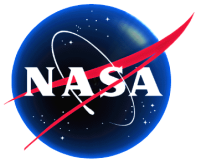


Remote Agent Experiment
DS1 Technology Validation Report

Douglas E. Bernard, Edward B. Gamble, Jr., Nicolas F. Rouquette,
Ben Smith, Yu-Wen Tung
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109

Nicola Muscettola, Gregory A. Dorias, Bob Kanefsky,
James Kurien, William Millar, Pandu Nayak, Kanna Rajan,
Will Taylor
Ames Research Center

http://rax.arc.nasa.gov

Table of Contents

Section Page

Figures

Figure Page

Tables

Table Page

Appendices

Appendix Page

EXTENDED ABSTRACT

Remote Agent (RA) is a model-based, reusable, artificial intelligence (AI) software system that enables goal-based spacecraft commanding and robust fault recovery. RA was flight validated during an experiment onboard Deep Space 1 (DS1) between May 17 and May 21, 1999.

Technology Overview

RA can operate at different levels of autonomy, allowing ground operators to interact with the spacecraft with immediate commands to the flight software, if needed. However, one of the most unique characteristics of RA, and a main difference with traditional spacecraft commanding, is that ground operators can communicate with RA using *goals* (e.g., "During the next week take pictures of the following asteroids and thrust 90% of the time") rather than with detailed sequences of timed commands. RA determines a plan of action that achieves those goals and carries out that plan by issuing commands to the spacecraft. Actions are represented with tasks that are decomposed on the fly into more detailed tasks and, eventually, into commands to the underlying flight software. When discrepancies are detected between the desired state and the actual state, RA detects, interprets, and responds to the anomaly in real time. More serious anomalies can be addressed with longer response times by generating a new plan of action while the spacecraft is kept idle in a safe configuration. When the new plan is generated, the spacecraft is taken out of the safe configuration and execution resumes normally.

RA differentiates itself from traditional flight software because it is *model-based*. In traditional software programs and expert systems, the programmer decides what the result of a program should be and writes down instructions or rules that attempt to achieve those results. The computer simply executes the instructions or fires the rules with no knowledge of what the intended result was or how it is achieving it. In the RA system, however, each component operates on *models,* general descriptions of the behavior and structure of the spacecraft it is controlling. Each RA component solves problems by accepting goals and using appropriate reasoning algorithms on its models to assemble a solution that achieves the goals. The reasoning algorithms are general-purpose and remain unchanged across different deployments of RA. For different applications, the parts that change are the models and possibly the problem-solving control knowledge needed by some RA modules to tune performance.

Remote Agent Component Technologies

Remote Agent integrates three separate technologies: an onboard Planner/Scheduler (PS), Smart Executive (EXEC), a robust plan-execution system , and the Mode Identification and Recovery (MIR) system for model-based fault diagnosis and recovery. These component technologies are described briefly below.

PS—PS generates the plans that RA uses to control the spacecraft. Given the initial spacecraft state and a set of goals, PS generates a set of synchronized high-level tasks that, once executed, will achieve the goals. PS consists of a heuristic chronological-backtracking search engine operating over a constraint-based temporal database. PS begins with an incomplete plan and expands it into a complete plan by posting additional constraints in the database. These constraints originate either from the ground, which imposes them directly on the goals, or from constraint templates (e.g., the camera must be pointed at an asteroid to take a picture of it) stored in a model of the spacecraft. PS queries domain-specific planning experts (specialized software modules such as Deep Space 1's navigation system) to access information that is not in its model.

EXEC—EXEC is a reactive, goal-achieving control system that is responsible for:

- Requesting and executing plans from the planner.
- Requesting/executing failure recoveries from MIR.
- Executing goals and commands from human operators.
- Managing system resources.
- Configuring system devices.
- System-level fault protection.
- Achieving and maintaining safe-modes as necessary.

EXEC is goal-oriented rather than command-oriented. A goal is defined as a system state being controlled that must be maintained for a specified length of time. As a simple example, consider the goal: keep device A on from time X to time Y. If EXEC were to detect that device A is off during that period, it would perform all the commands necessary to turn it back on. EXEC controls multiple processes in order to coordinate the simultaneous execution of multiple goals that are often interdependent. In order to execute each goal, EXEC uses a model-based approach to create a complex command procedure designed to robustly achieve the goal.

MIR—The MIR inference engine provides mode identification (diagnosis) and mode reconfiguration (recovery) functionality. To track the state of each component (called a mode) in the spacecraft, MIR eavesdrops on commands that are sent to the spacecraft hardware by EXEC. As each command is executed, MIR receives observations from spacecraft's sensors, which are then abstracted by monitors in the spacecraft's control software. MIR combines these commands and observations with declarative models of the spacecraft components to determine the current state of the system and to report it to EXEC. If failures occur, MIR uses

the same model to find a repair or workaround that allows the plan to continue execution.

The key idea underlying model-based diagnosis is that a combination of component modes is a possible description of the current overall state of the spacecraft only if the set of models associated with these modes is consistent with the observed sensor values. This method does not require that all aspects of the spacecraft state be directly observable, providing an elegant solution to the problem of limited observability.

Risks

RA is flight software and as such poses the same kind of risks posed by conventional flight software. The autonomous behavior implemented by RA is not qualitatively different from that displayed by conventional fault protection or attitude control. In all cases, the spacecraft is commanded on the basis of current state information rather than by direct operator commands. The behavior of RA can be predicted, within an envelope, just as the behavior of fault protection or attitude control can be predicted within certain bounds. Confidence in the RA's responses can be obtained through testing, just as confidence in fault protection or attitude control is obtained now.

A risk addressed by the experiment concerns the integration and testing of the technology. RA in a novel integration of three technologies; the application of these integrated technologies to spacecraft is also new. For this reason, there was no prior experience on development and validation methodologies for such a system. Another risk had to do with the integration of the AI technologies of RA, based on general-purpose search algorithms, together with real-time control software on a flight processor.

Validation Objectives

The first validation objective was to demonstrate RA's ability to autonomously operate a spacecraft with communication from ground limited to few high-level goals. This translated into specific objectives for PS, EXEC, and MIR. The second validation objective was to show that RA could be commanded with different levels of autonomy. This meant supporting all of the possible operation modes: using EXEC to run a traditional sequence of commands, preparing a plan on the ground and uplinking it to the spacecraft for execution, and providing closed-loop planning and execution onboard the spacecraft. The final validation objective was the first formulation of a development and testing plan for an autonomous flight software system.

Test Program and Results

The Remote Agent Experiment (RAX) consisted of using the RA technology to operate the DS1 spacecraft for several days. A series of operations scenario based on DS1 active cruise mode was developed. In these scenarios, RAX commanded a subset of the spacecraft subsystems: Ion Propulsion System (IPS), Miniature Integrated Camera and Spectrometer (MICAS), Autonomous Navigation (NAV), Attitude Control System (ACS), and a series of power switches.

The main scenario goals were to execute an IPS thrust arc, acquire optical navigation images as requested by the autonomous navigator, and respond to several simulated faults. The faults included minor ones that could be responded to without disrupting the current plan and more serious ones that required generating a new plan to achieve the remaining goals. A continuous integration approach was adopted in which new features or bug fixes were integrated in new releases only after the integrated system could successfully run the reference scenarios on all available testbeds. An extensive formal-testing program was conducted, separate from the software development process. Testing was distributed on several different platforms of different speeds, level of fidelity, and availability to the RA team. Test cases were targeted to the most available testbed that could validate them with the reasonable expectation that test results would hold on higher fidelity testbeds.

In spite of a couple of bugs that occurred during the flight experiment, RA successfully demonstrated 100% of its flight validation objectives.

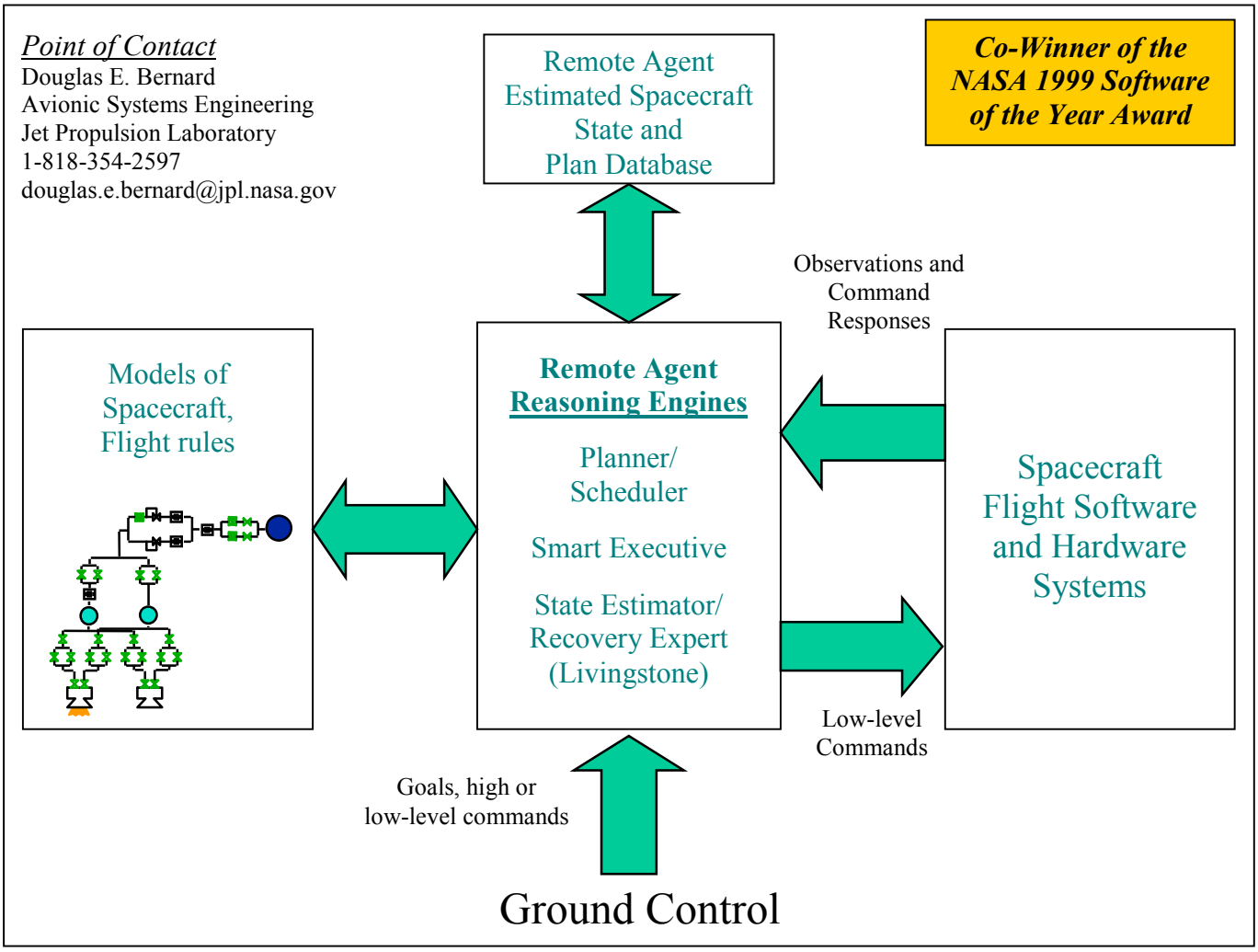
Applicability to future NASA missions

The Remote Agent technology is applicable to any future NASA mission that desires or requires autonomous operations. The RA reasoning engines can be used as-is on future missions. New domain models would be required for each mission.

FACT SHEET

Point of Contact
Douglas E. Bernard
Avionic Systems Engineering
Jet Propulsion Laboratory
1-818-354-2597
douglas.e.bernard@jpl.nasa.gov

Co-Winner of the NASA 1999 Software of the Year Award



Validation Objectives

- ✓ •Initiate and generate flexible plans on-board
- ✓ •Reject low-priority, unachievable goals
- ✓ •Execute plans generated both onboard and from ground
- ✓ •Confirm execution of commands
- ✓ •Demonstrate model-based failure detection and recovery
- ✓ •Maintain required spacecraft states in the face of failures
- ✓ •Re-plan following a failure
- ✓ •Generate back-to-back plans
- ✓ •Modify mission goals from ground
- ✓ •Execute low-level commands from ground
- ✓ •Update estimated spacecraft-state database from ground

Capabilities

- •Robust goal-based commanding
  - Planner expands high-level goals into flexible plans
  - Smart Executive decomposes plans into low-level spacecraft commands and monitors that the states commanded to are achieved and maintained
- •Fail-operational model-based fault recovery
  - Livingstone identifies faults and suggests recoveries that the Smart Executive uses to continue plan execution
  - If necessary, Executive requests the Planner/Scheduler to generate a new plan in light of failure

Applicability to Future Missions

Remote Agent technologies are generally applicable to missions that benefit from highly autonomous operation and are currently being applied to prototypes of future NASA missions including a space-based interferometer and an in-situ propellant production plant.

Remote Agent Experiment
DS1 Technology Validation Report

Douglas E. Bernard, Edward B. Gamble, Jr., Nicolas F. Rouquette, Ben Smith, and Yu-Wen Tung
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California

Nicola Muscettola, Gregory A. Dorais, Bob Kanefsky, James Kurien, William Millar, Pandu Nayak, Kanna Rajan, and Will Taylor
NASA Ames Research Center

1.0 The Remote Agent

Remote Agent (RA) is a model-based, reusable, artificial intelligence (AI) software system that enables goal-based spacecraft commanding, and robust fault recovery. This report describes the RA technology, its development and test history, and the DS1 flight experiment in which RA was validated. Whenever feasible, this report attempts to give guidance on how RA can be fruitfully employed in future science missions. Also highlighted are further technology developments and operational applications the team is currently pursuing.

1.1 Technology Overview

RA integrates three separate Artificial Intelligence technologies: automated planning and scheduling, robust multi-threaded execution, and model-based fault diagnosis and recovery.

1.1.1 Remote Agent Architecture—The RA architecture and its relation to flight software are 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 data from the spacecraft through a set of monitors that filter and discretize sensor values. The RA itself is comprised of three main components: a Planner/Scheduler (PS), a Smart Executive (EXEC), and Livingstone, a Mode Identification and Reconfiguration (MIR) system. An additional component, strictly related with PS, is the Mission Manager (MM). In addition, the RA team provided a clean interface to the rest of the FSW via the Remote Agent Experiment Manager (RAXM), which mediated all communication between RA and FSW and was included from the outset in the FSW design. RAXM provided a messaging conduit between RA and the rest of FSW, including interfaces to the planning experts, as well as to the monitors and the real time sequencer. This mechanism allowed RA to be cleanly bundled on top of the FSW much later in flight and also allowed a clear methodology for testing and validating the RA software on the ground.

The main functionalities provided by RA, how each individual RA component participates in the overall picture, and concrete examples of commanding and operations relative to DS1 are described below.

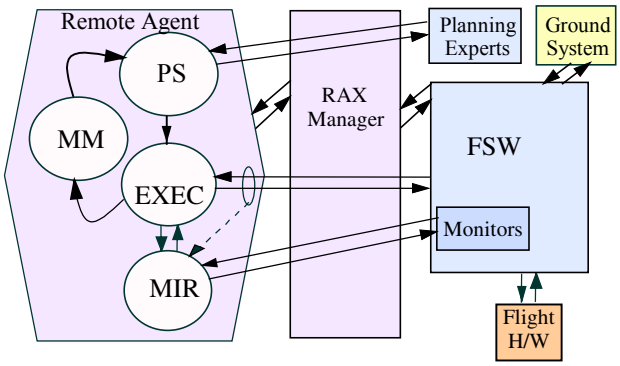


Figure 1. Remote Agent Architecture

RA can operate at different levels of autonomy, allowing ground operators to interact with the spacecraft with immediate commands to the FSW, if needed. However, what makes RA unique is that ground operators can skip formulating detailed timed-command sequences and communicate with RA at the *goal level*. Goals are stored in MM in a *mission profile* covering an extended period.

In principle, a mission profile could contain all goals for a mission, requiring no further uplink from ground. More realistically, mission operations will want to change goals (e.g., scheduled DSN communications can be modified on a week-by-week basis). This is easily done by uplinking commands to edit the mission profile. Goals typically contain few details of how they should be done. For example, the only DS1 Remote Agent Experiment mission profile goals were "Perform AutoNAV orbit determination (OD) activities for 1 hour every day," and "Thrust the IPS engine for at most 12 hours."

To translate high-level goals into a stream of commands to flight software, RA follows a two-step process. In the first step, MM selects goals for the next commanding horizon (typically covering several days) and sends them to PS. PS uses its model of the spacecraft to determine which detailed tasks should be selected and scheduled to achieve the goals. For example, in order to perform an OD, PS determines from the model that pictures of beacon asteroids need to be taken. In order to select these asteroids, the model instructs PS to interrogate the AutoNAV software as a *planning expert*. In general, PS will rely on several specialized services provided by software modules external to RA. In DS1, both AutoNAV and ACS provided information to PS

that was incorporated into plans. Going back to our example, observing an asteroid translates, according to the PS model, into taking a series of images of it with the Miniature Integrated Camera and Spectrometer (MICAS).

Therefor, PS schedules a "MICAS take Optical Navigation (OPNAV) module subsystem FSW images" task. Moreover, the model instructs PS that while images of an asteroid are being recorded, the attitude of the spacecraft must be compatible with the MICAS camera pointing at it. If this is not the case, the PS model instructs PS to schedule an appropriate turn, changing the attitude from the previous one to the desired one.

The brief example above points out another fundamental characteristic of all RA components: their fundamental reliance on explicit, declarative *models* of the spacecraft.

Although the level of detail varies between the different components, RA models are fairly abstract and focus on system level interactions—not detailed individual subsystems' or components' operation.

This approach has two advantages. First, this provides a method to capture system-level knowledge in a form that can directly command a spacecraft—no costly, error-prone translation into flight software is needed. At best, system requirements are translated into flight rules to check command sequence validity, not generate them.

Secondly, the more abstract models employed are less susceptible to changes when a detailed understanding of the behavior of each subsystem is gained during spacecraft development. Although they need to be adjusted to the new finding, abstract models usually remain structurally unchanged and, therefore, remain the synthesis procedures that RA components use to generate command loads.

Once PS has generated a plan for the next commanding horizon, EXEC receives it and incorporates it into the queues of tasks that it is currently executing. Tasks generated by PS tend to be fairly abstract. EXEC's responsibility is to synchronize the parallel execution of the plan's tasks according to the specifications contained in the plan and to further decompose each task, one at a time, into more detailed steps. This task decomposition eventually results in individual commands being sent, one at a time, to FSW. For example, the abstract task "MICAS take OPNAV images" is decomposed into commanding MICAS to take a number of snapshots while checking that MICAS is kept "ON" during the entire process.

Besides its goal-directed commanding and model-centered approaches, RA puts particular emphasis on robustness of execution and flexibility of response to faults. The mode identification (MI) component of MIR observes EXEC issuing commands, receives sensor observations from monitors, and uses model-based inference to deduce the state of the spacecraft and provide feedback to EXEC. The other component of MIR, mode reconfiguration (MR), serves as a recovery expert, taking as input a set of EXEC constraints to be established or maintained, and recommends a recovery action to EXEC that will achieve those constraints. MIR provides both the MI and MR functions using a single core algorithm and a single declarative model.

Fault protection in RA happens at two different levels:

- *Low-level fault protection loop.* This involves EXEC and MIR in the context of executing a single PS-generated task. Suppose that EXEC is commanding MICAS power on in order to ensure that MICAS is on during the "MICAS take OPNAV images" PS task. It does so by sending an appropriate command to the power driver. MI observes the command and, on the basis of its previous state estimate and its models, predicts the likely next state in which the system will be. This prediction provides a qualitative description of the sensor readings MIR should observe from the spacecraft (e.g., the switch sensor and current sensor should be consistent with MICAS being on). If the expected observations are not received, MI uses its model to hypothesize the most likely cause of the unexpected observations in terms of failures of the spacecraft's components. The information about the new state of the spacecraft hardware is sent to EXEC, which now asks MIR for an action to correct the problem. MIR now activates MR, which, using the same model, determines the least-cost system state that satisfies EXEC's request and one that is reachable from the fault mode. MIR then gives EXEC the first action in a possible sequence that will take the system to that state. Such a recovery might involve resetting a device, attempting a command again, or a complex reconfiguration of the spacecraft to enable a functionally redundant system. EXEC executes the recovery action, under the watchful eye of MIR, and receives further actions from MIR if needed by the recovery process. When the recovery is complete, EXEC continues executing the PS task in a nominal fashion. Note that during this entire process the original PS task is still active and in a "nominal" state. This depends on the time allocated to the task including enough slack to tolerate variations during execution that can be handled by low-level fault protection.
- *High-level fault protection loop.* This involves EXEC and PS. Assume that all recovery actions suggested by MR fail and no more recovery actions are available. MIR infers that MICAS is unusable and communicates this to EXEC. This means that there is no way to execute a command necessary for the success of the "MICAS take OPNAV images" task. Moreover, the assumed conditions for other tasks that may be present

in the plan in the future may now be invalidated. Therefore, EXEC terminates task execution with a failure, discards the rest of the plan, and immediately commands the spacecraft to enter an appropriate "RA standby" mode.[1] EXEC then activates PS by communicating to it the current state of the spacecraft and asks for a new plan. After receiving the initial state from EXEC and the goals from MM, PS generates a new plan that achieves the goals as best as possible within the new, degraded spacecraft configuration. When the plan is ready, PS sends it to EXEC. EXEC then exits the "RA standby" state and resumes normal operations by starting the execution of the new plan.

With the above capabilities, RA allows implementation of *fail-operational* behaviors under a much broader range than is possible in traditional spacecraft commanding. Traditionally, only critical sequences (e.g., Saturn orbit insertion for Cassini) are designed to tolerate a large number of faults without requiring "safing" of the spacecraft. This depends on the cost of analysis and implementation of these sequences. Therefore, in less critical mission phases, a fault event usually requires the intervention of the ground operations team to correct it. With RA, the cost of implementing these scenarios is significantly reduced, making possible an increase of mission productivity and a reduction of cost of operations.

1.2 Detailed Validation Objectives
Validation of a technology with the complexity and the pervasive systemic impact of RA required attention to several different aspects and dimensions.

The first and most obvious objective was to validate the fact that RA could autonomously command a system as complex as a spacecraft for an extended period of time. This translated into the following list of objectives for each RA component.

1.2.1 PS/MM Validation Objectives—
- Generate plans onboard the spacecraft.
- Reject low-priority, unachievable goals.
- Replan following a simulated failure.
- Enable modification of mission goals from ground.

1.2.2 EXEC Validation Objectives—
- Provide a low-level commanding interface.
- Initiate onboard planning.
- Execute plans generated onboard and from the ground.
- Recognize and respond to plan failures.
- Maintain required properties in the face of failures.

1.2.3 MIR Validation Objectives—
- Confirm executive command execution.
- Demonstrate model-based failure detection, isolation, and recovery.
- Demonstrate the ability to update MIR state via ground commands.

1.2.4 Other Objectives—Other validation objectives addressed the impact of the introduction of RA into a "traditional" spacecraft software architecture. From the outset, RA was designed to work in conjunction with existing FSW modules and not to replace them. As a result, fidelity control provided by RA depends on the scope and detail of the spacecraft models. The challenge was to demonstrate that such cooperative arrangement with FSW could indeed be carried out. This consisted of modeling within RA only a specific set of spacecraft subsystems and allowing conventional techniques of FSW control to deal with the remaining control modes of the craft. While there are no software or architectural limitations that would disallow RA to command all subsystems for an extended period of time, the fielding of RA on DS1 was also meant to provide a credible demonstration of the fact that autonomy concepts could be applied within a well-defined scope.

Even within the scope of the autonomy demonstration, it was important to show that adopting RA was not an "all or nothing" proposition and could be commanded with different autonomous-operation levels. Table 1 shows the possible RA autonomy levels, all the way from having EXEC issuing low-level commands from a low-level script analogous to a traditional command (autonomy level 2), to preparing a plan on the ground and uplinking it to the spacecraft for execution (autonomy level 3), to providing closed-loop planning and execution on the spacecraft (autonomy level 6). The DS1 autonomy experiment was designed from the outset to begin at level 3 to build confidence and then migrate to level 6.

Table 1. Autonomy Levels of RA

Level	Ground System	Onboard PS	Onboard EXEC
1	Prepare real-time commands	None	None (executed w/o EXEC involvement)
2	Prepare sequence	None	Execute sequence
3	Prepare plan, upload to EXEC as script	None	Execute plan; "Scripted mode"
4	Prepare plan, upload to planner as goals	Confirm and pass thru the planner	Execute plan; "Planner Mode"
5	Prepare plan, including some unexpanded goals	Complete the plan	Execute plan
6	Define goals	Prepare plan	Execute plan

[1] Note that this is a standby situation only from the perspective of RA. From the point of view of FSW, "RA standby" mode is not a fault mode and does not require FSW fault protection.

The final set of validation objectives involved the development process for autonomy software. This covered a number of separate items:

- Integration of RA with the DS1 FSW, a large and complex system written in a language (C) different from RA (Lisp).
- Adaptation of RA models and scenarios to reflect operational constraints imposed by the flight team, even late in the development process.
- Achievement of high-level of confidence by the DS1 spacecraft team by going through a rigorous test regimen dictated by the team on high-fidelity testbeds.

The level of achievement for each validation objective is discussed below.

1.3 Performance Envelope

Note that these performance and resource figures refer to RA as flown on Deep Space 1 in 1999 in Lisp. Each of the RA engines has been or is being re-architected and ported to C or C++. These new systems may exhibit significantly different performance characteristics:

- Memory—32 Mbytes memory peak, 20 average.
- CPU—
  - RAX ran at priority level just below that of DS1 sequencer (very low).
  - 20% of CPU when planner is idle (only EXEC and MIR are running).
  - 45% of CPU while planner is running (PS, EXEC, and MIR all running).
- The time required to generate plans depends on the plan's complexity. RAX plans took 50 to 90 minutes to generate.
- Telemetry—An average of 10 bits per second. This includes notification as each activity in the plan is executed, current diagnosis for each device monitored by MIR, and a summary of the planner's plan-generation progress. Similar telemetry would be needed for future science missions.
- File space—140 KB for support files, plus approximately 100 KB per stored plan, depending on plan complexity (proportional to number of activities in the plan). Compressed binary executable was 4 MB. At most one plan needs to be stored, though all plans were stored during RAX for validation purposes. RAX also generated a 1MB log.

1.4 Technology Details

RA consists of general-purpose reasoning engines and mission-specific domain models. The engines make decisions and command the spacecraft based on the knowledge in the models. This section describes the details of the reasoning engines and how they interact. The DS1 domain models developed for the flight experiment will be discussed in the flight experiment section.

1.4.1 Planner/Scheduler—PS provides the core of the high-level commanding capability of RAX. Given an initial, incomplete plan containing the initial spacecraft state and goals, PS generates a set of synchronized high-level activities that, once executed, will achieve the goals. In the spacecraft domain, planning and scheduling aspects of the problem need to be tightly integrated. The planner needs to recursively select and schedule appropriate activities to achieve mission goals and any other sub-goals generated by these activities. It also needs to synchronize activities and allocate global resources over time (e.g., power and data storage capacity). Subgoals may also be generated due to limited availability of resources over time. For example, it may be preferable to keep scientific instruments on as long as possible (to maximize the amount of science gathered). However, limited power availability may force a temporary instrument shutdown when other more mission-critical subsystems need to be functioning. In this case, the allocation of power to critical subsystems (the main result of a scheduling step) generates the subgoal "instrument must be off" (which requires the application of a planning step).

PS is able to tune the order in which decisions are made to the characteristics of the domain by considering the consequences of action planning and resource scheduling simultaneously. This helps keep the search complexity under control. This is a significant difference with respect to classical approaches both in Artificial Intelligence and Operations Research, where action planning and resource scheduling are addressed in two sequential problem-solving stages, often by distinct software systems (see [14]).

Another important distinction between PS and other classical approaches to planning is that, in addition to activities, the planner also "schedules" the occurrence of states and conditions. Such states and conditions may need to be monitored to ensure that, for example, the spacecraft is vibrationally quiet when high-stability pointing is required.

These states can also consume resources and have finite durations and, therefore, have very similar characteristics to other activities in the plan. PS explicitly acknowledges this similarity by using a unifying conceptual primitive, the ***token***, to represent both actions and states that occur over time intervals of finite extension. Examples of token semantics details are given further along in this section.

PS consists of a heuristic search engine that deals with incomplete or partial plans. Since the plans explicitly represent time in a metric fashion, the planner makes use of a *temporal database*. As with most causal planners, PS' beginning plan is incomplete; PS attempts to make the plan more complete by posting more constraints in the database.

These constraints originate from the goals and from constraint templates stored in a domain *model* of the

spacecraft. The temporal database and the facilities for defining and accessing model information during search are provided by the Heuristic Scheduling Testbed System (HSTS). The planning engine searches the possible plans for one that satisfies the constraints and achieves the goals. The action definitions determine the space of plans. The constraints determine which of these plans are legal and heavily prune the search space. The heuristics guide the search in order to increase the number of plans that can be found within the time allocated for planning. Figure 2 describes the PS architecture. Additional details on the planner algorithm and its correctness can be found in [10].

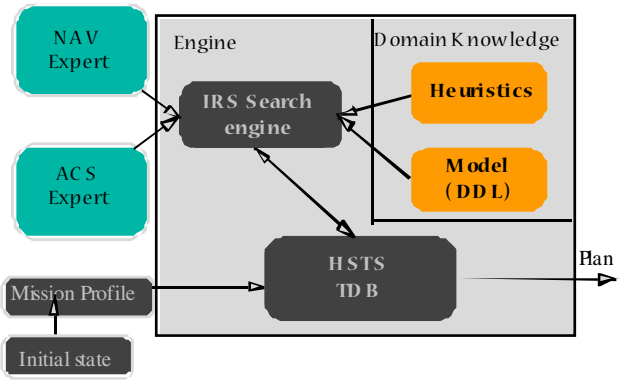


Figure 2. Planner/Scheduler Architecture

The model describes the set of actions, how goals decompose into actions, the constraints among actions, and resource utilization by the actions. For instance, the model will encode constraints such as "do not take MICAS images while thrusting" or "ensure that the spacecraft does not slew when within a DSN communication window." These constraints are encoded in a stylized and declarative form called the Domain Description Language (DDL).

In conventional modes of writing flight software, the constraints in the domain are mixed with the control information. In the model-based approach of RA, the domain model is a distinct entity that encodes the mission-specific flight rules. This means that (in the case of PS) not only are the core engines (the HSTS Temporal Database [TDB] and the Search Engine) reusable across missions, but that the model can be manipulated independently of any other piece of the flight code. (Note that since the heuristics search control information is model dependant, this module would be impacted also.) In addition, the richness of the representation and the declarative form of DDL ensures that mission/systems engineers can have a substantially easier job of understanding and verifying the implementation of the flight rules in RA than would have been possible in conventional FSW. These are some of the advantages that RA brings to a mission.

Each subsystem in the model is represented in the PS database as a set of dynamic *state variables* whose value is tracked over time. *Timelines* are treated as instantiations of state variables and are used interchangeably with state variables in this report. Each dynamic state variable can assume one or more values. A token is associated with a value of a state variable occurring over a finite time interval. Each value has one or more associated *compatibilities* (i.e., patterns of constraints between tokens). A legal plan will contain a token of a given value only if all temporal constraints in its compatibilities are satisfied by other tokens in the plan. Figure 3 shows an example of a set of compatibilities with temporal constraints.

```
(Define Compatibility
  ;; compats on SEP_Thrusting
  (SEP_Thrusting ?heading ?level ?duration)
  :compatibility_spec
  (AND
        (equal (DELTA MULTIPLE (Power) (+ 2416
                                          Used)))
        (contained_by (Constant_Pointing
                                          ?heading))
        (met_by (SEP_Standby))
        (meets  (SEP_Standby)))
)

(Define_Compatibility
  ;; Transitional Pointing
  (Transitional_Pointing ?from ?to ?legal)
  :parameter_functions
    (?_duration_ <- APE_Slew_Duration (?from
                          ?to ?_start_time_))
    (?_legal_    <- APE_Slew_Legality (?from
?to
                                ?_start_time_))
  :compatibility_spec
  (AND
        (met_by (Constant_Pointing ?from))
        (meets  (Constant_Pointing ?to))))

(Define_Compatibility
  ;; Constant Pointing
  (Constant_Pointing ?target)
  :compatibility_spec
  (AND
        (met_by (Transitional_Pointing *
?target
                                          LEGAL))
        (meets (Constant_Pointing ?target *
                                          LEGAL)))
 )
```

Figure 3. Temporal Constraints in DDL

The first compatibility indicates that the master token (which is at the head of the compatibility) is **SEP_Thrusting** (when the Solar Electric Propulsion [SEP] engine is producing thrust [2]), which must be immediately preceded and followed by a **SEP_Standby** token (when the

[2] Solar Electric Propulsion (SEP) is synonymous with IPS.

SEP engine is in a standby mode but has not been completely shut off). The master token must be temporally contained by a constant pointing token; the complete thrusting activity requires 2416 Watts of power. The **Constant_Pointing** token implies that the spacecraft is in a steady state aiming its camera towards a fixed target in space. **Transitional_ Pointing** tokens describe an activity when the spacecraft slews. Figure 4 gives a visual rendering of these compatibilities.

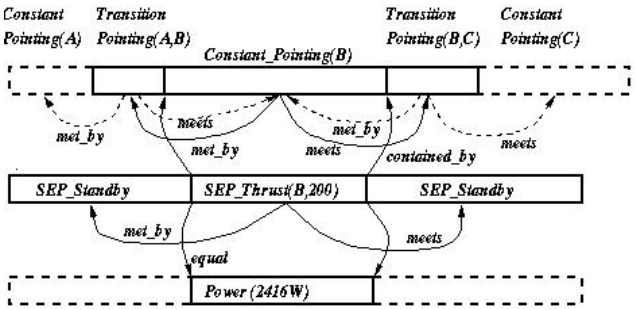


Figure 4. A Plan Fragment Formed by a DDL Model

The timeline approach to modeling is also driven by strong software engineering principles. In a complex domain with different individuals and organizations with varying expertise, timelines provide disparate views of the same domain model across organizational boundaries. For instance, the ground team might want to own and access timelines relating to communication coverage and when DSN access is available, while the attitude control team might want to place high-level goals on the attitude timeline.

Four distinct kinds of state variables are identified. A *goal* timeline will contain the sequence of high-level goals that the spacecraft can satisfy (e.g., the navigate goal described previously). Goal timelines can be filled either by ground operators or by onboard planning experts seen by PS as goal generators. For example, in order to generate the portion of the plan that commands the IPS engine, PS interrogates NAV, which returns two types of goals: the total accumulated time for the scheduling horizon and the thrusting profile to be followed. These two types of information are laid down on separate goal timelines.

Expected device-health information over time is tracked by *health* timelines. The expected profile is communicated by EXEC to PS in the initial spacecraft state. EXEC can communicate that the health of a device has changed even if no fault has occurred. Another kind of state variable is an *internal* timeline. These are only used by the planner to internally organize goal dependencies and subgoaling. Finally, an *executable* state variable corresponds to tasks that will be actually tracked and executed by EXEC.

The RAX PS treats all timelines and tokens within a simple, unified search algorithm. This has advantages. The ground team could force certain behaviors of the spacecraft by including in the mission profile explicit tokens on executable timelines. The additional tokens will be treated by PS as goals, will be checked against the internal PS model, and missing supporting tasks will be automatically expanded to create an overall consistent plan. This will greatly facilitate the work of the ground team. For DS1, such models were understandably more comprehensive and complex, with more timelines, tokens, and compatibilities between differing token types, and required careful consideration during modeling to ensure that interactions between timelines do not result in unanticipated and harmful behaviors generated by the planner.

When a science mission wants to fly the RA planner, primary tasks to be adapted to the mission will be:

- Perform knowledge acquisition to determine all the spacecraft flight rules.
- Encode these flight rules in the DDL model of the spacecraft.
- Design the search control heuristics that will be needed to ensure that the planner is able to produce a valid plan within specified resource (time, CPU) bounds.

Note that this is not to suggest that models can be or ought to be built in an all-or-nothing fashion. On the contrary, the team strongly believes that coming up with a viable plan encapsulating all domain flight rules is an incremental process (You build some and test some).

As mentioned previously, since the underlying search algorithm does not need to be rewritten, the mission will save costs in revalidating the control system and can confine itself to building and validating the model and search control heuristics. Efforts are underway at NASA's Ames Research Center to implement automated tools that will ensure that full coverage of the behaviors anticipated by the models is simulated during the modeling process. Additional efforts are also underway to automatically generate the heuristics from a given model of the domain. This will further allow mission designers and systems staff to build robust and complex models on their own without relying on the AI technologists themselves.

Additional details about the planner can be found in [5 to 7] and [10 to 12].

1.4.2 Executive—The Smart Executive (EXEC) is a multi-threaded, reactive-commanding system. EXEC is responsible for sending the appropriate commands to the various flight systems it is managing. EXEC can replace the traditional spacecraft sequencer or can be used in conjunction with a traditional sequencer to command a complex subsystem (e.g., interferometer).

EXEC is a multi-threaded process that is capable of asynchronously executing commands in parallel. In addition to a traditional sequencer's capabilities, EXEC can:

- Simultaneously achieve and maintain multiple goals (i.e., system states) by monitoring the success of commands it issues and reactively re-achieving states that are lost.
- Perform conditional sequencing. Commands can be dependent on conditions that occur at execution time.
- Perform event-driven commands, as opposed to traditional sequencers that are time-driven (i.e., taking a sequence of pictures based on the results of monitoring a range sensor).
- Perform high-level commanding and run-time task expansion. EXEC provides a rich procedural language, Execution Support Language (ESL) [1], in which spacecraft software/model developers define how complex activities are broken up into simpler ones. To increase robustness, a procedure can specify multiple-alternate methods to achieve a goal.
- Perform sequence recovery. In the event an executing sequence command fails, EXEC suspends executing the failed sequence and attempts a recovery, either by executing a pre-specified recovery sequence, such as reissuing the command or consulting a recovery expert (e.g., MIR). Once the desired state of the failed command is achieved, the suspended sequence is restarted.
- Execute a temporally-flexible sequence (or plan). In order to decrease the probability of a sequence failing, time ranges can be specified for executing and achieving the desired state for each command.
- Manage resources. As a multi-threaded system, EXEC can work on multiple tasks simultaneously. These tasks may compete for system resources within the constraints not already resolved by ground or the planner. EXEC manages abstract resources by monitoring resource availability and usage, allocating resources to tasks when available, making tasks wait until their resources are available, and suspending or aborting tasks if resources become unavailable due to failures (such as a device breaking). See [1] and [2] for a more detailed discussion.

Figure 5 illustrates key functions of EXEC.

EXEC achieves multiple tasks, sending the appropriate control commands (decomposed from high-level commands) to the flight software. The tasks also lock properties that need to be maintained. For example, if a task commands a switch ON, the switch property will be locked ON. Monitors (and MIR) determine if it is consistent to believe that the switch is ON. Since EXEC stores this state in its state database should the inferred state of the switch change, the database will be updated and an event created, thereby signaling a change. If the signaled event violates a property lock, an EXEC property thread interrupts those tasks that subscribed to that property lock. It will then attempt to achieve the state of the switch being ON using its own recovery mechanism or by consulting a recovery expert (e.g., MIR). If the switch cannot be turned ON in time, a hard deadline that is being tracked is missed; in response EXEC commands the spacecraft into a safe, wait state while it requests a new plan from the planner that takes into account that the switch cannot be turned ON.

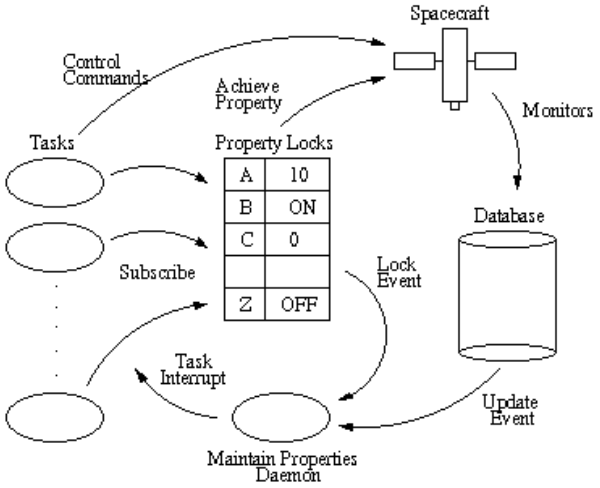


Figure 5. An Overview of the Remote Agent Executive

Recoveries may be as simple as sending another command to turn a switch ON, or may be complex, such as when multiple subsystems are tightly coupled. For example, consider two coupled DS1 subsystems: the engine gimbal and the solar panel gimbal. A gimbal enables the engine nozzle to be rotated to point in various directions without changing the spacecraft orientation. A separate gimbal system enables the solar panels to be independently rotated to track the sun. In DS1, both sets of gimbals communicate with the main computer via a common gimbal drive electronics (GDE) board. If either system experiences a communications failure, one way to reset the system is to power-cycle (turn on and off) the GDE. However, resetting the GDE to fix one system also resets the communication to the other system. In particular, resetting the engine gimbal to fix an engine problem causes temporary loss of control of the solar panels. Thus, fixing one problem can cause new problems. To avoid this, the recovery system needs to take into account global constraints from the nominal schedule execution, rather than just making local fixes in an incremental fashion; the recovery itself may be a sophisticated plan involving operations on many subsystems.

Domain-code developers use ESL to create high-level commands that EXEC decomposes and executes at run-time depending on the spacecraft state. The ESL code in Figure 6 illustrates multiple methods for achieving IPS thrusting at a desired level depending on the current state of execution. If IPS is in standby mode, ACS is commanded to change control modes only after the desired IPS thrust level has been confirmed.

```
(to achieve (IPS THRUSTING ips level)
 ((ips is in standby state p ips)
  (sequence (achieve (power on? 'ega-a))
  (command with confirmation
   (send-ips-set-thrust-level level))
  (command with confirmation
   (send-acs-change-control-mode
    :acs-tvc-mode))))
 ((ips in thrusting state p ips)
  (command with confirmation
   (send-ips-change-thrust-level level)))
 (t (fail :ips achieve thrusting)))
```

Figure 6. Multiple Methods in ESL for Achieving Thrust

EXEC and its commanding language, ESL, are currently implemented using multi-threaded Common Lisp. A new version of EXEC is currently under development in C/C++. The internal EXEC code is designed in a modular, layered fashion so that individual modules can be designed and tested independently. Individual generic device knowledge for RAX is implemented based on EXEC's library of device management routines to support addition of new devices and reuse of the software on future missions.

More details about EXEC can be found in [1 to 3] and [7].

1.4.3 Diagnosis and Repair—The diagnosis and repair engine of RA is the Mode Identification and Reconfiguration (MIR) system. MIR eavesdrops on commands that are sent to the onboard hardware managers by EXEC. As each command is executed, MIR receives observations from spacecraft sensors, abstracted by monitors in lower-level device managers for ACS, Bus Controller, and so on. MIR uses an inference engine called Livingstone to combine these commands and observations with declarative models of the spacecraft's components to determine the current state of the system (mode identification [MI]) and report it to EXEC. EXEC may then request that Livingstone return a set of commands that will recover from a failure or move the system to a desired configuration (mode reconfiguration [MR]). Figure 7 illustrates the data flow among the spacecraft, EXEC, and Livingstone.

MI is responsible for identifying the current operating or failure mode of each component in the spacecraft, allowing EXEC to reason about the state of the spacecraft in terms of component modes, rather than in terms of low-level sensor values. MR is responsible for suggesting reconfiguration actions that move the spacecraft to a configuration that achieves all current goals as required by PS and EXEC, supporting the run-time generation of novel reconfiguration actions. Though in RA, Livingstone is used only to recover following a component failure. Livingstone's MR capability can be used to derive simple actions to reconfigure the spacecraft at any time. Thus, Livingstone can be viewed as a discrete model-based controller in which MI provides the sensing component and MR provides the actuation component. Livingstone uses a single set of models and core algorithms to provide both the MI and MR functions.

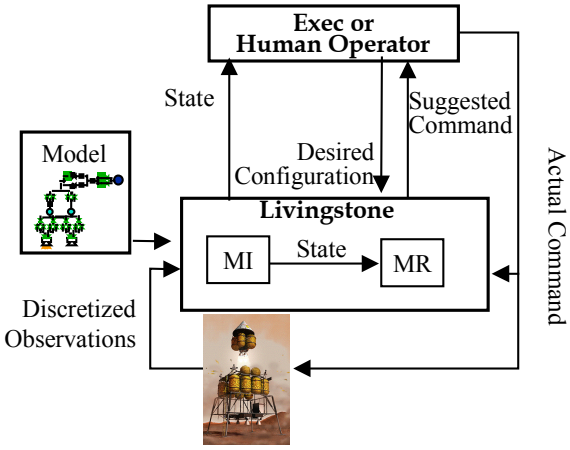


Figure 7. Livingstone Processing Cycle

To use Livingstone, one specifies how the components of interest are connected. For each component type, one then specifies a finite state machine that provides a description of the component's nominal and failure behavior.

Figure 8 graphically depicts a Livingstone model of the Cassini main-engine subsystem. An important feature is that the behavior of each component state or mode is captured using abstract, or qualitative, models [3, 4]. These models describe qualities of the spacecraft's structure or behavior without the detail needed for precise numerical prediction, making abstract models much easier to acquire and verify than quantitative engineering models. Examples of qualities captured are the power, data, and hydraulic connectivity of spacecraft components and the directions in which each thruster provides torque. While such models cannot quantify how the spacecraft would perform with a failed thruster, for example, they can be used to infer which thrusters are failed given only the signs of the errors in spacecraft orientation. Such inferences are robust since small changes in the underlying parameters do not affect the abstract behavior of the spacecraft.

Livingstone's abstract view of the spacecraft is supported by a set of fault-protection monitors that classify spacecraft sensor output into discrete ranges (e.g., high, low, nominal) or symptoms (e.g., positive X-axis attitude error). One

objective of the RA architecture was to make basic monitoring capability inexpensive so that the scope of monitoring could be driven from a system engineering analysis instead of being constrained by software development concerns. To achieve this, monitors are specified as a dataflow scheme of feature extraction and symptom-detection operators for reliably detecting and discriminating between classes of sensor behavior.

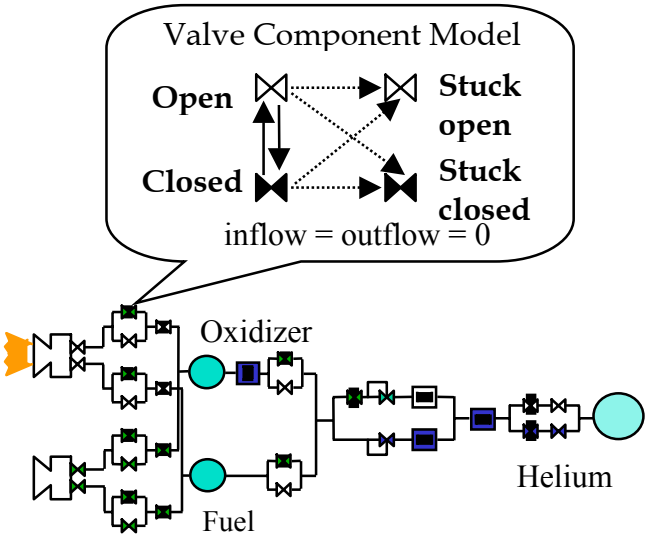


Figure 8. Livingstone Model of the Cassini Main Engine Subsystem

The software architecture for sensor monitoring is described using domain-specific software templates from which code is generated. Finally, all symptom detection algorithms are specified as restricted Harel-state transition diagrams reusable throughout the spacecraft. The goals of this methodology are to reuse symptom-classification algorithms, reduce the occurrence of errors through automation and to streamline monitor design and test.

It is important to note that the Livingstone models are not required to be explicit or complete with respect to the actual physical components. Often, models do not explicitly represent the cause for a given behavior in terms of a component's physical structure. For example, there are numerous causes for a stuck switch: the driver has failed, excessive current has welded it shut, and so on. If the observable behavior and recovery for all causes of a stuck switch are the same, Livingstone need not closely model the physical structure responsible for these fine distinctions.

Models are always incomplete in that they have an explicit unknown failure mode. Any component behavior that is inconsistent with all known nominal and failure modes is consistent with the unknown failure mode. Therefore, Livingstone can infer that a component has failed, though failure was not foreseen and left unmodeled because it was not possible.

By modeling only to the detail level required to make relevant distinctions in diagnosis (distinctions that prescribe different recoveries or system operations), a system with qualitative "common-sense" models that are compact and quite easily written can be described. Figure 9 provides a schematic overview of Livingstone's processing.

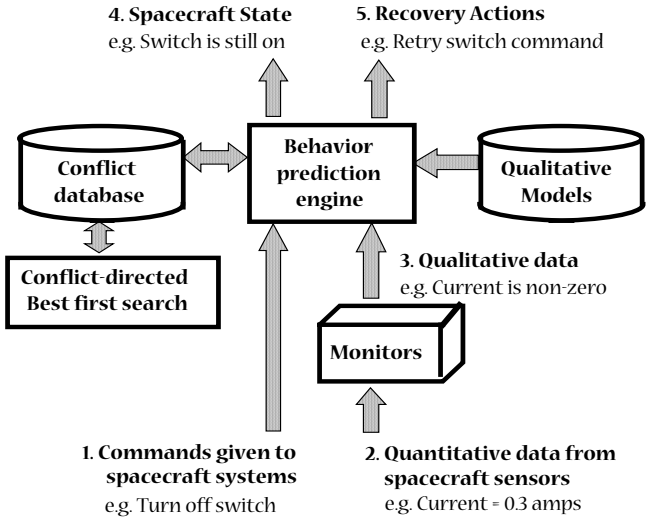


Figure 9. Schematic of Livingstone Processing

Livingstone uses algorithms adapted from model-based diagnosis (see [9]) to provide the above functions. The key idea underlying model-based diagnosis is that a combination of component modes is a possible description of the current state of the spacecraft only if the set of models associated with these modes is consistent with the observed sensor values. Following de Kleer and Williams [8], MI uses a conflict-directed, best-first search to find the most likely combination of component modes consistent with the observations. Analogously, MR uses the same search to find the least-cost combination of commands that achieve the desired goals in the next state. Furthermore, both MI and MR use the same system model to perform their function.

The combination of a single-search algorithm with a single model, and the process of exercising these through multiple uses, contributes significantly to the robustness of the complete system. Note that this methodology is independent of the actual set of available sensors and commands. Furthermore, it does not require that all aspects of the spacecraft state are directly observable, providing an elegant solution to the problem of limited observability.

The use of model-based diagnosis algorithms immediately provides Livingstone with a number of additional features.

First, the search algorithms are sound and complete, providing a guarantee of coverage with respect to the models used. Second, the model-building methodology is modular, which simplifies model construction and maintenance and supports reuse. Third, the algorithms extend smoothly to handling multiple faults and recoveries that involve multiple commands. Fourth, while the algorithms do not require explicit fault models for each component, they can easily exploit available fault models to find likely failures and possible recoveries.

Since the flight experiment, Livingstone has been ported to C++ and significantly improved in the areas of both MI and MR. The improved Livingstone is scheduled to be test flown on both the X-34 and X-37 experimental vehicles. Additional technical details about Livingstone can be found in [4] and at http://ace.arc.nasa.gov/postdoc/livingstone

1.5 Subsystem Interdependencies

The Remote Agent Experiment Manager (RAXM) is the flight software interface to the Remote Agent Experiment (RAX) and isolates the RA software from the rest of the FSW via a set of clean application programming interfaces (APIs).

In addition, RAXM provides a terminal in the point-to-point message-passing protocol used by the DS1 flight software (see Figure 1). RAXM in particular is tasked with handling three messages throughout the mission: RAX-START, RAX-STOP, and RAX-ABORT; RA software is operational only during the times between a RAX-START and either RAX-STOP or RAX-ABORT. RAX-START is used by RAXM to decompress the RAX Lisp image and initiate RAX control. The RAX-STOP is implemented to cleanly terminate RAX at the end of the experiment under nominal circumstance, while the RAX-ABORT is intended to kill the RAX process in the event of an abnormality detected by RAXM. At all other times, RAXM discards all incoming messages, allowing all FSW subsystems that interact with RAX to be ignorant of the RAX state.

When RA runs, RAXM handles and dispatches all incoming messages related to RA: some of the messages are handled by RAXM, others are passed through to RAX itself. Similarly, outgoing messages from RAXM can be due either to RAXM or to RAX itself.

Like the code for other flight software subsystems, RAXM is written in the C programming language and is part of the launch load. As a result, the interfaces for RAX needed to be specified early.

The computational resources (CPU fraction, memory space, telemetry buffers and downlink, etc.) required by RAXM when RA was not running were insignificant. This was, by design, a way to mitigate the impact of the RA technology demonstration on DS1.

1.6 Preparing Lisp for Flight

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

To fit within the 32 MB memory allocation and the CPU fraction constraints, the RA team thoroughly analyzed their code for memory and performance inefficiencies and applied 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 RA testing, peak memory usage was measured at about 29 MB. Upon completion of the transduction process, the RA 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 RA run, with the file being inflated directly into the 32-MB RA memory space. Use of this custom compression drastically reduced the file-uplink time and kept RA-file space usage within the agreed limits.

Added to the challenge of working within resource constraints was the challenge of working out the complicated interfaces between RA and the rest of the flight software. 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 analyzed many function-parameter variations.

Another problem arose in preparing the Lisp multi-threading 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, Lisp's

approach—to thread preemption to use a lower frequency wakeup call implemented with flight software timing services—had to be significantly changed.

Most of the late integration problems with RA Lisp arose because of the VxWorks port. As RA 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 RA 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. The entire undertaking benefited from the dedicated efforts of both Harlequin and the DS1 FSW team.

2.0 The Remote Agent Experiment

During the DS1 mission, the Remote Agent technology was validated with an experiment, the Remote Agent Experiment (RAX). The flight experiment was conducted between May 17, 1999, and May 21, 1999, and achieved all of the technology-validation objectives. However, the story is incomplete without reporting the valuable data gathered during development and testing on the ground. In the case of RA, this is particularly important since the technology is intended as a tool to support system engineering and operations for a mission, rather than simply provide the resulting autonomous capabilities. By quantitatively analyzing the history of RAX's development, we can evaluate how well the current state of the technology supports its ultimate goals. This can also help identify weak points that require further research and development.

RAX and the team attempt to evaluate the development and testing experience with respect to the features of the technology is described here. First, RAX must be put into the larger perspective of RA's technology evolution. Then the subsystems and fault modes modeled, the experiment scenarios, and the expected in-flight behavior are described. Then how RAX was developed and validated and the details of the flight experiment are discussed. Then the effectiveness and cost of development and testing are successively analyzed. The analysis is supported by the actual problem reports filed in the RAX problem-tracking system during development. Lessons learned conclude this section.

2.1 Historical Perspective

Development of the RA technology effectively started in May 1995. At that time, spacecraft engineers from JPL and Artificial Intelligence (AI) technologists from Ames Research Center (ARC) and JPL started working together on the New Millennium Autonomy Architecture rapid Prototype (NewMAAP), a six-month effort intended to assess the usability of AI technologies for onboard flight operations of a spacecraft 17]. NewMAAP yielded proof of concept of an autonomous agent that formed the fundamental blueprint for Remote Agent. NewMAAP also helped build the team of technologists that continued development of Remote Agent on DS1.

The successful demonstration of NewMAAP in November, 1995 led to the selection of RA as one of the components of the autonomy flight software for DS1. Between December 1995 and April 1997, the RA team was part of the DS1 flight software team. This led to the development of the three engines of the RA component technologies and included a substantial speed up of the MIR inference engine (see [4]), the design and implementation of the ESL language used by EXEC (see [1]), and the design and implementation of the heuristic search engine for PS together with the language to formulate search heuristics.

Regarding the overall Remote Agent architecture, the fault protection protocols were designed and implemented, both at low level (involving EXEC and MIR) and at the high level (involving EXEC and PS). During this period, the team acquired much of the high-level system knowledge needed to model DS1 cruise operations (including image acquisitions of beacon asteroids for AutoNAV, timed IPS thrusting, and file uplink and downlink) and other DS1 capabilities required for asteroid encounter activities.

In March 1997, the DS1 autonomy flight software was substantially overhauled and DS1 adapted the Mars Pathfinder (MPF) flight software as the basis for its flight software. Also, RA was re-directed to become an experiment operating for at most six days during the mission on a cruise scenario, including AutoNAV orbit determination and IPS-timed thrusting. RAX re-used much of the software developed during the previous autonomy flight software phase of DS1. RAX focused on the process of testing each RA component, integrating and testing them into the complete RA, and integrating and testing RA together with the DS1 flight software on the flight processor. Shortcomings found during the development and testing phases required several extensions and re-designs of domain models and the reasoning engines.

This document is focused solely on RAX and makes use of the detailed development and testing records maintained during this phase. However, when the technology readiness conclusion is presented, it will reflect the entire Remote-Agent's development history.

Table 2 shows the highlights of the RAX, starting with the RAX development effort after the redirection of the flight software to MPF. Due to this change, a requirement was

imposed on the RAX team to keep interactions with the flight team to a minimum. From the beginning, RAXM was identified as being the primary interface to RA and part of the launch load of DS1; delivery of RAXM was initiated by December after negotiating all interfaces with FSW. This was the only significant interaction the team had with the DS1 flight team till February 1999, three months prior to activation of RAX. Integration of RAX on the *Radbed* high fidelity testbed was completed during April 1998, which allowed the team to understand the timing characteristics of RA in flight. The RAX Software Delivery Review in September allowed the team for the first time to show the DS1 project the progress the team was making and explain the expected behavior of RAX during flight. November of the same year, barely five months before the experiment, was the first time RAX software ran on a *Papabed* after interfacing with the actual FSW. It took another month to actually produce a plan and execute it on this testbed. The RAX delivery entered the final deliverable phase in February 1999 with code development frozen and bug fixes under a strict change-control regime. RAX was finally initiated on DS1 on May 17, 1999.

Table 2. Significant Events for the RAX Project

Event	Date
Start of RAX development	April 1997
Delivery of RAX Manager to flight software	December 1997
RAX integrated on the flight processor	April 1998
Project Software Delivery Review	September 1998
DS1 launch	October 1998
First run of RAX with FSW on high-fidelity hardware simulation	November 1998
Beginning of M5 DS1 project phase	February 1999
RAX experiment	May 1999

Below is a detailed description of the DS1 subsystems modeled in RAX and the scenarios on which RA was exercised during RAX development and testing.

2.2 Domain Models

The team only developed domain models for the subsystems and fault modes that were necessary for the experiment. Table 3 describes the timelines modeled by the planner. Table 4 and Table 5 list the components and module models developed for MIR while Table 6 shows the modeled EXEC timelines. These models captured the following subsystems and resources:

- Ion Propulsion System
  Detect and command standby through thrusting states.
- Attitude Control Subsystem.
  PS planned attitude changes requested by NAV (IPS attitudes and beacon asteroids) or specified as goals in the mission profile. These attitudes were restricted in the model to slews that maintained the solar panels on-Sun. For the experiment, the NAV profiles and goals were specified to further limit the attitudes to either high gain antenna (HGA) at Earth (the default attitude and the IPS thrust attitude) or MICAS bore-sight at a beacon asteroid.
- MICAS.
  PS planned data takes and low-voltage power on/off. Switch status and commands were modeled, but the switch commands are not actually issued. (See the scenario description for why this is so.)
- Power.
  PS tracked predicted peak-power usage for each activity in the plan (e.g., IPS thrusting, MICAS on) and ensured that the total would never exceed the available power from the solar panels, as predicted by the operations team and supplied in the mission profile. MIR modeled a portion of the power distribution system and its relays in order to confirm operation of the switches commanded by RAX and distinguish between failures in the power system and erroneous sensor readings. MIR modeled switches not commanded by RA so that it could request the experiment be aborted if the power system was in a state out of scope for the experiment.
- Reaction Control System.
  MIR modeled the thruster pallets, thrusters, and valves of the RCS system in order to determine the health of the various components from errors in attitude and recommend which control mode to utilize.
- Data System.
  MIR modeled the 1553 bus and a subset of the remote terminal devices on it in order to monitor for remote terminal hangs and recommend resets. Resetting was limited to the Power Actuation and Switching Module (PASM) instrument. Other remote terminals were modeled in order to allow MIR to request the experiment be aborted if certain out-of-scope data-system problems occurred.
- Sensors.
  MIR modeled a subset of the switch position and current sensors onboard DS1 as fallible components in order to allow sensor failure as an explanation for unexpected observations.
- Remote Agent.
  PS models aspects of the operation of RA itself. For example, the Planner timeline allows PS to plan time for its next planning activity. The Special Activities timeline allows PS to schedule execution of scripts that (unbeknownst to RA) will cause simulated failures onboard the spacecraft.

Table 3. Summary of Planner Models for RAX

Subsystem	State Variables	Value Types	Compatibilities	Comments
MICAS	Executable: 2 Health: 1	7	14	Models the health, mode, and activity of the MICAS imaging camera. RAX demonstrates fault injection and recovery for this device as part of the 6-day scenario.
Navigation	Goal: 1 Executable: 1 Internal: 1	5	6	To schedule orbit determination (OD) based on picture taking activity.
Propulsion & Thrust	Goal: 2 Executable: 1 Internal: 1	9	12	Based on thrust schedule generated by the NAV module, the planner generates plans to precisely activate IPS in specific intervals based on constraints in the domain model and is the most complex set of timelines and subsystem controlled by the planner.
Attitude	Executable: 1 Health: 1	4	4	Enables the planner to schedule slews between constant pointing attitudes when the spacecraft maintains its panels towards the Sun. The targets of the constant pointing attitudes are imaging targets, Earth (for communication), and thrust direction (for IPS thrusting).
Power Manage-ment	Goal: 1 Internal: 1	2	1	Allows the planner to ensure that adequate power is available when scheduling numerous activities simultaneously.
Executive	Goal: 1 Executable: 1	2	7	Allows modeling of low-level sequences, bypassing planner models, giving Mission Ops the ability to run in sequencing mode with the RA.
Planner	Executable: 1	2	2	To schedule when EXEC requests the plan for the next horizon.
Mission	Goal: 1	2	2	Allows MM and PS to coordinate activities based on a series of scheduling horizons updatable by Mission Ops for the entire mission.

Table 4. DS1 Hardware Modeled as Components in MIR

Component Class	# in Model	Modes
ion propulsion system (IPS)	1	Standby, Startup, Steady State Thrusting, Shutdown, Beam Out, Controller Hung, Unknown
remote terminal	6	Nominal, Resettable Failure, Power-cyclable Failure, Unknown
attitude control	1	TVC, X for Y, Z for Y, X for Y Degraded, Z for Y Degraded, X for Y Failed, Z for Y Failed, TVC Failed, Unknown
switch	12	On, Off, Popped On, Popped Off, Stuck On, Stuck Off, Unknown
switch sensor	12	Nominal, Stuck On, Stuck Off, Unknown
current sensor	3	Nominal (reported value = real value), Unknown (values unconstrained)
thruster valve	8	Nominal, Stuck Closed, Unknown
thruster	8	Nominal, Unknown
propellant tank	1	Non-empty, Unknown (thruster hydrazine out or otherwise unavailable)
bus controller	1	Nominal, Unknown
vehicle dynamics	1	Nominal (this is a qualitative description of force and torque)
power bus	3	Nominal (failure considered too fatal and remote to involve in diagnosis)

Table 5. DS1 Hardware Modeled as Modules in MIR

Module	# in Model	Subcomponents
power relay	12	1 switch, 1 switch sensor
power distribution unit	1	12 relays, 3 power buses, 3 current sensors, 1 remote terminal
generic RT subsystem	3	1 remote terminal (models RT for devices MIR does not otherwise model)
IPS system	1	1 IPS, 1 remote terminal
thruster pallet	4	2 thrusters (X facing and Z facing)
reaction control system	1	4 thruster pallets
PASM subsystem	1	1 remote terminal

Table 6. Timelines and Their Respective Tokens by Module (EXEC's perspective)

Module	Timeline	Token	Description
ACS	Spacecraft Attitude	constant_pointing_on_sun	Point vector at Target, Solar Panels at Sun
		transitional_pointing_on_sun	Turn vector to Target, Solar Panels at Sun
		poke_primary_inertial_vector	Small attitude change
	RCS_Health	rcs_available	Maintain information on thruster status
	RCS_OK	maintain_rcs	Set and maintain desired RCS mode
MICAS (Camera)	MICAS_Actions	micas_take_op_nav_image	Take a set of navigation pictures
	MICAS_Mode	micas_off	Keep MICAS off
		micas_ready	Keep MICAS on
		micas_turning_on	Turn MICAS off
		micas_turning_off	Turn MICAS on
	MICAS_Health	micas_availability	Ensure MICAS is available for use
Op-Nav	Obs_Window	obs_window_op_nav	Wait for a specified duration
	Nav_Processing	nav_plan_prep	Send message to prepare navigation plan
PASM	PASM Available	pasm_monitor	Monitor the PASM switch
SEP	SEP	sep_standby	Achieve and maintain IPS standby state
		sep_starting_up	Achieve and maintain IPS start-up
		sep_thrusting	Maintain a thrust level
		sep_shutting_down	Stop thrusting and go to standby state
	SEP_Time Accum	accumulated_thrust_time	Monitor thrust time accumulated
	SEP_Schedule	thrust_segment	Specifies desired thrust level and vector
	SEP_Thrust Timer	max_thrust_time	Set a timer and stop thrusting if time reached
		thrust_timer_idle	Thrust timer is off
Planner	Planner_ Processing	planner_plan_next_horizon script_next_horizon	Request and get next plan from planner. Run the next scripted plan
General	EXEC Activity	exec_activity	Execute a low-level sequence file passed as a parameter
	EXEC_Eval	exec_eval_watcher	Process a specified script

2.3 Experiment Scenarios

The RAX experiment proposal contained a 12-hour scenario and a 6-day scenario. The 12-hour scenario was designed as a confidence builder for the DS1 project. The 6-day scenario was to be run following successful completion of the 12-hour scenario. Together, the 12-hour and 6-day scenarios demonstrate all RAX validation objectives and were used for all RAX integration and testing until the beginning of March 1999. Then the DS1 project levied additional constraints on how the spacecraft could be commanded and

specified that RAX should produce 12 hours of thrust or less. The team responded by developing a 2-day scenario that met the additional commanding constraints and provided 12 hours rather than 4 days of thrusting. The DS1 project viewed very favorably the group's ability to quickly respond with a new scenario for these new constraints. Each scenario is described below.

2.3.1 Twelve-hour Scenario—The twelve-hour scenario involves neither onboard planning nor thrusting with IPS. Rather, the plan is generated on the ground, uplinked to the spacecraft, and executed by EXEC and MIR. The scenario includes imaging asteroids with the MICAS camera to support optical navigation, a simulated sensor failure scenario, and demonstration of low-level commanding from a script through RAX to flip a switch. The planning of optical navigation imaging provides 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.

2.3.2 Six-day Scenario—The 6-day scenario includes both onboard planning and operating IPS and is a full-up test of RA. The scenario is divided into 2 planning horizons. At the start of the scenario, PS generates a plan for the first horizon that included MICAS imaging for optical navigation and IPS thrusting. Execution of the first plan also includes a ground command to modify the goals for the second horizon. At the end of the optical navigation window, PS plans to switch off the MICAS camera. However, a stuck-on-failure injection in the camera switch prevents RA from turning off the camera, leading to a plan failure. Repeated attempts to recover the problem fail. This leads to a replan, which produces a second plan with the camera being left on. The second plan also includes an activity to produce a plan for the second horizon (the third plan in the scenario), which is executed back-to-back with the second plan. While the second plan is being executed, the switch-failure injection is undone and ground informs MIR that the switch is now fixed. The execution of the third plan includes IPS thrusting, optical-navigation imaging, and two simulated failures, a communication failure on the 1553 bus and a thruster-valve-stuck-closed failure.

The MICAS stuck-on failure demonstrates how MIR and EXEC can make repeated attempts to recover a camera switch until it is deemed permanently stuck. The 1553 bus remote-terminal failure illustrates successful recovery of communication with a device by resetting its remote terminal (RT). In the ACS thruster-valve-stuck-closed failure, MIR infers from an attitude error and models of the spacecraft dynamics that one of a particular pair of thruster valves is stuck closed. MIR is then able to recommend that no matter which one of the two valves is stuck, switching ACS control modes will mitigate the problem.

2.3.3 Two-Day Scenario—In March 1999, the DS1 project analyzed the 6-day plan and decided that RA should not switch the MICAS camera off after each use due to concerns about thermal effects. In addition, RA would be required to produce at most 12 hours of IPS thrusting to ensure that DS1 would be on track for its asteroid encounter in July 1999.

The 2-day scenario was created that is 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 mission profile was adjusted so that PS would produce plans with only about 12 hours of IPS thrusting. This scenario is similar to the standard DS1 cruise phase, which consists of IPS thrusting punctuated with periodic optical-navigation activities. This baseline demonstrated RAX's basic commanding capabilities.

This scenario retains the simulated faults that exercise RAX's robust fault-response capabilities. Since the team could not depend on failures occurring during the experiment, failures were simulated by injecting false monitor readings consistent with the failures. While simulations are necessary for demonstration, the RAX is fully responsible for responding to real failures within its limited scope occurring during the experiment. To avoid potential conflicts between RAX and the flight-software fault-protection mechanism (FP) the RAX response threshold is a little lower than that FP to allow RAX to detect and respond to faults before FP does. If RAX fails to resolve a fault quickly enough, the FP response would be triggered (since the fault is still active). The FP response is to terminate RAX and resolve the fault.

2.4 RAX Development

RAX was developed on a number of platforms of decreasing processor speed and increasing level of hardware and software fidelity (see Table 7).

The team adopted a continuous-integration-development process with new software capabilities being first developed on the *UNIX* platform. Before they could be incorporated in a software build and be appropriately tagged, new features or bug fixes had to run to completion a representative set of scenarios. As time progressed, testbeds of higher and higher fidelity became available. As this happened, the requirements for acceptance of software modifications became more and more demanding since the scenarios had to run on all available platforms.

Besides the speed of the processors, another factor effecting productivity was the simulated-clock speed. The *UNIX, Babybed* and *Radbed* platform made use of a low-fidelity simulation developed by the RAX team, which essentially only simulated the message traffic and the delays in

receiving responses from flight software. This meant that the simulator was allowed to advance the clock at "warp" speed, simulating in a second, several minutes or hours of actual elapsed time. Time warping allowed the team to run to completion the full 6-day scenario in less than an hour, tremendously increasing the productivity during development and testing on such lower-fidelity testbeds.

Table 7. Development Testbeds for RAX

Platform	Fidelity	CPU/OS	Hardware	Availability	Speed
DS1 Spacecraft	Highest	Rad6000 VxWorks	Flight	1 for DS1	1:1
DS1 Testbed	High	Rad6000 VxWorks	Flight spares + DS1 sims	1 for DS1	1:1
Hotbench	High	Rad6000 VxWorks	Flight spares + DS1 sims	1 for DS1	1:1
Papabed	Medium	Rad6000 VxWorks	DS1 sims only	1 for DS1	1:1
Radbed	Low	Rad6000 VxWorks	RAX sims only	1 for RAX	1:1
Babybed	Very Low	PowerPC VxWorks	RAX sims only	2 for RAX	7:1
UNIX	Lowest	SPARC UNIX	RAX sims only	unlimited	35:1

Since the higher-fidelity testbeds could not be warped in time because of interfaces to the actual FSW code, it induced the team to devise reduced-length scenarios that would exercise in a few hours of actual clock time most or all of the functionalities included in the full, multi-day flight scenarios. These shorter scenarios led to exercising RAX under stress conditions complementary to those addressed by the formal test process. As a consequence, continuous integration over the course of testing and development led to the discovery and correction of a large quantity of RAX software problems. Table 8 shows the highlights of the testing on the various testbeds.

Table 8. Dates of RAX Readiness on Testbeds

Testbed	Date
UNIX	August 1997
Babybed	February 1998
Radbed	April 1998
Papabed	November 1998
Hotbench	March 1999
DS1 testbed	April 1999
DS1 spacecraft	May 1999

2.5 Ground Tests

To qualify RAX to run onboard the DS1 spacecraft, RAX underwent a rigorous program of formal tests. The tests covered nominal and off-nominal situations and exercised at all levels of fidelity available on the ground testbeds each Remote Agent component individually, the integrated RAX product, and RAX together with the flight software.

Autonomous systems like RA pose testing challenges that go beyond those usually faced by more traditional flight software. In fact, the range of possible behaviors exhibited by an autonomous system is usually very large. This is consistent with the expectation that the system operate robustly over a large range of possible values of system parameters. However, an exhaustive verification of all situations would require an unmanageably large 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. For example, in RAX the team could run only 10 tests in four weeks on the DS1 *Hotbench*. To cope with these time and resource limitations, the team employed a "baseline testing" approach to reduce the number of tests. Moreover, the team exploited whenever possible the lower-fidelity testbeds to validate system behaviors for which there was high confidence that the test results would extend to higher-fidelity situations. The high-fidelity testbeds were used mostly in nominal situations and under stress conditions requiring RAX to guarantee spacecraft safety.

The baseline scenario was the scenario that was expected to execute in flight initially the 6-day and 12-hour scenarios and subsequently the 2-day scenario. The team tested a number of nominal and off-nominal variations around these scenarios. These covered variations in spacecraft behavior that might be seen during execution and changes to the scenario that might be made prior to execution. Changes included variations to the goals in the mission profile, variations in when faults might occur, and variations in the FSW responses.

The architecture of RA allowed the team to run certain tests on lower-fidelity testbeds and be confident results would hold on higher-fidelity testbeds. Specifically, RA commands and monitors the spacecraft through well-defined interfaces with 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 let the team exercise a wide range of nominal and off-nominal behaviors on the *Babybeds* and the *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 the *Hotbench* and on the *DS1 Testbed*. Functional testing RA's PS component was a special case because it required extensive use of the *UNIX* testbeds.

The rest of this section describes the tests on each testbed.

2.5.1 UNIX—The PS team made extensive use of the *UNIX* testbed for PS unit testing throughout the formal testing process. Use of the *UNIX* testbed was critical since PS is a computationally intensive task and could not take advantage of time warping. Both in nominal- and fault-response situations, PS essentially operated as a batch process with practically no reliance on the underlying real-time system (e.g., timing services). This let the team repeatedly run a batch of 269 tests with several variations of initial states, goals of the planner, and model parameters (e.g., possible turn durations). Tests were repeated for each release of the RA software, providing a certain measure of regression testing for the PS software.

2.5.2 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. For PS, this involved a batch of 54 tests comprising some of the tests in the *UNIX* batch plus tests devised to test system-level responses of PS (e.g., response to invalid initial states or to an asynchronous kill message sent by EXEC). The repetition of the tests from *UNIX* both validated the complete functional equivalence of PS between *UNIX* and PPC and verified the acceptability of PS performance on the real-time architecture. MIR was exercised on a batch of 110 tests covering the likeliest failure contexts. The PS and MIR tests were used for testing EXEC. A suite of twenty additional scenarios exercised the system-level interaction of all modules. These tests were run rapidly on the *Babybeds* and *Radbed* with time warping. Running a scenario was a time-consuming and error-prone process. To alleviate this, an automated testing tool was designed 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.

2.5.3 Papabed Testing—Papabed was extensively used during development in order to integrate RAX with the DS1 flight software. In the context of the formal testing process, *Papabed* was used only to run six off-nominal system test scenarios on the "frozen" version of the RAX delivered to flight software for the flight experiment. These off-nominal scenarios corresponded to the situations that were most likely or had the potential for highest impact on the outcome of the experiment. No bugs were detected in these scenarios, probably because RA responses to off-nominal situations were well tested on the *Babybed*.

2.5.4 Hotbench and DS1 Testbed Testing—The *Hotbench* and *DS1 Testbed* were 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 onboard 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. The resource-consumption cases were tested by causing RAX to execute a Lisp script that consumed those resources. The team 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.

2.5.5 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,; therefore, full re-validation became increasingly infeasible as flight approached. 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.

2.5.6 Summary of Testing Resources—About 269 functional tests for PS were conducted on *UNIX* (repeated for 6 software releases), more than 300-*Babybed* tests (repeated for 6 software releases), 10-*Papabed* tests (run once), 10-*Hotbench* tests (repeated for two releases), and 2 *DS1-Testbed* tests (on the final release) over a period of 6 months with four half-time engineers. This figure includes design, execution, analysis of the test cases, and development of testing tools.

2.6 Ground Tools

To provide adequate coverage and visibility into the RA's onboard workings, a ground tools suite was designed to interface with the real-time RA-generated telemetry.

The two major goals of the RA ground tools were:

- To present a summary of the spacecraft status understood easily by the mission operations team.
- To present enough information about the inner workings of the RA software for the experiment team to quickly recognize and debug problems.

To support these goals, telemetry specific to Remote Agent was downlinked during the test. The RA-specific telemetry included contained:

- Planning events (e.g., planning started, finished, and progress messages).
- Sequence execution events (e.g., plan `p` is starting execution or plan step `x` started executing at time `t1`).
- Mode-interpretation events (e.g., valve `v` changed state from nominal to possibly-stuck-closed).
- Messages between RA components and between RA and other DS1 flight software.

This telemetry was event-oriented and largely incremental (i.e., interpretation of one message depended upon receipt of earlier messages). This design created some problems, as will be detailed later. In addition, there were heartbeat ("I'm still alive") messages from RA components visible in the normal DS1 telemetry. The next sections describe specific ground tools constructed to examine this telemetry.

2.6.1 PacketView—Remote Agent telemetry is event-based. PacketView displays these events in a simple one line per message format that is easily understandable to all members of the team. As such, PacketView was the most fundamental of the RA ground tools. The PacketView interface (see Figure 10) displayed four types of telemetry packet for each of the three RA modules and IPCO_RAXO onboard messages sent between the three RA components. These were displayed as color-coded scrolling lines of text. Missing packets, detected by gaps in the sequential packet numbering, were displayed as a single line in a unique color. Search capabilities were available based on message type and content.

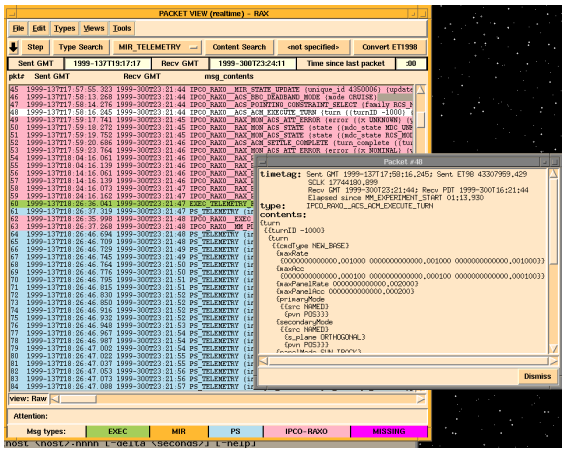


Figure 10. Packetview—Telemetry Packet Display

User-selectable dialogs presented "pretty printed" versions of the single-line packet entries. The "time bar" displayed the most recent "spacecraft sent" Greenwich Mean Time (GMT), the most recent "ground received" GMT time, and a running time since the last packet was received.

2.6.2 ExecView—ExecView visualized the execution status of plans onboard the spacecraft (see Figure 11). Different kinds of activity associated with different parts of the spacecraft appeared on separate timelines. For each kind of activity, specific planned events were represented as tokens appearing along the timeline. Tokens were color-coded to represent their execution status (i.e., in-the-future, currently-executing, completed, and completion-overdue).

As the plan was being executed by EXEC onboard the spacecraft, the start and finish times of the activities would be expected to change. Through the constraints, these changes would impact later activities. ExecView would propagate these changes downstream in the schedule, using the same propagation techniques used by the Planner.

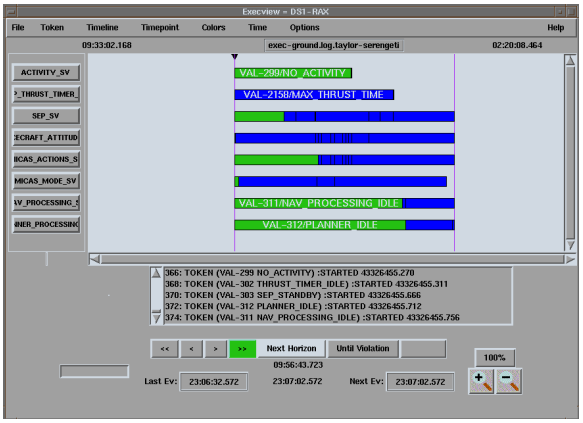


Figure 11. ExecView—Plan Execution Status

ExecView was designed initially as a debugging tool for validating EXEC development. As a result, it did not have support for handling missing telemetry packets during flight. As a result, it produced some erroneous conclusions during RAX concerning the state of plan execution. To make ExecView more useful, it will have to handle such missing data.

2.6.2 The Ground Planner—Of the three technology modules flown as part of RA, the spacecraft team was least comfortable with PS. To allow the DS1 team to gain confidence in the onboard planner, the RAX team used a ground twin of the planner. The ground planner was identical to the one onboard and was able to duplicate the onboard twin by tapping into the real-time telemetry available. It had access to other flight software resources via connection to the *Papabed*. This testbed accurately replicated the software onboard DS1 (although it did not replicate the hardware). Of particular importance to the planner were navigation-module and beacon-asteroid files describing targets for optical navigation and the portion of ACS that predicted the time required to change spacecraft orientation.

The ground planner was a useful tool in predicting the performance of the planner onboard and was especially useful as a confidence builder for mission staff unfamiliar with the working of an autonomous-planning agent.

2.6.3 PS Graph—PS Graph (see Figure 12) displayed the problem-solving trajectory taken by PS for each of the plans generated by the onboard planner. This took the form of an X-Y graph representing the search depth vs. number of search nodes visited for each successive step of the planner's search. The purpose of these plots was to provide a quick summary of the PS problem-solving process. For example, a trajectory that visits the same depth level several times while the search-node number increases indicates that the planner is backtracking. The persistence of this situation for a large number of steps is an indication that PS may be thrashing and that it will be unlikely to return a solution within the allotted amount of time. Another use of the PS-Graph plots is to compare telemetry-data trajectories generated during simulation runs of the ground planner twin.

Although very simple, the power of this tool's summarization and the insight level that it can provide during both RA development and operations in a stressful situation was surprising. As will be discussed in the flight experiment section below, PS Graph allowed the team to monitor an unexpected situation with PS and quickly identify the likely problem's cause. In the future, it is advisable to design several simple visualizations like PS Graph for the reduced ground team to support an autonomy mission.

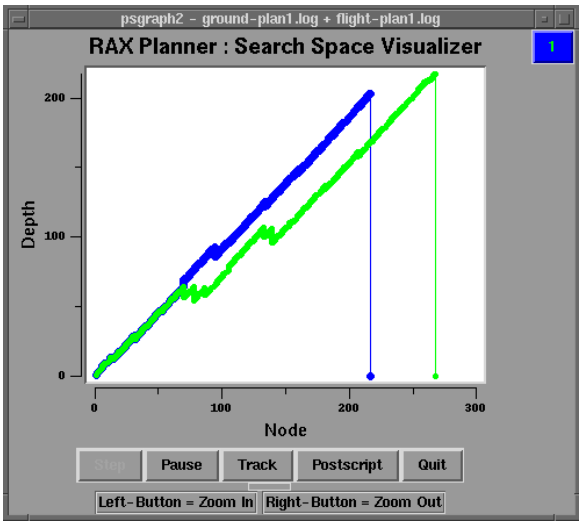


Figure 12. PS Graph—Planner Progress Display

2.6.4 Stanley and MIR—A version of MIR was also run on the ground. The purpose of this was to infer MIR's full internal representation of the spacecraft state from the telemetry that contained a much smaller subset. Specifically, it contained the set of independent variables in MIR's spacecraft model. The Stanley ground tool displayed a hierarchical schematic of the spacecraft's onboard components whose status was driven by the ground MIR (see Figure 13).

Components could be opened, to show more detail, or closed. The states displayed were blue ("ok - powered off"), green ("ok - powered on"), yellow ("recoverable failure"), purple ("degraded failure"), and red ("permanent failure"). Since Stanley assigned colors to all states, nominal and off-nominal, the user can tell at a glance the conditions of the devices. Stanley did not address the issue of displaying continuous values, such as a battery state-of-change.

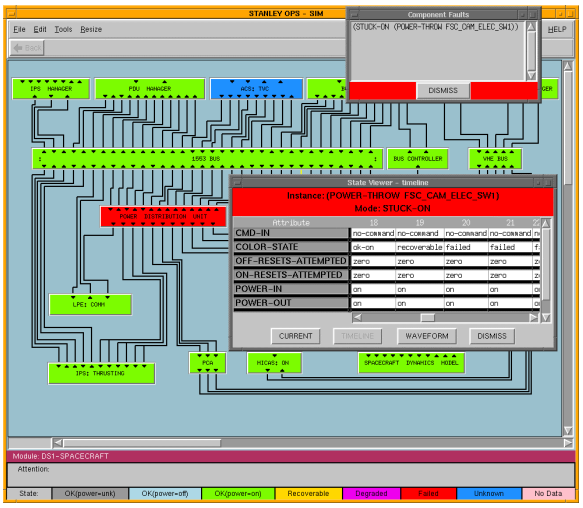


Figure 13. Stanley—Hardware Status Display

In addition to the color changes, detected component faults were reported by popping up an alert box. The alert box allowed the user to click on an entry, resulting in the schematic being opened down to the appropriate hierarchical level to show the local context of the fault. Histories of all state changes, important or not, were available at any time by clicking on components.

2.6.5 Predicted Events—In flying an autonomous agent, like RA, ground operators observing the spacecraft state via its telemetry may not be in a position to know precisely when certain events are to take place. It was nevertheless important to have a prediction of when RA planned to take various actions so that the appropriate subsystem stations at the mission operations center could be staffed for observability. Therefore, the team generated a Predicted Events File (PEF), which reported both the low-level commands RA would issue together with the high-level actions RA was asked to take.

2.6.6 Public Outreach via the Web—E-mailed summaries of events onboard presented in simple English and a Java applet timeline display on the Web, patterned after ExecView, were two additional tools to present RA's progress to the public. These tools are interesting because they required an even higher target for simplicity and understandability than did the flight controllers' tools.

Several recent missions have used pagers and e-mail to deliver notifications to the mission-operations team. The DS1 ground system, for instance, alerted operators by pager when a given measurement strayed outside a preset range or when fault-protection telemetry went into an unusual state. This was taken a step further in RAX by producing descriptions of important events in common English. The summarized descriptions were automatically posted to the RAX Web site (http://rax.arc.nasa.gov) and e-mailed in batches to a public mailing list. Two thousand subscribers received this e-mail during RAX. Terse descriptions were also sent to team members' alphanumeric pagers via e-mail.

An alternative Remote Agent-activity description (Figure 14) was also provided using horizontal timelines patterned after ExecView. This was implemented as a Java applet. The timelines in the top window represented major kinds of activity (e.g., attitude or camera-related activity). Along the timelines were tokens indicating particular activities (e.g., a turn), in effect displaying the plans generated onboard on a user's Web browser. Also included were controls to step through the timelines and an event-based summary similar to that provided in e-mail. The most interesting feature of this applet was its ability to show what RA planned to do at any time. The user could click on any event, and the applet would show what the RA planned to do at that time. This is interesting because the plan changed several times due to simulated faults. Thus the applet provided an historical overview of RA's re-planning activity and recreated conditions aboard the spacecraft for the general public.

Due to time pressure, the outreach tools were designed to handle the nominal scenario only (including the simulated faults). They did not accurately reflect the RAX software problems that occurred. They did, however, summarize activity during the new scenario without modification. These summaries are still available at the RAX web site: http://rax.arc.nasa.gov.

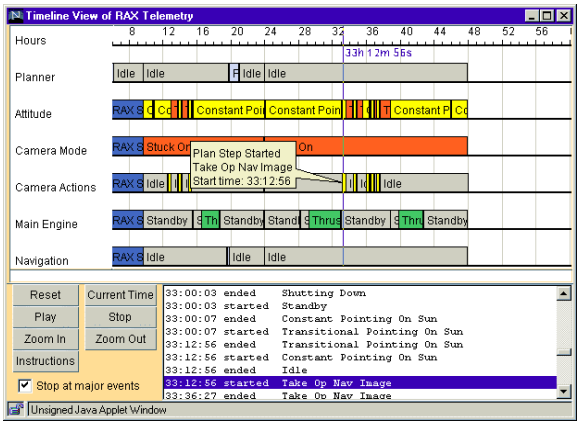


Figure 14. Timeline Applet

Additional details on the RAX Ground Tools can be found in [13].

2.7 Flight Test

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 17, 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.

2.7.1 Flight Test Part 1—The flight experiment started on Monday, May 17, 1999. At 11:04 am PDT, a telemetry packet was received confirming that the 2-day scenario had begun 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 among team members.

Figure 12 graphically shows the situation with the output of the PSGraph ground tool. The blue trajectory relates to a *Papabed* test that was run May 16, 1999 under identical condition to those of the flight test. The green trajectory describes what happened during flight. The deviation in the green trajectory from the 45° diagonal trajectory means that PS in flight backtracked significantly more than on *Papabed*. Since the conditions on the spacecraft were believed to be practically identical to those on the ground testbeds, there was no apparent reason for this discrepancy. Subsequently, the cause of this discrepancy was traced back to the spacecraft and *Papabed* differing on the contents of the AutoNAV file containing asteroid goals. Therefore, in flight PS was actually solving a slightly different problem than it had solved on the ground! Thus, this unexpected circumstance demonstrated 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.

2.7.2 Troubleshooting and Recovery—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, a 6-hour RAX scenario was generated to demonstrate the remaining 30% of the RAX validation objectives. This scenario included IPS thrusting, three failure scenarios, and back-to-back planning. This new scenario was designed, implemented, and tested, together with the patch, on *Papabed* overnight within about 10 hours. This rapid turnaround allowed the team to propose a new experiment at the DS1 project meeting 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 very low. Nonetheless, a contingency procedure was developed and tested that would enable the team to achieve most of our validation objectives even if the problem recurred.

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.

2.7.3 RAX Flight Part 2—The 6-hour scenario was activated Friday morning, May 21. The scenario ran well until it was time to start up IPS. Unfortunately, an unexpected problem occurring somewhere between FSW and RAXM caused a critical monitor value to be lost before it reached RA. The cause of this message loss has not been determined. The problem of lost-monitor values could have been avoided with periodic refreshes of the monitor values. This was deemed out of scope for the purposes of the experiment, and RA was known to be vulnerable to message loss. This vulnerability led RA's estimation of the IPS state to diverge from the true 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.

By executing the two flight scenarios, RAX achieved 100% of its validation objectives.

2.8 Effectiveness of the Development and Test Process

Progress in development and testing during the RAX project can be analyzed through the Problem Reports (PRs) filed between April 1997 and April 1999 (see Table 9).

A developer or a tester could file a PR, usually reporting a bug or requesting a change in the software behavior. A few PRs were reminders of activities or checks to be performed. PRs remained open until the developers addressed them. When a resolution to the report was filed (e.g., a bug fix was provided), the originator of the report would check the validity of the resolution. If accepted, the resolution was included in a formal release. A few PRs were suspended. This meant that the risk of the problem was assessed and considered acceptable within the limits of RAX.

Table 9. Number of PRs by Subsystem

Subsystem	Number of PRs
Planner/Scheduler	233
Executive	100
MIR	85
RAX Manager	22
System	77
Communication	22
Simulator	30
Others	11
Total	580

Figure 15 gives an idea of the temporal distribution of new PRs filed over the duration of the project. The last four columns (from January 1999 to April 1999) relate to problems that were submitted to the CCB process. Notice that the number of PRs in this period is still quite high (91). This depended in part on the fact that integration with flight software started in earnest in December 1999, with RAX running on *Papabed,* and that until then RA had only been operating interacting with low-fidelity simulators.

PRs can be divided into three categories:

- Modeling PRs required by domain-specific knowledge changes relative to the DS1 spacecraft subsystems.
- Engine PRs effecting RA's core reasoning engines.
- PRs related to other mechanisms such as the format of data file exchanged between RA components. This category also includes reminders and requests of change that were outside the scope of RAX.

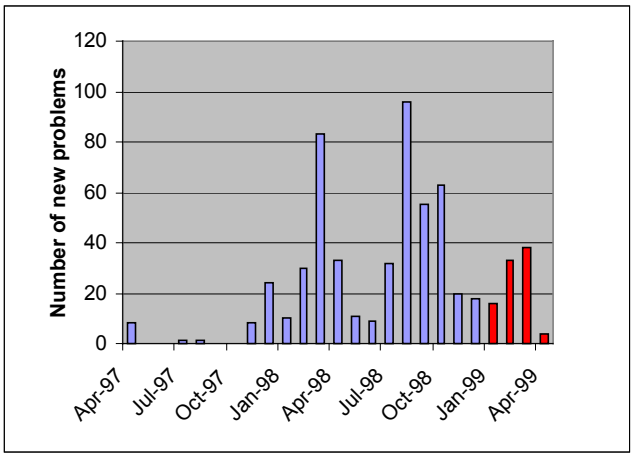


Figure 15. Temporal Distribution of Problem Reports

Figures 16, 17, and 18 describe the distribution of problems by category for each individual engine. The most stable RA subsystem was MIR. This stability manifested itself both in terms of the total number of Engine and Modeling PRs filed and in terms of the very few PRs of these categories filed in the last 4 months of the project. This was due both to the maturity of the MIR technology and to the fact that the problem addressed by MIR changed very little during the duration of the project.

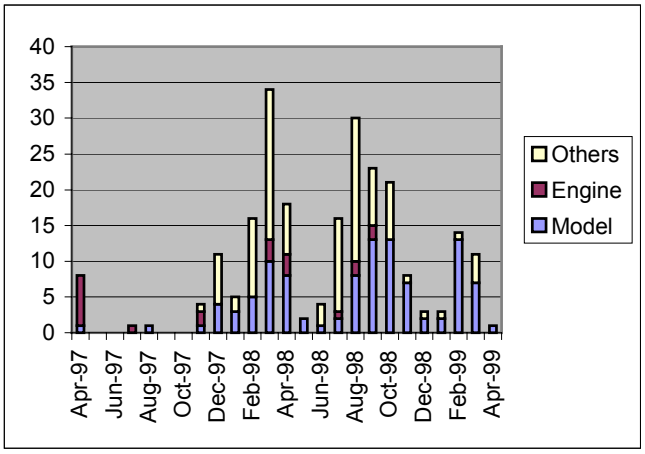


Figure 16. Planner PRs by Category

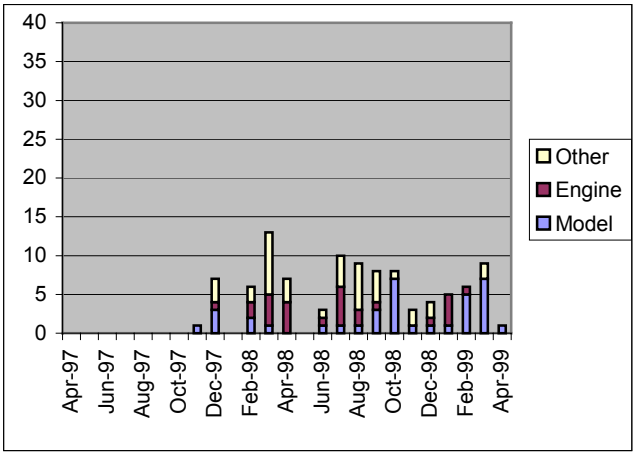


Figure 17. Executive PRs by Category

The command language used by EXEC, ESL, was developed prior to the RAX project and caused a negligible number of PRs. The majority of the EXEC PRs fell into the Other category and were related to integrating the PS and MIR modules. The next-largest category of PRs was model related. These tended to manifest themselves each time RA was integrated on a higher-fidelity testbed. Models for EXEC were undergoing modifications quite late (February 1999 to April 1999). This was primarily due to the fact that these months covered a period of intense activity on *Papabed* with the interfaces with the details of how flight software operated being finally communicated to the RAX team. This resulted in some localized changes in interface functions and in task-decomposition procedures. The effects

of these changes were typically localized at the EXEC level and did not propagate up to PS models. This confirms the possibility of developing RA even on the basis of an accurate but abstract characterization of the modeled system, with much of the high-level behaviors remaining stable when further details on the behavior of the system are known.

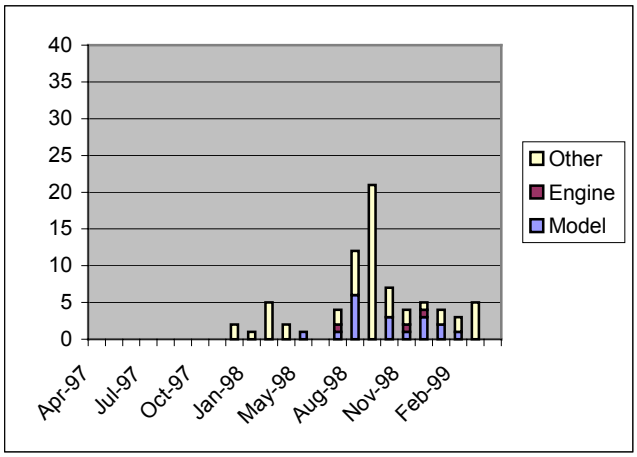


Figure 18. MIR PRs by Category

Both in the case of MIR and EXEC, testing was very effective at validating models. EXEC and MIR models have many non-interacting or loosely interacting components that can be tested independently. This reduces the number of test cases that are needed. Testing small model components independently—like the team did in RAX—should scale-up for larger, future science-mission models.

In the case of PS, a larger overall percentage of PRs (about 45%) were model related. More importantly, a large number of new problems were discovered during the last four months of the project, after the formal testing process had ended. The vast majority of these problems consisted of PS operating correctly but 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:

- 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 formal PS testing.
- Planning problems became more challenging in the transition from the 6-day 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.
- 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, the automatically-generated test cases used for PS validation only covered pair-wise interactions. Therefore, they could not 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, only the most crucial variation was focused on: the time at which re-plans might occur. The objective was to ensure that the planner could handle any re-planning 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 PS-model knowledge, boundary times at which the topology of the plans would change were manually identified. Twenty-five such boundary times were identified 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. This number of tests is more than four orders of magnitude smaller than the total of 172,800 possible re-plan times. 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.

The above test-suite-design methodology was used only toward the end of RAX, after the PS model and code had been frozen. However, 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.

Note that the number of PRs regarding the reasoning engines of PS, EXEC, and MIR was relatively small. For example, less than 10% of PS's PRs were Engine related and the last was filed in September 1998. However, the bug EXEC encountered during RAX shows that the engine validation methodology could have improved. In fact, the testing was primarily focused on validating the knowledge in the domain models. Tests were selected to exercise the domain models. By exercising RA on these test scenarios, the domain models and engines were effectively tested as a unit. However, especially for concurrent systems such as EXEC, a much better approach is to thoroughly, formally validate the logic of the engines through the use of formal methods [16]. Although expensive, this form of testing can give a high level of quality assurance on the core of the RA technology. Moreover, since the engines remain unchanged over a large number of applications, the cost of this testing can be amortized across several missions.

2.9 Costing

Figure 19 gives an overall view of the costing of RAX starting from October 1997, when tracking information was available. The figure describes costs based on development, testing, integration, and technical management activities. The Full Time Equivalence (FTE) exerted is shown on the Y-axis. Costing by FTEs is more appropriate in this case because of the differing accounting standards used at NASA's ARC and JPL.

The chart clearly shows the distinct development, testing, and integration efforts being partitioned in time; development efforts were clearly focused before the move to the high-fidelity testbeds. While testing and integrations efforts were ongoing activities, they came to dominate the latter part of the move to the testbeds. While the overall trend is a curve with diminishing figures, there are some features that need some explanation.

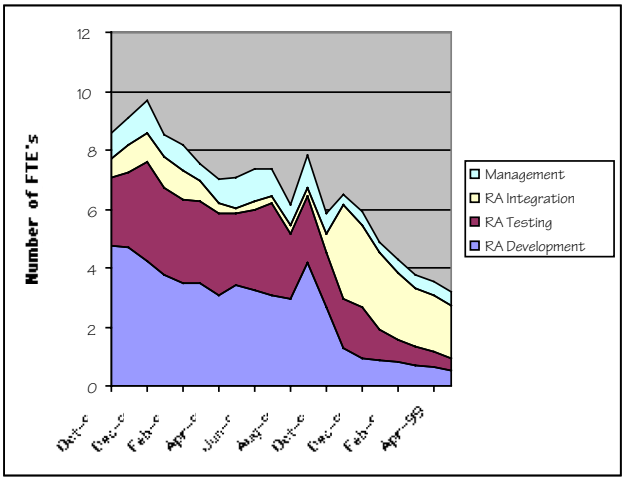


Figure 19. RAX Costing

The first peak in the October–December 1997 timeframe corresponds to the time when formal test plans were put together and *UNIX* testing began. In addition, RAXM was delivered to the flight team at this time. The peak, therefore, is categorized by these efforts and the resulting testing and bug fixing that took place.

The second peak in the August–October 1998 timeframe corresponds to a number of events. Primarily, this was dealing with new code deliveries to the planner engine to allow EXEC to deal more robustly with additional information in the plans. This increased effort highlights the extra individuals from outside the RA team who made these efforts possible. In addition, all team members were gearing up towards testing on the *Papabed*, the highest-fidelity testbed available at that time. Subsequent to that event, the curves show a deep decline, as expected, in the development efforts when the team focused more on integration and testing on the various testbeds available. Efforts dealing with integration, therefore, show a perceptible increase.

Lastly, the gap between the testing and integration efforts appears to be inverted in the December 1998 and May 1999 timeframe. The primary reason for this was our late arrival on the high-fidelity testbeds. This meant testbed integration had to be finished in half the normal time. It was also the case that working on these testbeds took time and effort beyond what that needed on the lower-fidelity testbeds (*Unix* and *Babybeds*) that were available early on. Configuration training and problem-detection also took substantial time and effort, causing a larger manpower effort for integration as shown.

The actual costs of the entire RA development effort was $500K for the NewMAAP demonstration (May to November 1995), $4.5 million during the DS1 autonomy FSW phase (December 1995 to March 1997), and $3 million for RAX (April 1997 to June 1999), for a total cost of $8 million.

2.10 Lessons Learned

The RA team learned valuable lessons in a number of areas including RA technology and processes, tools, and even autonomy benefits to missions.

2.10.1 Robustness of the Basic System—Model validation alone does not suffice; the rest of the system, including the underlying inference engines, the interfaces between the engines, and the ground tools, must all be robust. Given the resource constraints, the decision was made to focus our formal testing on model validation, with engine and interface testing happening as a side effect. This was a reasonable strategy: code that has been unchanged for years is likely to be very robust if it has been used with a variety of different models and scenarios. However, newer code does not come with the same quality assurance. Also, as the

deadlock bug in-flight showed, subtle-timing bugs can lay hidden for years before manifesting themselves.

Conclusion: The primary lesson is that the basic system must be thoroughly validated with a comprehensive test plan as well as formal methods, where appropriate, prior to model development and flight insertion. Interfaces between systems must be clean and well specified, with automatic code generation used to generate actual interface code, telemetry, model interfaces, and test cases; code generation proved enormously helpful in those cases where it was used.

2.10.2 Robustness of Model-Based Design—As mission development times become shorter and mission objectives more ambitious, it is less and less likely that an accurate model of each spacecraft component will be available early in the flight- and ground-software development cycle. Dealing with this uncertainty is a major problem facing future missions. By emphasizing qualitative and high-level models of behavior RA can help solve this dilemma. Qualitative, high-level models can be captured early in the mission lifetime and should need only minor adjustments when the hardware is better understood. Our experience on RAX essentially confirms this hypothesis. Initial spacecraft models used by PS, EXEC, and MIR were built early in the DS1 mission, before April 1997. During the following year and a half, EXEC and MIR models did not change and the PS model was only changed in order to support more efficient problem solving by the search engine, not in order to reflect new knowledge of the spacecraft behavior. In the last phase of the experiment preparation, when communications between the RAX team and the DS1 team resumed, adjustments were needed to finalize the interface between the low-level EXEC primitives and flight software.

Conclusion: Contrary to much concern, the type of qualitative, high-level models used by RA requires little tuning throughout the project. The usefulness of the models for software development has been validated.

2.10.3 Model Design, Development and Test—One of the biggest challenges was model validation. This was particularly true during validation testing, when even small changes in the models had to be carefully and laboriously analyzed and tested to ensure that there were no unexpected problems. In fact, in some cases it was decided to forego a model change and, instead, decided to institute flight rules that would preclude the situation that required the model change from arising. A related issue was that methods do not yet exist to characterize RA's expected behavior in novel situations. This made it difficult to precisely specify the boundaries within which RAX was guaranteed to act correctly. While the declarative nature of RA models was certainly very helpful in ensuring the correctness of models and model changes, the difficulty stemmed from unexpected interactions between different parts of the model (e.g., different parts of the model may have been built under different, implicit, conflicting assumptions).

Conclusion: The central lesson learned here is the need for better model-validation tools. For example, the automated test running capability that was developed proved to be enormously helpful, as it allowed the team to quickly evaluate a large number of off-nominal scenarios. However, scenario generation and evaluation of test results were time consuming. In some cases, the laborious process followed to validate model changes has provided the team with concrete ideas for developing tools that would dramatically simplify certain aspects of model validation. Preliminary work in the area of formal methods for model validation is also very promising. Finally, there is a need to develop better methods for characterizing RA's behavior with a specific set of models, both as a way of validating those models and as a way of explaining the models to a flight team.

2.10.4 Onboard Planning—Since the beginning of RA, onboard planning has been the autonomy technology that most challenges the comfort level of mission operators. Commanding a spacecraft with high-level goals and letting it autonomously take detailed actions is very far from the traditional commanding approach with fixed-time sequences of low-level commands. During RAX, the flawless demonstration of onboard planning has provided powerful feasibility-of-approach proof. Discomfort with the discrepancy between tested behavior and in-flight PS behavior during RAX was a surprising mirror of the autonomy critics' objections.

Conclusion: It is difficult to move past the mindset of expecting complete predictability from the behavior of an autonomous system. However, RAX has demonstrated that the paradigm shift is indeed possible. In the case of PS behavior during RAX, the point is not that the combination of pictures requested by NAV had never been experienced before, but that the problem-solving behavior that the planner used to achieve each individual picture goal had indeed been tested. Confidence in complex autonomous behaviors can be built up from confidence in each individual component behavior.

2.10.5 Design for Testability—System-level testing is an essential step in flight preparation. Designing RA to simplify and streamline system-level testing and analysis can enable more extensive testing, thus improving robustness. In RAX, system-level testing proved to be cumbersome. The primary reason for this was the absence of efficient tools to generate new mission scenarios; therefore, all system tests had to be variations on the nominal scenarios. Hence, to test a particular variation, one was forced to run a nominal scenario up to the point of the

variation: e.g., testing thruster failures during turns required at least 6 hours, since the first turn occurred about 6 hours into the scenario.

Conclusion: The difficulty of generating new mission scenarios is easily addressed: a graphical tool allowing visual inspection and modification of mission profiles, as well as constraint checking to ensure consistency, can dramatically simplify the construction of new mission profiles. Such a tool is now being constructed. Nonetheless, overall RA validation is still necessary to ensure that RA will properly handle each new mission profile (see below).

2.10.6 Systems Engineering Tools—Coding the domain models required substantial knowledge acquisition, which is a common bottleneck in Artificial Intelligence systems. It is better to have the domain expert code the models directly.

Conclusion: Develop tools and simplify the modeling languages to enable spacecraft experts to encode models themselves. Employ tools and languages already familiar to the experts. Organize the models around the domain (attitude control, power, etc.) rather than around the RA technology (PS, EXEC, MIR).

2.10.7 Mission Profile Development—RA is commanded by goals specified in a mission profile. For the experiment, constructing the profile was a "black art" that only one or two people on the RA team could perform. The mission planners and operations personnel must be able to specify goals themselves.

Conclusion: Simplify the specification of goals. When possible, use approaches already familiar to mission planner, such as graphical-timeline displays and time-ordered listings. Provide automated consistency checking.

2.10.8 Adaptability to Late-Model Changes—The spacecraft requirements and operating procedures changed throughout development and after launch. It was not possible to encode late changes, due to the regression-testing overhead that each change required.

Conclusion: The validation cost of model changes must be reduced. Some possibilities include tools to evaluate the consequences of model changes on testing. The models already support localized changes. Procedures are needed to uplink and install just those changes.

2.10.9 Ground Tools—Ground tools ought to be developed well in advance of the actual flight and be used as a primary means to test and understand how to operate complex systems. Given the late date of development of most of the ground tools, a good many of them felt not well integrated. As a result, only the tools displaying or interpreting data in the most obvious way were of high value.

2.10.10 Telemetry—In addition to an onboard textual log file downlinked at the end of the experiment or on request, RAX sent a stream of binary-telemetry packets, one for each significant event, that were displayed as color-coded text on the ground. Among other things, the telemetry let the team monitor all onboard communication among RAX modules and between RAX and FSW. This proved valuable in letting the team quickly diagnose the anomalies that occurred. It was immediately evident the reason RAX failed to turn off the ion engine was it had stopped executing the plan in some unanticipated manner; the team knew RAX was still running and could also rule out a plan abort or a failure to send just one command. Similarly, the upshot of the second anomaly was immediately narrowed down to a monitor message, which was either not sent or not received.

Conclusion: Ensuring sufficient visibility on all platforms, including in-flight, requires adequate information in telemetry. The best way to ensure this is to design the telemetry early and to use it as the primary, if not the only, way of debugging and understanding the behavior of the system during integration, test, and operations.

2.10.11 Team Structure for RA-Model Development—The RAX team was structured horizontally along engine boundaries. This meant that team members specialized in one of the PS, EXEC, and MIR engines and that each team was responsible for modeling all spacecraft subsystems for their engine. This horizontal organization was appropriate for RAX, since it was our first major experience in modeling spacecraft subsystems for flight. Hence, it made sense for engine experts to do all modeling for their engine. However, this organization has several shortcomings. Perhaps the most significant shortcoming was that knowledge of any one spacecraft subsystem (e.g., attitude control, ion propulsion, MICAS camera) was distributed across the three teams; one needed discussions with three individuals to get a complete understanding of how a subsystem was commanded by RA.

Conclusion: These shortcomings suggest an alternate structuring for a future SW team. Instead of a horizontal structure, teams might be organized vertically along spacecraft subsystem or domain-unit boundaries (e.g., a single team would be responsible for developing all models for ACS). This would ensure internal coherence of the resulting model. Furthermore, since modelers would need to understand how to use all three engines, they can make effective decisions on how best to model a subsystem to exploit the strengths of each engine and avoid information duplication.

While a vertical-team organization has its benefits, certain aspects of model development intrinsically involve managing and reasoning about global constraints: e.g., power allocation strategies, system-level fault protection. Hence, it is important to involve systems engineers to develop these global strategies.

2.11 Answers to a Project Manager's Questions

In August 1997, after a meeting between the RA team and DS1's project management, David Lehman, DS1 project manager, was asked how the RA team could convince a future science mission's project manager to use RA.

Lehman responded with a series of questions that a project manager would ask if he or she was just starting a new science-mission development.

Answering these questions after RAX would be a good way to summarize our current understanding of the technology. Also, the reader should keep in mind that these answers apply as well to other software frameworks comparable to RA in functionality and approach.

What does RA do to make my life easier?

It makes life easier because:

- It is possible to operate with a high level of autonomy during more phases of a mission outside of critical sequences.
- It provides a framework that facilitates the translation of system engineering requirements into operational code during the development phase of a mission.
- The RAX experience shows that RA can indeed operate autonomously and respond robustly to likely anomalies without intervention from a ground team and the associated delay due to round trip light time, diagnosing the problem, creating a command sequence, and validating it. This can translate into lower-operational costs and improved science.
- RA can reduce the need for communication with ground. This means less time on the highly subscribed DSN and further cost reductions.

Is RA a new technology?

RA is a novel integration of three technologies; their application to spacecraft is also new. Each of the component technologies in RA is an AI technology with a long history. Theoretical papers exist that demonstrate strong formal properties of some RA components [8][10]. Significant applications exist for each of the technology components. The most significant risk that was addressed by the overall RA development was the integration of the three technologies into a highly autonomous agent. The team believes that RAX demonstrates that successful integration.

Why is RA the best thing to do in order to make the spacecraft have autonomous operations?

Other systems exist that are comparable with some subset of the capabilities provided by RA; however, the other systems typically do not integrate all aspects of RA. For example, the team is not aware of any operational software for autonomous agents that contains an onboard planning and scheduling system.

One of the problems with FSW is that the FSW team is at the end of the "requirements food chain." Late requirements to FSW in turn results in increased costs and wasted efforts in the beginning of the project. With RA, more stuff must be put into the code, like the models of the hardware and how users want the spacecraft to operate. Therefore, this requirement should further exacerbate the standard FSW problem of the past. How can that be fixed?

Indeed, coding RA models requires a substantial up-front system engineering effort. The advantage of the declarative approach is that the impact of late model changes is lower compared to conventional flight software. Because of their abstract nature, the vast majority of RA models remained completely valid and operational throughout the project.

FSW is hard to test. FSW + RA should be even harder to test? How is that fixed?

The RAX experience confirms that testing FSW is hard. The bug that was found during flight shows that more attention and effort needs to be spent validating the basic engines. The validation cost is well worth the effort because the engines are components that can be re-used over many missions.

With respect to the capabilities provided by the domain models, our experience shows that testing of RA can be successfully layered. Testing RA can be separated from testing FSW. Also, internal to RA, different capability levels of abstraction can be separated, taking advantage of the each layer's different requirements. For example, low-level, real-time capabilities require testing on the slower real-time testbeds, while the higher-level functionalities can undergo extensive testing on readily-available, cheaper workstations.

With respect to coverage of possible RA behaviors, the experience is that the larger the space of possible combination of parameters and the higher the number of possible interactions between subsystems, the harder it is to guarantee that RA will work nominally under all circumstances. This is not surprising. Restricting harmful interaction by design is the standard problem that needs to be addressed by system and fault-protection engineers in a mission. RA does not make the problem go away. However,

during development RA can be used in simulation to provide a useful tool to explore system behavior under stress situations. This could help detect and fix potential problems with constraint interactions that are difficult to identify otherwise.

Mission operation is a big deal. However, most projects do not think about it until late in the development. Does RA offer any benefits here?

RA operates a spacecraft by generating plans that meet goals and flight rules. The development of goals and flight rules is intrinsic to RA development and forces issues to be worked hand in hand with flight software. The result is a tighter integration between mission operations and flight software, which is a good thing.

What parts of the operations phase is RA best suited for? Normal cruise phase when nothing is happening, flying around something when DSN is not in sight? Is RA needed during the whole mission, or only during some critical mission phases, like an orbit insertion?

In principle, RA can support all phases of a mission. This does not mean that all of its component technologies are suited for all phases. For example, the performance of the current implementation of PS[3] makes it unsuitable for closed-loop use with tight response times (seconds to a few minutes). However, PS could be very valuable for scheduling competing observation of different levels of for a long-term, observatory mission. In these situations, the conditions are more similar than those demonstrated in RAX, where the next plan can be generated while the current one is executing.

RAX demonstrated that RA is viable during mission-cruise phase. Although this was done on a reduced model of the spacecraft, the team believes that the scaling up factors in this case should be linear and within current RA-technology's reach. With respect to the potential use during a critical phase, EXEC's event-driven, conditional execution and MIR's model-based fault protection are best suited for onboard use. Also, even within its current performance characteristics, PS could be useful during the design phase of the scripts to be executed by RA. RAX gives some evidence that this is possible. However, the ultimate demonstration of these capabilities will require more work.

3.0 Future Applications

Future work regarding Remote Agent can be divided into three categories: fundamental improvements in the capabilities of its components, improvements in usability or deployability, and upcoming demonstrations or applications. Since the experiment, a significant effort has gone into basic research to improve future iterations of Remote Agent. For example, a more capable version of Livingstone has been developed that better handles ambiguity when tracking the state of the spacecraft. Livingstone now tracks a number of most-likely states the spacecraft could be in, given the observations it has received thus far. If new observations invalidate the possible states MIR considered most likely, it re-analyzes the commands that have been given and the possible failures in order to determine which previously unlikely states now explain the unexpected observations.

PS has a number of efforts underway to improve the underlying software implementation: it now has a new modular-software architecture that allows plugging various search techniques into the engine. Work is underway in model analysis that will allow early detection of domain-model inconsistencies. Analysis of static models is also being undertaken to automatically generate search-control instrumentation. The latter approaches will allow rapid prototype development of planner models by non-technologists using incremental model development via "what-if" analysis to vastly reduce development costs. It will also provide mission staff with a better understanding of how autonomy architectures will fit into the overall design of FSW.

Other efforts are also in place to redesign the system architecture to allow EXEC access to the planner temporal database and algorithms. A unified-modeling language is being developed with cleaner semantics to allow EXEC to respond to exogenous events more rapidly.

The architectural themes pioneered in Remote Agent are gaining more general acceptance in the flight-software and mission-operations communities [15]. Applying RA to the DS1 spacecraft provided a wealth of practical lessons about what was needed to create a sustainable autonomy-engineering process and make this technology usable for main-line mission development and operations. PS and MIR have been re-architected, modularized, and implemented in C++ rather than Lisp. These next-generation versions are in alpha testing at the date of this report. EXEC is expected to be re-architected and implemented in C by the end of calendar year 2000. RA team is now developing tools for graphically creating and debugging models, for automating much of the integration of RA with traditional flight software, and for allowing humans and autonomous software to interact more easily. The team is collaborating with software-verification researchers at NASA's ARC and at Carnegie-Mellon University to allow certain Remote Agent models to be analyzed to prove they cannot recommend undesired behavior. In short, these research and development efforts are designed to make RA and similar

[3] The RAX plans consisted of 15–25 executable activities, 50–80 tokens and 90–134 constraints. They took 50–90 minutes to generate using about 25% of the RAD6000 CPU.

technologies more capable, easier to use, and easier to test and validate.

RA technology is successfully being transferred beyond the original team, and several groups are currently building prototypes with RA to evaluate it. At NASA's Kennedy Space Center, Remote Agent applications are being developed to evaluate RA for missions involving in-situ-propellant production on the Mars 2003 lander or a future piloted mission. Applications for shuttle operations are being pursued as well.

At the Jet Propulsion Laboratory, RA is being evaluated as the baseline-autonomy architecture for the Origins Program interferometry instruments and is being used in the JPL-interferometry testbed. One early customer of this development may be New Millennium Program's Deep Space Three, a space-based interferometry mission that includes two or three spacecraft cooperating to make science observations. At Johnson Space Center, components of Remote Agent are being integrated into an ecological life-support testbed for human missions beyond Earth orbit. At Ames, Remote Agent technology is being incorporated into software for more robustly controlling planetary rovers. Along with Orbital Sciences Corporation, Ames is working to demonstrate Remote Agent as it applies to streamlining the checkout and operation of a reusable launch vehicle. This demonstration will fly on the X-34 vehicle. In collaboration with Boeing, a similar experiment will be flown on the X-37 vehicle.

4.0 Acknowledgments

This report describes work performed at NASA's Ames Research Center and at the Jet Propulsion Laboratory, California Institute of Technology, under contract to the National Aeronautics and Space Administration.

The Remote Agent Experiment would not have been possible without the efforts of the DS1 flight team and Harlequin Inc. In addition, the direct contribution of the following individuals is gratefully acknowledged:

Steve Chien, Micah Clark, Scott Davis, Julia Dunphy, Chuck Fry, Erann Gat, Ari Jonsson, Ron Keesing, Guy Man, Sunil Mohan, Paul Morris, Barney Pell, Chris Plaunt, Greg Rabideau, Scott Sawyer, Reid Simmons, Mike Wagner, Brian Williams, Greg Whelan, and David Yan.

In addition, the following individuals were instrumental in providing valuable support and encouragement throughout the duration of the development and flight of the Remote Agent:

Abdullah Aljabri, Martha Del Alto, Ralph Basilio, Magdy Bareh, Kane Casani, Rich Doyle, Dan Dvorak, Dan Eldred, Julio Fernandez, Peter Gluck, Jack Hansen, Ricardo Hassan, Lorraine Fesq, Ken Ford, Sandy Krasner, David Lehman, Frank Leang, Mike Lowry, Nicole Masjedizadeh, Maurine Miller, Alex Moncada, Mel Montemerlo, Peter Norvig, Keyur Patel, Bob Rasmussen, Marc Rayman, Mark Shirley, Martin Simmons, Helen Stewart, Gregg Swietek, Hans Thomas, Phil Varghese, and Udo Wehmeier.

The team also gratefully acknowledges Caelum Research Corp. and Recom Technologies' participation and support.

5.0 List of References

[1] E. Gat, "ESL: A language for supporting robust plan execution in embedded autonomous agents," in *Proc. 1997 IEEE Aerospace Conference*, 1997, pp. 319–324.

[2] E. Gat and B. Pell, "Abstract resource management in an unconstrained plan execution system," in *Proc. 1998 IEEE Aerospace Conference,* 1998 [CD-ROM].

[3] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith, "Robust periodic planning and execution for autonomous spacecraft," in *Proc. IJCAI-97*, 1997, pp. 1234–1239.

[4] B. C. Williams and P. P. Nayak, "A model-based approach to reactive self-configuring systems," in *Proc. AAAI-96*, 1996, pp. 971–978.

[5] N. Muscettola, "HSTS: Integrating planning and scheduling," in *Intelligent Scheduling,* M. Fox and M. Zweben, Eds. San Francisco: Morgan Kaufman, 1994, pp. 169–212.

[6] N. Muscettola, B. Smith, S. Chien, C. Fry, G. Rabideau, K. Rajan, and D. Yan, "Onboard planning for autonomous spacecraft," in *Proc. Fourth International Symposium on Artificial Intelligence, Robotics and Automation for Space (iSAIRAS-97)*, 1997, pp. 229–234.

[7] Muscettola N., P. P. Nayak, B. Pell, and B. C. Williams, "Remote Agent: To boldly go where no AI system has gone before," *Artificial Intelligence,* vol. 103, nos. 1–2, pp. 5–48, Aug. 1998.

[8] J. de Kleer and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, no. 1, pp. 97–130, 1987.

[9] D. S. Weld and J. de Kleer*, Readings in Qualitative Reasoning about Physical Systems*. San Francisco, CA: Morgan Kaufmann, 1990.

[10] A. Jonsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith, "Planning in interplanetary space: Theory and practice," *Proc. Fifth International AI Planning Systems (AIPS 2000*), 2000, pp. 177–186.

[11] B. Smith, K. Rajan, and N. Muscettola, "Knowledge acquisition for the onboard planner of an autonomous spacecraft," in *Knowledge Acquisition, Modeling and Management*, E. Plaza and R. Benjamins, Eds. New York, NY: Springer, 1997, pp. 253–268.

[12] N. Masjedizadeh, "Remote Agent: 1999 Co-Winner of NASA's Software of the Year Award" [Online document], 1999 May 16, [cited 2000 Jul. 6], Available HTTP: http://rax.arc.nasa.gov

[13] K. Rajan, M. Shirley, W. Taylor and B. Kanefsky, "Ground tools for autonomy in the 21st century, to appear in *Proc. IEEE Aerospace Conference,* 2000.

[14] D. Smith, J. Frank, and A. Jonsson, "Bridging the gap between planning and scheduling," to appear in *Knowledge Engineering Review*, 2000.

[15] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, "Software architecture themes in JPL's Mission Data System," in *Spaceflight Mechanics 1999: Proc. AIAA-99,* 1999.

[16] K. Havelund, M. Lowry, and J. Penix, "Formal analysis of a spacecraft controller using SPIN,"presented at the Fourth International SPIN Workshop, Paris, France, Nov. 1998; also NASA's Ames Technical Report, Nov. 1997.

[17] B. Pell, D. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. P. Nayak, M. D. Wagner, and B. C. Williams, "An autonomous spacecraft agent prototype," *Autonomous Robots,* vol. 5, no. 1, Mar., pp. 29–52, 1998.

Appendix A. List of Telemetry Channels and Names

The bulk of RAX monitoring and validation during the experiment was from the RAX telemetry on APID 9 & 10, channels W-500 to W-570, and the downlinked log files.

Channel	Mnemonic
W-500 through W-570	(RAX channels)
P-0300	LPE_PASM_mgr
APID 9 and APID 10	Monitored RAX behavior. Packets were in a RAX-specific format.
APID 45	Log files downlinked after the experiment (plan files and detailed execution trace).

The following channels were also activated for RAX:

Channel	Mnemonic
F-1048	FaultEnaStat
F-1052	BusSCstatus
F-1055	IPS_SCstatus
F-1057	PDS_SCstatus
F-1058	ACS_SCstatus
F-1060	RAX_SCstatus
F-1063	BusGDstatus
F-1066	IPS_GDstatus
F-1068	PDU_GDstatus
F-1069	ACS_GDstatus
F-1071	RAX_GDstatus
D-0149	buf_pkt_09
D-0150	sent_pkt_09
D-0165	buf_pkt_10
D-0166	sent_pkt_10
F-0716 through F-0727	

Appendix B. Date of Turn-on/off and Frequency of Data Capture

The Remote Agent Experiment first ran from May 17, 1999, 5 am PST to Wed May 19, 1999, 7 pm PST. It ran again from May 21, 1999, 7:15 am PST to 1:30 pm PST (RAX_STOP). The log files were downlinked by May 21, 1999 4:00 p.m. PST.

Appendix C: List of Acronyms and Abbreviations

AutoNAV	Autonomous Navigation subsystem of FSW
ACS	Attitude Control Subsystem of FSW
APE	Attitude Planning Expert subsystem of FSW
ARC	Ames Research Center
CCB	Change Control Board
CPU	Central Processing Unit (computer)
DDL	PS Domain Description Language
DS1	Deep Space 1 spacecraft
DSN	Deep Space Network
ESL	Executive Support Language
EXEC	Remote Agent Smart Executive
FTE	Full Time Equivalent
FSW	DS1 Flight Software
GMT	Greenwich Mean Time
HGA	High Gain Antenna
HSTS	Heuristic Scheduling Testbed System
IPS	Ion Propulsion System
JPL	Jet Propulsion Laboratory
MICAS	Miniature Integrated Camera And Spectrometer
MI	Mode Identification component of MIR
MIR	Remote Agent Mode Identification and Recovery module (Livingstone)
MR	Mode Recovery component of MIR
MM	Remote Agent Mission Manager module
NASA	National Aeronautics and Space Administration
NewMAAP	New Millennium Autonomy Architecture rapid Prototype
OD	Orbit Determination
OPNAV	Optical Navigation Module subsystem FSW
PASM	Power Actuation and Switching Module
PEF	Predicted Events File
PR	Problem Report
PS	Remote Agent Planner/Scheduler
RA	Remote Agent
RAX	Remote Agent Experiment
RAXM	RAX Manager
RCS	Reaction Control System
RT	Remote Terminal
TDB	HSTS Temporal Database
TVC	Thrust Vector Control