Design of the Remote Agent Experiment for Spacecraft Autonomy

Douglas E. Bernard[1], Gregory A. Dorais[3], Chuck Fry[3], Edward B. Gamble Jr. [1], Bob Kanefsky[3], James Kurien[3], William Millar[3], Nicola Muscettola[2], P. Pandurang Nayak[4], Barney Pell[4], Kanna Rajan[3], Nicolas Rouquette[1], Benjamin Smith[1], Brian C. Williams[5]

[1] Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
818-354-2597
douglas.e.bernard@jpl.nasa.gov,

[2, 3, 4, 5] NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035
[2] Recom Technologies @ Ames
[3] Caelum Research @ Ames
[4] RIACS @ Ames
650-604-4756
nayak@ptolemy.arc.nasa.gov

ABSTRACT—This paper describes the Remote Agent flight experiment for spacecraft commanding and control. In the Remote Agent approach, the operational rules and constraints are encoded in the flight software. The software may be considered to be an autonomous "remote agent" of the spacecraft operators in the sense that the operators rely on the agent to achieve particular goals.

The experiment will be executed during the flight of NASA's Deep Space One technology validation mission. During the experiment, the spacecraft will not be given the usual detailed sequence of commands to execute. Instead, the spacecraft will be given a list of goals to achieve during the experiment. In flight, the Remote Agent flight software will generate a plan to accomplish the goals and then execute the plan in a robust manner while keeping track of how well the plan is being accomplished. During plan execution, the Remote Agent stays on the lookout for any hardware faults that might require recovery actions or replanning.

In addition to describing the design of the remote agent, this paper discusses technology-insertion challenges and the approach used in the Remote Agent approach to address these challenges.

The experiment integrates several spacecraft autonomy technologies developed at NASA Ames and the Jet Propulsion Laboratory: on-board planning, a robust multi-threaded executive, and model-based failure diagnosis and recovery.

1. Introduction

Robotic spacecraft are making it possible to explore the other planets and understand the dynamics, composition, and history of the bodies that make up our solar system. These spacecraft enable us to extend our presence into space at a fraction of the cost and risk associated with human exploration. They also pave the way for human exploration. Where human exploration is desired, robotic precursors can help identify and map candidate landing sites, find resources, and demonstrate experimental technologies.

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

This paper describes a flight experiment which will demonstrate the Remote Agent approach to spacecraft commanding and control. In the Remote Agent approach, the operational rules and constraints are encoded in the flight software and the software may be considered to be an autonomous "remote agent" of the spacecraft operators in the sense that the operators rely on the agent to achieve particular goals. The operators do not know the exact conditions on the spacecraft, so they do not tell the agent exactly what to do at each instant of time. They do, however, tell the agent exactly which goals to achieve in a period of time as well as how and when to report in.

The Remote Agent (RA) is formed by the integration of three separate technologies: an on-board planner-scheduler, a robust multi-threaded executive, and a model-based fault diagnosis and recovery system.

This Remote Agent approach is being designed into the New Millennium Program's Deep Space One (DS1) mission as an experiment. The spacecraft (see Figure 1) will fly by an asteroid, Mars, and a comet.

The New Millennium Program is designed to validate high-payoff, cutting-edge technologies to enable those technologies to become more broadly available for use on other NASA programs. The experiment is slated to be exercised in October of 1998.

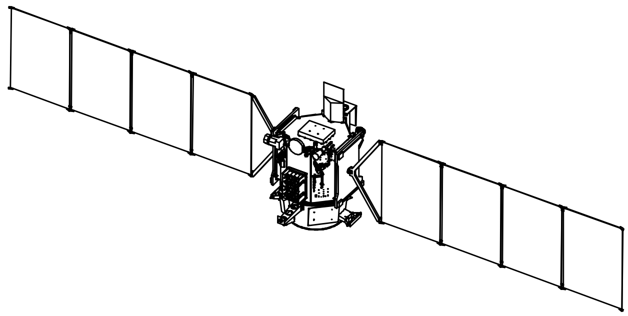


Figure 1. DS1 Spacecraft

Section 2 discusses the benefits to the spacecraft community from increased spacecraft autonomy and the motivation for this work. Section 3 outlines some of the challenges to acceptance of spacecraft autonomy and Section 4 introduces the Remote Agent design approach and architecture. Section 5 covers the particulars of the DS1 Remote Agent experiment. Section 6 discusses the functioning of each of the three technology components of the Remote Agent. Section 7 describes how the Remote Agent software is integrated into the separately-developed Deep Space One flight software. Section 8 describes how the Remote Agent experiment is tested prior to flight. Section 9 summarizes the paper and describes plans for future Remote Agent development.

2. NEED FOR AUTONOMY ON SPACECRAFT

The desire to increase the level of spacecraft autonomy comes from at least three separate objectives of spacecraft customers: taking good advantage of science opportunities, reducing spacecraft operations costs, and handling uncertainty—including ensuring robust operation in the presence of faults.

Taking Advantage of Science Opportunities

Our science customers would like the spacecraft to be able to modify its sequence of actions more quickly based on late-breaking information available on the spacecraft.

For example, an ultraviolet spectrometer on a comet flyby mission might identify a region of particular interest for intense scrutiny. With current technology, scientists have to make do with whatever pre-planned sequence of observations has been stored on-board and cannot reprogram any of those to examine more closely the newly identified region of interest. With a future RA, plans may be revised based on this new information hours or minutes before flyby. With ground-based control, a turnaround time of hours is impractical and a turnaround time of minutes is physically impossible due to the speed of light. See Figure 2.

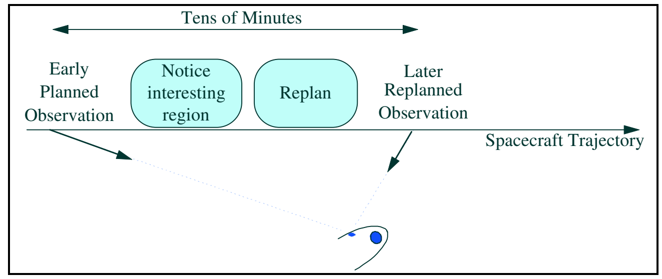


Figure 2. Fast replanning based on new information

Similarly, on the Mars Pathfinder mission, the science team requested the ability for the meteorology instrument, when it senses that a dust devil is passing, to tell the camera to take unplanned images aimed at the departing dust devil. It is difficult to see how this capability could coexist with time-tagged command sequences for the imaging planned for the rest of the day.

Reducing Spacecraft Operations Costs

Our funding sources are insisting that means be found to reduce operations costs. A fixed amount of funding is available from NASA for solar system exploration including spacecraft development and operations. When operations costs are reduced, more resources become available for developing a wider variety of interesting solar system exploration missions. Development of detailed spacecraft sequences accounts for the largest expenditure in operations budgets.

By commanding spacecraft at a higher level of abstraction, much of the sequence development task becomes the responsibility of the flight software, reducing ground operations costs. Some of the savings come from a change in how we think about operations planning. The old approach was that all spacecraft activities needed to be predicted and approved by ground controllers. The new thinking is that the ground controllers do not (always) need to know the low-level details of spacecraft activities but only the capabilities of the spacecraft and the high-level goals.

Ensuring Robust Operation in the presence of uncertainty

Our customers still require high reliability and the ability to respond to problems in flight. For existing spacecraft, the fault protection system often represents the most autonomous system on-board. Robust operation is desired in the presence of hard faults, degraded performance, and operator errors.

Traditional spacecraft, even in conservative designs, generally provide some minimal level of fault protection out of necessity. Otherwise, any major problem with attitude control, power, or antennas could by itself prevent ground controllers from diagnosing or correcting the problem. The Remote Agent is able to go a step further: after recovering from a fault, it can continue the mission, even if it involves replanning for degraded capability.

Another advantage of the Remote Agent derives from the nominal and failure modeling used by the fault diagnosis engine. For hard-coded fault protection designs, the domain knowledge is implicit rather than explicit. This means that we rely on the fault protection algorithm developers to understand the system, and abstract from that understanding a design for which symptoms to look for and what responses to take when they show up. In contrast, with model-based fault diagnosis, the fault protection software engineers explicitly model how the system behaves in nominal and failure cases. Fault diagnosis then becomes a search for likely diagnoses given observed symptoms. Since the spacecraft designers understand the details of the system behavior, there is an advantage to having them encode their knowledge explicitly at design time.

3. AUTONOMY TECHNOLOGY INSERTION REQUIREMENTS

It is not enough to build a better mousetrap; it won't catch any mice unless it gets used. There are similar issues for the insertion of higher levels of autonomy into spacecraft designs. The design must be developed with the needs of two sets of customers in mind: the spacecraft test engineers and the mission controllers.

Spacecraft Test

Conversations with spacecraft test engineers have raised a number of concerns that must be addressed in any autonomous system design process.

1. Determinism and non-determinism: Is the system non-deterministic? How do we test the system if we don't control its initial conditions in flight?

For the current Remote Agent design, the system is deterministic to the extent that the same set of inputs will yield the same outputs each time. The context for this question, however, is that we cannot predict the exact set of commands that the Remote Agent will use to achieve a set of goals far in the future since we cannot predict exactly what the spacecraft state will be at that time. This situation is common in another context, that of attitude control systems. We don't know exactly when a particular thruster will fire, but we do know that the system will fire thrusters as needed to achieve the higher level goal of holding the commanded attitude.

So how do we test such a system? For an attitude control system, we develop multiple scenarios and verify that the pointing error meets requirements in all situations. We also check that the propellant usage is acceptable while the requirements are being met. Continuing the analogy with an attitude control system, we develop multiple scenarios and test whether the high level goals are met, and analyze whether the resources required to do so were acceptable.

2. Earlier system behavior definition: The flight system is more complex, so more testing is needed earlier and the desired behavior needs to be defined long before launch. Some additional techniques are required. These are described in the testing section of this paper.

The concern about early definition may be valid depending on how much of the spacecraft behavior we choose to build into the flight software before launch. With the traditional sequence development approach, many sequences are developed after launch, so there is no opportunity to observe full end-to-end behavior in a test environment. With an on-board planner, we now have the opportunity to design and test the behavior before hand. It should be pointed out that this is an opportunity and not a requirement. For example, the Project may choose to delay final design of flyby scenarios until after launch. In this case, we should expect to update the on-board planner and mission goals at the time that the scenario is finalized and this may be after launch.

3. Test Plan coverage: How do we develop a test plan that assures adequate coverage? How should test cases be devised? What needs to be tested in system test? The core engines underlying the Remote Agent are unfamiliar to spacecraft test teams and could require large effort to test.

First, a distinction should be made between the Remote Agent infrastructure or engines and the mission-unique models. The Remote Agent infrastructure will be extensively analyzed and tested in pre-integration unit tests. At the system test level, the focus should be on whether the behavior of the Remote Agent meets the goals and constraints set for it.

As with any complex system, the test plan needs to include nominal cases, failure cases, and cases that test the boundaries of the system so that the operators learn where it will break. The planner can be challenged by overloading the number of tasks to be done in a short time. The executive may be challenged with a large number of tasks requiring immediate response, and fault protection may be challenged by examining its response to multiple, closely spaced failures. Planner unit tests will include examples using each constraint. Executive unit tests should explore each approach that might be used to achieve a task and fault protection tests still depend on devious testers to invent challenging scenarios.

A large variety of tests seeking extreme and boundary condition behavior is indicated when testing any complex software system.

A major advantage of the Remote Agent approach is that it depends on declarative hardware knowledge; in other approaches the hardware knowledge is captured Only implicitly. Explicit models come in handy at review time because the software engineer can sit with the hardware expert and review the declarative model of the hardware. This helps reduce errors in understanding between the hardware and software engineers.

Mission Operations

Mission operators or controllers have clearly expressed a number of requirements or desires with respect to fielding autonomous systems. These include:

1. Low level commanding: Operators should be able to have access to low-level control of spacecraft hardware unimpeded by the autonomous system.

As this requirement became clear, the Remote Agent design was modified to allow low-level hardware command access—potentially bypassing some autonomous capabilities and safeguards. Unless the Remote Agent is instructed in the context and goals of these low level commands, they need to be used carefully and when the spacecraft is in a low activity quiescent mode.

2. Ground override authority: An ability to command the spacecraft to revert to a low-level of autonomy mode if the controllers decide that they want to disable the autonomous feature.

This requirement is met on DS1.

3. Migration of autonomy capabilities: A sequence that allows demonstration of autonomous capabilities as ground system capabilities prior to fielding them on the spacecraft as on-board capabilities.

The Remote Agent experiment is being designed to meet this requirement by first engaging the executive as just another basic sequence engine, then allowing Remote Agent to execute a pre-computed plan sent from the ground, and finally enabling the on board planner, bringing the full DS1 Remote Agent level of autonomy to bear.

4. Behavior Prediction: The ability to predict (at some level) what the behavior of the spacecraft will be when the spacecraft begins to execute the on-board-generated plan.

There will be a copy of the on-board planner built into the ground system. This copy will be used to generate experience and rules of thumb as to what sets of goals are easily achievable and what sets are difficult to achieve for the on-board system based on these rules of thumb. The operators will define the goals for each mission phase and since the Remote Agent is closing the loop around these goals, the best prediction of spacecraft behavior is that the goals will be achieved on schedule.

The Remote Agent has been designed to support multi-level commanding and monitoring in order to enable ground controllers to adjust the level of autonomy they desire across different activities or mission phases [1].

4. REMOTE AGENT DESIGN APPROACH AND ARCHITECTURE

. The New Millennium Autonomy Architecture rapid Prototype (NewMaap) effort [2] identified the key contributing technologies: on-board planning and replanning, multi-threaded smart executive, and model-based failure diagnosis and repair. In NewMaap, we learned how to take advantages of the strengths and weaknesses of these three technologies and merge them into a powerful system. After successful completion of the prototype, the RA was selected as one of the NMP technologies for DS1. It will be uplinked to the spacecraft as a software modification and demonstrated as an experiment.

Fig. 3 shows the communications architecture for the Remote Agent's interaction with the rest of the spacecraft flight software. Note that all interaction with the hardware is the responsibility of the real-time software. The RA is layered on top of that software, but also gathers information from all levels to support fault diagnosis.

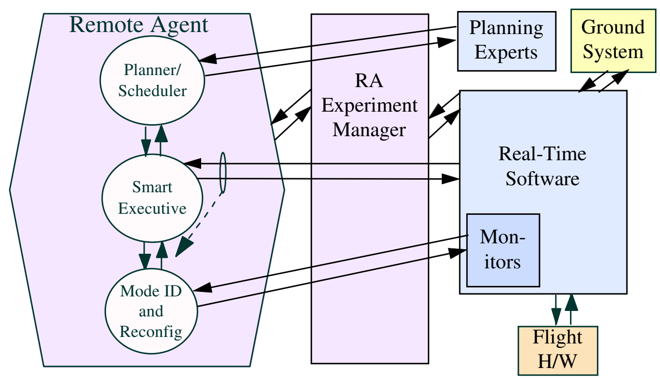


Figure 3. Remote Agent Communication Architecture

Several spacecraft commanding styles are possible. Goal-oriented commanding is the intended operating mode for most of an RA mission; provision has been made for updating the goals in flight. In a typical planning cycle, the executive is executing a plan and gets to an activity that can be interpreted as "time to plan the next segment." The executive calls the planner with the current and projected spacecraft state including the health of all devices. The planner/scheduler generates a new plan using priorities, heuristics, and domain models including system constraints. The planner sends this plan to an executive that creates an agenda of plan items and executes the agenda. Plan execution robustness is added by making use of the Model-based Mode Identification and Reconfiguration (MIR) system. The MIR system includes monitors, mode identification for nominal and failure conditions, communication of state to the executive and proposals of reconfiguration actions to take in the event of failures.

Each of the components of the Remote Agent will be described in more detail in Section 6, but first the Remote Agent experiment for the Deep Space One mission will be described in more detail.

5. THE DEEP SPACE ONE REMOTE AGENT EXPERIMENT

The Remote Agent eXperiment (RAX) for Deep Space One is a demonstration of RA capabilities. Since an alternate method of control is used for most of the mission, RAX is focused on demonstrating specific autonomy capabilities rather than controlling all aspects of spacecraft behavior. The Remote Agent controls the following spacecraft hardware and software: the camera for use in autonomous navigation, the Solar Electric Propulsion (SEP) subsystem for trajectory adjustment, the attitude control system for turns and attitude hold, the navigation system for determining how the actual trajectory is deviating from the reference trajectory and what SEP thrusting profile is needed to stay on the reference trajectory, the Power Amplification and Switching Module (PASM), for use in demonstrating fault protection capabilities.

Four failure modes are covered by RAX. These are:

F1. Power bus status switch failure

F2. Camera power stuck on

F3. Hardware device not communicating over bus to flight computer

F4. Thruster stuck closed

Mission Scenario

The Remote Agent experiment is executed in two phases, a 12 hour Phase One followed a couple of weeks later by a 6 day Phase Two.

In Phase One, we start slowly by first demonstrating the executive operating in the manner of a low level sequencer by accepting commands to turn devices on and off. Next, a "scripted" mode is demonstrated with execution of plans uplinked from the ground. The main demonstration here will be commanding the spacecraft to go to and stay in a known, safe, standby mode and then take a series of optical navigation (OpNav) images. In addition, Failure mode F1 will be demonstrated by injecting power bus switch status readings indicating that a power bus is unexpectedly off. The fault diagnostic system will examine this information along with other information that indicates that devices on the bus are still communicating normally with the flight computer and conclude that the failure is in the switch status measurement and not in the bus itself. No action will result. No planning or SEP thrusting are attempted in Phase One.

In Phase Two, we also start by demonstrating low level commanding, and then initiate on-board planning. Based on the spacecraft initial state and the uplinked goals, the planner will generate a three day plan including imaging for optical navigation, thrusting to stay on the reference trajectory, and simulated injection of faults to test out failures F2, F3, and F4. First the camera power stuck on failure (F2) is injected. When the executive is unable to turn off the camera when the plan so dictates, the executive realizes that the current plan should be aborted and replanning is indicated. This might be necessary, for example, because the initial plan's assumptions on power consumption are incorrect with the camera on when it should be off. The plan is declared failed, the spacecraft is sent to a standby mode while the planner is requested to replan based on the new information that the camera power switch is stuck on. When the new plan is received by the executive, execution resumes including navigation and SEP thrusting. Near the end of the three day plan, the planner is called to generate the plan for the next three days. This plan includes navigation and SEP thrusting as before. It also includes two simulated faults. First, a failure of a hardware device to communicate is injected (F3); the proper recovery is to reset the device without interrupting the plan. Next, a thruster stuck closed failure (F4) is simulated by injecting an attitude control error monitor above threshold. The correct response is to switch control modes so that the failure is mitigated.

RA Capabilities Demonstrated with DS1 RAX

The above scenario has been designed to demonstrate that the DS1 Remote Agent meets the following autonomy technology goals:

- Allow low-level command access to hardware
- Achieve goal oriented commanding
- Generate plans based on goals and current spacecraft state expectations
- Determine the health state of hardware modules
- Demonstrate model-based failure detection, isolation, and recovery
- Coordinate hardware states and software modes
- Replan after failure given new context

6. RA COMPONENTS

The major components of the Remote Agent are discussed below.

Planner/Scheduler

The highest level commanding interface to the Remote Agent is provided the Planner/Scheduler (PS). PS maintains a database of goals for the mission, the *mission profile,* that spans a very long time horizon, potentially the duration of the entire mission. Over the duration of a mission PS is iteratively invoked by the executive to return a synchronized network of high-level activities, the plan, for each short-term scheduling horizon into which the mission profile is partitioned. Typically each short-term horizon covers several days. When PS receives a request from EXEC, it identifies the next scheduling horizon, retrieves from the mission profile the goals relevant to that horizon, merges in the expected initial spacecraft state provided by EXEC into a incomplete, initial plan and generates a fully populated plan. PS sends that plan to EXEC for execution.

For RAX, Phase Two, the mission profile will cover 6 days and contain two scheduling horizons of three days each. RAX allows the specification of two kind of goals. One specifies the frequency and duration of the "optical navigation windows", the time during which the spacecraft is requested to take a set of asteroid pictures to be used for orbit determination by the on-board Navigator. The second type of goal specifies a "mini-sequence", i.e., a set of lower-level commands that EXEC will issue to the real-time software, and requirements to activate the mini-sequence with certain synchronization constraints with respect to other planned activities. A new plan will be requested of MM/PS in two situations:

- *nominal operations:* in this case EXEC reaches the activity `Planner_Plan_Next_Horizon` toward the end of the current scheduling horizon. EXEC will issue a request for a new plan. This request will define the new initial state as the expected final state from the plan currently in execution. This will allow seamless patching of the old and new schedule without any interruption of execution.
- *fault response:* if the fault protection system detects an anomaly that will impact the executability of future tasks in the plan, the EXEC will request a new plan to resume normal operations after having put the spacecraft in a safe standby mode. In this case the initial state describes the standby tasks or holding states for each subsystem modeled in the plan and health information describing possibly degraded modes for failed subsystems.

Notice that from the point of view of PS both the nominal and fault response case are handled exactly in the same way.

Ground controllers can add, modify, or delete goals from the mission profile by explicitly issuing a command to the mission profile. For example, in a mission in which the spacecraft communicated to Earth through the Deep Space Network, the final communication schedule allocated to the mission may become available only a few weeks ahead of time and it is possible that a schedule may change with a short notice (e.g., within a week). Ground controllers will need to communicate both of these situation to the spacecraft by issuing appropriate edit commands to modify the mission profile.

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. PS presents several features that distinguish it from other Artificial Intelligence and Operations Research approaches to the problem. For example, 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 subgoals 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 shut-down 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). The 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 typically addressed in two sequential problem solving stages, often by distinct software systems. Another important distinction between the Remote Agent PS and other classical approaches to planning is that besides 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.

PS consists of a heuristic search engine, the Incremental Refinement Scheduler (IRS) that operates in the space of incomplete or partial plan [6]. Since the plans explicitly represent time in a numeric (or metric) fashion, the planner makes use of a temporal database. As with most causal planners, PS begins with an incomplete plan and attempts to expand it into a complete plan by posting additional constraints in the database. These constraints originate from the goals and from constraint templates stored in a model of the spacecraft. The temporal database and the facilities for defining and accessing model information during search are provided by the HSTS system. For more details on PS and the HSTS system see [3] and [4]. Figure 4 describes the PS architecture.

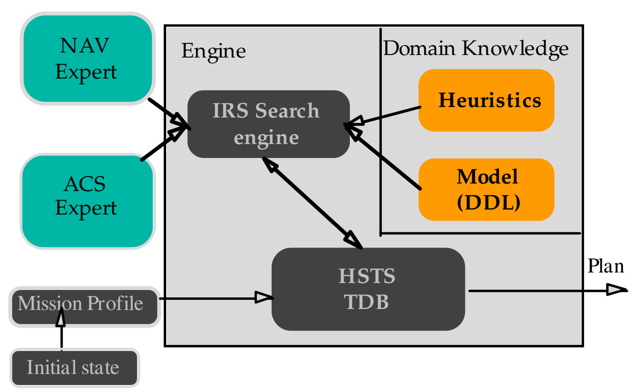


Figure 4. Planner/Scheduler Architecture

The coverage of the RAX model is described in Table 1. Appendix B gives a detailed description of the timelines and tokens needed by PS to handle the propulsion and thrust subsystems of the spacecraft.

Table 1 Summary of Planner Models for RA Experiment

Subsystem	*State Variables*	*Value Types*	*Compat-ibilities*	*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 the 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 (see Appendix B for details)
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 the Executive can request the plan for the next horizon.
Mission	Goal: 1	2	2	Allows the Mission Manager and the planner to coordinate activities based on a series of scheduling horizons updatable by Mission Ops for the entire mission

Each subsystem in the model is represented in the PS database. Each subsystem has a set of dynamic state variables whose value is tracked over time. 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. An example the atomic temporal constraints that belong to a compatibility can be expressed in English as "While the spacecraft is taking asteroid pictures requested by navigation, no ion thrusting is allowed".

In Table 1 we identify four distinct kinds of state variables. A *goal* timeline will contain the sequence of high-level goals that the spacecraft can satisfy (e.g., the `Navigate` goal described before). Goal timelines can be filled either by ground operators or by on-board 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. For example, in a previous fault situation ground controllers may have decided that the IPS engine is not trustworthy and therefore should not be considered operational until further tests have been run. PS will therefore generate plans that do not involve using the engine. They may decide to run some tests by posting appropriate goals in the mission profile and therefore not breaking nominal plan execution. After the tests they may decide that the IPS engine is trustworthy after all and may send a message to EXEC that it is again OK to thrust. EXEC will communicate this to PS through the health timeline in the next scheduling horizon, without needing to interrupt regular plan execution and put the spacecraft in standby mode.

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 a overall consistent plan. This will greatly facilitate the work of the ground team.

Table 2 gives quantitative information regarding the three plans that PS is expected to generate on board during the 6 day experiment. The *tokens* and *constraints* columns contain the number of tokens and pairwise temporal constraints (e.g., "token A starts between 1 and 2 minutes after token B") in the plan respectively. The first *CPU time* column reports the actual measured run time of PS on a PowerPC/VxWorks flight hardware testbed. The next column reports the estimated time to generate the same plans on the actual RAD6000 flight processor for DS1. The scale up factor of 40 is due to the lower speed of the RAD6000 with respect to the PowerPC (about one order of magnitude) and the allocation of only 25% of the CPU to the PS process.

Table 2. PS Metrics for Performance

Scenario	*tokens*	*con-straints*	*CPU time on PPC testbed (mm:ss)*	*Est. CPU time on RAD6000 (hh:mm:ss)*
First horizon	105	141	7:13	4:48:00
Replan in first horizon	69	66	4:01	2:40:00
Second horizon	126	192	13:49	9:12:00

Executive

The Smart Executive (EXEC) is a reactive plan execution system with responsibilities for coordinating execution-time activity. EXEC's functions include plan execution, task expansion, hardware reconfiguration, runtime resource management, plan monitoring, and event management. The executive invokes the planner and MIR to help it perform these functions. The executive also controls the lower-level software by setting its modes, supplying parameters and by responding to monitored events.

Task Expansion EXEC provides a rich procedural language, ESL [5], in which we define how complex activities should be broken up into simpler ones. A procedure can specify multiple alternate **methods** for goal achievement to increase robustness. If a selected method fails, EXEC will try any other methods applicable in the current context.

Resource Management As a multi-threaded system, EXEC works on multiple activities simultaneously. These activities 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 Ref. [8] for a more detailed discussion.

RAX Startup Upon startup, EXEC asks MIR to describe the current spacecraft configuration. Then EXEC puts the spacecraft into **standby mode**. Standby mode is a safe mode that guarantees sufficient power and ground communications as well as a thermally benign state. Once standby mode has been achieved, EXEC then begins its normal operational cycle.

Operational Cycle The top-level operational cycle, including the planning loop, is described as follows. EXEC requests a plan, by formulating a plan-request describing the current plan execution context. It later executes and monitors the generated plan. EXEC executes a plan by decomposing high-level activities in the plan into primitive activities, which it then executes by sending out commands, usually to the real-time flight software (FSW). EXEC determines whether its commanded activities succeeded based either on direct feedback from the recipient of the command or on inferences drawn by the Mode Identification (MI) component of MIR. When some method to achieve a task fails, EXEC attempts to accomplish the task using an alternate method in that task's definition or by invoking the Mode Reconfiguration (MR) component of MIR as a "recovery expert". If MR finds steps to repair the failing activity without interfering with other concurrent executing activities, EXEC performs those steps and then continues on with the original definition of the activity. If the EXEC is unable to execute or repair the current plan, it aborts the plan, cleans up all executing activities, and puts the controlled system into a stable safe state (called a "standby mode"). In situations where continued operation is allowed, EXEC then requests a new plan from PS while maintaining this standby mode until the plan is received, and finally executes the new plan.

Periodic Planning Cycle As shown in Figure 5, our approach separates an extensive, deliberative planning phase from the reactive execution phase, executing infrequently generated plans over extended time periods. How far in advance the system should plan is constrained by several factors, including uncertainty about the results of execution. We use the term "planning horizon" to describe the length of time into the future for which a plan is constructed. In normal operations, the RA would plan a week ahead of time, and when it comes near the end of the current plan it would start working on the plan for the next horizon. Since the actual RAX experiment lasts for only one week, the planning horizon is set considerably shorter (3 days).

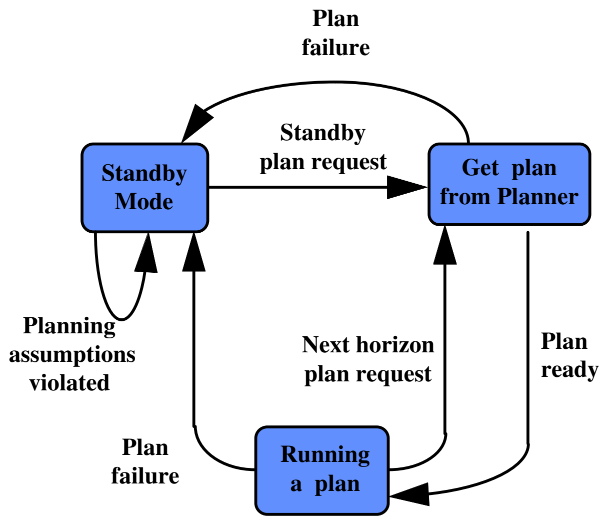


Figure 5 Executive Periodic Planning Cycle

We address the problem of generating initial states for the next planning round differently depending on the status of the currently-executing plan. Plans normally include the task of planning for the next horizon—i.e., the planner sets aside a good time for its own (next) computation. At this point, the executive sends to the planner the remainder of the current plan in its entirety, with annotations for the decisions that were made so far in executing it. The current plan serves as its own prediction of the future at the level of abstraction required by the planner. Thus, all the planner has to do is extend the plan to address the goals of the next planning horizon and return the result to the executive. The executive must then merge the extended plan with its current representation of the existing plan. The net result is that, from the executive's perspective, executing multiple chained plans is virtually the same as executing one long plan. This has the useful consequence that it enables the executive to engage in activities which span multiple planning horizons (such as a 3-month long ion engine burn) without interrupting them.

In the event of plan failure, the executive enters standby mode prior to invoking the planner, from which it generates a description of the resulting state in the abstract language understood by the planner. Note that establishing standby modes following plan failure is a costly activity with respect to mission goals, as it causes us to interrupt the ongoing planned activities and lose important opportunities. For example, a plan failure causing us to enter standby mode during the comet encounter would cause loss of all the encounter science, as there is no time to re-plan before the comet is out of sight. Such concerns motivate a strong desire for plan robustness, in which the plans contain enough flexibility to continue execution of the plan under a wide variety of execution outcomes. Executing a flexible plan is not easy, and draws on many capabilities of our "Smart" EXEC.

Plan Execution We now describe the plan execution capability of the executive in more detail. The planner represents spacecraft activity as a set of concurrent subsystems. Each independent component of a subsystem is conceptualized as a state variable, which can take on a series of different behaviors over time. A plan consists of one **timeline** for each state variable. Each timeline contains a sequence of constraints on the behavior of the state-variable. A **token** is a data structure which represents one part of a sequence on a timeline. A token has information about the desired behavior throughout the duration of the token, and also flexible constraints on when the token can start and finish. Lastly, the plan contains constraints to coordinate behavior across tokens on different timelines, called compatability constraints. An example of a compatibility constraint is one which says that a "take-picture" token may only be executed within the window during which the corresponding "keep-pointing-at-target" token is activated.

The EXEC is a multi-threaded process that is capable of asynchronously executing activities in parallel. EXEC has one thread for each timeline in a plan, and a procedure, called the token definition, for each type of token contained in the plan. A token definition procedure contains a precondition that must be met before the activity can start, a postcondition that must be met before the activity can finish, and a body which describes how the procedure is actually executed. To execute a plan, EXEC activates on the corresponding thread for each timeline the procedure corresponding to the first token on that timeline. EXEC tracks the status of all tokens in a data structure called an **agenda.** When a new token is able to start (because the previous token has finished and all other constraints are satisfied), EXEC terminates the previous token procedure and transitions to the next one. For example, once the token for turning to a target has completed, the token for constantly pointing at the target can then be activated. This enables the "take-picture" token on the camera timeline to be activated. Only when the picture activity has finished will the EXEC terminate the "keep pointing at target" token and transition to the token for turning to the next target attitude. The tokens executed by the RAX Executive are summarized in Appendix A.

In more detail, plan execution is achieved through the following cycle, as shown in Figure 6:

1. EXEC receives a new plan from the planner and updates the plan execution agenda.

2. EXEC chooses a new task (usually arising from a plan-level token) on the agenda that is ready for execution.

3. EXEC decomposes the task into a series of sub-tasks based on task definition models and current execution context. Sub-tasks are recursively decomposed down to the level of primitives. EXEC invokes MIR as a recovery expert to achieve tasks that have failed.

4. EXEC begins to execute a primitive task, for example by sending a command to the FSW or waiting until a condition becomes true.

5. (Not shown) FSW processes the command by making a change in a software parameter or device state. The monitor for the affected FSW component registers the change in

low-level sensor data and sends MI a new abstracted value for the state of the affected components. MI compares the command to the observations, infers the most likely actual nominal or failure mode of each component, and sends an update to EXEC describing the changes in any modes of interest to EXEC.

6. EXEC compares the feedback from external events, such as the MI mode updates, to the conditions specified in its task models to determine whether the command executed successfully. If so, it proceeds to take further steps to complete the high-level token. If the token is finished, EXEC updates its agenda and continues the cycle.

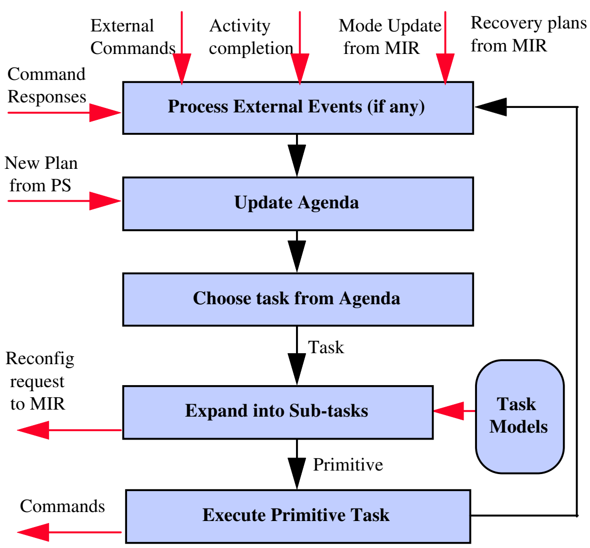


Figure 6. Executive Plan Execution Cycle

Hard command execution failures may require the modification of the schedule in which case the executive will coordinate the actions needed to keep the spacecraft in a "standby mode" and request the generation of a new schedule from the planner.

Architecture-support timelines Most timelines (and hence tokens) represent the activity of spacecraft subsystems external to the RA. However, the RA also contains two timelines used to support architectural features. First, the PLANNER-PROCESSING timeline describes the activity of the planner. The PLAN-NEXT-HORIZON token for this timeline corresponds to a state in which the planner is generating a new plan. EXEC executes this token by generating a plan request, sending it off to the planner, and then incorporating the new plan into the current execution context. This supports the model of planning with multiple horizons described above. The SCRIPT-NEXT-HORIZON token for this timeline is similar, except it directs EXEC to load and execute the plan defined in a file previously up-linked from ground. In this way ground controllers can also support back-to-back plans. This also supports the use of the automated planner running in closed-loop fashion either from the ground or on-board the spacecraft, hence supporting easy migration of planning capability from human-based, to automatic ground-based, to autonomous on-board planning.

Second, the EXEC-ACTIVITY timeline represents low-level activities that EXEC will perform that are lower-level than the tokens managed by the on-board planner. To execute the EXEC-ACTIVITY token, which takes a filename as an argument, EXEC simply loads and executes the referenced file. The file can contain arbitrary Lisp code, including any commands executable on the spacecraft. This timeline can be used to run EXEC in a mode corresponding to a traditional sequencer, by sending up a plan that contains only a sequence of EXEC-ACTIVITY tokens, each with low-level commands defined in a file. However, since this timeline runs concurrently with all the timelines defined for the planner, it also enables ground operators to require certain low-level activities to be inserted into whatever high-level plan is generated autonomously. EXEC also supports use of the EXEC-ACTIVITY procedure as an immediate function invocable by ground controllers. Hence, even in the middle of an autonomous plan execution, or in standby mode, ground operators can ask EXEC to run arbitrary low-level commands from a file and these can be tied to events rather than being linked to prespecified clock times. For the complete list of RAX timelines and tokens, see Appendix A.

Summary of Executive Capabilities Demonstrated in RAX We now summarize how the EXEC capabilities described above are demonstrated within the RAX scenarios.

First, EXEC demonstrates the multi-level commanding, allowing ground operators to specify low-level commands to the hardware as part of a sequence, to generate plans from ground, or to request and execute plans generated on-board the spacecraft. The low-level commanding and ground-based planning are demonstrated in Phase One of the RAX experiment, in which a plan is up-linked from the ground which contains both high-level activities (like turning to a target) and low-level activities (using the EXEC-ACTIVITY tokens to simulate the injection of various faults, and to turn PASM on and off).

Second, EXEC demonstrates plan request generation and execution. This is demonstrated from a currently executing plan where nothing has changed (nominal scenario), from a currently executing plan where a device health token has been updated (following the MICAS health update), and from a standby mode. As part of executing a plan phase two, EXEC demonstrates a number of important capabilities involved in token decomposition.

- EXEC demonstrates *context sensitive behavior* in the management of the ion propulsion system. Before executing a thrust command, EXEC requires that IPS is in standby mode. If it is already in standby mode, EXEC proceeds to the thrusting, otherwise it will put IPS into the standby mode before proceeding.
- EXEC demonstrates *time-driven* token durations. For example, it terminates a thrust segment based on a timeout, rather than external confirmation.
- EXEC demonstrates *event-driven* token durations, in which the picture tokens are not allowed to terminate

until the picture has actually finished, turn tokens are completed only upon receipt of turn-complete messages from the ACS, and the SEP-THRUSTING token is only completed upon a message from MIR that IPS is indeed in the thrusting state.

- EXEC demonstrates *goal-oriented achievement* (don't achieve things that are already true). Because the planner is unable to determine how many thrust segments are necessary to achieve the total desired thrust, it inserts thrust tokens into the plan which may not need to be executed. EXEC tracks how much thrust has been achieved, and only executes thrust tokens (and associated turns) for so long as thrust is actually necessary.
- EXEC demonstrates the *coordination of activity details* across subsystems that are below the level of visibility of the planner. There is a constraint that ACS be in thrust-vector-control (TVC) mode shortly after IPS has started thrusting. When EXEC commands IPS into thrusting mode, it also sends the command to ACS to enter TVC mode based on its own lower-level domain knowledge. Similarly, EXEC puts ACS back into Reaction Control System (RCS) control mode upon termination of a thrusting activity.

Third, EXEC demonstrates the ability to maintain required properties in the face of failures. In the thruster failure scenario, EXEC learns from an MIR state update that the current thruster mode is faulty. It invokes MIR with a recovery request and then executes MIR's recommendation to change to a degraded thruster control mode.

Fourth, EXEC demonstrates the ability to recognize plan failure, abort the plan, enter standby mode, and request and execute a replan. This occurs in the MICAS failure scenario, in which EXEC learns from MIR that MICAS is stuck on and cannot be turned off. EXEC requests a recovery from MIR so that it can turn MICAS off, but since there is no way to fix this problem MIR informs EXEC that is has no recovery. Since the plan requires MICAS to be off, EXEC aborts the plan, terminating a thrusting segment if necessary. It then enters a degraded standby mode, in which it leaves MICAS on despite the usual desire to turn off all unnecessary devices in standby mode, and requests a plan for the planner. In its plan request, EXEC informs the planner that MICAS is stuck on. Later, in executing the new plan, ground finds a way to fix MICAS and informs MIR of this fact. When EXEC learns from MIR that MICAS can now be shut off, this new information does not cause EXEC to abandon the plan, since the planner did not *require* MICAS to be broken. However, the next time EXEC asks for a plan, it informs the planner about the restored health of MICAS, so that the planner can now plan to switch MICAS off when desired. EXEC also demonstrates the ability to terminate plans based on an immediate command from the ground, in which case it enters whichever standby mode the command specifies.

Implementation EXEC is implemented on top of Execution Support Language (ESL) [5], which in turn is implemented using multi-threaded Common LISP. The internal EXEC code is designed in a modular, layered fashion so that individual modules can be designed and tested independently. Individual device knowledge for RAX is implemented based on EXEC's library of generic 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 References [6, 7. and 8].

Diagnosis and Repair

We refer to the Diagnosis and Repair engine of the Remote Agent as MIR, for Mode Identification and Reconfiguration, which emphasizes the model-based diagnosis and control flavor of the system. MIR eavesdrops on commands that are sent to the on-board hardware managers by the EXEC. As each command is executed, MIR receives observations from spacecraft sensors, abstracted by monitors in lower-level device managers for the Attitude Control Subsystem (ACS), Bus Controller, and so on. MIR combines these commands and observations with declarative models of the spacecraft's components to determine the current state of the system and report it to the Exec. A very simple example is shown schematically in Figure 7. In the nominal case, MIR merely confirms that the commands had the expected effect on spacecraft state. In case of failure, MIR diagnoses the failure and the current state of the spacecraft and provides a recovery recommendation. A single set of models and algorithms are exploited for command confirmation, diagnosis and recovery.

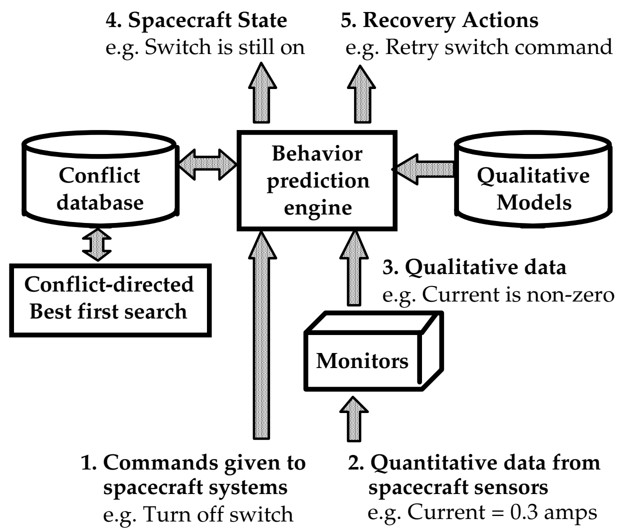


Figure 7. Information Flow in MIR

The RAX mission scenario demonstrates the following MIR capabilities: state identification throughout the experiment, diagnosis of sensor failure F1, diagnosis and recovery recommendations for device failures F2-F4, and overriding of a MIR diagnosis via a ground command.

F1 illustrates MIR's ability to disambiguate between a sensor failure and failure of the device being sensed. MIR combines power distribution models with the sensed nominal current draw and communication status of devices to conclude that the power switch must be on and that a switch sensor failure, though unlikely, has occurred.

Failures F2-F4 are diagnosed in a similar fashion and include the possibility of recovery. F2 focuses on repeated attempts to recover a camera switch until it is deemed permanently stuck. F3 illustrates successful recovery of communication with a device by resetting its remote terminal (RT). In F4, given only an attitude error and models of the spacecraft dynamics, MIR infers 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.

Since we cannot depend on failures F1-F4 occurring during the experiment, failures will be simulated by injecting false monitor readings consistent with the failures. The RAX will be expected to take the appropriate corrective actions, though none are necessary. Injecting simulated failures may seem senseless. However, in lieu of a guaranteed real failure, it provides greater confidence that the system is flight ready and will demonstrate that when the RA reacts to a failure the ground controllers will be able to observe, interpret, and, if necessary, override the actions it has taken. While simulations are necessary for demonstration, the RAX is fully responsible for responding to real failures within its limited scope occurring during the experiment. This raises an additional challenge regarding how the RAX will avoid conflicts with the flight software fault protection mechanism (FP), since both may be react to the same failure. Rather than negotiate a complex resolution strategy, the RAX was designed with a narrower notion of nominal operation than the FP (by tuning monitors appropriately), thus avoiding the conflict altogether. When the RAX is operational, it should always respond to and mitigate faults within its mandate before the FP monitors are triggered. If the RAX fails to do so, the FP will terminate the RAX upon being triggered.

The MIR component of the RA architecture, embodied in a system called Livingstone, consists of two parts: mode identification (MI) and mode reconfiguration (MR). MI is responsible for identifying the current operating or failure mode of each component in the spacecraft. Following a component failure, MR is responsible for suggesting reconfiguration actions that restore the spacecraft to a configuration that achieves all current goals as required by the planner and executive. Livingstone can be viewed as a discrete model-based controller in which MI provides the sensing component and MR provides the actuation component. MI's mode inference allows the executive to reason about the state of the spacecraft in terms of component modes, rather than in terms of low level sensor values, while MR supports the run-time generation of novel reconfiguration actions.

Livingstone uses algorithms adapted from model-based diagnosis [9, 10] 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 [10], 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.

Livingstone extends the basic ideas of model-based diagnosis by modeling each component as a finite state machine, and the whole spacecraft as a set of concurrent, synchronous state machines. Modeling the spacecraft as a concurrent machine allows Livingstone to effectively track concurrent state changes caused either by executive commands or component failures. An important feature is that the behavior of each component state or mode is captured using abstract, or qualitative, models [11, 12]. 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. In addition, abstract models can be reduced to a set of clauses in propositional logic. This form allows behavior prediction to take place via unit propagation, a restricted and very efficient inference procedure.

MIR's abstract view of the spacecraft is supported by a set of fault protection monitors which classify spacecraft sensor output into discrete ranges (e.g. high, low nominal) or symptoms (e.g. excessive attitude error). One goal of the RAX was to make basic monitoring capability inexpensive so that the scope of monitoring is driven from a system engineering analysis instead of being constrained by software development concerns. To achieve this, monitors are specified as a dataflow schema of feature extraction and symptom detection operators for reliably detecting and discriminating between classes of sensor behavior. Second, 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-detection algorithms, reduce the occurrence of errors through automation and streamline monitor design and test.

Table 3 illustrates the classes of components modeled by MIR for the DS1 spacecraft. For each we list the number of instances in the overall spacecraft model and the modes (states) the component can occupy. All told the MIR model represents fifty-seven components of twelve different types, their behavior, and their interconnections. For ease of modeling, MIR allows a set of components and a model describing their interconnection to be grouped into a module which can be treated as a unit. Table 4 illustrates the modules created to model DS1. For each we list the number of instances in the overall spacecraft model and the components or other modules the module contains.

Table 3. 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 4. 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

It is important to note that the MIR 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, MIR 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 which is inconsistent with all known nominal and failure modes is consistent with the unknown failure mode. In this way, MIR can infer that a component has failed, though the failure was not foreseen or was simply left unmodeled because no recovery is possible.

By modeling only to the level of detail required to make relevant distinctions in diagnosis (distinctions that prescribe different recoveries or different operation of the system) we can describe a system with qualitative "common-sense" models which are compact and quite easily written. Consider the stylized model fragment in Table 5 which describes some of the possible modes of a remote terminal.

Table 5. MIR Model Fragment for Remote Terminal

```
device remote-terminal
     power_input = rt_switch->power_output
     command_input = bus_controller->command_output
mode nominal:
 if ( power_input == OFF)  comm_status = NO_COMMUNCATION
 if ( power_input == ON)   comm_status = COMMUNCATION
mode resettable-failure:
 probability = LIKELY
 comm_status = NO_COMMUNCATION
        if (command_input == RESET) next mode = nominal
mode powercyclable-failure:
 probability = LESS-LIKELY
 comm_status = NO_COMMUNCATION
        if (power_input == OFF) next mode = nominal
mode unknown:
 probability = UNLIKELY
/* Note there is no model, so any unmodeled behavior is consistent */
```

This single model describes how a remote terminal's outputs behave nominally and during failure, what connections to other devices influence its behavior, and the expected effect of recovery actions such as RESET if the device is in the mode under consideration. If a remote terminal is not communicating, MIR will consider that it may no longer be nominal or it may not be receiving power input. When investigating the latter, MIR will generate a similar set of explanations for why a switch might fail to provide power given its model and connections. Additional technical details about Livingstone can be found in [13].

7. INTEGRATING RAX INTO THE FLIGHT SYSTEM

Integrating RAX with flight software is challenging because RAX represents a significant departure from traditional flight software. The differences are not only technical as described previously, but also practical and cultural. From the view of flight software these differences may manifest themselves in a number of ways—from uneasiness within the flight software developers to an actual increased risk in the flight software product. Fortunately, none of these differences nor their impacts are inherent limitations to RAX technology and thus, with sensitivity to the issues, RAX is successful as a high-level flight software control architecture.

Perhaps the single largest practical difference that RAX presents arises from the fact the RAX is implemented in Common Lisp whereas previous missions, and also the realtime software with which RAX interacts, use lower-level languages like C. Many issues arise some of which are fact others of which are myth; however, the most significant issue is that interfaces between RAX and FSW might need to be specified and shared in either or both of two languages.

The success of RAX required that these issues be addressed in a way that would allow traditional flight projects to be comfortable with RAX technology and also to mitigate the risk introduced by the new technology. The result is the "RAX Manager" flight software component.

The RAX Manager presents the RAX technology to the flight software with a traditional flight software interface. Like hardware device managers, the implementation behind the interface is of no concern once the interface is correct, the functionality is in place and the required resources are allocated.

The RAX Manager serves several different functions over the life cycle of the project.

1) At design time, the RAX Manager specifies the interface agreements between RAX and the flight project. The interfaces includes all of the following:

- Telemetry and Logging
- Ground-based Command Dictionaries
- Computational Resources (CPU Fraction, Memory Requirements, etc.)
- FSW messaging (function calling) interface
- Flight Rules
- Fault Protection responses
- Timing within the Mission Plan.

2) At implementation time, the RAX Manager shields the existence of CommonLisp in the RAX implementation from the flight software by presenting a "C" interface externally. Producing that interface and performing any necessary conversions to the RAX implementation language are the full responsibility of the RAX developers. The process was simplified dramatically by a RAX developed software package known as CLASH ("C and Lisp Abstract Syntax Harmony"). CLASH defines a language for use in declaring a message passing interfaces and provides a preprocessor program (i.e. a compiler) to translate the declared interfaces to "C" header files, "C"' code files, and Lisp code. CLASH also runs inside RAX and hides all aspects of the inter-module communication issues. Thus, there is one uniform interface for internal message passing among RAX components, external message passing between RAX and C modules, and even telemetry packet encoding. Simple compile-time declarations specify the interface and the location (internal or external) of the code implementing the corresponding interface.

3) At FSW testing time, the RAX Manager decouples RAX from the flight software and thus allows the launch-ready software to be tested in anticipation of the launch date and the RAX software to be tested in anticipation of the (later) experiment start date. The RAX testing can thus proceed after the launch much as many ground-generated traditional sequences are validated post-launch. The RAX Manager however, as a tiny subset of the RAX code, can be tested relatively early, on the flight software schedule.

4) At runtime, the RAX Manager mediates the message passing between RAX and flight software. There are two aspects to this. This first is that the RAX manager must both initiate and terminate the RAX experiment: the initiation happens as commanded from the ground; the termination as a result of either a ground command or an unanticipated fault having found its way into the non-RAX fault-protection subsystem. The second aspect is that the RAX Manager must discard any messages destined for RAX during those times when the RAX is not operational. For DS1, RAX is a relatively short-lived technology demonstration experiment, so the dominant runtime activity of the RAX Manager will be to simply discard any incoming messages. Of course, for the time between initiation and termination the RAX Manager passes most messages between RAX and flight software.

Through these four functions, the RAX Manager spans the entire flight project lifecycle and in so doing allows the RAX to address and mitigate the unique risks that arise in each phase.

8. TESTING RAX

Our approach to testing and validating the RAX not only exploits standard software testing practice, but also goes beyond it in a number of key areas. The foundation of a reliable, high quality system is laid with the design and specification of the interfaces between the different subsystems. To this end, we have formalized all RAX interfaces, both between RAX and the rest of the flight software and between the components of RAX, using CLASH. The use of CLASH has essentially eliminated a whole class of essentially syntactic errors such as discrepancies in the index used to identify a switch in an array, out of range values, and inconsistent interpretations of interface structures. Formalizing these interfaces has allowed us to focus our testing effort on finding and eliminating more subtle semantic errors.

RAX *System-level Testing*

Our principal approach to testing the RAX at the system level was the scenario-based testing of requirements. Testing of individual RAX modules used both scenario-based testing methods and a variety of other methods discussed later in this section. We started scenario-based testing by identifying the set of system-level requirements to be met by the RAX. We then designed a set of test scenarios, ensuring that each requirement is adequately tested by one or more of these scenarios. Scenario design started with the development of the 12 hour and 6 day scenarios to be demonstrated in flight. These scenarios include nominal operation, planning and executing back-to-back plans, and a variety of failure scenarios. Additional scenarios were developed as variations on this basic set of scenarios. Variations were generated both for nominal execution (e.g., varying the number of OpNav image goals per window, varying the available power from the solar arrays, and varying the slew times for turns) and for failures (e.g., varying the location, time, and number of failures).

An important aspect of the above approach is to have people intimately familiar with spacecraft and mission develop the scenario variations. This ensures that the different scenarios capture all likely variations in the nominal scenarios, and all credible failures. Furthermore, such people can identify situations that are likely to be challenging for the RAX, e.g., time or resource limited situations, critical sequences requiring precise timing, and failures that are hard to diagnose and recover from. Mission and systems engineers are in the best position to develop scenario variations. However, in order to avoid excessively taxing the systems engineer's time, our approach has been to have knowledgeable members of the RAX team develop the scenario variations, and have these variations be reviewed by DS-1 systems engineers. The limited scope of the RAX makes this approach feasible.

This basic approach to testing generalizes naturally to system-level testing of a Remote Agent being deployed for a complete mission. In particular, each mission usually consists of a number of different phases characterized by nominal scenarios. For example, the phases of the DS-1 mission include launch, ballistic cruise, cruise under ion thrusting, asteroid and comet flybys, and various validation experiments. Nominal scenarios for each of these phases can be developed and tested. Systems engineers can then use these nominal scenarios to develop scenario variations, including failure scenarios, to build confidence that the Remote Agent can effectively carry out all phases of the mission under a variety of different situations. The focus provided by the nominal scenario of each phase helps keep the system-level testing of the Remote Agent manageable.

Scenario-based testing of RAX is augmented with a variety of tools and processes to ensure effective testing. Specifically, we have developed a set of flight software and hardware simulators that support effective RAX testing prior to integration with the rest of the flight software. We have also developed tools for simulated time "warping", which allows the RAX and its associated simulators to skip over periods of time in which the RAX is idle. This allows us to test scenarios lasting for days or weeks of simulated time in a few minutes or hours of real time. Whenever possible, we have attempted to convert all tests into automatic regression tests requiring no manual intervention. This allows us to automatically run a battery of tests overnight, to ensure that every major release of the RAX passes all regression tests. Finally, we have installed a formal bug tracking system using the GNU GNATS system and a process for its use. Whenever a code error is discovered, it is logged in GNATS. Once the error is corrected, a regression test is created that fails before the code is corrected but passes with the corrected code. This regression test is then added to the set of regression tests.

In addition to the system-level testing described above, we also do extensive module feature tests on each of the RAX modules. These are described below.

Planner/scheduler module feature testing

The main requirements on the planner is that it produce a valid plan for all valid plan requests from the Executive and all legal behaviors of the plan experts, and successfully update the mission profile in response to an profile update request. The latter requirement can be tested directly with automated scenario-based testing.

The first requirement is somewhat harder to test. For any partial plan provided to the planner and any set of plan expert behaviors, the planner must either produce a valid plan before its computational resource bounds are exceeded (times out), or report that no plan can be generated within those bounds. For a plan to be valid, it must be consistent with the plan model. This requirement is tested by extensive scenario-based testing. The plans generated in each scenario are tested for correctness against the plan model by an automated constraint checker, and manual spot checking of plans. The constraint checker converts the plan model into a set of logical constraints. Each plan is checked to ensure that all of the constraints are met. The constraint checker also performs a coverage analysis to ensure that every rule in the plan model has been exercised by an adequate number of plans. Manual spot checking is done by displaying the plan as a modified GANTT chart with a plan viewing tool.

Even if a plan is valid with respect to the plan model, the plan model itself may be incorrect. The model may not express the knowledge that the model developer intended, or the developer may not have acquired the correct knowledge from the experts. The plan model must be verified with respect to the knowledge of appropriate experts. This is done by encoding the plan model into English specifications and confirming them with human experts. Another source of expert knowledge are the flight rules. These are English rules that state what actions can and cannot be performed on the spacecraft. For example, "never fire the IPS engines while taking optical navigation images". These rules can be converted into logical expressions and added to the set of constraints checked by the constraint checker. As a final test, a small representative set of plans are run through the executive to ensure that they execute correctly and that the spacecraft exhibits correct behavior.

Executive module feature testing

The modular, multi-level structure of the Executive (see Section 6) enables the Executive sub-modules to be tested independently and permits the Executive to be adapted to new missions with a minimal amount of change, primarily at the external devices level. Given the limited scope of the RAX, testing the higher levels of the Executive (i.e., the external device level and the top level control) is relatively straightforward. This gives us an opportunity to effectively test the lower levels of the Executive, providing a well-tested foundation for future missions. If it were necessary to redevelop and test the entire Executive for each mission, the high development cost could very well eliminate its selection on future missions.

As previously discussed, we use automatic regression tests whenever possible to test the Executive. Once such a test is started, manual intervention is not required and the test returns a pass or fail value. To facilitate this process, a simulator is used that was designed to check system-level properties and constraints while the Executive is running. For example, one constraint is that the MICAS camera is not to take a picture while the spacecraft is turning. Given this constraint, the simulator generates an error that will cause a test to fail if the simulated spacecraft is turning when it receives a command from the Executive to take a picture.

Unfortunately, not all testing can be done automatically. Determining if the Executive really did what it was supposed to do in certain situations often requires an expert to review the log generated by the Executive. This can be time consuming and errors may be overlooked. In order to address this problem, a visualization tool for validating Executive plan execution, called Planview, was developed at CMU by Simmons and Whelan [14]. Planview provides the user an overall view of all the executing timelines, highlights execution flaws, and allows the user to zoom in on an individual token showing its values and constraints.

Finally, a formal analysis approach is used to check if the Executive code violates design specifications [15]. In this approach, we create a formal model that characterizes the abstract behavior of critical Executive constructs (for example, those dealing with resource management). We also formalize design requirements that should be enforced whenever the constructs are used (for example, aborted activities must always give up any resources that were allocated to them). Then we run this abstract model through a formal model checker, which either proves that the formal model satisfies the design requirements or generates an example scenario where the requirement would be violated. Using this approach, errors in the Executive code were discovered that would have been very difficult to discover using the test methods described above. A major drawback of this approach is that it is time-consuming and has only been applied to a small part of the Executive. Decreasing the time and expertise required to perform this analysis is an ongoing research area.

Diagnosis and Repair module feature testing

MIR has four major categories of testable requirements: it must provide command confirmation to Exec, it must diagnose a set of failures, it must provide recoveries for those failures, and it must meet certain performance requirements. The majority of MIR testing is scenario based testing on a combination of simulators and real hardware. A scenario consisting of a sequence of spacecraft commands and resulting monitor values (real or simulated) is processed by MIR. At each point in the scenario, MIR's model of the spacecraft's state must agree with the spacecraft state predicted by the scenario commands. During the scenario, a failure is injected into the spacecraft simulation or hardware testbed, causing a set of monitor values to be reported to MIR. MIR's diagnosis and recovery are then checked against the injected failure and performance metrics are taken.

MIR testing scenarios derive from three sources. The first is devious human testers. We have developed tools to allow a user to easily write a scenario consisting of RAX command sequences, failure injections and, when not running on the hardware testbed, the expected monitor values. Human analysis of MIR's weaknesses provides the most stressful but most expensive test scenarios for the system. The second source is brute force automatic scenario generation. The RAX MIR models are small enough that many classes of tests can be performed exhaustively given a set of reasonable limiting assumptions and a fast spacecraft simulator. For example, given the simplicity of MIR's models, each failure can be injected in each combination of modes the model can achieve and automatically checked for correct diagnosis and recovery. The third source is informed automatic scenario generation. MIR models the spacecraft by modeling each component as a finite state automaton. A large amount of work has been done in the verification community in verifying that a finite automata (here the MIR models) correctly models a physical or software system (here the spacecraft simulator or hardware). In addition, a large amount of work has been done in the model-based diagnostics community in deriving tests that systematically sensitize each subsystem of an assembled system (here the simulator or hardware) and determine that diverge from their models. We are drawing on this work to build automatic test generators which will provide near-minimal length tests which will determine if a MIR model agrees with the hardware or simulator it models.

9. FUTURE WORK

A number of desirable Remote Agent features are planned for future Remote Agents that will not be part of the DS1 RA. These enhancements will further increase mission robustness, refine diagnostic capabilities, and simplify the process of representing and integrating knowledge throughout the software.

In our discussion of mission robustness, we discussed flexible planning and recovery capabilities. These capabilities will not help in cases where some preventative or preparatory action needed to be taken in the past to enable recoveries in the current situation. For example, if the primary engine breaks, the system may only be able to switch to the backup engine if it has been warmed up. Future Remote Agents will have the capability to anticipate such possible failures, or even opportunities, and to then build plans that provide the necessary resources so the system is prepared for many possible futures. A related capability in this vein is for the executive to understand the priorities in the plan, so that it can abandon individual tasks or threads of activity without failing the entire plan. This will enable high-priority activities to be completed even if low-priority activities fail.

In our discussion of diagnosis, we pointed out that the MIR system makes new inferences every time an action is taken or a new observation is made. In the event of failures, it will generate recoveries that may improve the situation. However, sometimes these actions taken during normal execution or even recovery will not present the right information to isolate the fault to an optimal level of detail. Our future work will develop methods for active testing, in which the system will conduct tests whose sole purpose is to help it improve its understanding of the state of the spacecraft. Examples of this capability include turning the spacecraft to see if a gyro is measuring turn rates correctly, and turning selected devices on and off to detect shorts.

In terms of knowledge engineering, we discussed how the various reasoning engines in the RA use different representations of knowledge. In many ways this is a necessary and useful feature, as it allows the planner to reason at a more abstract level than the executive, and the diagnosis system to reason at a more detailed level. While heterogeneous representations have a number of benefits, they also raise some difficulties. Most significant of these are the possibility for models to diverge rather than converge, and the need to duplicate knowledge representation efforts. Ideally, we would like to head toward an increasingly unified representation of the spacecraft, but we intend to do so always generalizing from powerful models capable of handling the complexities of our real-world domain.

Many of these technology advances are currently targeted for future Deep Space Missions of the New Millennium Program. Deep Space Three is a three spacecraft separated optical interferometer and Deep Space Four is a Comet nucleus Sample Return mission. Both are slated for launches in the early years of the new millennium.

10. ACKNOWLEDGMENTS

The work in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration and at the National Aeronautics and Space Administration's Ames Research Center.

The authors acknowledge the invaluable contributions of Guy K. Man and Robert D. Rasmussen for their work in defining the vision for spacecraft autonomy that evolved into this effort.

The Authors would also like to acknowledge the contributions of Guy K. Man, Richard Doyle, and Kane Casani of JPL, Gregg Swietek, Keith Swanson, and Ken Ford of NASA Ames, and Mel Montemerlo of NASA Headquarters for their leadership is seeing the necessity and possibility for advances in the area of spacecraft autonomy and their insight in recommending and supporting the approach that we took.

In addition to the authors, the early defining work on the Deep Space One Remote Agent was accomplished through the efforts of Erann Gat and Steve Chien of JPL and Michael Wagner, Ron Keesing, Chris Plaunt, Scott Sawyer, and Hans Thomas of NASA Ames.

APPENDIX A

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	Request and get next plan from planner.
		script_next_horizon	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.

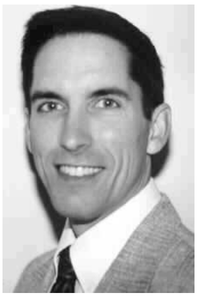
Additional tokens not listed above are used by the Planner as "placeholders" in the timelines. These placeholder tokens do not require EXEC to perform any activity.

Appendix B

Detailed Planner model for SEP

Timelines	*Tokens*	*Comments*
SEP_Schedule [Goal timeline]	Idle_Segment Thrust_Segment	SEP_Schedule is populated by NAV planning expert. Thrust_Segment defines time period with heading and thrust level. Several sequential segments constitute a schedule. Idle_Segments needed to pad the timeline to precisely position the thrust segments.
SEP_Thrust_Timer [Goal timeline]	Thrust_Timer_Idle Max_Thrust_Time	Max_Thrust_Time is returned by the NAV planning expert. It specifies the total burn duration to be achieved in the current planning horizon.
SEP_Time_Accum [Internal timeline]	Accumulated_Thrust_Time	Tracks the amount of time in the plan during which SEP is scheduled to thrust. Time accumulation occurs only during SEP_thrust tokens (see below).
SEP [Executable timeline]	SEP_Standby	SEP is ready but power to the grid is turned off. Tracks the amount of time since SEP was thrusting. Greater the time since last thrusted, longer the duration of the SEP_Starting_Up token. Follows SEP_Shutting_Down. Followed by SEP_Starting_Up. Schedule appropriate power consumption retrieved from on-board power table.
	SEP_Starting_Up	Prepares the Xenon tanks to allow thrusting. Duration of this token is dependent on when SEP was last thrusting and on previous thrust level. Requires attitude of spacecraft to be kept constant on requested thrust heading. Follows SEP_Standby. Followed by SEP_Thrusting. Schedules power consumption retrieved from on-board power table.
	SEP_Thrusting/FIRST	SEP engine is actually thrusting immediately after having been started up. Must be temporally contained in a Thrust_Segment token (see above) from which it receives requested attitude and thrust level. Attitude must be kept constant to requested attitude throughout the token. Communicates its duration to an Accumulated_Thrust_Time token to track total accumulation. Follows SEP_Starting_up. Followed by either SEP_Thrusting/NEXT or SEP_Shutting_Down. Communicates requested heading to SEP_Starting_Up and SEP_Shutting_Down (if appropriate). Schedules power consumption retrieved from power table.
	SEP_Thrusting/NEXT	SEP engine is continuing to thrust (without having shut down) after change of attitude. Follows SEP_Thrusting/NEXT. Followed by SEP_Thrusting/NEXT or SEP_Shutting_Down. A short duration turn in TVC mode is requested to change attitude at the very beginning of the token. Communicate requested heading to SEP_Shutting_Down if appropriate. Other constraints identical to those of SEP_Thrusting/FIRST. Schedules power consumption retrieved from power table.
	SEP_Shutting_Down	Turns off SEP. Requires spacecraft to be kept constant on requested thrust heading. Schedules power consumption from power table.

BIOGRAPHIES



Dr. Douglas E. Bernard *received his B.S. in Mechanical Engineering and Mathematics from the University of Vermont, his M.S. in Mechanical Engineering from MIT and his Ph.D. in Aeronautics and Astronautics from Stanford University. He has participated in dynamics analysis and attitude control system design for several spacecraft at JPL and Hughes Aircraft, including Attitude and Articulation Control Subsystem systems engineering lead for the Cassini mission to Saturn. Currently, Dr. Bernard is group supervisor for the flight system engineering group at JPL and team lead for Remote Agent autonomy technology development for the New Millennium Program.*

Dr. Gregory A. Dorais *is a Computer Scientist at the Computational Sciences Division of the NASA Ames Research Center. He received his B.S. in Management Information Systems from Oakland University, and received both his M.S. and Ph.D. in Computer Science from the University of Michigan. He has performed autonomous rover research at JPL and remote sensing research at General Motors Research.. His research interests include autonomous systems and machine learning.*

Chuck Fry *is a member of the technical staff of Caelum Research Corporation, under contract to NASA Ames Research Center's Information Sciences Division. Chuck majored in Information and Computer Science at the University of California at Irvine. His research interests include computer architecture, functional languages, and the social implications of the Internet.*

Dr. Ed Gamble *is a member of the Advanced Multimission Software Technology group at JPL. He received his bachelor's and master's in Electrical Engineering from UCLA. His doctorate was awarded in Electrical Engineering with a specialty in Artificial Intelligence from MIT. His interests have ranged from laser scattering in fusion plasmas and in critical phenomenon, to computational vision and integration of sensory information, and to programming languages and real-time systems. He is currently interested in spacecraft software architectures for reuse and autonomy.*

Bob Kanefsky *is a Senior Knowledge Engineer in the Computational Sciences Division at NASA Ames Research Center. He received a B.A. in Formal Systems at Stanford University. He designed and wrote several of the software tools used by the RA and DS1 flight software teams, particularly the abstractions for message-passing and Lisp telemetry. He recently supported the Mars Pathfinder science operations team, delivering an experiment super-resolution image processing algorithm he had helped develop, and writing a web-based sequence generator for preparing the necessary commands for the lander camera.*

James Kurien *is a Computer Scientist in the Computational Sciences Division at NASA Ames Research Center and is currently a doctoral candidate at Brown University. He holds Masters degrees from Brown University and Rensselaer. He has contributed to robot navigation research at Brown and model-based software environments at IBM Research.*

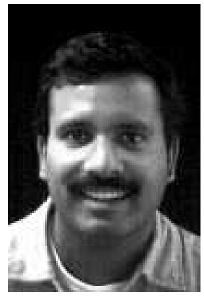
Bill Millar *is a Computer Scientist in the Computational Sciences Division at the NASA Ames Research Center, and is currently the system integration lead for the Remote Agent Experiment on Deep Space One. He holds a BMATH degree in Pure Mathematics from the University of Waterloo, Canada. His recent work and interests include automated problem decomposition, knowledge representation, model-based diagnosis, and visual modeling.*

QuickTime™ and a
Photo - JPEG decompresso
are needed to see this pictu

Dr. Nicola Muscettola *is a Senior Computer Scientist at the Computational Sciences Division of the NASA Ames Research Center. He received his Diploma di Laurea in Electrical and Control Engineering and his Ph.D. in Computer Science from the Politectnico di Milano, Italy. He is the principal designer of the HSTS planning framework and is the lead of the on-board planner team for the Deep Space 1 Remote Agent Experiment. His research interests include planning, scheduling, temporal reasoning, constraint propagation, action representations and knowledge compilation.*

Dr. Pandurang Nayak *is a Senior Computer Scientist at the Computational Sciences Division of the NASA Ames Research Center. He received a B.Tech. in Computer Science and Engineering from the Indian Institute of Technology, Bombay, and a Ph.D. in Computer Science from Stanford University. His Ph.D. dissertation, entitled "Automated Modeling of Physical Systems", was an ACM Distinguished Thesis. He is currently an Associate Editor of the Journal of Artificial Intelligence Research (JAIR), and his research interests include model-based autonomous systems, abstractions and approximations in knowledge representation and reasoning, diagnosis and recovery, and qualitative and causal reasoning.*

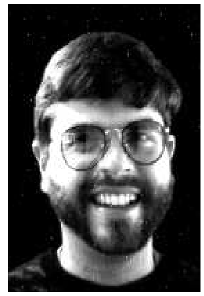
Dr. Barney Pell *is a Senior Computer Scientist in the Computational Sciences Division at NASA Ames Research Center. He is one of the architects of the Remote Agent for New Millennium's Deep Space One (DS-1) mission, and leads a team developing the Smart Executive component of the DS-1 Remote Agent. Dr. Pell received a B.S. degree with distinction in Symbolic Systems at Stanford University. He received a Ph.D. in computer science at Cambridge University, England, where he studied as a Marshall Scholar. His current research interests include spacecraft autonomy, integrated agent architecture, reactive execution systems, collaborative software development, and strategic reasoning. Pell was guest editor for Computational Intelligence Journal in 1996 and has given tutorials on autonomous agents, space robotics, and game-playing.*



Kanna Rajan is a member of the Planning and Scheduling group at NASA Ames Research Center. He holds a bachelors from the Birla Institute of Tech. and Science, Pilani, India and a Masters from the University of Texas, Arlington both in Computer Science. Prior to joining NASA Ames he was in the doctoral program in Computer Science at the Courant Institute of Mathematical Sciences at New York Univ. His primary research interests are in Planning, Robotics and Knowledge Representation.*

Dr. Nicolas Rouquette *is a Senior Computer Scientist in the Monitoring and Diagnosis Technology group, one of the Artificial Intelligence Groups at JPL. Nicolas received a Ph.D. from the Computer Science Department at the University of Southern California in 1995 under the advisement of Shankar Rajamoney, Paul Rosenbloom, David D'Argenio and, but not least, Michael Waterman. Nicolas received an M.S. in Computer Science from USC in 1989 and he is an electrical engineer graduate from ESIEE in Paris in 1989. Nicolas is responsible for the fault protection software for the Deep Space One spacecraft and its associated software monitors for the baseline fault protection as well as for the beacon and remote agent experiments. Nicolas is actively pursuing work and research in Monitoring, Model-Based Reasoning, Systems Engineering and Graph Theory.*

Dr. Ben Smith *is a member of the Artificial Intelligence group at JPL, and Deputy Lead of the JPL element of the DS1 planning team. He holds a Ph.D. in computer science from the University of Southern California. His research interests include intelligent agents, machine learning, and planning.*



Dr. Brian C. Williams *is Technical Group Supervisor of the Intelligent Autonomous Systems Group at the NASA Ames Research Center, and co-lead of the model-based autonomous systems project. He received his bachelor's in Electrical Engineering at MIT, continuing on to receive a Masters and Ph.D. in Computer Science. While at MIT he developed one of the earliest qualitative simulation systems, TQA, a hybrid qualitative/quantitative symbolic algebra system, MINIMA, and a system IBIS for synthesizing innovative controller designs. Williams was at Xerox PARC from 1989 to 1994, where he is best known for his work on the GDE and Sherlock model-based diagnosis systems. Williams received AAAI best paper awards in 1988 and 1997 for his work on qualitative symbolic algebra and incremental truth maintenance. He was guest editor for Artificial Intelligence in 1992, Chair of the AAAI Tutorial Forum in 1996 and 1997, and is currently on the editorial board of the Journal of Artificial Intelligence Research.*

REFERENCES

[1] Barney Pell, Scott Sawyer, Douglas E. Bernard, Nicola Muscettola, and Ben Smith, "Mission Operations with an Autonomous Agent," In Proc. of IEEE Aeronautics (AERO-98), Aspen, CO, IEEE Press, 1998 (To appear).

[2] B. Pell, D. E. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. Nayak, M. D. Wagner, and B. C. Williams, "A Remote Agent Prototype for Spacecraft Autonomy," SPIE Proceedings Volume 2810, Denver, CO, 1996.

[3] N. Muscettola, "HSTS: Integrating planning and scheduling," in Fox, M., and Zweben, M., eds, *Intelligent Scheduling*, Morgan Kaufman,

[4] N. Muscettola, B. Smith, S. Chien, C. Fry, G. Rabideau, K. Rajan, D. Yan, "On-board Planning for Autonomous Spacecraft," in Proceedings of the fourth International Symposium on Artificial Intelligence, Robotics and Automation for Space (i-SAIRAS 97), July 1997.

[5] Erann Gat, "ESL: A language for supporting robust plan execution in embedded autonomous agents," Proceedings

of the AAAI Fall Symposium on Plan Execution, AAAI Press, 1996.

[6] Barney Pell, Ed Gamble, Erann Gat, Ron Keesing, Jim Kurien, Bill Millar, P. Pandurang Nayak, Christian Plaunt, and Brian Williams, “A hybrid procedural/deductive executive for autonomous spacecraft,” In P. Pandurang Nayak and B. C. Williams, editors, Procs. of the AAAI Fall Symposium on Model-Directed Autonomous Systems, AAAI Press, 1997.

[7] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith, “Robust Periodic Planning and Execution for Autonomous Spacecraft,” In Procs. of IJCAI-97, 1997.

[8] Erann Gat and Barney Pell, “Abstract Resource Management in an Unconstrained Plan Execution System,” in Proc. of IEEE Aeronautics (AERO-98), Aspen, CO, IEEE Press, 1998 (To appear).

[9] J. de Kleer and B. C. Williams, “Diagnosing Multiple Faults," Artificial Intelligence, Vol 32, Number 1, 1987.

[10] J. de Kleer and B. C. Williams, “Diagnosis With Behavioral Modes," Proceedings of IJCAI-89, 1989.

[11] D. S. Weld and J. de Kleer, *Readings in Qualitative Reasoning About Physical Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[12] J. de Kleer and B. C. Williams, *Artificial Intelligence*, Volume 51, Elsevier, 1991.

[13] B. C. Williams and P. Nayak, “A Model-based Approach to Reactive Self-Configuring Systems,” Proceedings of AAAI-96, 1996.

[14] Reid Simmons and Greg Whelan “Visualization Tools for Validating Software of Autonomous Spacecraft,” In Proc. of the Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS), Tokyo, Japan, 1997.

[15] Klaus Havelund, Michael Lowry, and John Penix, “Formal analysis of a spacecraft controller using SPIN,” Technical report, NASA Ames Research Center, 1997.