Appears in *Proceedings of the 1999 IEEE Aerospace Conference*, Aspen CO.

# Validation and Verification of the Remote Agent for Spacecraft Autonomy

Ben Smith[1] William Millar[2] Julia Dunphy[1] [1]Yu-Wen Tung[1] Pandu Nayak[3] Ed Gamble[1] Micah Clark[1]

[1]Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
(818) 393-5371
benjamin.d.smith@jpl.nasa.gov

[2,3]NASA Ames Research Center,
MS 269-2, Moffett Field, CA 94035
[2]Caelum Research Corp. @ Ames
[3]RIACS @ Ames
(650) 604-0263
millar@ptolemy.arc.nasa.gov

*Abstract*—The six-day Remote Agent Experiment (RAX) on the Deep Space 1 mission will be the first time that an artificially intelligent agent will control a NASA spacecraft. Successful completion of this experiment will open the way for AI-based autonomy technology on future missions. An important validation objective for RAX is implementation of a credible validation and verification strategy for RAX that also "scales up" to missions that make full use of spacecraft autonomy.

Autonomous flight software presents novel and difficult testing challenges that traditional flight software (FSW) does not face. Since autonomous software must respond robustly in an immense number of situations, the all-paths testing approaches used for traditional FSW is not feasible. Instead, we advocate a combination of scenario-based testing and model-based validation. This paper describes the testing challenges faced by autonomous spacecraft commanding software, discusses the testing strategies and model-validation methods that we found effective for RAX, and argues that these methods will "scale up" to missions that make full use of spacecraft autonomy.

Among the key challenges for validating autonomous systems such as the RAX are ensuring adequate coverage for scenario-based tests, developing methods for specifying the expected behavior, and developing automated tools for verifying the observed behavior against those specifications. Another challenge, also faced by traditional FSW, is the scarcity of high-fidelity test-beds. The test plan must be designed to take advantage of lower-fidelity test-beds without compromising test effectiveness.

TABLE OF CONTENTS



## 1. INTRODUCTION

The Deep Space One (DS1) spacecraft, which launched October 24, 1998, is a technology validation mission. Unlike previous missions its primary objective is not to gather science observations, but to flight validate several new technologies. Successful validation of these technologies will remove a major obstacle to their use on more risk-averse science missions.

One of the new technologies is the Remote Agent (RA), an artificial-intelligence based architecture capable of autonomously commanding the spacecraft. The Remote Agent Experiment (RAX) will demonstrate autonomous operations of the DS1 spacecraft by the RA for a period of six days in the Spring of 1999. Successful flight validation of the RA will open the door to the use of autonomous spacecraft commanding technology on future science missions. A excellent description of the experiment design and validation objectives can be found in [1].

One of the Remote Agent's validation objectives is demonstrating a credible verification and validation (V&V) approach that will "scale up" from the experiment scope to missions that make full use of autonomy. The V&V approach for traditional flight software is not feasible for autonomous flight software.

Traditional FSW V&V is able to achieve very high confidence by testing each sequence (a time-ordered list of commands) before uplinking and executing it. Autonomous software is commanded by high-level goals rather than an explicit sequence. The software decides how to command the spacecraft in order to achieve those goals. The exact command sequence is difficult to predict in advance because the RA's decisions are conditional on onboard events. It is this ability to "close-loops" on-board instead of through the ground operations team that makes autonomy technology so valuable, but it also means that the traditional sequence validation approach is no longer feasible. A different V&V approach is needed. Autonomous software will not be accepted on risk-averse science missions without a credible V&V strategy.

### *Traditional FSW Testing*

For testing purposes, traditional FSW can be thought of as having two components: the onboard software and a *sequence*. The onboard software consists of low-level procedures for commanding aspects of the spacecraft hardware. The sequence is a time-ordered list of commands, where each command corresponds to one of the software procedures.

The commands are tested independently. This provides a moderate level of confidence in each of the commands, but it does not address interactions among commands. These interactions can be subtle and are often unsuspected. They may depend on the exact sequence and timing of prior commands, subtleties of the spacecraft state, etc.

Potential interactions and other subtle problems are detected by sequence testing. Sequences have a single execution path, or at most a small handful, which allows very detailed testing and interaction analysis to be focused on just those paths.[1] This is a very powerful approach, but it is only feasible when there are a small handful of execution paths and those paths are known in advance.

*Testing Autonomous Software*

Traditional FSW testing approaches are not suitable for autonomous software. In autonomous operations, the spacecraft is not given a sequence, but a set of high-level goals which the Remote Agent expands into lower-level commands. If problems arise during execution, the software automatically takes corrective action and selects alternate methods for achieving the goals. There are millions or billions of possible execution paths. It is impossible to identify the handful that will be needed and exhaustively test only those. The testing approach must provide high confidence that the software will perform properly for *all* of those paths.

The approach we have taken for RAX is scenario-based verification augmented with model-based verification and validation. The universe of possible inputs (goals, spacecraft state, device responses, timing, etc.) is partitioned into a manageable number of *scenarios*. The RA is exercised on each scenario, and its behavior verified against the specifications.

*Testing Challenges*

Scenario-based testing is hardly a novel approach, but there are a number of issues that must be addressed for this approach to provide the high confidence levels required by science missions.

First, the effectiveness of scenario-based testing depends on the coverage of the scenarios. Success on the tested scenarios must imply success on the untested scenarios with high confidence. The universe of possible input scenarios is very large, but the number of tested scenarios must be manageable. RAX has taken a parameter-based approach to characterizing the input space and measuring coverage. The RA's models were used to identify non-interacting or low-interaction regions of the parameter space, thereby reducing the number of parameter combinations that must be tested. Orthogonal arrays were used to design minimal-sized test suites with comprehensive coverage of the necessary parameter combinations.

The second challenge is that high-fidelity test beds are expensive and correspondingly scarce, but scenario-based testing involves many scenarios—more than can be run on the scarce high-fidelity test beds. The test suite must therefore allocate most of the scenarios on lower-fidelity (and more available) test-beds without compromising test effectiveness. RAX can be tested on lower-fidelity platforms due to the abstraction layers presented by the RA and FSW. The RA reasons about and acts upon an abstraction of the spacecraft presented by the FSW, which allows most of the RA behavior to be tested against that abstraction on lower-fidelity test-beds. The RA is composed of mission- and platform-independent reasoning engines, and mission-dependent models. Almost all of the engine requirements can be verified on lower fidelity platforms. The high-fidelity platforms test the remaining engine and RA requirements, and ensure that the RA as a whole operates correctly.

Third, since the RA interacts in a close-loop fashion with the spacecraft, the test-scenarios must be able to specify how the spacecraft will react in response to the RA. More specifically, testers must be able to inject events relative to the RA's actions and internal reasoning process. We designed a test controller that observes the RA's telemetry stream to determine what the RA is doing, and automatically injects events based on those observations. We believe this is a new approach in the spacecraft testing community, though similar approaches have been used for non-spacecraft applications.

The fourth challenge is automated verification. Determining whether the software responded correctly to a given scenario is difficult to automate, and laborious to perform manually. Developing formalisms for specifying the correct behavior and developing automated analysis tools that work with those specifications is therefore an important part of a credible testing strategy. The behavior of the RA is complex, and correctness of that behavior depends strongly on environmental details (spacecraft state, hardware response timing, etc). We designed several verification tools and methodologies for RAX, such as verifying abstractions of the behavior, and checking the behavior against safety and correctness constraints derived from DS1 requirements ("flight rules") and RAX requirements.

Finally, scenario-based testing provides relatively high confidence in the test artifact, but it can provide even higher confidence when combined with formal design-validation methods. For the Remote Agent Experiment, the scenario-based testing alone is more than adequate since the traditional FSW provides a number of "safety nets" that will abort the experiment if it behaves improperly. For risk-averse science missions, higher levels of confidence are

[1] The level of testing and analysis need not be the same for every sequence, and can depends on the importance of the sequence and a-priori confidence of the operations team in it. At the most detailed level, the sequence is executed on the flight test-bed in real-time, and the resulting telemetry and the sequence itself are analyzed by subsystem engineers and automated constraint checkers for negative interactions, etc.

needed. We have identified a number of these for the RA, and have provided proof-of-concept demonstrations in several cases, but these are not part of the RAX testing plan. We expect these model-based methods will be part of the testing story for science missions that make full use of spacecraft autonomy.

The remainder of this paper is organized as follows. Sections 2 and 3 describe the Remote Agent architecture and the experiment. Section 4 describes a number of test-case selection methods used for RAX, and discusses why these methods should scale up to missions where the universe of possible scenarios is much larger than it is for RAX. Section 4 discusses the automated verification tools and specification formalisms that we found effective. Section 5 describes the testbed allocation approach and lessons for larger-scale testing efforts. Automated testing and verification issues are covered in Section 6, and formal verification and validation methods in Section 7. We discuss the current status of RAX testing in Section 8 and conclude in Section 9.

## 2. Remote Agent Architecture

The Remote Agent [2] consists of three components: Executive (EXEC), Planner/Scheduler (PS), and Mode Identification and Reconfiguration (MIR). The planner is given a set of high-level goals from the ground operations team and from the onboard navigation system. PS generates a plan that achieves the goals while obeying safety constraints and resource constraints.

The fundamental execution units in the plan are tokens and timelines. Each activity in a plan is defined by a token, though not every token is an executable activity. Tokens also track spacecraft states and resources. A timeline is a sequence of tokens that specifies the evolution of that state variable over time. The plan has several parallel timelines. The plan specifies start and end time windows for each token, and temporal constraints among the tokens (before, after, contains, etc).

There are 18 timelines and 37 tokens defined in RAX. The executive tracks only 8 of these timelines and 22 tokens. The remainder are used by the planner for representing goals, untracked resources, and abstract activities that simplify the planning process but are not directly executable.

The executive executes the plan in a robust manner by issuing appropriate commands to the flight software. These are the same commands that would be used in traditional sequences. The executive guarantees that each activity in the plan will be executed within the temporal constraints specified in the plan. The temporal constraints are needed to synchronize the start and end times of activities on parallel timelines. For example, consider activities A and B with a start-time window of [5,10] and a temporal constraint that B must start two seconds after A. If A starts at time 6, then B must start at time 8, even though the start window for B is [5,10]. It is insufficient to guarantee only that A and B start within their time windows. See Figure 1.

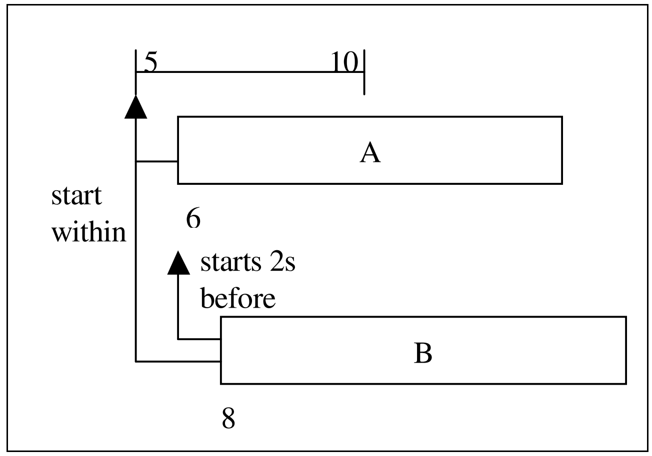


**Figure 1: Temporal Relations**

During execution, MIR observes the commands and the state of the spacecraft. If the commanded state differs from the observed state, MIR uses its model of the device to infer the most likely failure mode. It then suggests a repair activity to the executive, which carries it out. If the failure cannot be resolved within the constraints imposed by the plan, then the executive aborts the plan and requests a new one. The planner then generates a plan that achieves the remaining goals from the current state of the spacecraft.

The Remote Agent architecture is shown in Figure 2.

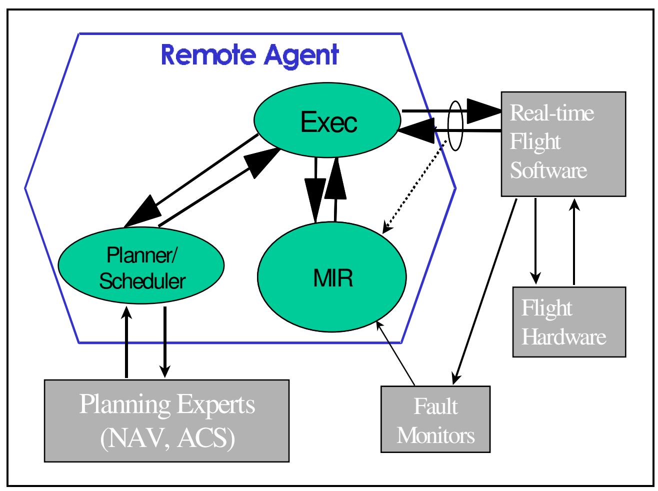


**Figure 2: Remote Agent Architecture**

## 3. Remote Agent Experiment

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 valve stuck closed

*Mission Scenario*

The Remote Agent experiment is executed in two phases, a 12 hour Phase One followed a few days 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. 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 exercise 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.

## 4. TEST CASE SELECTION AND COVERAGE

The effectiveness of scenario-based testing depends largely on how well the scenarios cover the requirements. Good coverage requires not only that the test suite exercise each requirement, but that the test-suite provide high confidence that if the requirement is satisfied on the test suite that the requirement will also be satisfied on all of the untested inputs.

In addition to providing good coverage, the test-suite must have a manageable number of tests. Finding the right balance between coverage and test-suite size can be difficult, and may involve trading risk (coverage) for manageability. The manageability of a test-suite depends on the availability of appropriate test-beds, the running time of the suite, and the analysis effort it entails.

The Remote Agent consists of three modules, the planner, the executive, and MIR. Each of the modules consist of a mission-independent engine and a mission-specific model. For RAX, the engines and models are tested as a unit by module-specific test suites. For future missions, the engines can be tested separately for increased confidence. Finally, a system test-suite exercises the RA as a whole.

The module test-suites were designed using a parameter-based approach. The universe of possible input scenarios is characterized by a multi-dimensional parameter space. A given assignment of values to parameters specifies a unique scenario. The test suite consists of a subset of the possible parameter values. The RAX test-suite uses three methods to achieve good coverage and manageability: abstracting the parameter space to focus on the relevant parameters and values, analyzing the RA models to identify independent regions of the parameter space and thereby reduce the number of parameter combinations that must be tested, and using orthogonal arrays to generate minimal-size test suites that cover those combinations.

The system tests provide confidence in the RA behavior as a whole. Since the system test-suite must be performed on scarce high-fidelity platforms, it can contain only a handful of scenarios. RAX is forced to trade risk (coverage) for feasibility. The system tests are therefore intended to verify the handful of requirements that could not be tested in the module test-suites on lower-fidelity platforms, and to exercise the module interactions. The system suite also serves as a "spot-check" of requirements verified in the module tests, to be sure that they are still satisfied in a high-fidelity system context. These issues are discussed further in Section 5.

The remainder of this section discusses the test-selection and coverage analysis approaches for the engines; the PS, MIR, and EXEC modules (engines plus models as a unit); and the system test-suite.

*Engine Verification*

Each component of the RA (EXEC, MIR, PS) consists of a domain-independent engine that reasons about and acts upon domain-dependent knowledge contained in a *model*. This separation is a key aspect of the RA design for testability.

Since the reasoning engines are domain-independent, they need not change from mission to mission. The reasoning and actions performed by the engines are also platform independent, with the exception of a handful of localizable performance and timing issues. This allows developers to provide one release of the engine software that can be used by several missions. Testers can validate the engine design and verify the platform-independent portions of the engines once. Only the platform-dependent requirements need to be verified for each mission. Since the testing cost is effectively amortized across several missions, the testers can provide correspondingly higher confidence in the engines.

A full-scale approach for verifying and validating the RA engines would be to identify key properties that the engines must enforce, perform a formal design analysis to ensure that the design does in fact enforce those properties, and then verify that the engine implementations satisfy the design requirements.

This process verifies that the engines enforce certain formal properties. For example, the EXEC engine ensures that the tokens (activities) in the plan will be dispatched and terminated at times consistent with the temporal constraints in the plan. The PS engine ensures that the plans it generates will obey all of the constraints encoded in the plan model. The MIR engine ensures that its diagnosis is the most probable one consistent with the model**.** These properties could be quite useful for validating the RA as a whole. Determining exactly what properties and formal validation methods would be most appropriate for this task is an area for future work.

Since RAX is the first user of these engines, and our testing resources are fairly limited, we could not invest the testing effort needed for this full-scale V&V approach. Instead, we treated each engine and its model as a single test artifact, and verified requirements on each of these artifacts with scenario-based testing. The following subsections describe how these test cases were selected As a proof-of-concept demonstration we used formal methods to validate selected properties of the EXEC as discussed in Section 7, but this was not part of the formal testing process.

Testing the engine and model as a single artifact is a valid approach, but it prevents amortizing the test cost across missions. In addition, this approach does not verify formal properties of the engine, and so these are not available for validating the RA as a whole. Future science missions can use either of these testing approaches, but we believe the amortized V&V approach discussed above is more powerful. The domain-independence of the engines makes this approach possible.

*Planner Test Case Selection*

The planner takes as input a requested plan start time, an initial spacecraft state, a set of goals (some of which come from the onboard navigator), and a set of constraints. It generates a plan that begins at the requested start time and achieves the goals from the initial state while obeying the constraints. The constraints are specified in the plan model and are largely fixed. However, a few of them can be modified by changing parameter settings, and fewer still are defined as external functions provided by onboard systems. (Specifically, the duration and legality of spacecraft *slews* (turns) are defined as functions provided by the attitude control subsystem.) The constraint parameters and the behavior of the ACS function must both be treated as inputs.

The planner's behavior is strictly a function of its inputs. Its behavior does not depend on the order or timing of events that occur while it is planning[2]. This makes it a good candidate for parameter-based testing (e.g., [4]). The input space is characterized by a multi-dimensional parameter space. Each assignment of values to parameters identifies a single point in the input space. The planner is tested on a carefully chosen subset of parameter values, and the resulting plan is checked against a list of plan correctness requirements as discussed in Section 6.

The test-suite must have good coverage, as defined by some metric, but not be too large to run and analyze. Based on our experience a suite of 200 to 300 plan-request scenarios is about the upper limit for a one-person testing effort, assuming an automated scenario-runner and adequate plan analysis tools.

A combination of approaches has proven effective for generating test-suites for the RAX planner. First, the parameter space is reduced by identifying equivalence classes of parameters and parameter values. The planner behavior is not expected to change qualitatively on inputs drawn from the same equivalence class, but is expected to change for inputs in different classes. Next, regions of high and low interaction in the reduced parameter space are identified by analyzing the planner model. Parameters from strongly interacting regions should be tested in combination, while fewer combinations must be tested from weakly interacting regions. Parameters from non-interacting regions can be tested independently. Finally, an orthogonal array-based algorithm generates a small (nearly minimal) size test-suite with comprehensive coverage of the identified parameter combinations.

*Parameter Space Construction*—The input space is characterized by a multi-dimensional parameter space such that there is a one-to-one correspondence between parameter settings and inputs. We term this the "true" parameter space. The true parameter space for the RAX planner is shown in Table 1. This space is infinite, and clearly infeasible for testing.

To produce a manageable number of test cases, it is first necessary to control the size of the parameter space. This was done by selecting parameters and parameter values that focus on aspects of the input space to which the planner is expected to be sensitive. We term this the "abstract" parameter space. Each parameter setting in this space specifies an equivalence class of inputs rather than a single input. The planner is expected to behave similarly on every input in a given class, but to have qualitatively different behavior for inputs drawn from different classes. Abstraction entails some risk, since there is no guarantee

[2] The onboard goals and ACS constraint functions are invoked during planning, but they always give the same results for the same inputs regardless of when they are called or in what order.

**Table 1**. True parameter space for RAX Planner

| Parameter | Input | Values | Distribution or Expected Value | Number of Values |
|---|---|---|---|---|
| Plan start time (relative to experiment start) | start time | 0-518,400 (6days in seconds) | uniform | 518,401 |
| initial state | initial state | all standby states, all legal end-states | uniform | infinite |
| NAV win duration | goal (ops) | integer > 0 | 3-5 hours. | maxint |
| NAV win goal start | goal (ops) | integer | 0 | 2 * maxint |
| NAV win period | goal (ops) | integer > 0 | 3 days | maxint |
| NAV win slack | goal (ops) | integer > 0 | 12 hours | maxint |
| Comm. window start time [6] | goal (ops) | 0-518,400 | 8 AM ea. day | 518,401 |
| Comm window duration [6] | goal (ops) | 0-518,400 | 8 hours | 518,401 |
| Image goals [20] | goal (nav) | see below | 8-10 | see below |
| Exposures/Image [20] | goal (nav) | small integer | 4 | ~10 |
| Exposure duration [20] | goal (nav) | 0-31 | 8-20 | 32 |
| Instrument Settings [20] | goal (nav) | 0-31 | 0-2 | 32 |
| Image target id [20] | goal (nav) | integer | uniform | maxint |
| IPS Maneuver Goals [12] | goal (nav) | see below | 1-2 | see below |
| Maneuver start [12] | goal (nav) | integer | 0 | maxint |
| Maneuver duration [12] | goal (nav) | integer > 12 hours | 7 days | maxint |
| Thrust profile [12] | goal (nav) | real x 4 | uniform | infinite |
| Thrust level [1] | goal (nav) | 0-99 | 70-100 | 100 |
| Solar Power available | constraint | 0-2500.0 | 2400-2500 | infinite |
| Slew Duration Function | constraint | all possible functions | -- | infinite |
| Misc. integer-valued parameters [15] | constraint | integer > 0 | expect one nominal value for each | maxint * 15 |
| Random Seed | constraint | integer | 15386 | 2*maxint |
| Exec Latency | constraint | small integer > 0 | uniform in 4-10 | maxint |

that the parameter space actually has these properties, but this risk is needed to construct a manageable test-suite.

Table 2 shows abstract parameter space for the RAX planner. Real or integer-valued parameters from the true space, and those with large numbers of values, were abstracted by selecting a small handful of discrete values to test. Where it was known which values were at boundaries of qualitative behavior regions of the planner, those boundary values were selected. In the other cases values were selected from the parameter's domain according to their expected distribution in operations. No attempt was made to select a statistically significant number of values. The abstraction makes several educated guesses, such as the qualitative behavior boundaries, the distribution of values in operations, and the number of parameter values to select.

The primary abstractions are as follows. The initial states are restricted to twelve canonical states that cover all of the qualitatively different initial states [all combinations of MICAS state (2), MICAS health (2), and final attitude (3)]. The planner is either insensitive to other variations, or those variations are not initial states that the exec would ever generate. The plan start time is restricted to ten boundary points: before, during, and after plan horizon boundaries; and before, during, and after op-nav windows.

Several parameters are set to fixed values. Most of these are design-time parameters that might change during testing and integration, but will not change during the experiment itself. The suite will be re-run if those parameters change. The remaining parameters control values that the planner does not reason about, but simply "passes through" to the executive. Testing a single arbitrary value is sufficient.

*Scaling Up*—There are a number of aspects to the planner and RA design that reduce the size of the parameter space, and thereby facilitate testing. It will be important to pay attention to these design decisions for future missions.

There are a vast number of possible initial states, but only ten or so that are qualitatively different. If the Exec cannot recover from a fault while executing a plan, it terminates the plan and achieves a standby state. This is a quiescent state in which the spacecraft can remain for an indefinite period. The exec then requests a new plan, using this standby state as the initial spacecraft state. There are only a

**Table 2**. Abstracted Parameter Space

| Parameter | True-space parameters | Values | Discretization | Cases (Discretized) |
|---|---|---|---|---|
| Plan start time | start time | 0-518,400 | 10 boundaries | 10 |
| MICAS state | initial state | ON, OFF | ON, OFF | 2 |
| MICAS health | initial state | OK, FAILED | OK, FAILED | 2 |
| Final attitude | initial state | infinite | Along IPS thrust vector, at op-nav target, cruise attitude | 3 |
| NAV win duration | same | integer > 0 | 2h, 3h, 4h, 6h | 4 |
| NAV win goal start | same | 0-518,400 | 1h, 2h, 4h | 3 |
| NAV win period | same | integer | fixed (nominal) | 1 |
| NAV win slack | same | integer | fixed (nominal) | 1 |
| LGA start time [6] | same | 0-518,400 | 0,+1day,+2days,…+5d ays | 1 |
| LGA duration [6] | same | integer | 8 hours | 1 |
| Number Image goals | image goals | 0-20 | 10, 20 | 2 |
| Exposures/Image * | same [20] | integer | 3,4,5 | 3 |
| Exposure duration* | same [20] | integer | 16,25,27 | 3 |
| Instrument Settings* | same [20] | integer | fixed | 1 |
| Image target id [20] | same | integer | random | 1 |
| Number IPS Maneuvers | IPS goals | 0-12 | 0-3 | 4 |
| First Maneuver start | Maneuver start [1] | integer | At plan start, After plan start, Before plan start, Before RAX start | 4 |
| Last maneuver end | Maneuver Start [2-12], Maneuver duration [2-12] | integer | At plan end, Before plan end, After plan end, or After RAX end | 4 |
| 1 hour break between maneuvers? | Maneuver Start [2-12], Maneuver duration [2-12] | None, b/t 1 and 2, b/t 2 and 3, b/t 1, 2 and 3. | None, b/t 1 and 2, b/t 2 and 3, b/t 1, 2 and 3. | 4 |
| Thrust profile* | same | infinite | arbitrary fixed value | 3 |
| Thrust level [1] | same | 0 | 70, 80, 90 | 3 |
| Solar Power Available | same | 0-2500.0 | 1500, 2400, 2500 | 3 |
| Slew Duration | slew duration function | 1s – 20min. | 2 ,5,10, 20 (minutes) | 4 |
| Misc. parameters [15] | same | integer | nominal values | 1 |
| Random Seed | same | integer | any three | 3 |
| Exec Latency | same | small integer > 0 | 1, 4, 10 | 3 |
| TOTAL | | infinite | | **6 billion** |

handful of standby states: one nominal state, and a small number of degraded states corresponding to various failures. For RAX, there are only four standby states including the nominal that correspond to all combinations of camera healthy or broken, and camera on or off.

In nominal operations, the initial state for a given plan is the end-state of the previous plan. This allows execution to continue seamlessly from plan to plan. The planner model is designed so that every plan ends in a relatively quiescent state similar to the standby state. There are only a small number of qualitatively different end-states.

A full-scale science mission will have a few more standby states and end-states, but not many more. The number will be proportional to the product of the health-states tracked by the planner. If the planner covered the entire DS1 mission, it would track only three more health timelines: IPS health, MICAS high-voltage switch health, and RCS thruster health for a total of 32 initial states.

The spacecraft is commanded by a handful of high-level goals that the planner expands into a plan for achieving them. The ground operations team specifies most of the goals, with a few coming from onboard systems (navigation for RAX). Goals tend to interact strongly, which means that they cannot be tested independently. On the other hand, missions are typically designed so that the spacecraft is only doing one or two things at a time. So although the goals interact, one can make assumptions about what combinations of goals will appear in practice. Each of those goal-sets can be tested independently.

*Test Suite Construction*—The test suite must provide adequate coverage, according to some metric, yet have a manageable number of cases. We use a combination of two approaches. First, we use *orthogonal arrays* [4] to generate a minimal-sized test-suite in which every parameter value and every pair of values appears in at least one test case, and every parameter value appears in about the same number of cases.

This approach detects *every* bug caused by a single parameter value or by an interaction of two parameter values. It will detect only some bugs caused by interactions of three or more parameter values. The risk of this approach is that it assumes that the majority of bugs are due to one or two parameter values.

The PS test-suite was constructed using an orthogonal arrays approach. The RAX test-suite contains three sub-suites generated with orthogonal arrays: one for the twelve-hour experiment, and one each for the six day replan cases and the six-day back-to-back plans. The twelve-hour suite has 24 test cases (many of the above parameters are fixed for the twelve-hour experiment), and the other suites have about fifty cases each.

*Coverage Metrics*—Constructing this test suite required several assumptions and abstractions, and each of these introduces some uncertainty in the coverage. In order to assess this risk, we developed several orthogonal coverage metrics with which to evaluate the test suite and its assumptions.

One metric is whether there is at least one test case from each set of inputs for which the planner behavior is expected to be qualitatively different. Whether the planner behaves qualitatively similarly or differently to a set of inputs depends on the constraints in the planner model. If the inputs differ on plan elements that are interact strongly (have many constraints among them), then the planner behavior is likely to differ dramatically. If the inputs differ on weakly interacting plan elements, then the output plans are likely to be similar. With this metric, one identifies all the strongly interacting plan elements and the combinations of "true" parameter values that control these elements. The abstract parameter space should not have settings that correspond to these combinations. If it does, then settings in the abstract space do not correspond to equivalence classes where the behavior is qualitatively similar.

The test-suite coverage is measured with respect to this metric by identifying the combinations of abstract parameter values that control the strongly interacting regions. The test-suite should have at least one test case for each combination. The orthogonal array algorithm can be extended to include these test cases (e.g., [5]), or they can simply be appended to the test-suite with no attempt to minimize.

We performed a very rough interaction analysis to identify the most heavily interacting goals, initial states, and constraint parameters. The test suite contains at least a few test cases from each of these interaction regions. Additional work is needed to implement the interaction analysis metric.

A second metric is how well the test suite exercises all of the requirements. If a requirement is trivially satisfied for some test-case, then that case does not exercise the requirement. A third related metric is how well the test-suite exercises all of the knowledge in the plan model. The plan model consists of constraints of the form "if token A appears in the plan, then token B must also appear in the plan and be in the following temporal relation to A." Token A is called the master token, and B is called the target token. The constraint is exercised if and only if the master token appears in the plan. It is therefore easy to determine which constraints were exercised by examining the tokens in the resulting plan. As an additional check, the plan maintains temporal relations, which makes it possible to tell whether a master/target pair occurred by accident or as a result of exercising a constraint in the model. The coverage of the test-suite is proportional to the percentage of the total constraints exercised.

It is difficult to predict which inputs will exercise a given constraint or requirement, though one can often make a good guess. For the RAX planner, we use these two coverage metrics to measure the coverage of the suite after running it, and then add test-cases if needed. The third metric is analogous to code-covering metrics which are also used for post-hoc coverage analysis.

*Single Variation Test Suites*—The orthogonal array-generated test suite provides excellent coverage with a handful of test cases. However, it is difficult to identify which parameter caused a problem and file a meaningful bug report. To address this problem, we constructed a second suite of test-cases in which each case is identical to a baseline scenario in every parameter value but one. Since the planner is known to perform correctly on the baseline case, any problems are very likely to be caused by the one parameter that changed.

In practice, these "single variation" cases catch most of the initial bugs. The orthogonal-array suites are useful for identifying additional bugs once the single variation cases all pass.

*MIR Test Case Selection*

The MIR module is responsible for the following aspects of autonomous control:

- Confirmation of EXEC commanding
- Detection and Diagnosis of Failures
- Generation of Recovery Sequences
- Reporting State Change to Ground Control
- Incorporating model updates from Ground Control

There are two additional classes of MIR responsibilities that are a feature of the Remote Agent being run as a technology validation experiment:

- Abortion of the experiment when in a state that is beyond scope
- Robustness to real failures during failure demonstration scenarios

The details of these responsibilities, together with their performance and resource usage constraints, are captured into a set of MIR requirements. The goal during test case selection is to define a set of test scenarios that provide confidence that the test artifact meets all requirements.

In principle, the scenario space is bounded from the MIR perspective given that models of the FSW, hardware, and environment are finite state automata, that transitions in these automata take time, and that the duration of the mission is bounded. In practice, this scenario space is prohibitively large for any manner of exhaustive testing, even in simulation. The challenge is therefore to identify a representative sample of the scenario space and exercise the test artifact against that sample, with the ability to claim that requirements met in the sample imply adequate confidence that they are met in the large. In this section, we first provide a descriptive feel for the scenario space, then we provide a more concise parameterization for the space, and finally describe how we exploit this parameterization to select a test suite that meets our needs in terms of coverage and prioritization.

*Scenario Space Description*—To provide a feel for the scenario space facing MIR during RAX, we frame a scenario as a discrete time series over the finite set of events that include: EXEC commands to FSW, FSW monitor events to MIR, Clock timeout notifications to MIR, communication between MIR and EXEC, and communication between MIR and Ground. For instance, consider the following fragment of a scenario:

| | | |
|---|---|---|
| **Event-0** | Command-1: | Power On Camera |
| **Event-1** | to Clock: | Command-1 Timer |
| **Event-2** | Monitor-1: | Camera Status On |
| **Event-3** | Clock: | Command-1 Timeout |
| **Event-4** | to Exec: | Camera On, Healthy |
| **Event-5** | to Ground: | Camera On, Healthy |

In this fragment, MIR first observes the EXEC requesting that the Camera be powered On, and requests that a timer be started to allow any monitored values to stabilize in response to the command. MIR is then notified by a monitor that the Camera switch sensor is reporting that the switch has transitioned to the On position, and subsequently MIR is notified that the command timer has expired. MIR concludes that the Camera is now On, and reports this state change to both Exec and Ground. Now consider the following fragment further downstream:

| | | |
|---|---|---|
| **Event-145** | Monitor-83: | NEB Current Nominal |
| **Event-146** | Command-56: | Power Off Camera |
| **Event-147** | to Clock: | Command-56 Timer |
| **Event-148** | Monitor-84: | NEB Status Off |
| **Event-149** | to Clock: | Monitor-84 Timer |
| **Event-150** | Clock: | Monitor-84 Timeout |
| **Event-151** | to Ground: | NEB Sensor Pop Off |
| **Event-152** | Clock: | Cammand-56 Timeout |
| **Event-153** | to Exec: | Camera Stuck On |
| **Event-154** | to Ground: | Camera Stuck On |
| **Event-155** | from Exec: | Recovery? Camera Off |
| **Event-156** | Recovery: | Power Camera Off |
| **Event-157** | Command-57: | Power Off Camera |

In this off-nominal scenario fragment, two failures have occurred: a non-essential power bus (NEB) switch sensor is erroneously reporting that the NEB switch has popped off, and the Camera power switch is stuck in the On position. The NEB switch sensor failure is neither recoverable nor serious, and demonstrates MIR's ability to disambiguate sensor failure from failure in the sensed device. The Camera switch failure is used here to demonstrate recovery. Looking at the event sequence, MIR first receives a monitor report from the NEB current sensor indicating that current is in the Nominal range. The Exec is then observed to request that the Camera be powered Off, and MIR requests that a timer be started to allow monitors to stabilize. MIR then receives a monitor report indicating that the NEB switch sensor is reporting that the NEB switch is in the Off position. This report is inconsistent with the current desired state of the spacecraft, so MIR starts an additional timer going to allow monitored values to stabilize before initiating a diagnosis. The monitor timer then expires, and MIR diagnoses the switch sensor to be faulty based primarily on these facts: the NEB current sensor is reporting Nominal, the NEB switch must be in the On position to draw current, and a NEB switch sensor failure is more likely than failure of *both* the NEB current sensor and the NEB switch. MIR thus reports to Ground that the NEB switch sensor has popped off (the Exec does not control this device and thus is not informed). MIR next receives notification that the Command-56 timer has expired, and diagnoses the Camera switch to be stuck On, for no monitor report indicates the Camera switch sensor observing the switch transition to the Off position, and a faulty Camera switch is more likely than a faulty switch sensor. MIR reports the diagnosis to both Exec and Ground. In the remainder of the fragment, Exec requests that MIR suggest how to recover from the Camera switch failure to achieve the goal of Camera Off, MIR suggests that Exec re-issue the command to power it off, and finally Exec follows that advice.

These examples are intended to make the notion of a scenario more concrete, but provide only a glimpse into the reasoning capabilities of MIR. Next, consider the following curious fragment:

| | | |
|---|---|---|
| **Event-145** | Clock: | Command-56 Timeout |
| **Event-146** | Clock: | Command-56 Timeout |
| **Event-147** | Command-56: | Power Off Camera |

Here the Clock is reporting the same command timeout twice, and prior to the command itself having been issued. Both of these are anomalies that are legal scenarios according to our current time series definition. A tester would expect very little of MIR in this case, for design assumptions about its environment have clearly been violated; the action in such cases is to track down an external problem in hardware, FSW or a simulator. From the MIR testing perspective, then, we constrain the candidate scenario space and consider only those scenarios that behave within certain design assumptions.

In addition, there are scenarios that are possible given design assumptions but that are considered beyond the scope of the experiment. In some of these scenarios, MIR is responsible for identifying that the experiment is outside of scope and must request that the experiment be aborted; in others, there is no such requirement, and FSW fault protection will abort the experiment if necessary. In selecting test scenarios for MIR, we consider only the former in the candidate scenario space.

*Parameterized Scenario Space*—By exploiting structure in the scenario space, as induced by design assumptions about the environment and experiment scope, it is possible to more concisely characterize the scenario space. The parameterization of the MIR scenario space is shown in Table 3.

Values within each entry of the table are mutually exclusive, thus a single value is drawn from the range shown in each entry. The scenario space can therefore be viewed as a 34-tuple (one for each device parameter), each ordinate of which is itself a 5-tuple specifying a value for Behavior, State, Timing, Attempt and Context. The strong structural claim being made here about the scenario space is that all scenarios passing through such a configuration can be viewed as equivalent from the MIR perspective.

It is important to explicitly capture all environmental assumptions that we use to justify this parameterization, though we do not report on the details here, for doing so would require a level of description of the domain that is beyond the scope of this paper. These assumptions can serve to focus model checking of properties of subsystems in the environment. For example, MIR testing makes assumptions about the nature and frequency of EXEC commanding to justify this parameterization, and such properties therefore warrant special attention during EXEC testing.

*Coverage and Prioritization*—The parameterized scenario space is prohibitively large for any manner of exhaustive testing of our artifact, even on low fidelity testbeds that support rapid simulation of the environment. (Exhaustive testing of our *design* is another matter. We are in the process of applying model-checking techniques to validate our design. We are attempting this using both the parameterized scenario space, and the less structured time series representation described in the previous section.)

We take the set of candidate test cases to be the space of all possible scenarios; that is, every configuration in the parameterized representation is considered for inclusion in the test suite. There are reachability and controllability assumptions made here about the testbed, for some possible scenarios cannot be performed on some higher-fidelity testbeds: the testbed may not support injection of a desired fault into real hardware or software, or there may be insufficient observability or responsiveness in the testbed to time such an injection appropriately during a transitional period of interest in the scenario. We make these assumptions during test case selection and test the artifact on the highest-fidelity testbed capable of executing the scenario, or perhaps prune such test cases from the test suite if no such testbed exists. It is worth noting here the guideline that scenarios that are difficult to produce in a testbed environment for the above reasons are for the same reasons rather unlikely to occur in flight.

Our current approach to selecting a covering sample of the parameterized scenario space is qualitative. We note first that every entry in the table has nominal values, or those that can be considered such; we label all such configurations involving only nominal values as the *nominal scenario*. To this scenario, we add all *single variations*, consisting of configurations whose parameter values differ from a nominal one in at most one entry. We then include any *multiple variations* deemed to provide additional interesting interactions, as obtained by manual inspection of the MIR models.

Traditional FSW development and testing often makes the simplifying assumption that multiple variations are beyond scope, and the justification cited is usually that the likelihood of multiple variations in negligible. The justification for our current qualitative approach to coverage is also guided by this tradition, and is deemed acceptable for RAX particularly given testing resources available. Another traditional justification for the single variation scope is that the risk in supporting multiple variations due to added complexity in the software outweighs the risk inherent in the variation itself; this claim is justified in part by noting that the matter is traditionally addressed by ground operations. An autonomous system will therefore have to go a step beyond single variation scope to gain wide acceptance. We take our test suite a step further into a reasonable number of multiple variations to provide additional confidence, as we believe this is a necessary requirement in any long-term testing story for an autonomous system.

We are currently pursuing a more quantitative approach to test case selection. Given our parameterization above, we augment it with information available in the MIR models about the likelihood of various off-nominal entries occurring. Given a mission profile, or even a probability distribution over possible mission profiles using the parameterization discussed in the Planner Test Case Selection section, it is possible to measure the expected

**Table 3:** MIR Scenario Space

| Parameter | Behavior | State | Timing | Attempt | Context |
|---|---|---|---|---|---|
| Camera Power Throw | Responsive, Unresponsive | On, Off | Nominal, Delayed | 1-3 Tries | Startup, Standard, 4 Demos |
| Camera Status Throw | Nominal, Stuck | On, Off | Nominal | 1-3 Tries | Startup, Standard, 4 Demos |
| PASM Power Throw | Responsive, Unresponsive | On, Off | Nominal, Delayed | 1-3 Tries | Startup, Standard, 4 Demos |
| PASM Status Throw | Nominal, Stuck | On, Off | Nominal | 1-3 Tries | Startup, Standard, 4 Demos |
| 8 Static Power Throws | Uncommanded | On, Popped Off | Nominal | None | Startup, Standard, 4 Demos |
| 8 Status Throws | Nominal, Stuck | On, Off | Nominal | None | Startup, Standard, 4 Demos |
| 8 Thruster Valves | Uncommanded | Open, Stuck Closed | Nominal, Delayed | None | Startup, Standard, 4 Demos |
| ACS MDC State | Nominal, Unexpected | Sun Standby, Earth Standby, RCS, TVC | Nominal | 1 Try | Startup, Standard, 4 Demos |
| IPS/DCIU State | Nominal, Unexpected | Standby, XFS Init, Thrust, Safe | Nominal | 1 Try | Startup, Standard, 4 Demos |
| LPE Remote Terminal | Responsive, Unresponsive | Comm., No Comm. | Nominal, Delayed | 1-3 Tries | Startup, Standard, 4 Demos |
| 3 Other Remote Terminals | Uncommanded | Comm., No Comm. | Nominal, Delayed | None | Startup, Standard, 4 Demos |

amount of time spent in each of the nominal configurations above. The expected time in each nominal configuration together with the likelihood of off-nominal values occurring in that configuration yield an estimate on the likelihood of entering an arbitrary configuration during the mission; this estimate can then be used to focus test case selection on highest likelihood configurations.

The quantitative approach outlined here also enables an "any-time" approach to testing by prioritizing test runs according to the highest likelihood configurations; this prioritization is valuable in an environment of uncertain testbed availability, enabling testers to deal optimally both with testbed shortages and with any unexpected opportunities in testbed availability that may arise.

*EXEC Test Case Selection*

The executive carries out plans provided by the planner, and responds to fault diagnoses and repair suggestions from MIR. Accordingly, the majority of executive testing consists of exercising the EXEC on the planner and MIR test cases and verifying that the EXEC behaves properly.

A few EXEC requirements are verified with unit tests. In particular, the EXEC has a procedure for each token in the plan. The procedures are largely independent of each other, and so unit testing provides high confidence that the procedures are correct.

*System Test Case Selection*

RAX is required to be able to react positively to a predetermined set of fault conditions that can occur on the spacecraft. It is also required to abort when requested to by the flight software and to ensure that the present spacecraft state matches the state that is asserted by the plans. There are also a handful of lower-level safety, performance, and timing requirements that must be met. In all, there are 66 system requirements.

The system tests consist of 22 scenarios that exercise these conditions. The scenarios consist of segments of the nominal RAX scenario, the nominal experiment in its entirety, and a handful of the most likely off-nominal scenarios. Each scenario configures the FSW simulators or actual hardware simulators to provide desired behavior, and then the RA is run normally. The RA generates a log file, which testers analyze to verify that the RA behavior is correct.

The system scenarios exercise all of the requirements in the highest-likelihood conditions, but do not test them in all the possible situations that could occur. The module tests on the low-fidelity test-beds provide confidence in the RA behavior under the off-nominal conditions. The scarcity of high-fidelity test beds forces us to trade risk for feasibility.

This approach to system testing assumes that there is one nominal scenario that is known prior to execution. This is certainly true for RAX, where the scenario is almost entirely under control of the experiment team and the only anomalies that are likely to occur are those simulated as part of the experiment (though RA must also respond properly if those anomalies occur naturally). The system suite therefore provides relatively high confidence in RAX.

**Table 4.** DS1 Test-beds

| ***Platform*** | ***Fidelity*** | ***CPU*** | ***Hardware*** | ***Availability*** | ***CPU speed*** |
|---|---|---|---|---|---|
| **Spacecraft** | Highest | Flight | Flight | 1 for DS1 | 1:1 |
| **DS1 Testbed** | High | Flight | Flight spares + DS1 simulators | 1 for DS1 | 1:1 |
| **Hotbench** | High | Flight | Flight spares + DS1 simulators | 1 for DS1 | 1:1 |
| **Papabed** | Med | Rad6k | DS1 simulators only | 1 for DS1 | 4:1 |
| **Radbed** | Low | Rad6k | RAX simulators only | 1 for RAX | 4:1 |
| **PowerPC** | Lowest | PPC | RAX simulators only | 2 for RAX | 10:1 |
| **Unix** | Minimal | Sparc Ultra | RAX simulators only | unlimited | 40:1 |

- The flight CPU is a radiation hardened RS-6000 chip (Rad6k) running on the flight bus, memory, etc.
- The Papabed and Radbed run on a Rad6k chip bus, but have some non-flight bus and memory components.
- The PowerPC (PPC) is a non-hardened, off-the-shelf RS-6000 chip with higher clock speed than the Rad6k.
- The RAX simulators were written by the RAX team, and are of lower fidelity than the DS1 simulators.

The same approach should scale to science missions. Science missions have a nominal mission plan. The system test-suite will consist of this scenario combined with the most likely off-nominal scenarios. The purpose of the system tests is to gain confidence that the software behaves properly as a whole, interacts properly with the actual FSW and hardware, and that there are no timing or performance problems. Module tests on lower fidelity platforms are necessary to gain confidence that the RA responds robustly to off-nominal situations. This approach still entails some risk, perhaps more than flight project mangers are willing to accept. Additional testing or formal V&V methods may be needed to reduce the perceived risk and the actual risk to acceptable levels.

## 5. ALLOCATING TESTS TO TEST BEDS

The DS1 flight project has a number of test beds ranging in fidelity (with respect to the spacecraft) and scarcity, as shown in Table 4. The highest fidelity platforms, such as the spacecraft itself, are scarce and testing time on them is limited. CPU speed decreases with increasing fidelity,[3] further limiting the effective testing time. Lower fidelity platforms are more numerous and have faster CPUs, but tests performed on them must be combined with a strong argument that additional fidelity will not change the outcome.

High fidelity test beds are also the most difficult to configure and instrument for a given test, whereas lower fidelity test-beds are generally the easiest. This means that some tests can only be performed on lower fidelity test beds.

A key challenge is deciding how to allocate tests to platforms in order to maximize confidence in the Remote Agent. Traditional FSW also faces this challenge, but to a lesser degree since there are fewer tests and most of them must be run on high fidelity test-beds.

The general approach we have taken for RAX is to allocate most of the module (PS, MIR, and EXEC) tests to the low-fidelity PPC test-beds, with timing and performance issues being tested on Radbed. Scarce access to high-fidelity test beds is reserved for system level tests.

### *RA architecture supports multi-fidelity testing*

The Remote Agent architecture is well suited to testing at multiple fidelity levels. Each element of the RA (EXEC, MIR, PS) consists of a domain-independent reasoning engine, a domain-specific model, and a handful of implementation-specific interfaces with the flight software. Low fidelity platforms (such as Unix) can verify almost all of the engine requirements since the way in which the engines reason about and act upon the knowledge in the models does not change with increasing fidelity. The remaining handful of timing and performance requirements can be independently verified on high-fidelity platforms.

The RA interacts with the spacecraft via FSW interfaces. The FSW abstracts the details of the spacecraft. The Remote Agent reasons about and acts upon the spacecraft at that level, according to the knowledge in the RA's models. This abstraction layer makes it possible to verify the models on low to medium fidelity platforms.

The FSW presents an abstraction to the RA that is relatively easy to capture in a simulator. The RA interacts with the FSW simulator on low or medium fidelity platforms. This is sufficient to verify that the RA reasons and reacts correctly with respect to the abstracted spacecraft.

Testers can configure the FSW simulator to produce behavior that would be difficult to elicit from the actual FSW. This allows the low-fidelity tests to more thoroughly exercise the RA than would otherwise be possible.

[3] Flight processors are typically one or two generations behind the state-of-the-art due to the need for radiation hardening and the need to select a CPU at the beginning of the project (at least two years before launch).

This argument is used extensively for Planner testing, since the PS test-suite is the most time-intensive of all the RA modules. In order to obtain adequate coverage, the test suite must have about 200 plan generation cases. Plans take four hours to generate on flight CPUs, but less than five minutes to generate on a Sparc Ultra/1 which means it would take over a month to run through the entire suite on a high-fidelity test-bed, but only sixteen hours to run through on Unix.

The only planner input that changes on higher fidelity test-beds is the exact content of the goals provided by on-board systems such as NAV and ACS. This means that modulo timing and performance issues, the planner behavior is expected to behave identically on Unix and the spacecraft for any given input. This permits comprehensive planner testing on Unix platforms with full expectation that the results will scale up to other platforms. A handful of tests are repeated on higher fidelity platforms to address performance issues and to be sure that the plans are identical (i.e., no anomalies are introduced by platform-dependent implementation differences, the CPU, or because we are using a RAMdisk instead of an NFS file system).

At some point the RA must be tested in conjunction with the actual FSW. The FSW may have drifted from its original specifications, or may have subtle behavior not captured by the simulator. These tests require test-beds that support the FSW. For DS1 this means Papabed or the high-fidelity testbeds. If the FSW ran on lower-fidelity test-beds such as the PPC, we could test the RA/FSW interactions on those testbeds.

Finally, the RA/FSW combination must be run on the test-beds with the highest fidelity spacecraft hardware and simulators (Hotbench or DS1 testbed) in order to verify timing and performance issues, and to ensure that there are no subtle hardware behaviors that violate their FSW abstraction.

A test-suite with adequate coverage is too large to run on the scarce high-fidelity test-beds. The approach we are taking for RAX is to devise high-coverage suites for each of the RA modules (EXEC, MIR, and PS), and run these on the low fidelity PPC and Radbed platforms. A fourth test-suite, called the system tests, covers system requirements and provides minimal coverage of the most critical module requirements. This suite is run on higher fidelity platforms. Since these platforms are scarce, and tests take longer to run on them, the system test suite is very small (about 20 tests). The correspondingly low level of coverage is only acceptable because of the comprehensive module testing on lower fidelity platforms. Table 5 shows the allocation of RAX requirements to testbeds.

*Experiment Design for Testability*

The system tests consist of the nominal RAX scenario, plus a handful of the most likely off-nominal scenarios. Each test configures the FSW simulators or actual hardware simulators to provide desired behavior, and then the RA is run normally. The RA generates a log file, which testers analyze to verify that the RA behavior is correct.

The scenarios vary in duration from twelve hours to six days. The high-fidelity testbed schedules have very few contiguous time slots large enough to accommodate these scenarios, and many of the scenarios will have to be repeated (detect a bug, fix it, repeat test).

Testbed availability therefore imposes very strong constraints on the number of scenarios in the system test-suite, and the running time of those scenarios. Most test slots are one day long, with three-day slots the next most common, and only two or three opportunities for six-day runs. Correspondingly, the running time of scenarios in the test suite should follow this distribution. The number of scenarios of each length should be one half to one third the number of available slots for that running time, to allow for re-running scenarios and slots lost due to downtime or preemption.

**Table 5.** RAX Test Allocation

| Requirement | Test Suite | Platform |
|---|---|---|
| Timing (execution, fault response, etc) | System | DS1 Testbed |
| Resource utilization | System | DS1 Testbed |
| Performance (e.g., planning duration) | System | DS1 Testbed |
| Plan generates plan in four hours | System | DS1 Testbed |
| RAX Aborts when commanded | System | PPC, confirm on DS1 Testbed |
| Planner generates correct plans w.r.t. real NAV goals and ACS constraint functions | System | Papabed, DS1 Testbed |
| Planner generates correct plans w.r.t. simulations of NAV goals and ACS constraint function. | PS | Unix, PPC |
| MIR diagnoses faults correctly | MIR | PPC |
| MIR suggests correct fault recovery | MIR | PPC |
| Exec dispatches tokens according to temporal constraints in plan | EXEC | PPC |
| Exec issues proper commands to execute plans | EXEC | PPC |

Since the test-suite can have only a few scenarios, each scenario must exercise several requirements. We identified the events that we wanted to exercise within a single run, and selected scenario start and end times or events accordingly. However, two issues make it difficult to define and execute these scenarios.

First, the critical events in the experiment are temporally distant, which makes it difficult to pack several of them into a short run. On low-fidelity test-beds the simulators can be jumped ahead ("warped") to the next interesting event, but this is not possible on high-fidelity test-beds. Even if it were, at least some scenarios would have to be run without warping, since this is the way it will run in flight.

The second issue is starting and ending the scenario at the selected boundary points. The high-fidelity test-beds operate as much like the spacecraft as possible. This means the RA can only be operated as it would in flight. That is, the RA starts in one of a few initial states, requests a plan and executes it. The plan can be a pre-specified plan if the RA is in "scripted" mode, or generated on-board if the RA is in "autonomous" mode. The RA cannot start in the middle of a plan or in an initial state outside of the specified set. The RA cannot begin in the middle of a plan, even if that is the most logical place to do so from a testing standpoint.

For RAX we have managed to design a test-suite in spite of these restrictions, largely because there are only a few critical events that need to be tested and because the experiment has very short plan horizons that happen to fit the available time slots (12 hours, 2.5 days, and 3 days). Future missions will have more critical events and longer plan horizons, which will make it more difficult to design an adequate system test suite. One possibility is to define mini-scenarios that exercise the critical events within a short time span. This entails some risk, since the software is not tested on the nominal flight scenario.

## 6. AUTOMATION ISSUES

Automation is an important component of modern testing. For RAX we developed automated test runners to reduce the time and workforce needed to collect test results after each major release (every 3-4 weeks), and to ensure reproducible test-cases for purposes of debugging and regression testing.

We also developed automated tools for analyzing the test results and checking them against requirements. Because RAX must react robustly to a wide variety of situations, it is often infeasible to specify the expected output for each input scenario. Instead, we had to develop customized analysis tools or perform the analysis by hand.

The following sections describe the automation methods we found useful for testing RAX.

*Testing Process*

After producing a relatively stable and full-featured version of RAX, the RAX team split into development and testing sub-teams. Every three to four weeks the development team releases a new version of RAX along with a release document listing the bugs that the developers believe they have fixed.

The testers run the entire test suite on the release and archive the execution logs for future reference. The suite is fully automated for repeatability and to reduce workforce requirements. The next section discusses the automated test-running infrastructure in more detail. Running through the test suite take about five to eight work days, with each of the test teams (EXEC, MIR, PS, SYSTEM) allocated one three-hour time window on the PPC platform per day.

The testers analyze the logs using special-purpose verification tools. These are discussed in the following sections. Bugs are reported via GNATS™, a web-accessible bug tracking database in the GNU tool chain. Bugs are initially filed as "open," and move through "analyzed" to "feedback" when the developers believe they have resolved the problem.

Bugs are not "closed" until (a) the bug has been declared fixed in an official release, and (b) the tester who filed the initial report has verified the fix with an appropriate test. The test results are referenced in the database when the bug is closed. A bug may also be "suspended" to indicate the bug may be safely ignored for some reason (e.g., the behavior is technically incorrect but harmless, or there exists a work-around).

Figure 3 summarizes the testing procedure. The following sections describe the test-running infrastructure and verification tools in more detail.

*Automation methods: Running tests*

We are given a test artifact, and a set of scenarios with which to exercise the artifact. The goal is to quickly and reliably determine how the artifact behaves when faced with each scenario. Making this determination requires having the ability to embed the *unmodified* test artifact into a testbed so that it can be operated in the *standard* way, and it requires having the ability to both observe and control the testbed environment in such a way that forces the test artifact through the scenario of interest to collect the necessary data.

Running a single test scenario in a testbed generally involves having a test operator perform the tasks outlined in Figure 4.

The purpose of spelling out this test procedure it twofold. First, it should be apparent that the test operator is playing the roles of ground operator and, to the extent required, the environment. With some testbeds, little is required of the test operator to control the environment if the testbed simulator is able to support scripting and timed fault injection. In other cases, the test operator is required to monitor the test artifact as it progresses through the scenario, and must manually inject a failure while the system is observed to be in some context of interest, by exploiting an observable of which the simulator is not aware; this is traditionally necessary when the observation is based on the telemetry stream, for instance.

Second, the procedure serves to make clear that running a test scenario is a rather involved, time-consuming, and potentially error-prone task; this is of course a function of the testbed: some operations above (uplink, for instance) are trivial on lower-fidelity testbeds. However, even on lower-fidelity testbeds, running an arbitrary scenario places a significant demand on the test operator. Operator errors are costly, usually requiring a restart of the above process and with it the loss of valuable testbed time. Worse, the operator error may not have been noticed during the test run, resulting in a data set from a scenario other than the one intended. Such errors will then have to be caught downstream in the testing process during the data analysis phase, at a much higher cost.

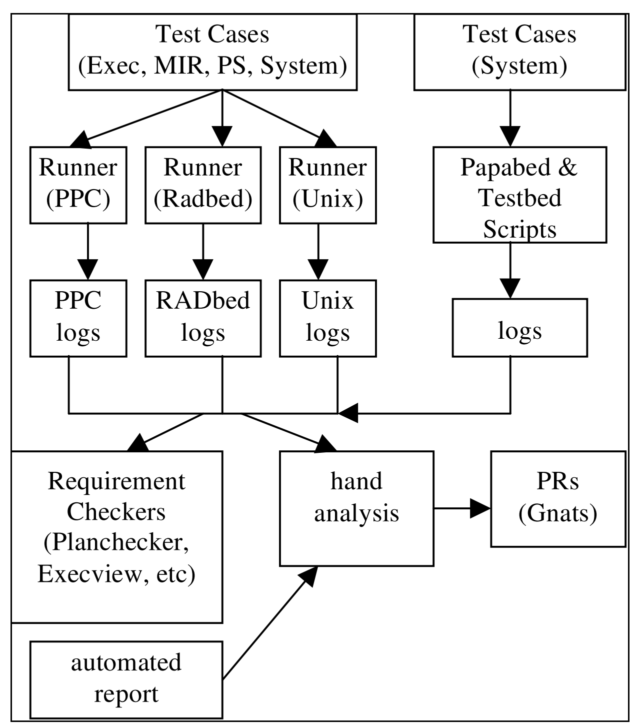


**Figure 3: Testing Process**

™ GNATS © is a trade-mark of Free Software Foundation, Inc.

Our goal is to quickly and reliably acquire data from hundreds of test scenarios on a sufficiently frequent basis to keep pace with major releases of new test artifacts, and to enable feedback in a manner timely enough to impact the following release. To accomplish this, we have developed a *test controller* to automate this process of running the test artifact through an arbitrary scenario of interest.

The test controller accepts as input a formal specification of a test scenario (the first step in the test operations procedure). The scenario specification fixes values for the parameters discussed in the Planner and MIR Test Case Selection sections, thus dictating the mission profile to be used and the behavior of the environment during the scenario. In addition, numerous other parameters facilitating test operations are provided; we outline some of the most important in the remainder of this section, and highlight the associated benefits.

One challenge in running through a test suite using an automatic test runner is in knowing when it is safe to stop the current test and move onto the next. To address this, a condition is provided in the scenario specification for automatically stopping the test based on what is observed in the telemetry stream during a run. As a safeguard, a test timeout duration is also provided to allow the test controller to stop the test in the event of unexpected behavior. The value in the stopping condition is that the time it takes to run the test is dramatically reduced when the software performs as expected. The timeout is intended only as a conservative bound on the time the test will take, and it must be viewed a waste of testbed resources if the test times out. Such control and safeguarding is critical in planning and performing effective overnight runs on scarce testbed resources.

1. Read and understand a description of the current test scenario
2. Reset the testbed:
    a. Hardware, FSW, and any simulators in standard bootup state
    b. Prepare testbed for RAX uplink:
        - Issue testbed commands to configure the hardware and simulator
        - Issue ground sequence to configure FSW
    c. Issue ground sequence to Uplink RAX software
3. Start RAX
    a. Issue testbed commands to configure the hardware and simulator
    b. Issue ground sequences to configure the FSW for RAX startup
    c. Wait until RAX Start time
    d. Issue ground sequence to Start RAX
4. As the current test scenario description dictates, in a timely manner:
    a. Issue testbed commands to simulate any off-nominal events
    b. Issue scenario-related ground sequences to RAX
    c. Issue ground sequence to Stop RAX
5. Ensure RAX correctly downlinked all validation logs
    a. Issue ground commands to salvage missing logs
    b. Move test data files out of downlink area into permanent storage
6. Reset the testbed for the next test scenario

Another way of speeding up test runs is to speed up the environment; many simulators support this. The important constraint to keep in mind when designing a test is to ensure that the speedup is otherwise irrelevant to the outcome of the test. We have developed a warping mechanism that is able to determine when it is safe to warp the testbed to a future time; this is possible, for example, when the test artifact is going to be idling for an extended period. In addition to warping, the scenario specification enables various simulator parameters to be specified to reduce the time it takes to execute a turn, to take a picture, etc.

The ability to automatically monitor the decoded telemetry stream in real-time during a test opens the door to a broader class of context-sensitive fault-injection. The scenario specification provides a language for specifying actions (simulator or ground commands) to be taken in terms of what is being observed in the telemetry stream. By actively monitoring the telemetry stream, the test controller is able to track the progress of the test artifact; when it enters the context of interest, a fault can be injected into the simulator. This provides a much finer-grained and reliable control over the testbed than is traditionally possible, and thus enables a greater set of test scenarios to be performed on each testbed.

Our approach raises awareness with respect to designing telemetry content to improve testability and operability. For example, engineers might wish to add certain telemetry content to increase observability into the artifact by exposing in telemetry key aspects of its internal state; this remedies certain aspects of the reachability and controllability issues raised in the MIR Test Case Selection section.

Complementary to augmenting telemetry content, one may wish to provide some form of instrumentation to customize the data collected during a run. The RA is equipped with a general mechanism for adding such instrumentation on the fly as part of its interface with ground operations. This mechanism can be exploited during testing equally well, and serves to minimize any non-standard interaction with the test artifact during testing that may have otherwise been necessary.

Finally, our approach to specifying test scenarios is designed with the need to run the same scenarios on multiple testbeds in mind. For instance, telemetry content is a property of the test artifact, and is therefore an invariant of the testbed on which the artifact is exercised;

thus, having a test strategy that defines scenarios in terms of this testbed invariant is a very useful way of easing migration from one testbed to another.

More generally, we have attempted to make scenario specifications a testbed invariant. We have developed an API for our test controller that provides abstractions for ground commands and simulator commands required to exercise an arbitrary scenario on any testbed. Only the API is reimplemented when migrating to a new testbed. The test suite itself, as a collection of scenario specifications, need not change during a migration. We have succeeded in seamlessly migrating our test suite from a PowerPC testbed to a higher-fidelity RADbed using this approach. This portability does have inherent limits, of course: one cannot expect higher-fidelity testbeds involving flight hardware to support injection of all faults supported in simulation-based testbeds.

*Planner Verification*

The planner generates a plan from an initial spacecraft state, a set of goals, and constraints. The main requirement on the planner is that the plan meet a long list of correctness requirements. Plans can be several hundred kilobytes long, and are not human-readable. There are about two hundred plans in the test-suite, and the entire suite must be analyzed once a month. There is clearly a need for automated plan verification tools.

We have developed two such tools for RAX. They both use the same basic approach, which is to read the plan into an assertions database and then verify that the assertions satisfy constraints expressed in first order predicate logic (FOPL). These tools were implemented in AP5 [3], a language that supports these kinds of FOPL operations.

The plan correctness requirements are FOPL statements that specify constraints that must hold among plan elements. For example, one constraint is that the plan must not contain OP_NAV_WINDOW tokens if the MICAS camera switch is stuck in the off position (as specified by the MICAS_HEALTH token). This is encoded as the FOPL statement "for all opnav window tokens *w* there exists a MICAS_HEALTH token *h* such that *h* contains *w*."

Some of these constraints correspond directly to *compatibilities* in the planner model. Other constraints do not map to a single compatibility, but are satisfied by some collection of compatibilities.

The first tool verifies the constraints that do not correspond to a single compatibility in the plan model. If the plans in the test suite satisfy these constraints, we have high confidence that the planner model correctly enforces the constraints for all plans. It is much easier to verify that a given plan satisfies a constraint than it is to write compatibilities that enforce the constraint for all plans. For the RAX planner, there are about 40 constraints in this category.

The second tool verifies that the plan satisfies every compatibility in the model. The primary purpose of this tool is to verify that the planning engine obeys the semantics of the plan model. The secondary purpose is to verify all the constraints that correspond to a single compatibility. The model is inspected to verify that each constraint does in fact have a corresponding compatibility. If so, the constraint is satisfied since the engine guarantees that every compatibility is satisfied in the plans it generates.

This tool verifies plans as follows. Roughly speaking, compatibilities are of the form "if token A exists in the plan, then there also exists a token B such that the temporal relation R holds between A and B." A plan satisfies a compatibility if for every token of type A there exists a token of type B in the specified temporal relation, and the relation is specified explicitly in the plan. It is very important that the temporal relation appear explicitly in the plan. First, the exec uses the relations in the plan to maintain the proper relationship among executing tokens. Second, the existence of an explicit relationship makes it clear that the tokens did not satisfy the relation by accident, thereby avoiding false "PASS" results. False PASS results are more dangerous than false FAIL results, since the FAIL results are all investigated to identify the bug that caused it, but the PASS results are not. False FAIL results will therefore be detected and corrected, but false PASS results will not be corrected.

The verification tools provide high confidence that the planner generates plans that satisfy the correctness conditions. It is also necessary to validate the correctness conditions themselves. A minimal approach is to have appropriate system engineers review the conditions. We took this approach for RAX. A more systematic approach would be to augment the review process with formal design-validation methods such as SPIN [6] to ensure that the correctness conditions guarantee a small handful of high-level safety and "liveness" conditions.

This was done for RAX by having appropriate DS1 engineers review the correctness conditions.

Final approach is to validate the model itself. Since model is declarative, we rely on inspection-based validation by experts.

*Execution Verification*

During execution, the Remote Agent generates data in the form of telemetry and (for the experiment) an additional validation log file detailing behavior and key aspects of its internal state; EXEC, MIR, and System testing focuses on this data during the analysis phase to determine if requirements are met. There are twenty-two system-level scenarios, and several hundred scenarios for EXEC and MIR. These data files are human-readable, though rather long (over a megabyte), making a thorough manual inspection of every execution trace infeasible.

There are two levels at which requirements checking can occur: first, one can attempt to verify that every requirement is met by every execution trace; alternatively, one can note that each execution trace was produced by running a scenario that was included in the test suite to test a very focussed subset of the requirements, and thus focus only on verifying that small subset.

We have adopted the latter approach for RAX, primarily because it turns out to be quite manageable given the experiment scope. The key is that each execution trace need only be examined in very localized regions to determine if the relevant requirements have been met, for the tester has knowledge of the scenario and where it is intended to deviate from nominal. The execution trace from the nominal scenario is examined in its entirety.

One incompleteness in the above approach is that focussing exclusively on regions of the execution trace that are expected to deviate from nominal (given the scenario) will fail to detect deviations in other (supposedly nominal and uninteresting) regions. One solution is to specify a “gold standard” execution trace that specifies the correct behavior during the nominal scenario. The gold standard is then contrasted against the current execution trace, and all deviations are examined.

This solution comes at a cost of having to again examine each execution trace in its entirety, but here the property being detected is simply “deviation from nominal” rather than a complete check on all requirements. This intermediate solution strikes the right balance. We have developed a differencing tool that takes an abstraction of the execution trace and highlights deviations with the gold standard. Expected deviations are checked for pertinent requirements, unexpected ones are debugged.

While we believe the approach we have taken is the right one for RAX given testing resources, we are well aware that it will not scale up to larger missions. We have taken the opportunity to begin pursuing in parallel a longer term approach. Specifically, we have found that special-purpose visualization tools will go a long way towards helping to focus attention on key aspects of the trace, and can present information in an intuitive fashion. We have had significant external support in developing the following tools:

- **Planview**
  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 Carnegie Mellon University by Simmons and Whelan [9]. 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.

- **Log checker**
  This Perl-based tool reads the validation log and performs information filtering, analysis, statistics and preliminary diagnosis. It checks the log against several system level requirements automatically while helping the human reader to get necessary information easily to check against other requirements.

- **Packet View**
  This Tcl/Tk tool reads the decoded telemetry stream and displays it at various levels of granularity, providing support for color-coding packets and probing for other properties.

- **Stanley Ground Ops**
  This Tcl/Tk tool provides a hierarchical schematic representation of the MIR models, accepting as input the decoded telemetry stream and displaying the resulting state changes that occur within MIR as execution progresses.

With these tools, a tester can analyze a log much more quickly than is possible by analyzing the raw logs.

*Flight-rule Checking*

One of the system requirements is that the Remote Agent obey all of the DS1 flight rules. Flight rules are requirements on how the spacecraft is commanded. They typically address issues such as subsystem interactions (e.g., always turn off the high-power instrument switch before turning on the low-power switch to avoid damaging the instrument), and operational work-arounds for bugs or limitations in the flight software.

For RAX we will address this requirement by generating sequences from the execution log for several of the system test scenarios. We are developing a tool that extracts spacecraft commands from the log file, and convert them into a sequence file, a JPL SASF (Space Activity Sequence File) format. This file can then be fed into the DS1 uplink sequence constraint checker, which verifies that the sequence is consistent with the flight rules. This is the same checker that validates the traditional sequences used for the rest of the DS1 mission. The sequences can also be confirmed manually by subsystem engineers.

This verification step is needed to build confidence in the RA for this first mission, but it should not be needed for testing autonomy software on future science missions. Instead, the flight rules should be encoded in the planner model. The planner will guarantee that any plan it generates will obey these rules. This guarantee can be stated as a property of the planner and validated with formal methods. This is a much stronger proof that the RA will not violate flight rules than spot-checking a handful of scenarios.

## 7. MODEL VERIFICATION AND VALIDATION

We have identified a number of formal methods approaches for verifying and validating the RA models and engines. Portions of the EXEC engine were validated using a SPIN-based model-checker that identified a number of subtle interactions that would not have been caught with scenario-based testing [8]. The planner model and the planner engine are verified by generating plans for a variety of initial states and goals, and using a plan-checker to verify that the plans meet a validated set of constraints (this was discussed in Section 6). The MIR team is investigating methods for automatically generating minimal-length test suites directly from the MIR models.

With the exception of the plan-checker, time limitations have prevented us from incorporating these methods into the RAX testing strategy in any significant way. Further work is needed to fully develop these methods and integrate them into the testing strategy for future missions that utilize the RA technology.

*Design-analysis of the EXEC*

A formal analysis approach is used to check if the Executive code violates design specifications [8]. 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.

This approach discovered errors in the Executive code 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.

*Automatic generation of minimal-length tests*

In the MIR Test Case Selection section, a parameterization for the MIR scenario space was provided, and a method for test case selection offered. The result is a prioritized set of configurations that are to be tested. Each configuration corresponds to an equivalence class of scenarios, any one of which can be selected and executed to test MIR.

Several issues arise: given a configuration, what is the best scenario to select to represent the corresponding equivalence class? Can we and how do we instruct a testbed to follow that scenario? To answer these questions requires having a model of testbed capabilities, and an ability to exploit that model to generate a plan that will bring the testbed into that configuration.

For RAX, we developed these plans manually, not at high cost given the complexity of the domain. We are investing an approach that would exploit the MIR models themselves to generate minimal-length plans to use in controlling a testbed.

## 8. STATUS OF RAX TESTING

As of October 1998, we have performed the low-fidelity test suites on four RAX releases. Between December and March we will perform the system test suite on the high-fidelity test beds. We have been able to achieve this testing effort with a four-person testing team working approximately half-time.

## 9. CONCLUSIONS

Verifying and validating autonomous systems raises a number of issues not faced by traditional flight software. Traditional flight software (FSW) can focus testing efforts on the small handful of known execution paths, whereas autonomous software must provide high confidence that it will behave correctly in all situations. To provide this confidence, the test suite must have adequate coverage of the requirements and input space. Some of the standard coverage metrics and test-suite construction methods are applicable to the RA, but in some cases new metrics and methods were needed. We identified a number of these that we found useful for testing RAX.

Test suites with good coverage also have a relatively large number of test cases, at least with respect to traditional FSW. Since high-fidelity test beds are scarce on flight projects, it was necessary to distribute the tests among high and low fidelity platforms. Several aspects of the Remote Agent architecture made this feasible. We expect that future missions can use a similar approach.

The complexity of autonomous systems makes it is difficult to specify and verify the expected behavior. We identified a number of methods for specifying the expected behavior, and developed tools for automatically verifying the observed behavior against those specifications.

A full-scale testing effort should also include formal validation methods to provide even higher confidence in the RA. We identified a few such methods and performed proof-of-concept demonstrations. Expanding upon these methods is an area for future work.

Overall, the RAX testing effort has identified several issues that arise when testing autonomous spacecraft commanding systems, and demonstrates a credible verification and validation approach that will scale up beyond the scope of the Remote Agent Experiment. Successful validation of the RAX will open the door to use of this exciting technology on future science missions, and perhaps encourage the development of new mission classes that are only possible with autonomous spacecraft.

## 10. ACKNOWLEDGMENTS

This paper describes work performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract from the National Aeronautics and Space Administration, and by the NASA Ames Research Center. This work would not have been possible without the efforts of the rest of the DS1 team, Doug Bernard, Greg Dorais, Ed Gamble, Chuck Fry, Bob Kanefsky, Jim Kurien, Nicola Muscettola, Kanna Rajan, and Will Taylor.

## REFERENCES

[1] Doug Bernard, Greg Dorais, Chuck Fry, Edward

Gamble Jr., Bob Kanefsky, James Kurien, William Millar, Nicola Muscettola, P. Pandurang Nayak, Barney Pell, Kanna Rajan, Nicolas Rouquette, Ben Smith, and Brian Williams, “Design of the Remote Agent Experiment for Spacecraft Autonomy,*”* In *Proceedings of the 1998 IEEE Aerospace Conference*, Aspen CO

[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] D. Cohen. “Compiling Complex Database Transition Triggers,”. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 225-234. Portland, Oregon. ACM Press, 1989.

[4] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, “The Combinatorial Design Approach to Automatic Test Generation”, *IEEE Software*, pp. 83-88, September 1996. (Also voted one of the best papers at Seventh International Conference on Software Reliability Engineering [ISSRE], Oct. 30 to Nov.2, 1996).

[5] D. M. Cohen, S. R. Dalal, M. L. Friedman, and G. C. Patton, “The AETG System: An Approach to Testing Based on Combinatorial Design”, *IEEE Transactions on Software Engineering*, Vol., 23 No. 7, pp. 437-444, July 1997.

[6] Holzmann, G. Tutorial on SPIN/Promela. *Computer Networks and ISDN Systems*, **25**:981-1017, 1993.

[7] James Clarke “Automated Test Generation from a Behavioral Model”, May 1998, Software Quality Week conference, volume 1, section 2T1.

[8] Lowry, M., Havelund, K., and Penix, J., “Verification and Validation of AI systems that Control Deep-Space Spacecraft,” in Foundations of Intelligent Systems, Proceedings ISMIS-97: 10th Int’l Symp. Methodologies for Intelligent Systems, Lecture Notes in Artificial Intelligence, No. 1325, Springer-Verlag, 1997.

[9] 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

***Ben Smith*** *is a member of the Artificial Intelligence group at JP. He is Deputy Project Element Manager for RAX, 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.*

***William Millar*** *is a computer scientist at the NASA Ames Research Center, and holds a degree in pure mathematics from the University of Waterloo. He is leading the MIR testing effort and designed the automated test controller.*

***Julia Dunphy*** *received her Master’s and Bachelor’s degrees in Physics and Mathematics from Cambridge University, UK, (’63) and her doctorate in Theoretical Physics from Stanford in ‘67. After a career in magnetic recording research in the 70s, she switched to software development and was a cofounder of a small company which provided development software for the then infant microcomputing industry. She now works as a contractor to JPL in the areas of design research and network computing. Her interests include the field of collaborative engineering design infrastructures and automatic source code generation. She holds several patents and has published over two dozen papers in various areas, such as magnetic recording, error-correction coding, and control of robotic vehicles (for the Mars Pathfinder Rover).*

***Yu-Wen Tung*** *is a member of the RAX Executive team at JPL, where he is leading the Executive testing effort. He received his M.S. and Ph.D. degrees in Electrical Engineering with an emphasis in Computer Engineering from the University of Southern California. His research interests include spacecraft software engineering, autonomy, computer graphics and parallel computing. He is a reviewer for the ACM Computing Review Journal.*

***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.*

***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.*

***Micah Clark*** *was a member of the RAX system testing team as a co-op student at the Rensselaer Polytechnic Institute, and is now a Member of the Technical Staff at JPL where he continues to work on the Remote Agent Experiment.*