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 any plan.

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?

As planning systems are fielded in mission-critical applications, it becomes increasingly important to address these issues. We describe the major issues 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.

At the highest level there are two ways that a planner can fail. It can fail to generate a plan (*converge*) within stated time bounds, or it can generate an incorrect plan. We used empirical testing to detect these kinds of failures. We ran the planner on several inputs and used 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.

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.

The RAX test selection strategy required 289 cases to adequately exercise a narrow space of inputs similar to those we expected to see in operation. This was sufficient for the RA experiment, but broader coverage—with correspondingly more test cases— would be needed for operational profiles outside of the experiment scope. 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. Better test selection strategies and more powerful automation methods are needed to permit broader coverage for a reasonable cost. This paper identifies several open issues in these areas, and suggests ways to address them.

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 and the open test selection issues. We then discuss the test automations employed for RAX, 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). The other components are the Executive (EXEC), Mission Manager (MM), and Mode Identification and Reconfiguration (MIR).

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.

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. 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.

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 (opnav) images and communication activities.

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.

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

Test Selection Strategy

The key test selection issue is achieving adequate coverage with a manageable number of cases. One selection strategy is to analyze the domain model to identify input values that would fully exercise the model according to some coverage metric. Although the validation and verification literature is full of coverage metrics for conventional systems, to our knowledge no such metrics exist for planner domain models.

Having neither a metric nor the time to devise one, we instead used a black-box selection 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 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. Large input spaces can be tested tractably since the number of pair-wise cases grows only logarithmically in the number of parameters—proportional to $(v/2)\log_2 k$ for k parameters with v values each (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

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

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, and its similarity to the baseline case made it easier to identify the causal defect. 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: $1 + n(v-1)$ for n parameters with v values each.

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 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 values tested.

Parameters 6-11 specify the initial state, Parameters 14-18 specify the IPS thrusting and MICAS imaging goals requested by the onboard Navigator, and Parameter 20 specifies the duration of spacecraft slew (turns) activities in the plan as computed by the attitude control planning expert (APE). This Parameters 12, 13, and 19 specify the mission profile input. 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 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, the profile had to be changed radically at the last minute for operational reasons. 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.

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

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.

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. An additional 22 cases exercised the planner interfaces.

id	Parameter	Constraint Sets (req'd 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

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

Table 5: Test Set Sizes

Test Selection Challenges

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. Objective metrics are needed to evaluate new and existing test strategies. 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.

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.

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 3 values for each parameter would require 175 cases, and 5 values would require 337. All-values testing would require 884 and 1700 cases respectively.

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. Mission profiles would need to be generated automatically from the parameter values since manual generation is infeasible. One issue is that 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 is subject to the same effectiveness issues as the narrower all-pairs testing. That is, there could be coverage gaps from ad hoc value selection, and from not testing higher-order parameter combinations. Coverage metrics would help answer that question.

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. 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 not immediately obvious how to invert that mapping in a way that produces a reasonable number of cases.

One possible metric would be to measure the number and strength of goal interactions exercised by the test cases. 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

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 6: Test Effort in Work Weeks by Task

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) and Alpine (Knoblock 1994). 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.

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 6. This section discusses the automations that we found effective, the human bottlenecks, and opportunities for further automation.

Test Automation Tools

We employed two test automation tools: a test harness, a plan correctness oracle, and a trivial plan convergence oracle (the case succeeds if and only if the planner creates a plan within the time limit). The test harness converted the parameters in each test case into planner inputs, ran the planner on them, and saved the output. The full test suite could be run automatically in about 16 hours. The plan correctness oracle (Feather & Smith. 1999) reads a plan into an assertions database and then verifies that the assertions satisfy requirements expressed in first order predicate logic (FOPL). This tool was critical since inspecting plans manually for correctness would have been prohibitively time consuming and error prone.

Analysis Costs

The two analysis tasks are determining whether a test case has failed, and why. The first task was performed by the automated test oracles. 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.

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 effort of 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.

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. The test harness and test cases then had to be updated to support the new inputs. Making these changes only required a day or two. The bulk of the effort was caused by undocumented interface changes which managed to creep into most of the software releases. Planner inputs that were correct before the change could be incorrect after it, resulting in cases that fail when they should have succeeded or vice versa. Some of these errors 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 re-run and re-analyzed after fixing the harness.

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. These were used when 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. Their values are set automatically by propagation from other parameters. This reduced the number of impactful parameter changes from 30 to 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.

Conclusions

The main requirements for the Remote Agent planner were to generate a plan within the time limit, and that the plan be correct. These requirements were verified by running the planner on several input cases and automatically checking the results for convergence and plan correctness. Correctness was measured against a set of requirements reviewed 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 was sufficient for the experiment, but may not scale to broader operational contexts. Formal planner coverage metrics are sorely needed to make the best use of available cases and objectively balance risk (coverage) against cost.

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.

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

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.; 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.

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. On-board 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.