Challenges and Methods in Testing the Remote Agent Planner

Ben Smith
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
benjamin.smith@jpl.nasa.gov

Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
martin.s.feather@jpl.nasa.gov

Nicola Muscettola
NASA Ames Research Center
MS 269-2
Moffet Field, CA 94035
mus@ptolemy.arc.nasa.gov

Abstract

The Remote Agent Experiment (RAX) on the Deep Space 1 (DS1) mission was the first time that an artificially intelligent agent controlled a NASA spacecraft. One of the key components of the remote agent is an on-board planner. Since there was no opportunity for human intervention between plan generation and execution, extensive testing was required to ensure that the planner would not endanger the spacecraft by producing an incorrect plan, or by not producing a plan at all.

The testing process raised many challenging issues, several of which remain open. The planner and domain model are complex, with billions of possible inputs and outputs. How does one obtain adequate coverage with a reasonable number of test cases? How does one even measure coverage for a planner? How does one determine plan correctness? Other issues arise from developing a planner in the context of a larger operations-oriented project, such as limited workforce and changing domain models, interfaces and requirements. As planning systems are fielded in mission-critical applications, it becomes increasingly important to address these issues.

This paper describes the major issues that we encountered while testing the Remote Agent planner, how we addressed them, and what issues remain open.

Introduction

As planning systems are fielded in operational environments, especially mission-critical ones such as spacecraft commanding, validation of those systems becomes increasingly important. Verification and validation of mission-critical systems is an area of much research and practice, but little of that is applicable to planning systems.

Our experience in validating the Remote Agent planner for operations on board DS1 raised a number of key issues, some of which we have addressed and many of which remain open. The purpose of this paper is to share those experiences and methods with the planning community at large, and to highlight important areas for future research.

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

At the highest level there are two ways that a planner can fail. It can fail to generate a plan within stated time bounds[1] (*converge*), or it can generate an incorrect plan.

Plans are correct if they command the spacecraft in a manner that is consistent with accepted requirements. If the domain model entails the requirements, and the planner enforces the model, then the plans will be correct. One must also validate the requirements themselves to be sure they are complete and correct.

Ideally we would *prove* that the domain model entails the requirements: that is, prove that the model will always (never) generate plans in which particular conditions hold. This may be possible for some requirements, but is almost certainly undecidable in general.

A more practicable approach, and the one we used for RAX, is empirical testing. We first had spacecraft engineers review the English requirements for completeness and accuracy. We then generated several plans from the model and developed an automated test oracle to determine whether they satisfied the requirements as expressed in first order predicate logic. A second (trivial) oracle checked for convergence. If all of the test cases converge, and the test cases are a representative sample of the possible output plans (i.e., have good coverage), then we have high confidence that the planner will generate correct plans for all inputs.

The key issue in empirical testing is obtaining adequate coverage (confidence) within the available testing resources. This requires a combination of strong test selection methods that maximize the coverage for a given number of cases, and strong automation methods that reduce the per-test cost. Complex systems such as planners can potentially require huge numbers of test cases with correspondingly high testing costs, so this issue is particularly critical for planners.

We developed a number of test automation tools, but it still required six work-weeks to run and analyze 289 cases. This high per-test cost was largely due to human bottlenecks in analyzing results and modifying the test

[1] Since the search space is exponential there will always be inputs for which a plan exists but cannot be found within the time limit. Testing needs to show that the planner will converge for all of the most likely inputs and a high proportion of the remaining ones.

cases and automations in response to domain model changes. This paper identifies the bottlenecks and suggests some ways of eliminating them.

With only 289 cases it was impossible to test the planner as broadly as we would have liked. To keep the test suite manageable we focused the test effort on the baseline goal set most likely to be used in operation. This strategy yields high confidence in inputs around the baseline but very low confidence in other goal sets. This risk is appropriate when there is a baseline scenario that changes slowly and becomes fixed in advance of operations, as is common in space missions. Late changes to the baseline could uncover new bugs at a stage where there is insufficient time to fix them.

This risk could be reduced with formal coverage metrics. Such metrics can identify coverage gaps. Even if there are insufficient test resources to plug those gaps the tester can at least address the most critical gaps with a few key tests, or inform the project manager as to which inputs to avoid. Coverage metrics also enable the tester to maximize the coverage of a fixed number of tests.

To our knowledge no such metrics exist for planning systems and we did not have time to develop one of our own for testing RAX. Instead we selected cases according to an informal coverage metric. Since test adequacy could only be assessed subjectively we used more cases than were probably necessary in order to reduce the risk of coverage gaps. Formal coverage metrics for planning systems are sorely needed to provide objective risk assessments and to maximize coverage.

The rest of this paper is organized as follows. We first describe the RAX planner and domain model. We then discuss the test case selection strategy, the effectiveness of that strategy, and the opportunities for future research into coverage metrics and test selection strategies. We then discuss the test automations we employed, the demands for human involvement that limited their effectiveness, and suggest automations and process improvements that could mitigate these factors. We conclude with an evaluation of the overall effectiveness of the Remote Agent planner testing, and summarize the most important open issues for planner testing in general.

RAX Planner

The Remote Agent planner (Muscettola *et al.* 1997) is one of four components of the Remote Agent (Nayak *et al.* 1999; Bernard *et al.* 1998). The other components are the Executive (EXEC) (Pell *et al.* 1997), Mission Manager (MM), and Mode Identification and Reconfiguration (MIR) (Williams & Nayak 1996).

When the Remote Agent is given a "start" command the EXEC puts the spacecraft in a special idle state, in which it can remain indefinitely without harming the spacecraft, and requests a plan. The request consists of the desired plan start time and the current state of the spacecraft. The desired start time is the current time plus the amount of time allocated for generating a plan (as determined by a parameter, and typically between one and four hours).

The Mission Manager extracts goals from the *mission profile*, which contains all the goals for the experiment and spans several plan horizons. A special *waypoint* goal marks the end of each horizon. The MM extracts goals between the required start time and the next waypoint goal in the profile. These are combined with the initial state. The MM invokes the planner with this combined initial state and the requested plan start time.

The planner expands the initial state into a conflict-free plan using a heuristic chronological backtracking search. During the search the planner obtains additional inputs from two on-board software modules, the navigator (NAV) and the attitude control subsystem (ACS). These are also referred to as "plan experts." When the planner decides to decompose certain navigation goal into subgoals, it invokes a NAV function that returns the subgoals as a function of the goal parameters. The planner queries ACS for the duration and legality of turn activities as a function of the turn start time and end-points.

The fundamental plan unit is a *token*. These can represent goals, activities, spacecraft states, and resources. Each token has a start and end timepoint and zero or more arguments. The tokens exist on parallel *timelines*, which describe the temporal evolution of some state or resource, or the activities and goals related to a particular state. Some RAX timelines are attitude, camera mode, and power. The domain model defines the token types and the temporal and parameter constraints that must hold among them.

Nominal Execution. If the planner generates a plan the EXEC executes it. Under nominal conditions the plan is executed successfully and the EXEC requests a new plan. This plan starts at the end of the current plan, which is also the start of the next waypoint in the profile.

Off-nominal Execution. If a fault occurs during execution, and the EXEC cannot recover from it, it terminates the plan and achieves an idle state. This removes the immediate threat of the fault. Depending on the failure, it may only be able to achieve a degraded idle state (e.g., the camera switch is stuck in the off position). It then requests a new plan that achieves the remaining goals from the achieved idle state. As with other requests, the required start time is the current time plus the time allowed for planning.

RAX DS1 Domain Model. The domain model encodes the knowledge for commanding a subset of the DS1 mission known as "active cruise" that consists of firing the ion propulsion (IPS) engine continuously for long periods, punctuated every few days by optical navigation (op-nav) images and communication activities.

Goal Type	Arguments
waypoint	HZN,EXPT_START,EXPT_END
navigate	frequency (int), duration (int). slack (int)
Comm	none
power_estimate	amount (0-2500)
exec_activity	type, file, int, int, bool
sep_segment	vector (int), level (0-15)
max_thrust	duration (0-inf)
image_goal	target (int), exposures (0-20), exp. duration (0-15)

Table 1: Goals

state timeline	initial values
EXEC_ACTIVITY	0,1,2
ATTITUDE	Earth, image, thrust vector
MICAS_SWITCH	ready, off
MICAS_HEALTHY	true, false

Table 2: Variable Initial State Timelines

The goals defined by the domain model are shown in Table 1. The initial state consists of an initial token for each of the timelines in the model. The legal start tokens for most timelines are fixed. Table 2 shows the non-fixed timelines and the set of legal start tokens for each one. Finally, the domain model defines 11 executable activities for commanding the IPS engine and MICAS camera, slewing (turning) the spacecraft, and injecting simulated faults. The latter allow RAX to demonstrate fault recovery capabilities, since actual faults were unlikely to occur during the experiment.

Test Selection Strategy

The key test selection issue is achieving adequate coverage with a manageable number of cases. Test selection should ideally be guided by a coverage metric in order to ensure test adequacy. Coverage metrics generally identify equivalence classes of inputs that result in qualitatively similar behavior with respect to the requirement being verified. A set of tests has full coverage with respect to the metric if it exercises the test artifact on one input from each class.

The verification and validation literature is full of coverage metrics for mission-critical systems (e.g., code coverage), but to our knowledge there are no coverage metrics specifically suited to planning systems. The most relevant metrics are those for verifying expert system rule bases. The idea is to backward chain through the rule base to identify inputs that would result in qualitatively different diagnoses (e.g., (O'Keefe & O'Leary 1993)). Planners have more complex search engines with correspondingly complex mappings, and a much richer input/output space. It is unclear how to invert that mapping in a way that produces a reasonable number of cases.

Since we did not have a planner-specific coverage metric for RAX, we instead used a black-box approach that has been successful in several conventional systems. The idea is to characterize the inputs as an n-dimensional parameter space and use orthogonal arrays (Cohen *et al.* 1996) to select a manageable number of cases that exercises all pair-wise combinations of parameter values. These tests can be augmented as needed with selected higher-order combinations. Since the number of pair-wise cases is logarithmic in the number of parameters, systems with eighty or more parameters can be tested with just a few hundred test cases. Specifically, the number of cases is proportional to $(v/2)\log_2 k$ for k parameters, each with v values (Cohen *et al.* 1997).

One disadvantage of this all-pairs selection strategy is that each test case differs from the others and from the nominal baseline input in several parameter values. That often made it difficult to determine why a test case failed, especially when the planner failed to converge.

To address this problem we created a second test set in which each case differed in only one parameter value from the nominal baseline, which was known to produce a valid plan. This "all-values" test set exercised each parameter value at least once. If one of these cases failed, it was obviously due to the single changed parameter. Its similarity to the baseline case made it easier to identify the causal defect. Analysis of failed all-pairs cases was simplified by initially diagnosing them with the same causal defects as failed all-values cases with which they shared a parameter value. Further diagnosis was undertaken only if the all-pairs case still failed after fixing the bug.

The reduction in analysis cost comes at the expense of additional test cases. The all-values test set grows linearly in the number of parameter values. Specifically, there are $1 + \Sigma_{i=1}^{n}(v_i - 1)$ cases for n parameters where parameter i has v_i values.

RAX Test Selection

We now discuss how the all-pairs and all-values test selection strategies were employed for RAX. The planner has the following inputs: a set of goals, which are specified in a mission profile and by the on-board navigator; an initial state; a plan start time; slew durations as provided by the ACS plan expert; and two planner parameters—a seed for the pseudo-random number generator that selects among non-deterministic choices in the search, and "exec latency" which controls the minimum duration of executable activities.

Each of these inputs is specified as a vector of one or more parameter values. The goals and initial states are specified by several parameters, and the other inputs are specified by a single parameter each. Several of the parameters, such as plan start time, have infinite or very large domains. It is clearly infeasible to test all of these values, so we selected a small subset that we expected to lie at key boundary points. This selection was *ad hoc* based on the intuition of a test engineer

id	Parameter	Values Tested	Range
1	experiment start	3	integer
2	plan start	10	integer
3	profile	12h, 6day, 2day	same
4	random seed	3 seeds	integer
5	exec latency	1, 4, 10	0-10
6	micas switch	off, ready	same
7	micas healthy	true, false	same
8	micas healthy (prior plan)	true, false, n/a	same
9	attitude	SEP, Image, Earth	same
10	end last thrust	-2d, -1d, -6h	integer
11	end last window	-2d, -1d, 0	integer
12	window duration	1,2,3,4,6 hours	integer
13	window start	0, 1, 2, 4	integer
14	targets/window	2, 20	0-20
15	images/target	3, 4, 5	3-5
16	image duration	1, 8, 16	1-16
17	SEP goals	6 configurations	infinite
18	SEP thrust level	6, 12, 14	15
19	SPE	1500,2400,2500	2500
20	slew duration	30, 120, 300, 400, 600, 1200	30-1200

Table 3: Tested Parameters

familiar with the domain model, or simply high, middle, and low values in the absence of any strong intuition. Table 3 shows the full list of parameters, the range of values each can take, and the subset of those values tested.

The initial state input consists of one token for each of the initial state timelines, with the exception of the MICAS health timeline which can initially have two adjacent tokens if the health changed while executing the prior plan. Parameters 6-9 specify the initial tokens and arguments for each of the non-fixed timelines shown in Table 2 . The initial tokens can start at or before the plan start. With the exception of the SEP and navigation window timelines the start times have no impact on planning and are set to the plan start for testing. Parameters 10 and 11 control the start times of the two exceptional tokens.

There are two goal inputs: the mission profile and the goals requested by the onboard navigator. The navigator goals specify the IPS thrusting it needs to achieved the desired trajectory, and the asteroid images it needs to determine the spacecraft position along that trajectory. This input is specified by Parameters 14-18.

The mission profile input is specified by Parameters 12, 13, and 19. These generate mutations of the two baseline mission profiles that we expected to use in operations: a 12 hour confidence-building profile that contained a single optical navigation goal and no IPS thrusting goals, and a six day primary profile that contained all of the goal types in Table 1. The mutations

id	Parameter	Constraint Sets (required values) 1	2	3
2	plan start	0	$\neq$ 3 days	3 days
3	profile	12-hr	6-day	6-day
8	micas healthy (prior plan)	none	none	*
9	attitude	Earth	Earth	*
10	end prior thrust	0	0	*
11	end prior window	0	0	*
17	SEP goals	null goal	*	*
18	SEP thrust level	0	*	*

Table 4: Constraint Sets

were designed to cover possible changes to the least stable elements of the profiles. Since the profiles are finalized prior to operations, and we had control over their contents, focusing on mutations of these profiles seemed a reasonable strategy. As it turned out, for operational reasons out of our control the profile had to be changed radically at the last minute. We reduced the horizon from six days to two, deleted five goals and changed the parameters and absolute (but not relative) placement of others. The goal types and overall profile structure remained the same. Fortunately, no new bugs were exposed by the new profiles since there would have been little time to fix them. Testing a broader range of profiles would have mitigated that risk. Broader test strategies are discussed in the next section.

The final input, specified by Parameter 20, is the duration of spacecraft slews (turns) computed by the attitude control planning expert (APE). The planner invokes APE to determine the duration of each slew activity as a function of the turn end-points and the spacecraft position at the start of the turn. Since position over time (trajectory) is not known until flight, we had to test the range of possible slew durations.

RAX operational requirements imposed three constraints among the parameter values as shown in Table 4. The test generator considered these constraints to avoid generating impossible cases. Constraint set one enforces the operational requirement that plans generated from the 12 hour profile will never have SEP goals, will start at the horizon start, and will have one of the four RAX idle states as the initial state. The second and third constraints enforce the following requirement. The plan start time is always one of the horizon boundaries (horizon waypoint goals) except when the exec requests a replan after a plan failure. In that case the exec first achieves one of the four RAX idle states, which becomes the initial state for the replan. So if the plan start is not a horizon boundary, constraint set two restricts the initial state parameters to the four idle states. When the plan start is at the horizon boundary for the six-day plan, all initial states are possible. This situation is reflected by the third constraint set.

	1	2	3	Total
all-pairs	24	61	41	126
all-values	23	51	45	119

Table 5: Test Set Sizes

The all-pairs and all-values test cases were generated automatically from the parameters and constraints described above. The constraints were satisfied by generating one test set for each constraint set. The sizes of the resulting test sets are shown in Table 5. These were augmented by 22 cases to exercise the planner interfaces.

Test Effectiveness

The selected tests were ultimately successful in that the on-board planner exhibited no faults during the experiment, and the tests provided the DS1 flight managers with enough confidence to approve RAX for execution on DS1. However we still have no objective measure of the delivered reliability. It seems likely that there were a number of coverage gaps, though again we have no way to measure that objectively. This section makes some informed guesses as to where those gaps might be and suggests some ways of addressing them.

Effectiveness metrics needed. Objective metrics are needed to evaluate new and existing test strategies. Defect coverage can only be estimated since the actual number of defects is unknowable. One method is to inject faults according to the estimated defect distribution for a system and evaluate how well different test strategies detect them. Another possibility is to evaluate test selection strategies against various planner coverage metrics. This assumes higher test coverage is correlated with higher defect coverage, which is not necessarily true. Empirical data from several planning applications would be needed to confirm the correlation.

Value selection was *ad hoc*. Many parameters had large or infinite domains, and so only a few of those could be tested. That selection was *ad hoc*, based primarily on the tester's intuition. This undoubtedly left coverage gaps. One way to close the gap is to select values more intelligently based on a coverage metric. The metric would partition the values into equivalence classes that would exercise the domain model in qualitatively different ways. This would ensure adequate coverage while minimizing the number of values per parameter, and therefore minimizing the number of test cases.

For example, one bug detected serendipitously during integration depended upon the specific values of three continuous parameters: the time to start up the IPS engine, the time to the next optical navigation window, and the duration of the turn from the IPS attitude to the first asteroid. An equation relating these parameters can crisply identify the boundary values that should be exercised. Value selection based solely on the tester's intuition is likely to miss many such interactions. Some possible metrics are discussed below.

bugs found	pairs - values	all values	pairs + values	other
convergence	12	20	32	1
correctness	5	37	42	9
interface	3	25	28	24
engine	5	22	27	30
total	25	104	129	64

Table 6: Defect Coverage by Test Type

Was all-pairs testing sufficient? All pairs testing will detect any bug exercised by one or two parameter values, but only some bugs exercised by interactions of three or more parameters. For example, the IPS bug discussed above was an interaction among three parameter values and was not detected by all-pairs testing. An open question is whether or not these bugs represent a significant fraction of the total defects. Assuming they are significant, the next open question is how to detect them with a manageable number of test cases.

Table 6 shows the defects detected by the all-pairs, all-values, and other tests. The other tests include a set of 22 interface tests and bugs discovered incidentally during development. The all-pairs and all-values tests detected 88% of the correctness and convergence bugs, but only half of the interface and engine bugs. This data also show that all-pairs testing detects only 20% more bugs than all-values testing alone. One reason for this sub-linear increase may be that many defects are exercised by many parameter value combinations, so that testing all values will find them. The table also shows that all-values misses more convergence bugs than correctness bugs with respect to all-pairs. This might be because convergence bugs are often caused by interactions among several domain constraints and are therefore less likely to be exercised by a single value.

These results suggest that defect detection increases sub-linearly with the number, m, of parameter combinations tested. That is, all-pairs detects fewer new bugs than all-values, and all-triples detects even fewer new bugs. The point of diminishing returns is probably reached at some small value of m. An effective strategy might therefore be to test m-wise combinations of parameter values and augment these with a few carefully selected higher order combinations. The test set should still be tractable for small m since the number of cases is proportional to $(v/2)\log_m k$ for k parameters with v values each. Selection of additional cases would have to be guided by a coverage metric based in turn on a formal analysis of the domain model. Tester intuition is probably insufficient, as evidenced in value selection.

	Constraint Set			Total
	1	2	3	
all-pairs, $v = 3$	43	65	67	175
all-pairs, $v = 5$	103	116	118	337
all-values, $v = 3$	270	303	311	884
all-values, $v = 5$	526	583	591	1700

Table 7: Test Set Sizes for All Goal-Pairs

A coverage metric could also help select a value for m that best balanced coverage against number of cases.

Broader goal coverage needed. RAX planner testing focused on mutations of the baseline profile. Bugs exercised only by other goal sets would not have been detected. For example, transitioning from the 6 day scenario to the 2 day scenario compressed the schedule and eliminated the slack time between activities. This led to increased backtracking which caused new convergence failures.

Exercising the full goal space would eliminate this coverage gap. It is also necessary for future missions, which must be confident that any goal set (profile) they provide will produce a valid plan. The challenge is how to provide this coverage with a manageable number of test cases.

One possibility is to create parameters that could specify any mission profile and perform all-pairs testing on this space. This would require at least one parameter for the start time, end time, and arguments for up to k instances of each goal type. For $k = 3$ the RAX model would require 140 parameters. These would replace parameters 12-19 of Table 3. Testing $v = 3$ values for each parameter would require 175 cases, and $v = 5$ values would require 337 cases as shown in Table 7.

This indicates that all-pairs testing of the full goal space is feasible, and that all-values testing might be feasible with sufficient test resources. Some additional issues would still need to be addressed, though these are relatively straightforward. First, profiles must be generated automatically from parameter values. There were only a few profile mutations for RAX so they were generated manually, but this is infeasible when there are hundreds of cases.

Second, some parameter vectors specify unachievable or impossible goal sets that would never occur in practice. These cases have to be automatically identified and eliminated to avoid the high analysis cost of discriminating test cases that failed due to impossible goals from those that failed due to a defect. Determining whether an arbitrary goal set is illegal is at least as difficult as planning, but it should be possible to detect many classes of illegal goals with simpler algorithms (e.g., eliminate goals that are mutually exclusive with any one or two domain constraints).

Although all-pairs testing of this parameter space is feasible, it may or may not be effective. As discussed above it may be necessary to test every k-wise combinations of parameter values for some small $k \geq 2$, and/or create additional cases to exercise key goal interactions.

Formal Coverage Metrics Needed

Formal coverage metrics are sorely needed for planner validation. Metrics based on analyses of the domain model can indicate which parameter values and goal combinations are likely to exercise the domain model in qualitatively different ways. Formal metrics can identify coverage gaps and inform cost-risk assessments (number of cases vs. coverage).

Formal coverage metrics, such as code coverage, have been developed for critical systems but to our knowledge no metrics have been developed for measuring coverage of a planner domain model. This is clearly an area for future research. A few possibilities are discussed below.

Constraint coverage. One possible coverage metric is the number of domain model constraints exercised. This is analogous to a code coverage metric. For a given plan, it determines which constraints it uses, and how those constraints were instantiated. A good test suite should exercise each instantiation of each constraint at least once.

Goal-Interaction coverage. This coverage metric is targeted at exercising combinations of strongly interacting goals. Since testing all combinations is intractable, the idea is to analyze the domain model to determine how the goals interact, and only test goal combinations that yield qualitatively different conflicts. For example, if goals A and B used power, we would test cases where power is oversubscribed by several A goals, by several B goals, and by a combination of both goals.. The coverage could be adjusted to balance risk against number of cases. One could limit the coverage to interactions above a given strength threshold.

This metric would extend on prior work on detecting goal interactions in planners to improve up the planning search, such as STATIC (Etzioni 1993), Alpine (Knoblock 1994) and Universal Plans (Schoppers 1987). STATIC generates a problem solving graph from the constraints and identifies search control rules for avoiding goal interactions. Alpine identifies interactions to find non-interacting sub-problems, and universal plans (Schoppers 87) derive reactive control rules from pair-wise goal interactions. These methods are designed for STRIPS-like planning systems and would have to be extended to deal with metric time and aggregate resources, both of which are crucial for spacecraft applications. One of the authors (Smith) is currently pursuing research in this area.

Slack metric. Another approach being pursued by one of us (Muscettola) is to select plan start times by analyzing the slack in the baseline plans. This approach

Task	Effort
Update/debug cases, tools	3.0
Run cases and analyzers	0.1
Review analyzer output	1.5
File bug reports	0.5
Close bugs	0.5
Total	5.6

Table 8: Test Effort in Work Weeks by Task

was used to manually select plan start times once the final baseline was frozen just prior to the experiment.

Using our knowledge of the PS model, we manually identified boundary times at which the topology of the plans would change. We identified 25 such boundary times and generated a total of 88 test cases corresponding to plans starting at, near, or between boundary times. This led to the discovery of two new bugs. Furthermore, analysis of the test results showed that PS would fail to find a plan at only 0.5% of all possible start times.

Test Automation

Automation played a key role in testing the Remote Agent planner. It was used for generating tests, running tests, and checking test results for convergence and plan correctness. Even so, the demand for human involvement was high enough to limit the number of test cases to just under three hundred per six week test period, or an average of ten cases per work-day.

The biggest demand for human involvement was updating the test cases and infrastructure following changes to the planner inputs, such as the domain model and mission profile. The next largest effort was in analyzing the test results. The test effort by task is shown in Table 8. This section discusses the automations that we found effective, the human bottlenecks, and opportunities for further automation.

Testing Tasks

The Remote Agent software, including the planner, was released for testing every six to eight weeks. The planner was exercised on the full set of test cases. A typical test cycle consisted of the following activities.

The tester updates the set of test cases as required by any changes to the planner input space. The test harness is updated to accommodate any new inputs or interface changes. The harness invokes the planner on each test case and collects the output. The tester makes sure that the cases ran properly, and re-runs any that failed for irrelevant reasons (e.g., the ACS simulator did not start).

The test results are analyzed by two oracles. The first checks for convergence, and the second for plan correctness. The oracles say that a requirement failed, but not why it failed. The tester reviews the output to determine the apparent cause and files a bug report.

Finally, the analyst confirms purported bug fixes from the previous release as reported in the bug-tracking database. Each bug has one or more supporting cases. The analyst determines whether those cases passed, or whether the bug is still open. In some instances, the tester may have to devise additional tests to confirm the bug fix.

Test Automation Tools

We employed several test automation tools for validating the Remote Agent planner, which are summarized below.

- **Test Harness.** The harness invokes the planner with the inputs for a given test case. Since the planner is embedded in RAX the harness invokes the planner by hijacking the RAX internal planner interfaces, which are primarily file and socket based. It converts the parameter values for each test case to input files: an initial state file, a planner parameter file (seed and latency), and a parameter file for the ACS and NAV simulators. The mission profiles were too difficult to generate automatically and were constructed by hand. The remaining inputs are sent over socket connections. After running the planner it collects the output, which consists of the plan file (if any), time spent planning, search trace, the initial state generated by the mission manager, and the simulator and harness output.
- **Plan Correctness Oracle.** The oracle reads a plan into an assertions database and then verifies that the assertions satisfy requirements expressed in first order predicate logic (FOPL). This tool (Feather 1998; Feather & Smith. 1999) was implemented in AP5, a language that supports these kinds of FOPL operations.

  The oracle also verified that the plan engine enforced the plan model by automatically converting the domain constraints into equivalent FOPL statements and checking the plan against them. Constraints 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." This maps onto an equivalent FOPL requirement: $A \rightarrow B \wedge R(A, B)$.

Analysis Costs

The two analysis tasks are determining whether a test case has failed, and why. The first task was performed by automated test oracles. A trivial oracle tested for convergence: if the planner generates a plan within the time limit the test cases passes, otherwise it fails. Automatically determining plan correctness requires a more sophisticated oracle. The oracle must measure correctness against some specification. The model is one such specification, but there is little point in validating the model against itself. We need a second specification that can be easily validated by spacecraft experts. We developed a small set of requirements in first order predicate logic (FOPL) and had spacecraft experts validate

their English translations. The automated test oracle determined whether individual plans satisfied those FOPL statements. This approach is not foolproof: the requirements may be incomplete, or their English translations may be incorrect. Methods to deal with these vulnerabilities are needed.

Once the oracles have identified the failed test cases, the next analysis task is to determine *why* they failed. For each failed test case, the analyst determines the apparent cause of the failure. Cases with similar causes are filed as a single bug report. This initial diagnosis provides guidance for finding the underlying bug, and is critical for tracking progress. If the analyst simply stated that the planner failed to generate a plan on the following fifty test cases, there could be fifty underlying bugs or just one. The initial diagnoses provide a much better estimate of the number of outstanding bugs.

Analyzing the test cases took eight to ten work-days for a typical test cycle and were largely unautomated. To determine why a plan failed to converge the analyst looked for excessive backtracking in the search trace or compared it to traces from similar cases that converged. Plan correctness failures also required review, although it was somewhat simpler (2-3 days vs. 8-10) since the incorrect plan provided context and the oracle identified the offending plan elements.

Automated diagnosis could reduce these efforts, especially for determining why the planner failed to generate a plan. There has been some work in this area that could be applied or extended. Howe (Howe & Cohen 1995) performed statistical analyses of the planner trace to identify applications of repair operators to states that were strongly correlated with failures. Chien (Chien 1998) allowed the planner to generate a plan, when it was otherwise unable to, by ignoring problematic constraints. Analysts were able to diagnose the underlying problem more quickly in the context of the resulting plan.

Analysis costs could also be reduced by only running and analyzing tests that exercise those parts of the domain model that have changed since the last release. One would need to know which parts of the domain model each test was intended to exercise. This information is not currently provided by the all-pairs strategy, but could be provided by a coverage metric: a test is intended to exercise whatever parts of the model it covers. A differencing algorithm could then determine what parts of the model had changed, where the "parts" are defined by the coverage metric.

This capability would also allow one to assess the cost of testing proposed model changes. This is an important factor in deciding how (or even whether) to fix a bug near delivery, and in assessing which fixes or changes to include in a release.

Impact of Model and Interface Changes

About half of the test effort in each cycle were the result of changes to the planner inputs and interfaces, which includes changes to token definitions in the domain model and changes to the baseline mission profiles. Modifications to the interfaces necessitated corresponding changes to the test harness, and sometimes to the input parameter table from which the test cases were generated. Most of the effort was spent not in making these changes, which generally took just a day or two, but in debugging them and identifying undocumented changes.

Version	days	tokens	parameters		relations	
			chg	new	chg	new
009	6	27		34		13
011	6		1		6	
015	6		0		1	
019	6		0		10	-1
026	6	+8	1	+9	8	+9
029	6			+5	0	
FLT 03	6		1		0	
FLT 05	2	-12,+3	10	0	15	-7
FLT 07	2		5		7	+2

Table 9: Profile Evolution

Unimplemented or incorrectly implemented changes resulted in test cases with unintended planner inputs. These inputs could cause a test case to fail when it would have succeeded with the intended inputs, or to succeed when it would have failed. Some of these were obvious, and detected by dry runs with a few test cases. Others were more subtle and not detected until the analysis phase, at which point the cases had to be rerun and re-analyzed after fixing the harness.

The planner interfaces consisted of the experiment and plan start times, and files for the initial state, profile, planner parameters (seed and exec latency), and simulator parameters (for NAV and ACS). One or more of these changed for every release, sometimes without notice. The instability of the interfaces was largely due to the tight development schedule combined with parallel development and testing.

The profile and initial state inputs were particularly mutable as shown in Table 9 and Table 10. These files are comprised of tokens, and if these token definitions change in the domain model, these input files must also change.

This experience indicates that it is preferable to maintain stable interfaces throughout testing if at all possible. If interfaces must change, they should do so infrequently and any decision to change them must consider the test impact. Appropriate software engineering practices can help minimize interface changes. Automation can also help reduce the impact of changes when they do occur. We present a few possibilities below. The first two were used successfully for RAX.

Private Parameters. To minimize the impact from token parameter changes, we created the notion of a *private* parameter in the domain specification language.

Token	Type	Public	Private	Version
navigate	goal	+3		026
op nav window	goal		+8	026
no op nav	goal		+4	026
waypoint	goal		+1	026
no op nav	goal		+1	027
op nav window	goal		+1	029
exec goal	goal	+1		029
sep timer idle	init		−1	019
SEP standby	init		+1	019
no activity	init		+1	029
micas health	init		+1	029
MICAS ready	init		−1	029
RCS turn	internal	+5		019
sep turn	internal	+1		026
exec activity	internal		+1	029

Table 10: Token Parameter Changes

These were used when new parameters were added to propagate values needed by new domain constraints or heuristics, the most common reason for adding new parameters to the model. Private parameters do not appear in the initial state or profile, but are added automatically by the MM. Their values are set automatically by propagation from other parameters. This reduced the number of impactful parameter changes from 30 to 10 as shown in Table 10.

Special Test Interfaces. To reduce the impact of changes to the initial state tokens and the format of the initial state file, both of which changed frequently, we negotiated an alternative testing interface to the initial-state generating function in the EXEC code. The test harness constructed an initial state by sending appropriate inputs to those functions, which then created the initial state in the correct format with the correct tokens. The idea of negotiating stable testing interfaces applies to testing complex systems in general, and should ideally be considered during the design phase.

Automated Input Legality Checks. The effort of identifying unintended mission profile and initial state inputs could have been greatly reduced by automatically checking their legality. One could imagine automating these checks by using an abstraction of the domain model to determine whether a set of goals are achievable from the specified initial state.

The debugging effort for other inputs could have been reduced in a similar fashion. The syntax and semantics of each input file could be formally specified and automatically verified against that specification. This would have detected interface changes (the input files would be invalid) and eliminated most of the debugging effort by detecting input file inconsistencies early and automatically.

Conclusions

The main requirements for the Remote Agent planner were to generate a plan within the time limit, and that the plan be correct. The validation approach for the Remote Agent planner was to invoke the planner on several test cases, and automatically check the results for convergence and plan correctness. Correctness was measured against a set of requirements developed by the planning team and validated by system and subsystem engineers. The cases were selected according to an "all-pairs" selection strategy that exercised all pairs of input parameter values. The selected values were at key boundary points and extrema. They were selected informally, based on the tester's knowledge of the domain model.

The tests focused on mutations of the two baseline mission profiles (goal sets) we expected to use in operations. This approach reduced the number of test cases, but was vulnerable to late changes to the baseline. A better approach would be to exercise the goal space more completely. There are a number of open research opportunities in this area. Formal coverage metrics are sorely needed for planners. Such metrics could guide the test selection and inform decisions on balancing risk (coverage) against cost (number of cases). Clever metrics may also be able to reduce the number of cases needed for a given level of coverage as compared to the straightforward metrics used for the Remote Agent planner. Finally, effectiveness metrics are needed to estimate and compare the defect coverage of various test strategies.

The number of manageable cases could be increased by reducing the demand for human involvement. Analysis costs were high because of the need to provide initial diagnoses for cases where the planner failed to generate a plan, and the need to review the plan checker's output. Changes to the planner interfaces, including changes to the model, also created an overhead for updating and debugging the test harness. We suggested a number of ways to mitigate these factors.

Automated diagnosis methods would eliminate one bottleneck, especially methods for determining why no plan was generated. Methods for identifying illegal inputs, especially illegal goals and initial states, would eliminate some of the test-case debugging effort, as would process improvements for limiting interface changes.

The Remote Agent was a real-world, mission-critical planning application. Our experience in validating the Remote Agent planner raised a number of key issues. We addressed several of these, but many issues remain open. As planning systems are increasingly fielded in critical applications the importance of resolving these issues grows as well. Hopefully the Remote Agent experience will spark new research in this important area.

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 Remote Agent Experiment team and the other two members of the test team, Todd Turco and Anita Govindjee.

References

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

Chien, S. 1998. Static and completion analysis for knowledge acquisition, validation and maintenance of planning knowledge bases. *International Journal of Human-Computer Studies* 48:499–519.

Cohen, D.; Dalal, S.; Parelius, J.; and Patton, G. 1996. The combinatorial design approach to automatic test generation. *IEEE Software* 13(5):83–89.

Cohen, D.; Dalal, S.; Fredman, M.; and Patton, G. 1997. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23(7):437–444.

Etzioni, O. 1993. Acquiring search control knowledge via static analysis. *Artificial Intelligence* 62:255–302.

Feather, M., and Smith., B. 1999. Automatic generation of test oracles: From pilot studies to applications. In *Proceedings of the Fourteenth International Conference on Automated Software Engineering (ASE-99)*, 63–72. Cocoa Beach, FL: IEEE Computer Society. Best Paper.

Feather, M. 1998. Rapid application of lightweight formal methods for consistency analysis. *IEEE Transactions on Software Engineering* 24(11):949–959.

Howe, A. E., and Cohen, P. R. 1995. Understanding planner behavior. *Artificial Intelligence* 76(2):125–166.

Knoblock, C. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68(2).

Muscettola, N.; Smith, B.; Chien, C.; Fry, C.; Rajan, K.; Mohan, S.; Rabideau, G.; and Yan, D. 1997. Onboard planning for the new millennium deep space one spacecraft. In *Proceedings of the 1997 IEEE Aerospace Conference*, volume 1, 303–318.

Nayak, P.; Bernard, D.; Dorais, G.; Gamble, E.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Rajan, K.; Rouquette, N.; Smith, B.; Taylor, W.; and Tung, Y. 1999. Validating the ds1 remote agent. In *International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS-99)*.

O'Keefe, R., and O'Leary, D. 1993. Expert system verification and validation: a survey and tutorial. *AI Review* 7:3–42.

Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1997. Robust periodic planning and execution for autonomous spacecraft. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*.

Schoppers, M. 1987. Universal plans for reactive robots in unpredictable environments. In *IJCAI 87*.

Williams, B., and Nayak, P. 1996. A model-based approach to reactive self-configuring systems. In *Proceedings of the thirteenth national conference on artificial intelligence (AAAI-96)*, 971–978.