A Parametric Testing Environment for Finding the Operational Envelopes of Simulated Guidance Algorithms

Anthony C. Barrett
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, M/S 301-260
Pasadena, CA 91109

ABSTRACT

The ever-increasing size and complexity of aerospace systems often drive developers to validate using modeling and Monte Carlo simulations confined around expected points of operation in a hyper-dimensional parameter space. This paper describes an alternative that explores large regions of the parameter space with explicit coverage guarantees, searching for *n* parameter relations that characterize a system's performance envelope.

KEY WORDS: Parametric testing, treatment learning, simulation, performance envelope, hyper-dimensional test space, support vector machine.

INTRODUCTION

As aerospace systems increase in size and complexity, they are often validated using modeling and Monte Carlo simulations confined around expected points of operation. This limited exploration of a system's possible operating region is due to each simulation's large number of input parameters as well as the arbitrarily complex interplay of parameters that affect an aerospace system's performance in simulation. Our testing infrastructure widens this exploration by using several synergistic techniques for systematically testing large areas of the parameter-space and analyzing the results to find critical parameter interactions. The components of the testing infrastructure described here are illustrated in figure 1. First the test case generator is given a set of initial coverage requirements. With these requirements, a set of tests are computed and fed to a test runner, which interacts with a system simulation. We use a simple mechanism for summarizing and classifying tests from simulation logs, and a result analyzer takes classified tests and determines variable interactions that drive simulations to target behaviors.

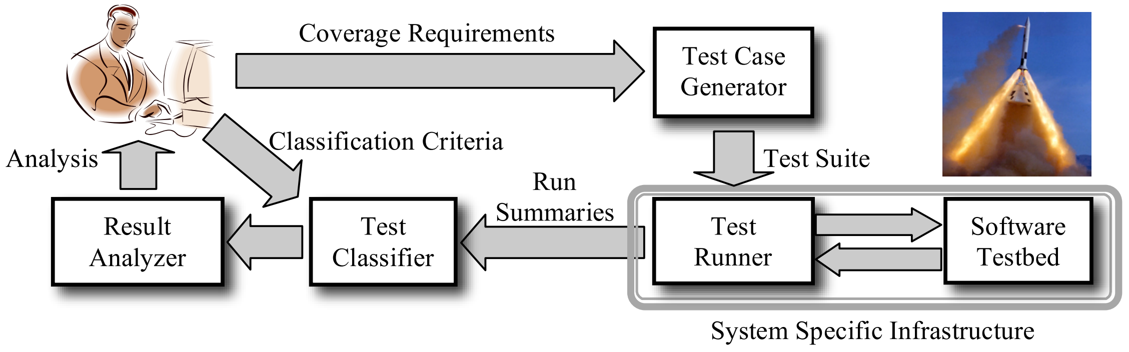

Figure 1: The components of the test infrastructure and how they interact during mixed-initiative testing

More precisely, the test case generator is based on n-way combinatorial testing among targeted sets of parameters (Barrett and Dvorak 200). The output of the test case generator a suite of tests that exhibits user targeted coverage guarantees over lower dimensional projections of the full parameter space. This facilitates providing coverage guarantees for finding interactions among relatively small numbers of parameters with limited numbers of simulations. The test suite is then fed to a test runner that feeds each test to the simulation and then summarizes the results for subsequent classification. Given classified tests with targeted coverage guarantees, we use both treatment learning with TAR3 (Menzies and Hu, 2003) and support vector machine learning with SVM-light (Joachims 1999) to determine simple relationships between parameters that define an aerospace system's operational envelope. In addition to learning rules the system displays the results in 2 or 3 dimensional scatter plots for visualizing features of the operational envelope.

The currently implemented testing environment is a set of interacting shell scripts that coordinate the components to work with any simulation developed with the TRICK simulation development toolkit (Paddock et al. 2003) by replacing its Monte Carlo testing facility. This paper's running example came from work with the Advanced NASA Technology Architecture for Exploration Studies (ANTARES) launch abort simulation (Williams-Hayes 2007), a TRICK-based testbed.

The next four sections describe a four-step loop for exploring a test space. First the test space needs to be either defined or redefined based on prior analysis. Second, a set of classified tests that satisfy some coverage criteria need to be added to the space. Third, analysis algorithms are applied to the to search for destructive parameter interactions. Once these interactions are known, visualization provides human insight that may lead back to the first step with a redefinition of the test space. Finally, after presenting the four-step loop, this paper ends by describing related work and future directions.

DEFINING THE TEST SPACE

Defining the test space starts with the system being tested, which comes in the form of an implemented test simulation. This simulation has K input parameters whose combined settings determine how the system will evolve over time, and each simulation run generates a sequence of M-element vectors characterizing that evolution. For instance, the model of a rocket launch has a number of input parameters (e.g. payload mass, propellant mass, launch time of day, wind velocity, etcetera). All of these parameters combine in a simulation to determine where that rocket would be, how fast it would be moving, its current mass, and other values for every instant of time after take off. Given this more precise definition of a test simulation, the task becomes a matter of defining a way to classify whether or not a simulation violates a requirement, and determine which combinations of input parameters drive the simulation to violate the requirement. As such, defining the test space involves specifying the possible inputs and how to classify the output simulation.

Initially, the system simply took the K input parameters and ranges of allowed values for each, but that proved to be too simplistic. Working with the simulations rapidly showed that some parameters are redundant. Instead of editing a rather complex simulation to remove redundant input parameters, enforced equality constraints became a useful feature. Thus a text file defined the initial test space, and each line defined an input parameter in terms of either a range of acceptable values or some other pre-defined input parameter. While other constraints are possible, equality was enough for our current needs.

With the input parameters defined, the next issue involves how to classify test results. Each test simulation result is a log file with a sequence of state vectors, but the objective of running a simulation is to classify its defining test vector. Thus a sequence of thousands of state vectors needs to be condensed into an element from a finite set of classes. This condensation takes the form of a two-step process (Figure 2) where each simulation log file is first analyzed to generate a few floating-point numbers that summarize its contained behavior. Given these summaries, the suite of test vectors is collectively analyzed to generate classifications. In both steps, expressions of the following extended Backus-Naur form (EBNF) are used to perform the condensation.

```
expression = variable “:” summary “{“ equation [ “:” test ] “}”
summary = “max” | “min” | “first” | “last” | “sum” | “avg” | “high(” n “)”
```

In the case of the first step, each `variable` is a summary variable. The `equation` and `test` are respectively a floating-point equation and Boolean test over state vector elements. The semantics are to take the sequence of state vectors, throw away those that do not satisfy the optional test, apply the equation to each of the remaining test vectors to compute a sequence of floating point numbers. The `summary` directive is then applied to compute the value. When the directive is `max` or `min` this value is respectively the maximum or minimum of the sequence. Similarly, `first` and `last` respectively extract the first and last value from a sequence. Where `sum` adds all the elements, `avg` computes their average. The last directive is `high(n)`, which computes a value that separates the highest `n` percent of the elements from the rest. For example `high(50)` computes the median.

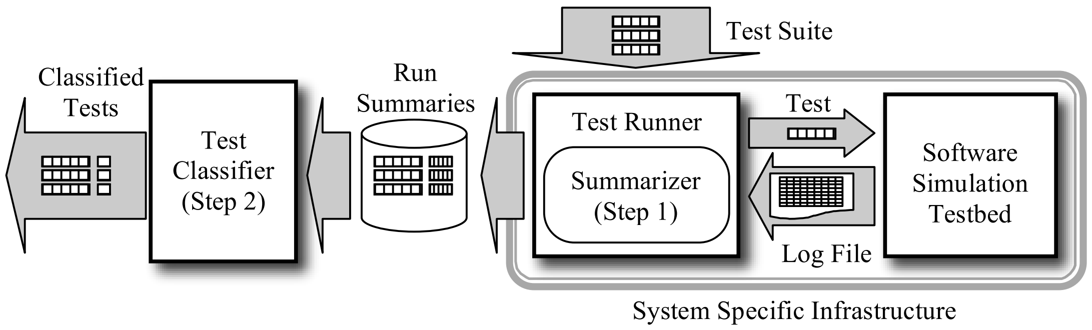

Figure 2: Test classification is a two-step process where each test is summarized in isolation and then all tests are collectively classified using summary information.

Where the first step computes the values of summary variables for each simulation from state vectors, the second step starts by computing global summary variables for all simulations from summary variables of each simulation. As such, the expressions for the second step have equations and tests over simulation summary variables, and the summary directive has the same effect. Finally, the second step ends by combining global summary variables with simulation summary variables to determine classes. The classification is defined using the following form where either the first successful test determines the `className` or the last `className` is used as a default.

```
classification = { className ":" test ";" } className ";"
```

The objective of this simple language is to give an analyst an easy way to classify tests and alter the classifications if needed. Also, the implementation in involves translating the classification code into C++ and compiling the result. This results in extremely fast classification and reclassification to support adjustments motivated by discovering properties in the test space.

The principle of keeping things simple and fast diverges from classical classification literature by motivating fast simple tools for mixed initiative analysis. In general, the focus of this work is to keep things as simple and usable as possible. The subsequent sections more complex components, but in each case they are very controllable and generate simple results.

ADDING TESTS

As previously mentioned, our environment uses a combinatorial alternative to random testing, which enables coverage guarantees with relatively few tests. For instance combinatorial techniques enable exercising all interactions between pairs of twenty ten-valued inputs with only 212 tests. More precisely, any two values for any two parameters would appear in at least one of the 212 tests. While this number of tests is miniscule compared to 10^{20} possible exhaustive tests, anecdotal evidence suggests that they are enough to catch most coding errors. The underlying premise behind the combinatorial approach can be captured in the following four statements, where a factor is an input, single value perturbation, configuration, etc.

- The simplest programming errors are exposed by setting the value of a single parameter.
- The next simplest are triggered by two interacting parameters.
- Progressively more obscure bugs involve interactions between more parameters.
- Exhaustive testing involves trying all combinations of all parameters, but is often intractable.

So errors can be grouped into families depending on how many parameters need specific settings to exercise the error. The m-factor combinatorial approach guarantees that all errors involving the specific setting of m or fewer factors will be exercised by at least one test. In our effort to determine a system's performance envelope we likewise endeavor to characterize the boundary in terms of relationships between as few parameters as possible.

From a geometric viewpoint, system testing involves exploring a K dimensional feature space in order to find and characterize regions of interest. These regions denote those inputs that drive a simulation to violate one or more requirements. One approach is to perform a Monte Carlo test where the K input parameters are generated at random. While this works in principle, it requires a huge number of tests to explore the input space. The approach taken here is an alternative that uses extensions to pairwise testing, an approach that attempts to evenly explore interactions of pairs and larger collections of parameters with a relatively small number of tests. Essentially, a pairwise test implies that any projection of the generated tests onto a plane evenly covers that plane (Figure 3).

Given this geometric perspective, there is a natural tension between resolution and coverage when using a limited number of tests since the number of required tests is proportional to the product of the resolution and the coverage. For this reason testing typically starts with a broad coverage low resolution test suite (a large grid) and then continues with narrower-coverage higher-resolution grids to explore found regions of interest.

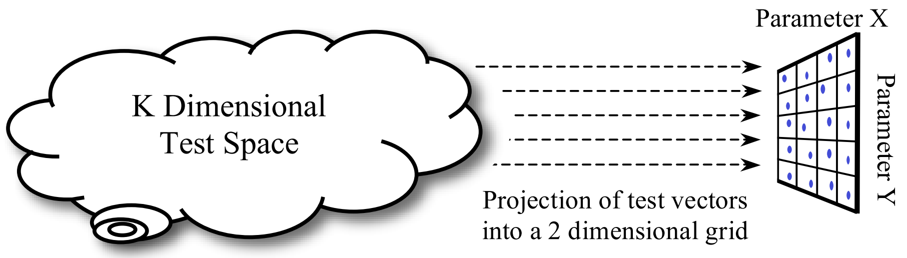

Figure 3: In pairwise testing, any projection of the generated K-parameter tests onto a two-parameter grid has at least one test in each grid box.

More precisely, a command with the following EBNF generates a test suite, where `granularity` and `nary` are integers that respectively specify the number of grid lines and the coverage dimensionality. The optional list of subsequent parameters is used to limit the coverage requirements to projections involving those stated parameters. Otherwise, coverage requirements for all projections are assumed.

```
    "addtests" granularity nary { parameter }
```

ANALYZING TESTS

Once the test space is populated by a set of classified tests, with instances of target and non-target classes, the issue is to determine the parameter settings that drive a test toward a target class. One way do this is to visually inspect all 2D and 3D projections in search of relationships, but that becomes arduous as the number of parameters increases. For instance, the Constellation Program's ANTARES launch abort simulation (Williams-Hayes 2007) has 143 input parameters, with 14 redundant. Given this number, there are 16,512 two-dimensional projections and 2,097,024 three-dimensional projections. Given this number, visual inspection is out of the question. Some analysis tools are needed to identify which projections to focus on, as well as identifying higher-dimensional relationships without simple visualizations.

In the infrastructure, an analyst makes this determination using the following `learn` command that invokes the TAR3 treatment learner. This command takes a set of classified tests and learns rules that emphasize the target class. It works by adaptively gridding each parameter into `granularity` ranges with equal numbers of tests. The learned rules refer to contiguous ranges in this grid and a rule can refer to one through `nary` parameters. Finally, each learned rule must match at least a specified `percent` of the tests with the target class.

```
    "learn" granularity nary percent
```

Textually, the returned rules have the following EBNF format, specifying conjuncts of range bounds on 1 to `nary` parameters. The `granularity` determines the values that each $\texttt{loBound}_x$ and $\texttt{hiBound}_x$ can take, and each rule must match the given `percent` of the target class. Finally, the `worth` is a measure of how much the rule increases the ratio of the target class over all tests that match the rule.

```
i “worth=” worth “[“ parameter_1 “=[“ loBound_1 “..” hiBound_1 “)]”
                 { “[“ parameter_i “=[“ loBound_i “..” hiBound_i “)]” }
```

Returning to our geometric viewpoint, this learner finds 1 through `nary` dimensional boxes and each box must contain more than percent of the tests with the last class. This means that making percent too small will result in learning rules that are too specific, resulting in finding numerous rules that are artifacts of the limited number of tests. On the other hand, making percent too large may result in not being able to learn any rules at all. A good rule of thumb is to see how to rules evolve when varying the percentage. Ultimately, the objective is to find useful projections of the test space onto 2 or 3 dimensions for visualization.

The motivation behind using TAR3 was a desire to learn simple approximate rules that can be visually inspected and easily understood. The sacrifice is that the learned rules are only approximate, but they are enough to focus a tester’s attention on the appropriate projections. For instance, after performing 66 tests with the ANTARES pad abort simulator, there was enough information to determine an interaction between the rotational inertias about the yaw and pitch axes of the launch abort system (figure 4). When the difference reaches a given threshold, the launch abort simulations start exhibiting undesirable accelerations that lead to system failures.

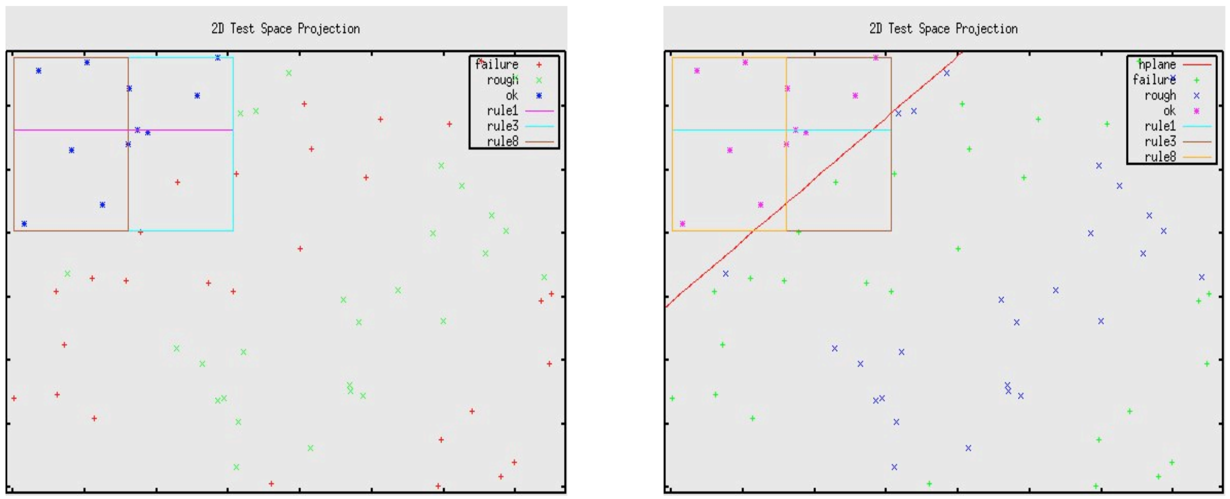

Figure 4: Projections of 66 classified tests onto two dimensions, showing how success depends on a relationship between two parameters as well as the hyperplane denoting that relation.

While the TAR3-based learning mechanism finds interacting variable settings that lead to target outcomes, more information is desired when linear interactions are observed. For this reason the svm_light support vector machine learner was integrated into the infrastructure using the following two commands for learning a linear relation and then simplifying it, where `ratio` is a number between 0 and 1.

```
“SVMlearn” ratio
“SVMsimplify” ratio
```

The first command takes all of the K-dimensional test points and attempts to learn a K parameter linear equation, a hyperplane. As in all support vector machine learners, the computed hyperplane is such that it maximizes the distance between the hyperplane and its closest positive and negative test points. This distance is called the "margin", and the floating-point `ratio` for `SVMlearn` denotes the how much error to accept in order to enlarge the margin. This argument primarily applies when the test points are not linearly separable and is typically set to 1 when the points are linearly separable.

When the hyperplane is computed it is typically too complex for direct inspection. For this instance the `SVMsimplify` command was created. It first determines the most relevant parameter and then removes all parameters that have a lesser impact as specified by `ratio`. For instance, applying `SVMlearn` to our 66 tests resulted in generating a 128-parameter hyperplane, but applying SVMsimplify to remove all parameters that have less than a 0.1 impact resulted in the two-parameter hyperplane displayed on the right of figure 4. While this hyperplane does separate the OK tests from all of the rest, the simplification appreciably reduced the separation margins, which hints at another lesser interacting parameter. Applying `SVMsimplify` with a 0.01 ratio resulted in regaining the original separation margin and determining a third parameter involving the rotational inertia of the crew module. This expanded relationship is illustrated in figure 5.

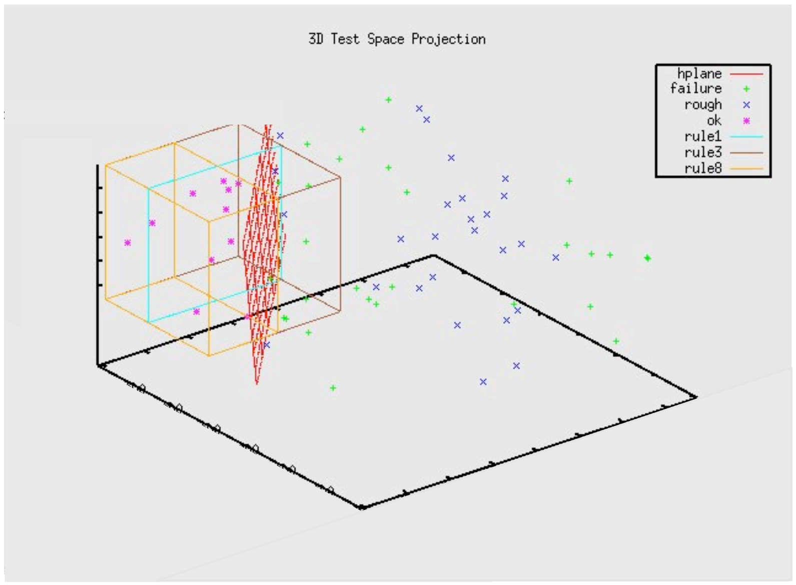

Figure 5: Projection onto three dimensions to illustrate possible minor participation of third parameter in driving simulation to failure.

RESPONDING TO DISCOVERIES

Once a margin on the performance envelope has been detected, adjusting the arguments to the learning commands facilitates improved characterization. Once an interesting constraint is determined, the natural question becomes, "Is that the only constraint?" At any point in the mixed initiative process, a test engineer has the option to add new tests and change the region of interest in the test space. The following two commands are used respectively to either alter the entire test space by either expanding it or contracting it or shift/restrict the test space with respect to a single parameter by explicitly altering its bounds. Both of these commands can mask out particular tests that fall outside the focus/restrict bounds. First, the `focus` command either expands or contracts the focus of the entire test space around its middle point. While factors greater than one expand the test space, factors less than one contract it. Invoking this command with the factor "1" will always reset the space to its original values. Second, the `restrict` command explicitly sets the range of a parameter, thus shifting the test space. These commands alter the operation of the learning and the projection commands to only consider the tests in the currently included region. Restricting a parameter can also return its bounds to the default focused parameter file settings by using a "`-`" instead of a number in the `loBound` or `hiBound` arguments.

```
    “focus” factor
    “restrict” parameter (loBound | “-“) (hiBound | “-“)
```

For instance, after characterizing the margin in figures 4 and 5, a test engineer can expand the test space in search of other margins. The left side of figure 6 illustrates the result of expanding the test space by a factor of two and the right side illustrates the result of adding more tests once the space is expanded. In this case, a second margin was detected in the upper left corner of the expanded test space projection. To further explore that margin, a test engineer can restrict the space to exclude the margin on the lower right and then add tests to determine the new margin.

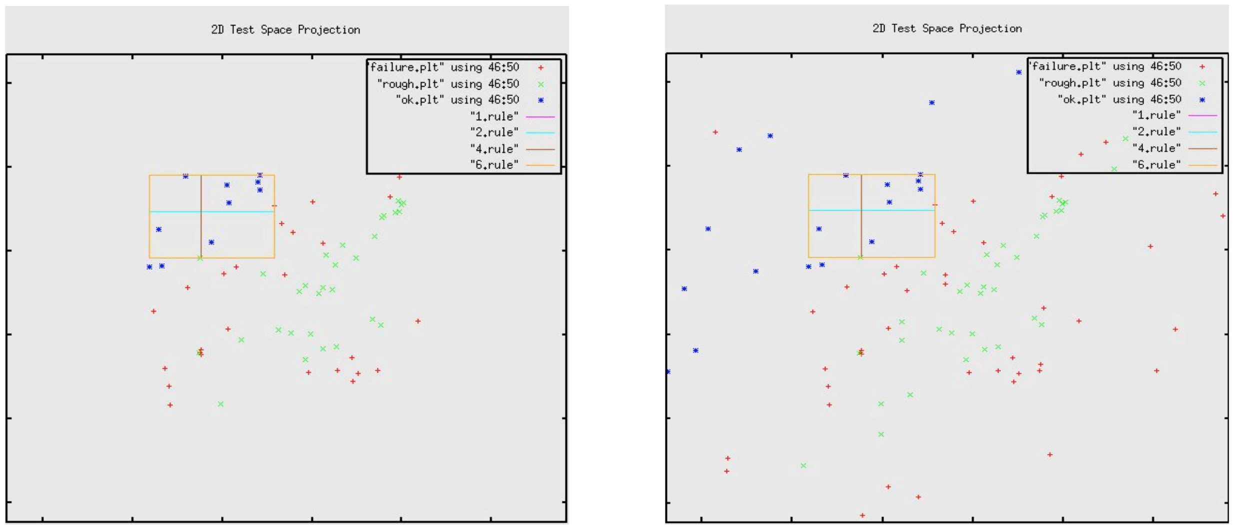

Figure 6: Projections of classified tests onto two dimensions after first focusing out by a factor of two and then adding more tests to find another performance envelope constraint above the OK region.

SUMMARY & CONCLUSIONS

Due to ever increasing size and complexity of aerospace systems, they are often validated using modeling and Monte Carlo simulations confined around expected points of operation. This paper presents an alternative approach based on combinatorial testing that facilitates searching for the envelope where a system is driven to failure. Through using combinatorial techniques, this approach uses relatively few tests to provide explicit coverage guarantees over large regions of a test space. These coverage guarantees synergistically combine with treatment learning techniques to facilitate discovering rules that characterize the envelope in terms of simple relationships with 2D and 3D visualizations. Finally, this work has been applied to finding features of the ANTARES pad abort system's performance envelop, and future work will focus on other simulation based analyses of aerospace systems.

ACKNOWLEDGEMENT

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The author would also like to thank Daniel Dvorak, Karen Gundy-Burlet, Tim Menzies, and Johann Schumann for discussions contributing to this effort.

REFERENCES

Barrett, Anthony and Daniel Dvorak, "A Combinatorial Test Suite Generator for Gray-Box Testing," In Proceedings of IEEE SMC-IT 2009, July 2009.

Joachims, Thorsten, "Making large-Scale SVM Learning Practical" In Advances in Kernel Methods – Support Vector Learning. B. Scholkopf, C. Burges, and A. Smola (editors) MIT-Press. 1999.

Menzies, Tim and Ying Hu, "Data Mining for Very Busy People," Computer, vol. 36, no. 11, pp. 22-29, November 2003.

Paddock, Eddie J., Alexander Lin, Keith Vetter, and Edwin Z. Crues, "TRICK®: A Simulation Development Toolkit," Proceedings of AIAA Modeling and Simulation Technologies Conference and Exhibit, August 2003.

Williams-Hayes, Peggy, "Crew Exploration Vehicle Launch Abort System Flight Test Overview." Proceedings of AIAA Guidance, Navigation and Control Conference and Exhibit. August 2007.

BIOGRAPHY

Dr. Anthony Barrett is a senior member of the Artificial Intelligence Group at the Jet Propulsion Laboratory, California Institute of Technology where his R&D activities involve planning & scheduling, execution, testing, diagnosis, and multi-agent coordination applied to autonomy for controlling clusters of spacecraft. He holds a B.S. in Physics, Computer Science, and Applied Mathematics from James Madison University and both an M.S. and PhD in Computer Science from the University of Washington. His research interests are in the areas of planning, scheduling, V&V, and execution/diagnosis in the context of single and multi-agent systems with a primary focus on autonomous space systems.