g., domain-specific heuristics used to guide search, mission activity models used by a planning and scheduling mechanism, and spacecraft subsystem hardware behavior models used for model-based diagnosis and recovery. Trust in the automation depends on the belief that this knowledge has been specified correctly. This underscores the importance of engaging personnel with appropriate expertise (system engineer, subsystem engineers, scientists) during the design and development of the intelligent automation software, and of performing thorough V&V on the models to assure their completeness and correctness.

Part of the challenge is to bridge the gap in expertise between technologists who understand the intelligent control engine and specialists in the controlled system(s). The technologist's role centers most naturally on the design and prototype development of the general-purpose engine. Final implementation and integration into the mission software may be assigned to the project software engineering team, in order to retain confidence that all mission software meets the project's criteria for software quality and engineering practice. Domain experts have a more significant role to play in the design and implementation of the domain model software. System engineers, subsystem engineers and scientists are the authoritative sources for domain knowledge in their respective domains. The question remains, how is this knowledge imparted to the software? Whose responsibility is it to translate domain-specific models into application software for use by an intelligent automation engine?

One option is to have the technologist (or software engineer) perform this translation. This option becomes necessary when the domain model must be represented using a complex, difficult formal notation in order to be usable by the engine component. In this case, the technologist is in the best position to understand and manipulate the formalisms to capture the appropriate domain-specific behavior. Unfortunately, the technologist's likely unfamiliarity with the details of the domain area introduces risk of model inaccuracy. Moreover, this approach is really only effective once the system design has matured and the required automation behaviors are well-understood by the domain experts. This means delaying development of the domain model software development to later phases in the project lifecycle when design iterations are typically most costly and most difficult to carry out. An example of this type of implementation strategy was adopted for the DS-1 Remote Agent Experiment (RAX). The

domain models used by the various components of the Remote Agent software (Planning and Scheduling, Execution, and Mode Identification and Reconfiguration) were developed by the corresponding autonomy software researchers working in concert with the DS-1 spacecraft system engineers [Bernard *et al.*, 2000]. Much iteration was required to correctly capture the system behavior, some of which was as a result of changes in the system engineers' understanding of the system and its operational requirements.

A second option is to have domain experts translate their domain-specific knowledge into domain models, allowing these experts to be directly responsible for its correctness. However, as noted above, this presents a significant challenge if domain modeling involves complex formalisms as system/subsystem engineers and scientists are not generally trained in the use of such formalisms. Similarly, if domain knowledge is incorporated directly into running code, requiring experts to specify the knowledge requires a great deal of oversight by mission software engineers to ensure that best software practices are applied, and that appropriate care is taken in the development of real-time embedded software. This type of implementation strategy is common in the area of guidance, navigation and control (GN&C), where estimation and control algorithms (e.g., Extended Kalman Filters and attitude controllers) are frequently developed by the GN&C analysts themselves, with some help and oversight from the flight software team.

Recent and ongoing research in the field of "executable models" opens up a third strategy for the implementation of domain model software. Given that domain experts already frequently use expressive design representations to capture their domain knowledge (e.g., StateCharts [Harel, 1987] used to depict behavioral models of component hardware, UML models of software behavior), it makes sense to implement general-purpose automation engines that can directly parse, interpret and reason about these representations. One of the important findings from RAX was that it would be desirable to have domain experts encode the domain models directly. A concluding recommendation from the RAX Final Report was to "develop tools and simplify the modeling languages to enable spacecraft experts to encode models themselves," to "employ tools and languages already familiar to the experts," and to "organize the models around the domain (attitude control, power, etc.)" rather than around the RAX technology components. The *model-based programming* paradigm [Williams *et al.*, 2003], for example, has been developed as an attempt to leverage modeling representations that are already familiar to domain experts. Another option is to use augmented procedure-based formalisms that embed model information need by the reasoning engine. This builds on the naturally intuitive quality of procedural formalisms, allowing reliable specification of complex behaviors by individuals lacking expertise in formal reasoning methods [Freed *et al*. 2003].

Separating the automation software into engine and model components has many benefits: once an appropriate general-purpose engine has been developed and validated, it can be reused from mission-to-mission, leading to potentially significant cost savings in software development. Once the engine has been validated, the recurring software V&V problem is effectively reduced to the problem of model validation. Though still a significant challenge, this presents fewer difficulties than code validation in general as the relative readability and inspectability of domain models allows experts to confidently inspect, discuss and critique this knowledge. While this is a very promising area of ongoing work, a complete validation of the approach via full-scale demonstration in a mission context has yet to be done.

IV. Variable Autonomy – Minimizing the Need for Leaps of Faith

Decisions about how much automation to incorporate are a fundamental part of system design. An enormous literature exists to provide guidance on such decisions, define tradeoffs and warn of potential pitfalls. In one particularly well known example [Sheridan 1992], design options are defined by 10 degrees of automation which range from "the computer offers no assistance: humans must do it all" to "the computer selects and executes the task, ignoring the human." Other work has focused specifically on automation in spaceflight systems with the goal of developing a methodology for making these design decisions for given missions and systems [Proud, Hart and Mrozinski, 2003].

Looking at level of automation as exclusively a design-time decision is not appropriate for spaceflight systems. Mission managers will typically choose to operate with minimal automation in certain conditions, especially in the early stages of a mission or system deployment when uncertainty about system performance is greatest. Trust in the automation (and in the system as a whole) should increase with time and experience, especially if the system supports fine control over the degree of autonomy exercised by the intelligent controller.

For example, the traditional approach to spacecraft control is based on timed command sequences. A mission manager might find it acceptable to allow flight automation the flexibility to adjust the timing of actions in the sequence. Such a change has important advantages and significant operational consequences [Pell and Shafto, 2004]. One can imagine a sequence of small steps leading to complete autonomy: from flexible timing to flexible

ordering – i.e. allowing the automation to reorder actions if appropriate; from there to allowing the automation to decide between alternative methods (sets of actions) for achieving a goal. Next, mission personnel might allow the automation to decide which goals to pursue and which to abandon. Next, they could allow it to interrupt/resume actions and insert behaviors to smooth the transitions. At each step, the intelligent controller is allowed more degrees of freedom in selecting and organizing its activities. And at each step, its behavior becomes less predictable, but more adaptive, more responsive and more suitable for complex responsibilities. Providing such fine control in the executive allows gradual increases in trust to translate into small increments in operational autonomy.* The alternative is to require mission managers to make large leaps of faith, an approach that is at odds with the challenges and practice of developing spaceflight systems and one that has met with little past success.

The term *variable autonomy* refers to the ability of intelligent control software to support changes in degree of automation. More specifically, the goal of a variable autonomy software architecture is to allow systems to operate with dynamically varying levels of independence, intelligence and control. A human user, another system, or the autonomous system itself may adjust the system's "level of autonomy" as required by the current situation. A system's variable autonomy can involve changes in:

- The complexity of the commands it executes and the constraints it enforces.
- The resources (including time) consumed by its operation.
- The circumstances under which the system will either override or allow manual control.
- The circumstances under which the system will request user information or control.
- The number of subsystems that are being controlled autonomously and how they are coupled.
- The allocation of responsibility to select, sequence, perform or terminate tasks among systems and users.

Variable autonomy provides the flexibility to adapt to changing environments, user goals, and resources. This flexibility is important in order to successfully deploy autonomous system solutions for highly complex real-world problems. For example, advanced life support systems for space habitats will need to operate autonomously in most conditions as vigilant monitoring of these systems would be a poor use of astronaut time (and also intolerably dull). But reduced autonomy would likely be required in off-nominal and maintenance conditions. Another context in which variable autonomy is likely to prove critical is in coordinating the behavior multiple systems (e.g. a group of mobile robots). A human operator might normally supervise the group at a high level, allowing each system or entity a great deal of autonomy, but take direct control of an individual to carry out a difficult task or handle an emergency. In each case, the control software may be completely in control or it may be supervising manual control or it may be somewhere in between. The key goal of variable autonomy is to minimize the *necessity* for human interaction, but maximize the *capability* to interact.

A. Variable Autonomy Issues

Effective variable autonomy requires an autonomous control system that can perform routine operations autonomously yet give control to a human to perform specialized or difficult operations. The advantage of a variable autonomy system is that people's unique (and in space applications, expensive) capabilities can be brought to bear when needed most and not during tedious, repetitive and routine operations. However, a challenge for variable autonomy is that the control software cannot always confirm what the state of the world or of the task is when the human finishes his or her portion of the job. This can make it difficult and dangerous for the system to resume autonomous operation. As an extreme example, the human may forget some crucial aspect of their portion of the task (such as turning a valve) that the system is expecting to be accomplished. Less drastic, but probably more commonplace would be subtle side effects of the human's performance such as putting a tool in a slightly different orientation than is expected by the robot. In either case, the problem is that the system's models of the world and of the task are not consistent with the real state of the world and of the task.

Effective variable autonomy also requires that the autonomous control system know when a human should be performing a task and when it can safely perform a task. Ideally, the control system would plan both its own activities and those required of the human with whom it will be trading control. In this case, the control system would proceed autonomously until reaching the point where human intervention is required; the system would then inform the human and safely wait until the human is ready to accept control. The control system should also recognize situations when the prescribed control is not effective and stop to ask for human assistance even if human

* Note that increments in level of autonomy needed to gain trust should not be expected to yield continuous benefits such as reduced cost or increased productivity [Hawken *et al.*, 1999].

assistance was not originally required for the action. The control system would then need to account for the human intervention and replan its task.

Just as there are issues in designing the control software to accommodate interaction with a human, there are also issues in supporting the human's understanding of when and how to interact. Even when operating in a fully autonomous mode, the user should be able to quickly assess the state of the system being controlled as well as the control activities required to achieve the goal state. When problems occur, the user should be able to determine what went wrong, how the problems affect ongoing and planned activities, and whether manual action is required. When novel opportunities occur, the user should be able to deviate from the encoded control scheme temporarily. When performing joint human-computer tasks, even closer task coordination is required. The human and autonomous system must maintain a shared understanding of the state of the overall task, to ensure that each performs assigned activities in a correct and timely manner. These requirements to assist humans interacting with autonomous systems affect the design of the control software. New types of information must be made available to the user as well as new ways of influencing control activities. This influence can range from changing the parameters or instrumentation used for autonomous control to issuing control commands. These requirements must be met without impacting the ability of the control software to achieve real-time control performance.

B. Variable Autonomy Principles

Through experience we have begun to identify several principles for building effective variable autonomy systems. These principles include:

- Retrofitting manually controlled hardware for variable autonomy is costly and time-consuming. If variable autonomy is desired then hardware systems must be designed with this goal in mind. Ideally hardware and software control systems should be designed at the same time.
- Sensing is necessary for any sort of closed-loop control, but especially critical for variable autonomy systems where the control system is expected to make higher-level decisions. In particular, there must be sufficient and appropriate sensor coverage to capture the relevant state of the environment and the system.
- The previous two principles lead naturally to another: The designers should strive for the highest level of autonomy possible, and fall back on human capabilities where necessary. This will lead to a natural understanding of the limitations of the system, and highlight where and how improvements in technology could be used to improve the system.
- "Cognizant failure" and "conditional execution" are necessary capabilities at all levels. Cognizant failure refers to the control system's ability to recognize when it is failing at a task and take appropriate action, such as trying another strategy or asking for help. Conditional execution is the requirement that the system verify that certain preconditions are met before proceeding with an action.
- An effective variable autonomy system needs a component capable of planning and resource management. This component must keep track of constraints and preconditions, despite possible changes in level of autonomy or the trading of control between the human and the control system. To the extent possible, planning for human intervention at the appropriate times simplifies the coordination of human and autonomous activities.
- An intelligent user interface is required that can combine information from various levels of abstraction and interpret it for the user. Conversely, if the user decides to take control, the interface must provide the ability to control the system at various levels of abstraction. The interface should provide pertinent information at a fast enough rate such that the user can spot trouble in time, and should quickly alert the user when the control system needs help.
- Control autonomy may need to be adjusted at different levels of abstraction for different control situations. When changes in control initiative can be anticipated, the planning capability can designate the level of autonomy via allocation of tasks to either human agents or autonomous software agents. When changes in control initiative are in response to a control situation, the executive capability can provide alternative methods for a human or autonomous software agent. When changes in control initiative are in response to a novel or unexpected situation, the human may need to change the level of autonomy to manage the situation manually.
- When possible, the design of the automated software should include a means of determining the state of the environment that can be altered by manual tasks. One of the more effective approaches is to execute manual tasks through the automated control software, when manual tasks are amenable to computer-based control. We have used other strategies, including monitoring states during manual tasks and updating states after manual tasks complete. In cases where the effects of manual tasks cannot be sensed directly, indirect

means of determining state (e.g., inferred from other activities, requesting confirmation from the user) may be required.

- The user interface to assist joint task performance between a human and autonomous software agent should represent data about the state of the task shared by all participating agents. Information needed for such a view includes explicit representation of tasks, the agent responsible for a task, and the expected effects of the task.
- This common grounding makes it easier to hand over tasks among agents and assists in detecting when tasks do not have the desired effect (mismatch between expected and actual task effects). The interface should also provide a means for exchanging information and coordinating activities (such as notification of pending manual tasks).

Variable autonomy is necessary for any application of autonomous control technology that needs to interact with humans. Humans who rely on the autonomous control system will want to be able to take control of it at various times and at various levels. They will also want insight into what the system is doing even when all is going well.

V. The Test of Time – Developing Proven Technologies for Intelligent Control

People are understandably averse to having complex software used for the first time on their mission in a mission-critical role. Intelligent control software is thus more likely to be trusted and used if it has proven itself in other applications, preferably including applications similar to a given proposed use and preferably over an extended period. Long deployments help build trust in the underlying automation framework for a variety of reasons. They provide time to shake out bugs, improve performance and refine the system. They further provide opportunity to analyze performance and failure modes, study and try alternative designs and compile documentation of users' experiences, all giving concrete evidence of software reliability. Long deployments also lead to more people becoming familiar with the technology, thus providing a source of technical guidance and staffing for new deployments. In this section we present a case study of such a long-duration deployment, and discuss factors that have emerged as affecting the level of trust in the autonomy software.

A. Case Study: Intelligent Control for Advanced Life Support

A good example of extended use of intelligent control technologies is the use of 3T [Bonasso *et al*., 1997] to provide automation for an advanced water recovery system (AWRS) at NASA's Johnson Space Center over a seven year period (from 1998 to the present. 3T consists of three "tiers" of software. These combine to form a powerful automation capability whose flexibility and complexity present trust-related challenges representative of those we expect for intelligent automation in general.

The top tier of the 3T autonomous control architecture is a hierarchical task net planner that specifies tasks required to achieve control objectives and designates those tasks for execution. The middle tier, a reactive executive, provides high-level control services that translate planner-generated tasks into low-level "primitive" actions at runtime. Services include decomposing tasks into subtasks based on stored procedures, sequencing actions based on priority and other factors, recognizing and responding to anomalies and ensuring that each action's timing requirements are met. Actions selected by the executive are carried out by "skills," software modules that transform primitives specified by the executive into continuous control and monitoring activities at the hardware level. The set of skills used in a given application constitute the third tier of the architecture.

3T applications run autonomously due in large part to the principle of "cognizant failure" [Gat, 1998] embodied in each level of the architecture. The skills level notifies the executive when it loses any of the states it must achieve; the executive uses alternative procedures when the primary methods fail, ultimately putting the control system in a safe state. The planner can synthesize alternative plans in light of the failures of the lower two tiers.

Prior to supporting the AWRS developments, the 3T team from JSC's Automation and Robotics Division had applied the architecture several times in support of limited tests controlling a portion of the life support control system [Shreckenghost *et al.,* 1998; Lai-fook & Ambrose, 1997]. While this familiarized some users with 3T, there were different user teams for each test and therefore no continuing corporate memory of the advantages of intelligent control software

With the advent of advanced water recovery systems developed in 1998 in support of the space station, the 3T team became an integral part of the advanced life support team. This provided an opportunity to develop a complete suite of control software from the ground up. In the summer of 1999 the bottom two layers of 3T were used to provide autonomous control for a single subsystem -- a second-generation biological water processor -- during a

450-day 24/7 test. Then in January 2000 the advanced water research group received ALS funding for the year long AWRS test, involving four advanced water recovery subsystems: 1) a biological water processor (BWP) to remove organic compounds and ammonia; 2) a reverse osmosis (RO) subsystem to remove inorganic compounds from the effluent of the BWP; 3) an air evaporation system (AES) to recover additional water from the brine produced by the RO; and 4) a post processing system (PPS) to bring the processed water to within potable limits. The intelligent control system had to handle upwards of 200 sensors and actuators, run 24/7 and be completely autonomous. The 3T team provided intelligent control technologies for the AWRS from buildup through the long running testing phase until April 2002. From then until the present they have continued providing 3T control for the follow-on PPS Optimization Tests.

The use of the 3T system for control of the AWRS was a resounding success for applied AI. The resulting software ran unattended for 98.75% of the test period (6684 of 6768 hours), averaging on the order of only 6 hours downtime per month. In an environment where experimental hardware is being tested, this achievement is especially notable and is directly attributable to the hybrid deliberative-reactive nature of the control design.

A more subtle but longer lasting effect was also achieved: the life support engineers changed their modus operandi from one of vigilant monitoring of life support systems to intermittent remote supervision. At the beginning of the test, the using engineers required a human in attendance overnight to monitor the control system. After three months of operation, the life support group determined that the system could be run overnight unattended. This shift to depending on advanced software to manage life support is a clear result of the users learning to trust the intelligent control system.

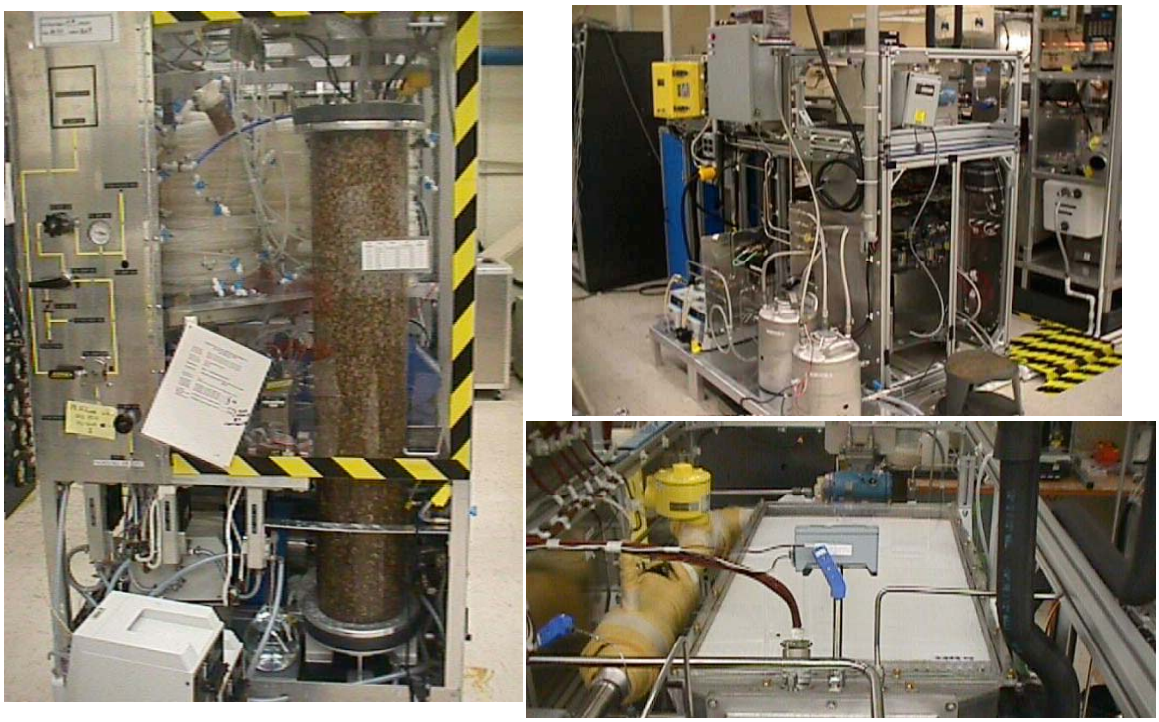

Figure 1. The AWRS subsystems. At the left is biological water processor (BWP). Upper right is the rack containing the Reverse Osmosis (RO) subsystem in the rack bottom, the air evaporation subsystem (AES) at the top of the rack, and the post processing system (PPS) in the rack's left rear. The bottom picture is a close up of the wick in the AES.

B. Factors Leading to Trusted Autonomy

A number of developments occurred during this experience that helped engender trust in ALS engineers.

1. Adjustable Autonomy

Although the 3T controls for WRS were able to run with full autonomy, during hardware build-up, functional testing, and for the first three months of operation, test engineers wanted to be able to command the system or its subsystems at all levels of operation. The 3T team provided the test team with interactive interfaces used for 3T code testing. These interfaces included commands for activating individual pumps, valves and relays, for executing agent-level procedures such as executing an RO purge, and to start or stop the autonomous operation of any subsystem, such as running the BWP in a stand-alone mode.

Being able to suspend parts of the control system's autonomy was important as well. For example, mid-way through the test it became necessary for the BWP engineers to manually purge the individual tubes in the nitrifier portion of the BWP. This purging often resulted in a low-pressure condition that would trigger a low-pressure automatic shutdown (ASD) from the BWP skills. To prevent the ASD during staff purge operations, the 3T team modified the ASD procedure to check the state of a memory flag for staff purging. When the flag was present, the BWP agent would put out the ASD warning but would take no action. Interactive text was added to the WRS display that could be triggered to set the staff purge flag in memory and start a twenty-minute timer. At the end of the twenty minutes, the timer code would remove the flag and the ASD would function normally.

2. Equipment Degradation

By their very nature, life support systems are long running, carrying out their prescribed processing for weeks or months. Anomalies are rare, but when they occur they must be detected and processed to prevent often catastrophic results. This long duration characteristic has several trust implications. The twelve months of WRS operation saw the slow degradation of pressure transducers, flow meters, a dew point sensor, the AES blower and the main RO feed pump. Sometimes the failure brought the test to a halt; at other times, the degradation was gradual and difficult to detect, generating intermittent symptoms. Degradation sometimes occurred over a number of months. Neither the water team nor the controls engineers had the experience to determine if the various problems stemmed from software or hardware. There were few utilities in place to help the team capture intermittent events, so a great deal of time was spent adding trace code and studying the results. After about six months, the test team's familiarity with each subsystem grew to the point where the causes of these anomalies were easily ascertained. For the PPS Optimization Tests, the 3T team has developed and is using a complex event recognition subsystem, which recognizes the structure of anomalies and tracks their substructures to the underlying logged data. With the controls team aiding the test team through these problems the test team was able to better understand the limits of the intelligent control system and to ask for specific modifications that would improve its contribution to the test.

3. Safety Shutdowns

No matter the number of precautions taken to prevent system failure, there will always be the possibility of variables outside the bounds of the intelligent control system. Chief among these were network problems and power failures. Every six weeks or so, over the course of a twelve-month test, the control system would experienced random faulty data packets coming from the wider JSC network. These would produce a data set that would cause the executive to stop communicating with the skills. This event inevitably occurred after the last nightly check by the control engineers, and before the laboratory personnel arrived in the water lab six hours later. With the executive down, its messages would build up in the communications server and after about an hour, the communications server would crash, bringing down all skills connected to it.

When the skills died they left the last settings on the pumps and valves. In one instance, the failure caused a pump to drain a gas-liquid separator and to push air into the ion-exchange beds of the PPS, resulting in a need to repack the bed. In another instance, the gas-liquid separator was pumped dry by the RO and the RO drew air at high pressure into its membranes, rendering them useless.

The solution to these network failures was to make the skills aware of the loss of communications with the executive and execute a safing of their respective subsystem. To this end the 3T team developed watchdog timers in the skills layer. If the elapsed time since the last executive communication was greater than a predetermined time, say five minutes, the skills would put the subsystems in a protected state, e.g., the AES would turn off its heaters and the condensate pump; the BWP would reconfigure itself to stand alone and turn both the feed and effluent pumps off. The watchdog timers, instituted soon after the restart of the first test point, protected the WRS from network failures through out the remainder of the test. The ease with which the skills could be modified contributed to building trust in the intelligent control technologies.

4. Logging State Speeds Restarts

Loss of power to the water facility occurred several times during the course of the test. In these instances, the valves would remain in their last commanded state but all pumps would go to an off state. The primary dilatory effect was the loss of the bacteria colonies in the BWP if the feed water was not restored to the BWP in a timely manner. The watchdog timers took care of a power loss to the executive computer, but for a full-power loss, the life support engineers eventually learned to resuscitate the bacteria if the time lapse was less than six hours.

However, with the loss of power as well as the numerous times the WRS had to be halted due to hardware failure -- about twenty-five times during the course of the test -- it became important to be able to restart the system quickly, without having to manually determine the system state before the power loss or the hardware failure. Thus, the 3T team wrote a procedure to log the internal state of the agents every thirty seconds. When the staff restarted the system, the executive read the last logged state of all the subsystems and the current state via the skills then determined how to resume operations. This was especially easy for 3T because each primitive procedure checks the state of the hardware before commanding it, and the sequencer will skip steps in a procedure that have been obviated by outside or serendipitous events. Thus to restart the RO for example, the sequencer might determine from the logged state file that RO was last stopped twenty minutes into its secondary phase, command the RO skills to set the valve configurations for secondary if they were not left in that state, check to see that all the pumps are on for secondary operation, advance the phase timer to +20 and resume monitoring secondary phase processing. Again, the ease with which the executive could be modified, and its resulting stable operation engendered continuing trust by the test team.

5. Supporting Intermittent Monitoring

As stated earlier, the life support engineers came to trust the 24/7 autonomous operations of the AWRS via 3T. But even this happy state of affairs generated problems that had to be overcome in order to maintain trust of the intelligent control system. The main issue was that while the life support engineers continued to use data logs for scientific analysis, they rarely monitored the running hardware systems. So when an anomaly occurred, the staff took considerable time to come up to speed on the state of the AWRS before and after the anomaly. The 3T team thus made a number of additions to the 3T system to alert the staff to failures and to help them quickly gain an accurate view of the situation.

Fed by 3T's distributed data management system, graphical user interfaces (GUIs) were made available on the user's desktops (see Figure 3) to provide a quick overview, via the subsystem schematic, and from which additional details could be called up on demand. The more commonly desired data (tank levels, pump speeds) were displayed directly on the schematic, and additional information was available (units, human-readable device name, component values for a computed value) by clicking on the schematic component in question. The user could also display skill logic specifications -- a part of the configuration management of the control code -- allowing operators to refresh

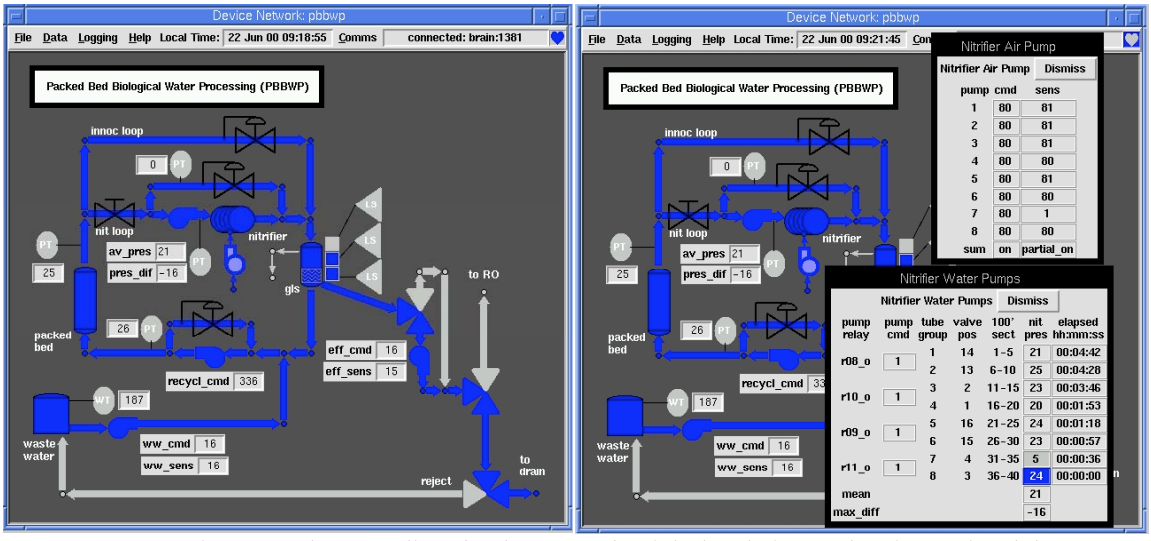

Figure 2. The GUI client for the BWP, the right-hand picture showing optional data

their understanding of how the controls worked. In addition, the lead controls engineer wrote detailed operations notes during functional testing which the remainder of the controls team used to assess the health of the software agents. These operations notes were also made available from the GUI schematics displays.
It was also important for the WRS engineers to be able to review performance history to detect system anomalies or indicators that an anomaly was developing. The water team made extensive use of the data logs to support anomaly detection and performance analysis. The logs could be displayed in a table from the GUIs and variables could be plotted for viewing performance over time.

Intermittent monitoring also requires that the intelligent system recognize when failures occur and notify the human expert in a timely fashion. Initially, the GUIs subscribed only to device level data from the agents in 3T. Consequently, the primary anomaly that could be detected was a loss of data connection between the skill manager and the GUIs, signaled by both audible and visual alarms. Later, the watchdog timers were added, which indicated the health of the communications between the skills and RAPs layers of 3T. The user could then use information pop-ups on the GUIs to see how long the agents and the sequencer had been out of contact with one another. This information would also go to an error log, with timestamps allowing the user to examine performance history just prior to the anomaly.

C. Case Study Summary

The described experience highlights the tenet that using intelligent control technologies over extended periods generates trust in operators and mission stakeholders. Trust didn't come quickly. 3T had been used successfully in earlier life support tests spanning the four years prior to the AWRS testing, providing an initial comfort level for choosing 3T for the AWRS. It took three months of operations before the life support engineers trusted the intelligent control system to operate unattended. Specific adjustments – not fundamental changes but changes critical to user compatibility – had to be made to the intelligent control system to foment and maintain operator trust throughout the long testing period. These included providing adjustable autonomy, additional safety measures, a quick restart/resume capability, and additional utilities to support the paradigm shift from vigilance to intermittent supervision. Most of these adjustments would not have been recognized, let alone implemented had the intelligent control system been used for a shorter period.

VI. Conclusion

This history of automation in most systems is marked by gradual adoption. Early on, the benefits are least, the risk highest and prior practice exerts the most influence. But ultimately, benefits in cost, safety and productivity make automation a normal and relatively uncontroversial system element. No commercial jet would be built today without advanced automation, nor would a telecommunications facility, modern naval vessel or automobile manufacturing plant. Such systems perform or support activities analogous to those we wish to carry out in space. However, adopting advanced automation into spaceflight systems presents particularly acute challenges. There is a constant tension between emphasis on the enormous difficulty of surviving and operating in space at all and emphasis on realizing exploration goals with the most powerful, flexible tools we can construct. Mission failures lead to an understandable focus on avoiding risk. Notable, recent failures attributed to software intensify concern about software in particular, especially complex software of the sort needed for advanced automation.

But the risks are balanced by an equally great need, ultimately, to realize the benefits advanced automation can provide. It is critical, therefore, to lay groundwork for reliable, trusted, intelligent control. Fortunately, a great deal of prior and ongoing work addresses this need. One element of this work is to obtain a better understanding of the specific barriers to adoption of intelligent control technology. A comprehensive effort to obtain this understanding and subsequently define needed systems engineering practices and standards, would involve technologists, mission managers, system and subsystem engineers, and mission stakeholders of all kinds. No such study has yet taken place, but the combined experience of many individuals involved in limited deployments of advanced automation on space systems indicates several important areas. In this paper, we have focused on 3 of these areas – minimizing the risk of software design errors through verification and validation, minimizing the need to adopt automation too rapidly through variable autonomy, and proving system qualities over time in long deployments. We have also documented a significant case study that holds many lessons for trusted spaceflight automation. Continued efforts in these and other areas should lead to the best of all worlds: highly capable, highly reliable spaceflight systems allowing us to safely and economically extend our presence in space.

References

Artho, C., Drusinsky, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Rosu, G. and Visser, W."Experiments with Test Case Generation and Runtime Analysis", *Proceedings of Abstract State Machines 2003*, Lecture Notes in Computer Science 2589, Springer-Verlag, March 2003.

Bernard, D. E.; Dorais, G. A.; Fry, C.; Jr., E. B. G.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Nayak, P. P.; Pell, B.; Rajan, K.; Rouquette, N.; Smith, B.; and Williams, B. C. "Design of the remote agent experiment for spacecraft autonomy," In *Proceedings of the IEEE Aerospace Conference,* 1998.

Bernard, D. et al., "Final report on the Remote Agent Experiment," *Proceedings of the New Millenium Program DS-1 Technology Validation Symposium*, 2000.

Bonasso, R. P., Firby, J. R., Gat, E., Kortenkamp, D., Miller, D. P., and Slack, M. G. 1997. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence* 9: 237-256.

Brat, G. and Klemm, R. "Static Analysis of the Mars Exploration Rover Flight Software," *First International Space Mission Challenges for Information Technology*, Pasadena, California, July 2003.

Brat, G., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, P., Venet, A., Visser, W., Washington, R. "Experimental Evaluation of Verification and Validation Tools on Martian Rover Software"**,** *Formal Methods in System Design*, September - November 2004, Volume 25, Issue 2-3, 167-198.

Chien, S. et al., "The EO-1 Autonomous Science Agent," *Proceedings of the Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004)*, 2004.

Chien, S., Knight, R., Stechert, A., Sherwood, R. and Rabideau, G. "Using iterative repair to increase the responsiveness of planning and scheduling for autonomous spacecraft," *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, 2000.

Cobleigh, J.M., Giannakopoulou, D., and Pasareanu, C.S. "Learning Assumptions for Compositional Verification", in *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*. April 2003, Warsaw, Poland. Springer, LNCS.

D. Harel, "Statecharts: A Visual Formulation for Complex Systems," *Science of Computer Programming*, Vol. 8, No. 3, 1987, pp. 231–274.

Denney, E., Fischer, B. and Schumann, J. "Adding Assurance to Automatically Generated Code"**.** *Proceedings the 8th IEEE International Symposium on High Assurance Systems Engineering* (HASE 2004). Tampa, Florida, 25-26 March 2004.

Feather, M., Fesq, L., Ingham, M., Klein, S. and Nelson, S. "Planning for V&V of the Mars Science Laboratory Rover Software," *Proceedings of the 2004 IEEE Aerospace Conference*, 2004.

Freed, M., Matessa, M., Remington, R. and Vera, A. (2003) How Apex Automates CPM-GOMS. In *Proceedings of the 2003 International Conference on Cognitive Modeling*. Bamberg, Germany.

Gat, E. 1998. Three-Layer Architectures. In *Mobile Robots and Artificial Intelligence*, Kortenkamp, D., Bonasso, R. P., and Murphy, R., Eds.: AAAI Press.

Giannakopoulou, D., Pasareanu, C., and Barringer, H., "Assumption Generation for Software Component Verification", in *Proc. of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*. September 2002.

Giannakopoulou, D., Pasareanu, C., and Cobleigh, J.M. "Assume-guarantee Verification of Source Code with Design-Level Assumptions", *Proc. of the 26th International Conference on Software Engineering (ICSE'2004)*,May 2004.

Havelund, K. and Rosu, G. "Java PathExplorer - A Runtime Verification Tool", *The 6th International Symposium on AI, Robotics and Automation in Space*, May 2001

Havelund, K., Johnson, S. and Rosu, G. "Specification and Error Pattern Based Program Monitoring"**,** *European Space Agency Workshop on On-Board Autonomy*, Noordwijk, Holland, 17-19 October 2001

Havelund, K., Lowry, M. and Penix, J. "Formal Analysis of a Space Craft Controller using SPIN", *IEEE Transactions on Software Engineering*, Vol. 27, No. 8, August 2001.

Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W. and White, J.L. "Formal Analysis of the Remote Agent Before and After Flight", *Proceedings of the 5th NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000.

Jonsson, A.K., Morris, P., Muscettola, N., Rajan, K. and Smith, B. "Planning in interplanetary space: Theory and practice," *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, 2000.

Lai-fook, K. M. and Ambrose, R. O. 1997. Automation of Bioregenerative Habitats for Space Environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2471-2476. Albuquerque, NM: IEEE.

Lowry, M., Havelund, K. and Penix, J. "Verification and Validation of AI Systems that Control Deep-Space Spacecraft", In *The Proceedings of the 10th International Symposium on Methodologies for Intelligent Systems (ISMIS'97)* October, 1997.

Mehlitz, P.C. and Penix, J. "Design for Verification: Using Design Patterns to Build Reliable Systems," Workshop on Component-Based Software Engineering, International Conference on Software Engineering, May 2003.

Pell, B. and Shafto, M. "Adjustable Autonomy in NASA's Exploration Vision", In *Proceedings of the First AIAA Intelligent Systems Conference*, Chicago, IL. , October, 2004.

Proud, R.W., Hart, J.J and Mrozinski, R.B. "Methods for determining the level of autonomy to design into a human spaceflight vehicle: a function specific approach," *Proceedings of the 2003 Conference on Performance Metric for Intelligent Systems*, 2003.

Schreckenghost, D., Bonasso, R. P., Kortenkamp, D., and Ryan, D. "Three Tier Architecture for Controlling Space Life Support Systems," *Proceedings of the IEEE Symposium on Intelligence in Automation and Robotics*: IEEE, 1998.

T.B. Sheridan. *Telerobotics, Automation and Human Supervisory Control.* The MIT Press. 1992.

Tsamardinos, I., Muscettola, N. and Morris, P. "Fast transformation of temporal plans for efficient execution," *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 1998.

Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. "Model Checking Programs", *Automated Software Engineering Journal*, Volume 10, Number 2, April 2003

Williams, B.C. and Nayak, P. "A model-based approach to reactive self-configuring Systems," *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.

Williams, B.C. Ingham, M., Chung, S. and Elliott, P. "Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers," *Proceedings of the IEEE*, Vol. 91, No. 1, 2003, pp. 212–237.