VALIDATING THE DS1 REMOTE AGENT EXPERIMENT

P. Pandurang Nayak[†] **Douglas E. Bernard**[¶] **Gregory Dorais**[‡]

Edward B. Gamble Jr.[¶] **Bob Kanefsky**[‡] **James Kurien**[||] **William Millar**[‡]

Nicola Muscettola[§] **Kanna Rajan**[‡] **Nicolas Rouquette**[¶]

Benjamin D. Smith[¶] **William Taylor**[§] **Yu-wen Tung**[¶]

Abstract

This paper describes the validation of the Remote Agent Experiment. A primary goal of this experiment was to provide an on-board demonstration of spacecraft autonomy. This demonstration included both nominal operations with goal-oriented commanding and closed-loop plan execution, and fault protection capabilities with failure diagnosis and recovery, on-board replanning following unrecoverable failures, and system-level fault protection. Other equally important goals of the experiment were to decrease the risk of deploying Remote Agents on future missions and to familiarize the spacecraft engineering community with the Remote Agent approach. These goals were achieved by successfully integrating the Remote Agent with the Deep Space 1 flight software, developing a layered testing approach, and taking various steps to gain the confidence of the spacecraft team. In this paper we describe how we achieved our goals, and discuss the actual on-board demonstration in May, 1999, when the Remote Agent took control of Deep Space 1.

1 Introduction

May, 1999, represents a milestone in the history of the development of spacecraft autonomy. In two separate experiments, the Remote Agent, an AI software system, was given control of an operational spacecraft and demonstrated the ability to respond to high level goals by generating and executing plans on-board the spacecraft, all the time under the watchful eye of model-based fault diagnosis and recovery software.

[†]RIACS, NASA Ames Research Center, MS 269-2, Moffett Field, CA 94035.
Corresponding author: nayak@ptolemy.arc.nasa.gov

[§]Recom Technologies, NASA Ames Research Center, MS 269-2, Moffett Field, CA 94035.

[‡]Caelum Research, NASA Ames Research Center, MS 269-2, Moffett Field, CA 94035.

[¶]Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109.

[||]NASA Ames Research Center, MS 269-2, Moffett Field, CA 94035.

Current spacecraft control technology relies heavily on a relatively large and highly skilled mission operations team that generates detailed time-ordered sequences of commands or macros to step the spacecraft through each desired activity. Each sequence is carefully constructed in such a way as to ensure that all known operational constraints are satisfied. The autonomy of the spacecraft is limited.

The Remote Agent (RA) approach to spacecraft commanding and control puts more "smarts" on the spacecraft. In the RA approach, the operational rules and constraints are encoded in the flight software and the software may be considered to be an autonomous "remote agent" of the spacecraft operators in the sense that the operators rely on the agent to achieve particular goals. The operators do not know the exact conditions on the spacecraft, so they do not tell the agent exactly what to do at each instant of time. They do, however, tell the agent exactly which goals to achieve in a specified period of time.

Three separate Artificial Intelligence technologies are integrated to form the RA: an on-board planner-scheduler, a robust multi-threaded executive, and Livingstone, a model-based fault diagnosis and recovery system [5; 4]. This RA approach was flown on the New Millennium Program's Deep Space One (DS1) mission as an experiment. The New Millennium Program is designed to validate high-payoff, cutting-edge technologies to enable those technologies to become more broadly available for use on other NASA programs.

The DS1 Remote Agent Experiment (RAX) had multiple objectives [2]. A primary objective of the experiment was to provide an on-board demonstration of spacecraft autonomy. This demonstration included both nominal operations with goal-oriented commanding and closed-loop plan execution, and fault protection capabilities with failure diagnosis and recovery, on-board replanning following unrecoverable failures, and system-level fault protection. These capabilities were demonstrated using in-flight scenarios that included ground commanding and simulated failures.

Other equally important, and complementary, goals of the experiment were to decrease the risk (both real and perceived) in deploying RAs on future missions and to familiarize the spacecraft engineering community with the RA approach to spacecraft command and control. These goals were achieved by a three-pronged approach. First, a successful on-board demonstration required integration

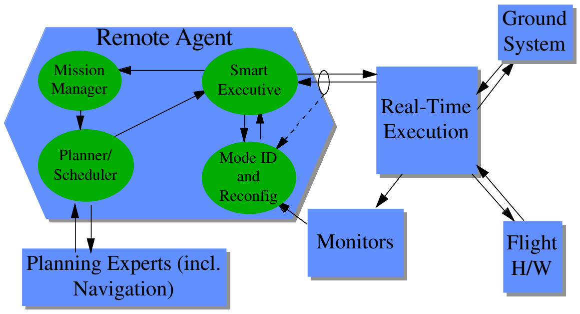

Figure 1: Remote Agent architecture.

of the RA with the spacecraft flight software. This integration provided valuable information on required interfaces and performance characteristics, and alleviates the risk of carrying out such integration on future missions. It also served to familiarize systems engineers and flight software engineers with the integration of RAs with traditional flight software. Second, a perceived risk of deploying RAs is related to its ability to synthesize new untested sequences in response to unexpected situations. We addressed this risk by demonstrating a layered testing methodology that serves to build confidence in the sequences synthesized by the RA in a variety of nominal and off-nominal situations. Third, the experiment was operated with close cooperation between RA team members and DS1 ground operators. This served to familiarize the ground operations community with benefits and costs of operating a spacecraft equipped with an RA.

The RAX was successfully executed on-board DS1 during the week of May 17–21, 1999. There were a few surprises along the way, which are discussed in a later section. These surprises pointed out some areas for improvement for future deployments of the remote agent. They also gave the team an opportunity to show off a number of benefits that the technology provides in terms of robust execution despite unexpected events, the ability to query the system to understand its state, as well as the ability to rapidly create and execute new mission profiles.

The remainder of the paper is organized as follows: Sections 2 and 3 describe the RA and the RAX scenarios; Section 4 describes RAX flight preparations for flight; Section 5 discusses the flight experiment itself, including surprises and the responses to these surprises; finally, Section 6 summarizes the paper.

2 Remote Agent Architecture

The RA architecture and its relation to flight software is shown in Figure 1. Viewed as a black-box, RA issues commands to real-time execution flight software (FSW) to modify spacecraft state, and receives state information through a set of monitors (MON) that filter data streams into a set of abstract properties. The RA itself is comprised of four components: a Mission Manager (MM), a Planner/Scheduler (PS) [3], a Smart Executive (EXEC) [6], and a Mode Identification and Reconfiguration module (MIR) [8].

MM formulates near-term planning problems based on a long-range mission profile representing the goals of the mission. MM extracts goals for the next scheduling horizon, combines them with a projected spacecraft state provided by EXEC, and formulates a planning problem for PS. This decomposition into long-range mission planning and shorter-term detailed planning enables RA to undertake an extended mission with minimal human intervention.

PS takes as input a plan request from MM and produces a flexible, concurrent temporal plan for execution by EXEC. PS constructs plans using domain constraints and heuristics in its knowledge base; planning experts participate in the planning process by requesting new goals or answering queries posed by PS.

EXEC executes a plan by decomposing the high-level plan activities into primitives, sending out commands, and monitoring progress based on direct feedback from the command recipient or on inferences drawn by MIR. If some task cannot be achieved, EXEC may attempt an alternate method or may request a recovery from MIR. If the EXEC is unable to execute or repair the current plan, it cleanly aborts the plan and attempts to bring the spacecraft into a safe state while requesting a new plan from MM.

MIR is responsible for mode identification (MI) and mode reconfiguration (MR). MI observes EXEC issuing commands, receives events from MON, and uses model-based inference to deduce the state of the spacecraft and provide feedback to EXEC. MR serves as a recovery expert, taking as input a set of EXEC constraints to be established or maintained, and uses declarative models it shares with MI to recommend a single recovery action to EXEC.

All communication between RAX and the flight software was mediated by the RAX manager, a software task belonging to DS1 flight software. The RAX manager was also responsible for starting the RAX Lisp task at the start of the experiment. When RAX is terminated, either normally or by ground controllers, the RAX manager immediately stops any further communication between RAX and the flight software, and then stops the RAX Lisp task. The ability to tightly control RAX activity through the RAX manager was an important factor in convincing the DS1 project that ground controllers could easily recover control of the spacecraft from RAX.

3 Remote Agent Experiment scenarios

The design of the RAX scenarios was driven by the need to demonstrate the RAX validation objectives. The RAX scenarios were originally designed in mid-1997, and were largely unchanged until early 1999. However, in response to new operations constraints levied by DS1 and an unexpected anomaly during the experiment, we were forced to significantly redesign the scenarios. Our ability to quickly redesign the RAX scenarios provides objective evidence of the flexibility of the RAX technology. In this section we describe the validation objectives and the various RAX scenarios.

3.1 RAX validation objectives

The DS1 project required formal validation objectives from each of the 12 technologies being validated on DS1. The validation objectives for RAX were broken down into specific objectives for each of the three engines as follows.

PS's validation objectives were to: (a) generate plans on-board; (b) reject low-priority, unachievable goals; (c) replan following a failure; (d) generate back-to-back plans; and (d) enable modification of mission goals from ground. EXEC's validation objectives were to: (a) provide a low-level commanding interface; (b) initiate on-board planning; (c) execute plans generated both on-board and on the ground; (d) recognize and respond to plan failure; and (e) maintain required properties in the face of failures. MIR's validation objectives were to: (a) confirm executive command execution; (b) demonstrate model-based failure detection, isolation, and recovery; and (c) demonstrate ability to update MIR state via ground commands.

3.2 Original RAX scenarios

The original RAX scenarios consisted of a 12 hour scenario and a 6 day scenario. The 12 hour scenario was designed as a confidence builder for the DS1 project. It involved neither on-board planning nor thrusting with the Ion Propulsion System (IPS). Rather, the plan was to be generated on the ground, uplinked to the spacecraft, and executed by EXEC and MIR. The scenario included imaging asteroids with the MICAS camera to support optical navigation, a simulated sensor failure* scenario, and demonstration of low-level commanding to flip a switch. The planning of optical navigation imaging provided the planner the opportunity to reject low-priority, unachievable goals since the optical navigation windows had time only to image a subset of the asteroid goals.

The 6 day scenario was to be run following successful completion of the 12 hour scenario. The 6 day scenario included both on-board planning and operating the IPS, and was the full-up test of RA. The scenario was divided into 2 horizons. At the start of the scenario, PS generated a plan for the first horizon which included MICAS imaging for optical navigation and IPS thrusting. Execution of the first plan also included a ground command to modify the goals for the second horizon. At the end of the optical navigation window PS planned to switch off the MICAS camera. However, a stuck on failure injection in the camera switch prevented RA from turning off the camera, leading to a plan failure. This led to a replan, which produced a second plan with the camera being left on. The second plan also included an activity to produce a plan for the second horizon (the third plan in the scenario), which was to be executed back-to-back with the second plan. While the second plan was being executed, the switch failure injection was undone and ground informed MIR that the switch is now fixed. The execution of the third plan included IPS thrusting, optical navigation imaging, and two simulated failures, a communication failure on the 1553 bus, and a thruster valve stuck closed failure.

Together, these two scenarios demonstrate all RAX validation objectives.

3.3 2 day RAX scenario

The 12 hour and 6 day scenarios were used for all RAX integration and testing until the beginning of March, 1999. At that point, we were informed by the DS1 project that they did not want us to switch off the MICAS camera due to concerns about thermal effects. Furthermore, we were required to provide only about 12 hours of IPS thrusting, to ensure that DS1 would be on track for its asteroid encounter in July, 1999. These changes meant that the 6 day scenario had to be changed at this late date, since it switched off the camera 3 times (not including the failed attempt during the failure injection) and thrusted for a total of about 4 days. We responded by developing a 2 day scenario. The 2 day scenario was similar to a compressed 6 day scenario, except that the simulated MICAS switch failure was active for the whole duration of the scenario. This prevented RA from ever switching off the camera. Furthermore, the 2 day scenario had only about 12 hours of IPS thrusting. Our ability to quickly develop a new scenario in response to these new constraints was viewed very favorably by the DS1 project.

3.4 6 hour RAX scenario

An anomaly was encountered while executing the 2 day scenario on-board DS1 which led to early termination of the 2 day scenario (see Section 6). At this time, approximately 70% of the RAX validation objectives had been achieved. To achieve the remaining 30% of the objectives, we quickly put together a 6 hour scenario which included IPS thrusting, three failure scenarios, and back-to-back planning. This scenario was executed on the spacecraft a little over 2 days later, thus completing RAX validation. The remarkable thing about this scenario was not just that we could quickly design and test it at such short notice, but rather that the DS1 project had already gained enough confidence in the RA that they allowed on-board execution of this new scenario within days of conception!

4 Preparing the Remote Agent Experiment for flight

We took a number of steps to prepare RAX for flight. In this section we highlight some of the key steps, including preparing the Lisp for flight, testing RAX, software change control, special considerations involved in testing PS, and the operational readiness tests. A comprehensive discussion of our integration methodology, a central element in preparing RAX for flight, is beyond the scope of this paper. Suffice it to say that developers acted as front-line testers during our various integration efforts, and hence identified and resolved a significant number of bugs (often unreported in our formal problem reporting system). As a result, formal testing on high fidelity platforms found few bugs, since most of the problems on these platforms had been discovered and resolved during integration.

4.1 Preparing Lisp for Flight

One important aspect of the RAX preparation for flight was the preparation of Lisp for flight. The RAX software development and runtime environment was based on CommonLisp, in particular the Harlequin Lispworks product []. The use of Lisp was appropriate given the background of the RAX developers, the early inheritance of code libraries, and the hardware independence of the high-level software interfaces between RAX and the rest

*All failure scenarios were simulated failures, though they appeared to be real to RAX.

of flight software. However, with the choice of Lisp came some unique challenges. These challenges fell into two rather broad categories: resource constraints and flight software interfaces.

Like all spacecraft, DS1 placed constraints on computational and telecommunication bandwidth (both uplink and downlink) resources. For computational resource, DS1 has a total of 128 MB RAM, 16 MB EEPROM, and a 20 MHz RAD6k. During the RAX experiment time, the uplink and downlink data rates were about 1 kbps and 4 kbps, respectively. Based on early estimates, RAX was allocated 32 MB of RAM, 16 MB of file space and up to 45% of the CPU. At the time of this allocation it was not clear if RAX could meet these resource constraints.

To fit within the 32 MB memory allocation and the CPU fraction constraints, the RAX team thoroughly analyzed their code for memory and performance inefficiencies and employed a "tree-shaking/transduction" process to the Lisp image. The analysis is, of course, common for any high performance software. However, transduction is Lisp-specific and arises from the tight coupling of the Lisp runtime and development environments. Transduction removes the unneeded parts of the development environment, e.g., the compiler, debugger, windowing system. The result is a significantly smaller image, both in terms of file system and runtime memory. During RAX testing, peak memory usage was measured at about 29 MB, which was more than was actually observed in flight.

To reduce the uplink time and the spacecraft file system usage, we employed a custom Lisp image that supported ground-based compression and spacecraft-based decompression. Upon completion of the transduction process the RAX Lisp image was compressed by a factor of about 3 to 4.7 MB and uplinked to the spacecraft. Onboard decompression was initiated at the start of each RAX run, with the file being inflated directly into the 32 MB RAX memory space. Use of this custom compression drastically reduced the file uplink time and kept the RAX file space usage within the agreed upon limits.

Besides the resource constraints, we also dealt with a complicated flight software interface. The flight software was written in the 'C' programming language and ran on the VxWorks operating system. Lisp and 'C' interacted through Lisp's foreign function interface. This interface was the source of many early problems, primarily caused by discrepancies between data structure alignments assumed by the Lisp and 'C' compilers. These problems were quickly discovered and resolved with the help of an extensive test suite that tested a large number of function parameter variations.

Another problem arose in preparing the Lisp multithreading system for flight. Originally, the Lisp thread scheduler relied on a high frequency external, periodic wakeup call, issued at interrupt level. However, this went against the design principles of the DS1 flight software. Hence, we had to significantly change Lisp's approach to thread preemption to use a lower frequency wakeup call implemented with flight software timing services.

Most of the late integration problems with RAX Lisp arose because of the VxWorks port. As RAX moved from testbed to testbed, ever closer to the final spacecraft configuration, low-level Lisp problems arose. The problems were consistently of two types: a function assumed by Lisp to be present was not present or a function was present but did not perform as expected by Lisp. The first type of problem was resolved by consistent application of a detailed RAX and FSW build process. The second type of problem was addressed on a case-by-case basis. Solutions to these problems were made difficult due to the reduced debugging visibility as testbeds assumed the spacecraft configuration. We benefited from the dedicated efforts of both Harlequin and the DS1 FSW team.

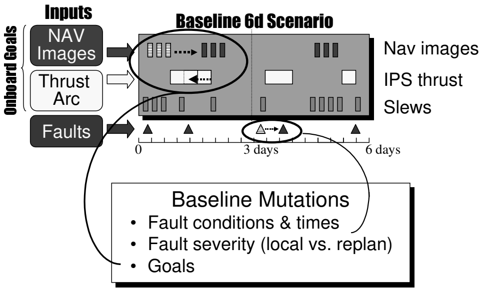

Figure 2: Baseline Variations

4.2 Testing the Remote Agent

Autonomous systems, such as the RA, need to respond robustly in a wide range of situations. In order to verify that they respond correctly in all situations would require a huge number of test cases. To make matters worse, the tests should ideally be run on high-fidelity testbeds, which are heavily oversubscribed, difficult to configure correctly, and cannot run faster than real time, e.g., we could run only 10 tests in four weeks on one of DS1's high-fidelity testbeds. To address these problems, we employed a "baseline testing" approach to reduce the number of tests, and exploited several lower-fidelity testbeds to increase the number of tests we could run [7].

The baseline scenarios we used were the 12 hour and 6 day scenarios discussed above. We tested a number of nominal and off-nominal variations around these scenarios. The variations comprised variations in spacecraft behavior that we might see during execution and changes to the baseline scenario that might be made prior to execution. This included variations to the goals in the mission profile, variations in when faults might occur, and variations in the FSW responses (see Figure 2).

The tests were distributed among the low, medium, and high fidelity platforms. The two low-fidelity platforms were the "babybed" and "radbed". The babybed had a non-radiation hardened PowerPC CPU and ran with simulators written by the RAX team. The radbed was identical to the babybed, except that it had a flight RAD6000 CPU. The medium fidelity platform was the "papabed", which had a flight CPU, bus, and memory and official DS1 simulators. The highest-fidelity platforms, the "hotbench" and "testbed", had flight computers and were connected to flight hardware (flight spares)

Platform	Fidelity	CPU	Hardware	Availability	Speed
Spacecraft	Highest	Rad6000	Flight	1 for DS1	1:1
DS1 Testbed	High	Rad6000	Flight spares + DS1 sims	1 for DS1	1:1
Hotbench	High	Rad6000	Flight spares + DS1 sims	1 for DS1	1:1
Papabed	Med	Rad6000	DS1 simulators only	1 for DS1	1:1
Radbed	Low	Rad6000	RAX simulators only	1 for RAX	1:1
PowerPC	Lowest	PowerPC	RAX simulators only	2 for RAX	7:1

Table 1: DS1 Testbeds

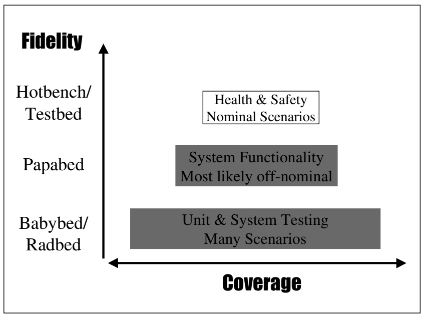

Figure 3: Pyramid Testing Approach

where feasible (see Table 1).

The architecture of RA allowed us to run certain tests on lower-fidelity testbeds and be confident that their results would hold on higher-fidelity testbeds. Specifically, the RA commands and monitors the spacecraft through well-defined interfaces with the FSW. Those interfaces were the same on all platforms, as were the range of possible responses. Only the fidelity of the responses improved with platform fidelity. This allowed us to exercise a wide range of nominal and off-nominal behaviors on the babybeds and radbed, test the most likely off-nominal scenarios on the papabed, and test only the nominal scenarios and certain performance and timing related tests on hotbench and testbed. This "pyramid" approach to testing is summarized in Figure 3.

The remainder of this section describes the tests on each of the testbeds, and discusses the effectiveness of our testing approach given the benefit of hindsight.

Babybed and radbed testing

Each of the RA modules devised a test suite of nominal and off-nominal scenarios that isolated and exercised key behaviors in each module. This involved testing about 200 variations of the initial state and goals of the planner, while exercising MIR in hundreds of the likeliest failure contexts. The PS and MIR tests were used for testing EXEC, and the system-level interaction of all modules was exercised by a suite of twenty additional scenarios. These tests were run rapidly on the babybeds and radbed, with simulators that permitted faster than real-time execution and exploited RA's ability to "warp" over long periods of idle time. Even with this increased speed, running a scenario was a time-consuming and error-prone process. To alleviate this, we designed an automated testing tool that accepted an encoded scenario description as input, controlled the simulator and ground tools to execute the scenario, stopped the test when appropriate by monitoring the telemetry stream, and stored all logs and downlinked files for later examination. This rapid data collection led to a total running time of about one week for all tests, since tests could be scheduled overnight and required no monitoring. Analyzing the results of the tests, however, was still a time consuming process. These tests were run after each major RAX software release. We identified (and resolved) over 800 bugs in six months.

Papabed testing

Once we delivered a "frozen" version of RA, we ran six off-nominal system test scenarios on the papabed. These corresponded to the most likely and highest-impact scenarios. No bugs were detected in these scenarios, probably because RA responses to off-nominal situations were well tested on the babybed.

Hotbench and testbed testing

The hotbench and testbed was reserved for testing the nominal scenarios, and for testing a handful of requirements for spacecraft health and safety. RAX was designed with a "safety net" that allowed it to be completely disabled with a single command sent either by the ground or by on-board FSW fault protection. Hence, the only ways in which RAX could affect spacecraft health and safety was by consuming excessive resources (memory, downlink bandwidth, and CPU) or by issuing improper commands. We tested the resource consumption cases by causing RAX to execute a Lisp script that consumed those resources. We guarded against improper commands by having subsystem engineers review the execution traces of the nominal scenarios, and doing automated flight rule checking. The nominal scenarios were run in conditions that were as close to flight-like as possible.

4.3 Software change control

As the date of the flight experiment drew closer, our perspective on testing changed. Throughout 1998 the main goal of testing was to discover bugs in order to fix them in the code. Starting in January 1999 the discovery of a bug did not automatically imply a code change to fix it. Instead, every new problem was reported to a Change Control Board (CCB) composed by senior RAX project members. Every bug and proposed fix was presented in detail, including the specific lines of code that needed to change. After carefully weighing the pros and cons of making the change, the board voted on whether or not to allow the fix. Closer to flight, DS1 instituted its own CCB to review RAX changes.

As time progressed, the CCB became increasingly conservative and the bias against code modifications significantly increased. This is demonstrated by the following figures. In total, 66 change requests were submitted to the RAX CCB. Of these, 18 were rejected amounting to a 27% rejection rate. The rejection rate steadily increased as time passed: 8 of the last 20 and 6 of the last 10 submitted changes were rejected.

The reason for this increase in conservatism is easily explained. Every bug fix modifies a system that has already gone through several rounds of testing. To ensure that the bug fix has no unexpected repercussions, the modified system would need to undergo thorough testing. This is time consuming, especially on the higher fidelity testbeds, so that full revalidation became increasingly infeasible as we approached flight. Therefore, the CCB faced a clear choice between flying a modified RAX with little empirical evidence of its overall soundness or flying the unmodified code and trying to prevent the bug from being exercised in flight by appropriately restricting the scenario and other input parameters. Often, the answer was to forego the change.

4.4 Testing the PS module

As discussed above, the PS module had undergone extensive testing throughout 1998 using variations of the 12 hour and 6 day scenarios. To generate these variations, we started by identifying the parameters that define a scenario. Test cases were generated using the "Latin squares" method [1] that ensured every pair of parameter values occurred in some test case. This approach was very effective in finding bugs, and resulted in a majority of the 211 PS problem reports filed in that period.

However, as we entered 1999, new problems were discovered in PS outside of the formal testing process. This resulted in 22 change requests submitted to the RAX CCB, a little over 9% of the total PS problem reports. The vast majority of these problems consisted of PS operating correctly but being unable to find a plan within the allocated time limit since its search was "thrashing". These problems were particularly serious since they could easily arise in off-nominal situations during flight.

There were several reasons for this situation:

1. The ranges of some parameters turned out to be different than those assumed by PS testing, e.g., PS testing assumed turn durations were at most 20 minutes, while actual turns could take over an hour. This created stress situations not considered by PS testing.

2. Planning problems became more challenging when we transitioned from the 6 day scenario to the 2 day scenario. The temporal compression led to the disappearance of slack time between activities. In the 6 day scenario PS could exploit this slack to achieve subgoals without backtracking. In the 2 day scenario backtracking became necessary, revealing additional brittleness in the PS chronological backtracking search.

3. A more fundamental issue was the independence between the PS test generator and the structural characteristics of the domain model. This led to the test generator missing a number of stress cases. For example, one problem depended upon the specific values of three continuous parameters: the time to start up the IPS engine, the time to the next optical navigation window, and the duration of the turn from the IPS attitude to the first asteroid. An equation relating these parameters can crisply characterize the stress situations. Unfortunately, automatically generated test cases based on covering pairwise interactions of parameter values cannot reliably detect such problems.

Given the late date at which these new problems were discovered, it was not feasible to modify the test suite to test extended variations around the new baseline. Instead, we focused on the just the most crucial variation: the time at which replans might occur. The objective was to ensure that the planner was robust to any replanning contingency. Two steps were needed to accomplish this. First, the new 2 day scenario was designed to guarantee that the harmful constraint interactions of the PS domain model would be avoided under any hypothetical replanning contingency. The idea was to ensure that PS could always return a plan within the given time limit. Second, a new PS test suite was carefully designed and run to ensure that this was indeed the case.

The design methodology for this new PS test suite is instructive. Exhaustive generation of all possible plans was clearly impossible. Instead, using our knowledge of the PS model, we manually identified boundary times at which the topology of the plans would change. We identified 25 such boundary times and generated a total of 88 test cases corresponding to plans starting at, near, or between boundary times. This led to the discovery of two new bugs. Furthermore, analysis of the test results showed that PS would fail to find a plan at only about 0.5% of all possible start times. Although the probability of this failure was extremely low, contingency procedures were developed to ensure that the experiment could be successfully continued even if this PS failure actually occurred.

We used the above test suite design methodology only toward the end of RAX, after the PS model and code had been frozen. However, we believe that this (currently manual) analysis method can be generalized and extended to provide an automatic PS testing procedure throughout the development process for new application domains.

4.5 Operational Readiness Tests

In addition to testing RAX, preparing RAX for operations involved preparing operational procedures and sequences for running the experiment and identifying contingency procedures. The operational readiness tests (ORTs) were a "dress rehearsal" of the procedures and contingencies, and were intended to familiarize the operations team with the procedure and to identify problem areas.

We performed two ORTs. The first ran through the first several hours of the 12 hour scenario and was primarily intended to exercise the procedures for starting RAX. This involved configuring the spacecraft, filesystem, and memory to the state required to start RAX. The second ORT ran through the entire 2 day scenario. The operations team monitored key events in the scenario, with breaks in between. This proved to be an effective way to monitor the experiment without unduly taxing the operations team. During actual spacecraft operations we followed a similar approach, though the RA team monitored the experiment around the clock.

The other purpose of the ORT was to exercise the RAX ground tools in an operations environment. During the two ORTs, RAX was run on the hotbench and the data was sent to workstations in the mission control center, some of which were running the RAX ground tools. The tools performed well, although we did identify a number of shortcomings, which we proceeded to resolve prior to flight.

5 The Remote Agent Experiment in flight

RAX was scheduled to be performed on DS1 during a three week period starting May 10, 1999. This period included time to retry the experiment in case of unexpected contingencies. On May 6, 1999, DS1 encountered an anomaly that led to spacecraft safing. Complete recovery from this anomaly took about a week of work by the DS1 team, both delaying the start of RAX as well as taking time away from their preparation for the asteroid encounter in July, 1999. In order not to jeopardize the encounter, the DS1 project also decided to reclaim the third RAX week for encounter preparation, leaving only the week of May 17th, 1999, for RAX. However, to maximize the time to try the more important 2 day experiment, they agreed to go ahead with the 2 day experiment without first doing the confidence building 12 hour experiment. This decision was strong evidence that the DS1 project had already developed significant confidence in RAX during pre-flight testing.

On Monday, May 17th, 1999, at 11:04 am PDT, we received a telemetry packet that confirmed that the 2 day RAX scenario had started on DS1. Shortly thereafter, PS started generating the first plan. The first plan was generated correctly, but not before an unexpected circumstance created some apprehension in us.

PS telemetry indicated that PS was generating the plan following a different search trajectory than what we had observed in ground testing. Since the conditions on the spacecraft were practically identical to those on the ground testbeds, there was no apparent reason of for this discrepancy. Subsequently, the cause for this discrepancy was traced back to the spacecraft and papabed differing on the contents of the file containing asteroid goals; PS was actually solving a slightly different problem than it had solved on the ground! Thus, this unexpected circumstance allowed us to demonstrate that PS problem solving was robust to last minute changes in the planning goals, increasing the credibility of the autonomy demonstration.

The 2 day scenario continued smoothly and uneventfully with the simulated MICAS switch failure, the resulting replan, long turns to point the camera at target asteroids, optical navigation imaging during which no communication with DS1 was possible, and the start of IPS thrusting. However, around 7:00 am on Tuesday, May 18, 1999, it became apparent that RAX had not commanded termination of IPS thrusting as expected. Although plan execution appeared to be blocked, telemetry indicated that RAX was otherwise healthy. The spacecraft too was healthy and in no apparent danger. The decision was made to use EXEC's ability to handle low-level commands to obtain more information regarding the problem. Once enough information had been gathered, the decision was made to stop the experiment. By this time an estimated 70% of the RAX validation objectives had already been achieved.

By late Tuesday afternoon the cause of the problem was identified as a missing critical section in the plan execution code. This created a race condition between two EXEC threads. If the wrong thread won this race, a deadlock condition would occur in which each thread was waiting for an event from the other. This is exactly what happened in flight, though it had not occurred even once in thousands of previous races on the various ground platforms. The occurrence of this problem at the worst possible time provides strong impetus for research on formal verification of flight critical systems. Once the problem was identified, a patch was quickly generated for possible uplink.

Following the discovery of the problem, we generated a 6 hour RAX scenario to demonstrate the remaining 30% of the RAX validation objectives. This new scenario was designed, implemented, and tested, together with the patch, on papabed overnight within about 10 hours. This rapid turn around allowed us to propose a new experiment at the DS1 project meeting on Wednesday. The DS1 project decided to proceed with the new scenario. However, they decided not to uplink the patch, citing insufficient testing to build adequate confidence. In addition, based on the experience on various ground testbeds, the likelihood of the problem recurring during the 6 hour test was deemed to be very low. Nonetheless, we developed and tested a contingency procedure that would enable us to achieve most of our validation objectives even if the problem were to recur.

The DS1 project's decision not to uplink the patch is not surprising. What was remarkable was their ready acceptance of the new RAX scenario. This is yet more evidence that the DS1 project had developed a high level of confidence in RA and its ability to run new mission scenarios in response to changed circumstances. Hence, although caused by an unfortunate circumstance, this rapid mission redesign provided unexpected validation for RA.

The 6 hour scenario was activated Friday morning. The scenario ran well until it was time to start up the IPS. Unfortunately, an unexpected problem in some supporting software failed to confirm an IPS state transition, thus causing RA to (correctly) stop commanding the IPS startup sequence. The underlying cause of this problem was still under investigation as of May 28, 1999. Since this situation was out of scope for RAX, the resulting RA state was inconsistent with spacecraft state. Fortunately, the discrepancy proved to be benign. Hence, RA was able to continue executing the rest of the scenario to achieve the rest of its validation objectives.

As a consequence of the two flight scenarios, RAX achieved 100% of its validation objectives.

6 Summary

The primary goal of RAX was to demonstrate that Artificial Intelligence technologies could achieve high-level autonomous control of a spacecraft including:

- goal-oriented commanding;

- closed-loop planning and execution;
- spacecraft state inferencing and failure detection;
- closed-loop model-based failure diagnosis and recovery;
- on-board re-planning as a response to unrecoverable failures; and
- system-level fault protection.

Familiarizing the spacecraft engineering community with these technologies and laying the foundation for more extensive applications of RA were also important goals. These goals were achieved by the design of RA, its integration with the DS1 flight software on spacecraft testbeds, its layered testing, two operational readiness tests with ground control personnel, and succesful commanding of the spacecraft during the week of May 17-21, 1999.

As a result of the Remote Agent project, we believe that the willingness of NASA missions to deploy highly-autonomous systems has increased. Moreover, the NASA Ames Research Center and the Jet Propulsion Laboratory have recognized this contribution by nominating RA for NASA's prestigious Software of the Year award.

Acknowledgments

We gratefully acknowledge the DS1 team and Harlequin, without whom the Remote Agent Experiment would not have been possible. We would also like the thank the many past contributors to the Remote Agent adventure and its many supporters over the past four years. This paper describes work performed at the NASA Ames Research Center and at the Jet Propulsion Laboratory, California Institute of Technology, under contract from the National Aeronautics and Space Administration.

References

[1] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, pages 83–88, September 1996.

[2] Douglas E. Bernard *et al.* Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference*, 1998.

[3] Nicola Muscettola. HSTS: Integrating planning and scheduling. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[4] Nicola Muscettola, P. Pandurang Nayak, and Brian C. Williams Barney Pell. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103:5–47, 1998.

[5] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An autonomous spacecraft agent prototype. *Autonomous Robotics*, 5(1), March 1998.

[6] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. Robust periodic planning and execution for autonomous spacecraft. In *Proceedings of IJCAI-97*, 1997.

[7] Benjamin Smith, William Millar, Julia Dunphy, Yu wen Tung, P. Pandurang Nayak, Edward B. Gamble Jr., and Micah Clark. Validation and verification of the remote agent for spacecraft autonomy. In *Proceedings of the 1999 IEEE Aerospace Conference*, 1999.

[8] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*, pages 971–978, 1996.