Test And Evaluation by Genetic Algorithms

Alan C. Schultz, John J. Grefenstette, and Kenneth A. De Jong
Navy Center for Applied Research in Artificial Intelligence

THIS APPROACH USES MACHINE LEARNING TECHNIQUES TO EVALUATE AUTONOMOUS-VEHICLE SOFTWARE CONTROLLERS. A SET OF SIMULATED FAULT SCENARIOS IS APPLIED TO A CONTROLLER, AND A GENETIC ALGORITHM SEARCHES FOR SIGNIFICANT COMBINATIONS OF FAULTS.

AUTONOMOUS VEHICLES WILL require sophisticated software controllers. Given a vehicle simulation and an intelligent controller for that vehicle, what methods are available for testing the controller's robustness? Validation and verification are not enough: The controller might perform as specified, but the specifications could be incorrect; that is, the vehicle might not behave as expected. Testing all possible situations is obviously intractable. And techniques for testing isolated, low-level controllers do not apply to the vehicle as a whole.[1]

Also, traditional controller tests are labor intensive and time consuming. Some methods require simulated vehicle missions to test the controller's robustness under various conditions. After designing a *fault scenario* that will cause particular failures during a simulated mission, the test engineers observe the vehicle's resulting behavior and then refine the fault scenario to better exercise the controller. This cycle is repeated until the engineers are confident that the vehicle's behavior will be appropriate in the field. Implicitly, the engineers are searching for interesting fault scenarios.

We propose a machine learning technique to automate this process: We subject a controller to an adaptively chosen set of fault scenarios in a vehicle simulator, and then use a genetic algorithm to search for fault combinations that produce noteworthy actions in the controller. We have applied this approach to find a minimal set of faults that produces degraded vehicle performance, and a maximal set of faults that can be tolerated without significant performance loss.

Fault scenarios

A *fault scenario* is a description of faults that can occur in a vehicle, and the conditions under which they will occur. It can also include information about the environment in which the vehicle is operating. Each scenario has two main parts (see Figure 1). The *initial conditions* give starting states for the vehicle and environment, such as initial speed, attitude, position, and so on. The *fault rules* map current conditions to fault modes.

Each fault rule also has two parts. The *triggers* on the left represent the conditions that must be met for the fault to occur. Each trigger measures some aspect of the current state of the vehicle, the environment, or other faults that might be activated at that time. A trigger has a low and a high value; if the measured quantity is within the trigger's range, that trigger is "satisfied." When all triggers in a rule are satisfied, the *fault mode* (the right side of the rule) is instantiated in the vehicle simulation. Each fault mode also has two parts: The *fault type* describes the vehicle subsystem that will fail, and the *fault level* describes the failure's severity. Thus, after

0885/9000/93/1000-0009 $3.00 © 1993 IEEE

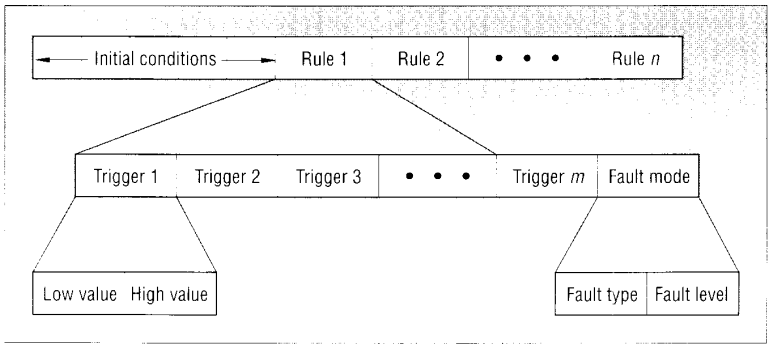


Figure 1. Representation of a fault scenario.

```
*** Initial conditions ************

Set   wind speed          = 8
Set   wind direction      = 58
Set   altitude            = 1,460
Set   distance            = 2,462
Set   horizontal offset   = 36
Set   velocity            = 121

*** Rule 1 ***********************

If    -63.00  <=  velocity [x]  <=   64.00  And
      -56.00  <=  velocity [y]  <=  -28.00  And
     -254.00  <=  velocity [z]  <= -224.00  And
   -16,220    <=  position [x]  <=  -860     And
       13     <=  position [y]  <=   832     And
    2,040     <=  position [z]  <=  2040     And
     -325     <=  pitch         <=   635     And
   -1,370     <=  yaw           <=  -100     And
     -900     <=  roll          <=  -857     And
        7     <=  elapsed time  <=  2,016    And
        0     <=  last fault    <=  2,044    And
        0     <=  thrust        <=    72     And
       16     <=  flaps         <=    40     And
                  M_drag         =   NA       And
                  C_flaps        =   NA       And
                  C_rudder       =   Clear    And
                  S_azimuth      =   NA       And
                  C_elevator     =   NA       And
                  S_elevation    =   NA       And
                  C_rollers      =   Set      And
                  S_roll         =   NA

Then  Set fault type = S_roll
      Set fault value = -0.232

*** Rule 2 ********************
If ...
```

Figure 2. Part of a fault scenario file.

reading the initial conditions and setting those variables, the system examines each rule at each simulated time step to see if any triggers are satisfied; if they are, it instantiates that rule's fault mode with the given amount of degradation.

Evaluating a fault scenario. When test engineers search for interesting fault scenarios, they apply an evaluation criterion to measure each scenario's usefulness. To automate the search process, we must explicitly define an evaluation function that can measure the fitness of each scenario. This can be difficult, because evaluation criteria are often based on informal judgments.

One approach measures the difference between the controller's actual performance in a scenario against some form of ideal response. The ideal response could be approximated based on knowledge of the causal assumption behind the fault mode (that is, a certain sensor has failed and should be recalibrated or ignored), or it could be based on an expert controller's actions, or it could simply be to return to nominal performance of the mission plan in the least amount of time. Computing the ideal response might require information that is not available to the controller. Although this approach is a more automated way to identify problem areas for the controller, it also requires a substantial effort to design software that can compute ideal responses.

A second approach measures fitness based on the likelihood and severity of fault conditions. The highest fitness indicates the most likely set of faults that cause the controller to degrade to a specified level. This approach is useful if probability estimates of the various fault modes are available when the evaluation function is being constructed. Unfortunately, many of the fault modes that long-endurance autonomous vehicles would encounter are of low probability and have the same order of magnitude. Therefore, this approach would not work in practice.

A third approach rewards fault scenarios occurring on the boundary of the controller's performance space. That is, a set of fault rules would receive a high fitness rating if it causes the controller to degrade sufficiently, but some minor variation would not. This fitness function would help us identify "hot spots" in the controller's performance space, but we would have to evaluate several scenarios for each fault specification, and each evaluation would require a complete mission simulation. Depending on the computation cost, this approach might not be feasible.

A fourth approach defines and searches

for *interesting* scenarios. There are several ways to define "interesting" in the context of an intelligent controller, each producing a separate function. One interesting class of scenarios are those in which minimal fault activity causes a mission failure or vehicle loss. The dual of that class comprises scenarios in which maximal fault activity still permits a high degree of mission success. Using this approach, we implemented an evaluation function for the automated testbed: Maximizing this function produces fault scenarios that yield a minimal level of fault activity, while minimizing this function finds fault scenarios that allow a high level of fault activity that can still be tolerated without significant performance loss.

Autonomous vehicles

Our end goal is to evaluate the robustness of an autonomous underwater vehicle, but the full-scale AUV was not available in the early stages of developing this technique. Our initial development therefore focused on a controller for an autonomous air vehicle, although the general method can be applied to AUVs as well. The domain involved a medium-fidelity, three-dimensional simulation of a jet aircraft that flies to and lands on an aircraft carrier and is guided by an intelligent controller. The simulation, called AutoAce, can also control environmental conditions, including constant wind and wind gusts.

The autonomous controller, which flies the aircraft and lands it on the deck, was designed using a subsumption-based architecture.[2] The controller is composed of individual behaviors, operating at different levels of abstraction, that communicate among themselves. Top-level behaviors include *fly-craft* and *land-craft*. At a lower level, behaviors include *fly-heading* and *fly-altitude*. The lowest level behaviors include *hold-pitch* and *adjust-roll*. After the initial design, optimization techniques were used to improve the controller such that it could successfully fly and land the aircraft in constant wind and in wind gusts.

Modeling faults. We introduced three classes of faults into the simulation. A *control fault* occurs when an actuator fails to perform an action commanded by the controller. In the control faults we modeled, the elevators, rudder, ailerons, or flaps might fail to reach a commanded angle. The fault level is a percentage of the total range for that actuator. For example, since the normal range for flaps is 0° to 40°, a 10-percent fault level is 4° positive error, while a fault level of −10 percent would yield an actuator set 4° lower than expected.

Sensor faults are failures of vehicle sensors or detectors: When the controller tries to read a sensor, it receives erroneous information either because of noise or sensor failure. For our experiments, we modeled pitch, yaw, and roll sensor faults. For example, a pitch sensor fault represents a failure in the sensor that determines the vehicle's current pitch in degrees from the horizon. The fault level for a sensor fault is expressed as a percentage of ±180°, since that is the total range these sensors might return. Thus, a fault degradation of −10 percent in the pitch sensor means that the reported value is 18 degrees below the actual one.

FOR THIS PROJECT, WE WERE MORE INTERESTED IN COLLECTING A LARGE NUMBER OF INTERESTING FAULT SCENARIOS THAN IN FINDING THE SINGLE, MOST INTERESTING ONE.

Model faults are not directly related to sensors or actuators; they usually involve physical aspects of the vehicle. For example, a leak in an AUV is a model fault. The current simulation has only one model fault, which represents increased drag due to vehicle damage. In general, the fault level depends on the type of model fault. In the case of drag, degradation is expressed as a percent increase in drag, from none to an amount that is reasonable in this domain.

In addition to these three classes, faults can be identified as persistent or nonpersistent. Persistent faults do not cease, while nonpersistent ones must be reinstantiated at each time step to continue. For example, actuators and sensors tend to have intermittent failures and can return to a fault-free state, so they would be modeled as nonpersistent. On the other hand, increased drag due to damage cannot be undone and is modeled as a persistent fault.

Trigger conditions. In our AutoAce experiments, each fault rule has 21 triggers:

(1-3) components of the velocity vector (x, y, and z);
(4-6) absolute position in space (x, y, and z);
(7-9) attitude (pitch, yaw, and roll);
(10) flap setting;
(11) thrust setting;
(12) elapsed time since mission began;
(13) time since last fault was instantiated; and
(14-21) current state of each fault: active, not active, or not important.

The rules are tested in each cycle to see if a fault should be instantiated. Figure 2 shows part of a fault scenario file for the AutoAce system.

Setting initial conditions. We modified the simulator to read a fault scenario file at startup. The first group of items in the file are the initial conditions, which describe the environment and the aircraft's starting configuration. We restricted the range of initial conditions so that no setting by itself could cause the vehicle to fail; all aircraft failures had to come from the instantiation of vehicle faults. When the simulation starts, the aircraft begins its mission approximately two nautical miles from the carrier and then proceeds to land. The initial conditions are

- constant wind speed (knots);
- wind direction (degrees);
- initial altitude (feet);
- initial distance from the carrier (nautical miles);
- horizontal offset (in feet), or how well the aircraft is lined up with the carrier (zero means it is perfectly lined up); and
- initial forward velocity (feet per second).

Genetic algorithms

To search for fault scenarios, we use a class of learning systems called *genetic algorithms* (described in this issue's introduction on pages 5-8 and elsewhere[3-7]). A GA simulates the dynamics of population genetics by maintaining a knowledge

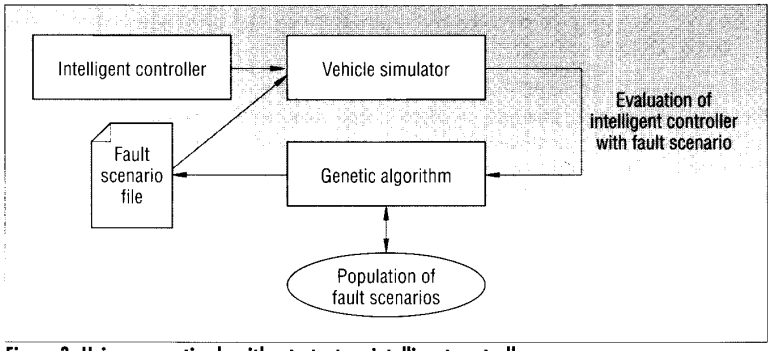


Figure 3. Using a genetic algorithm to test an intelligent controller.

base of fault scenarios that evolves over time in response to the vehicle's simulated performance. An evaluation function captures the structure's fitness, as described earlier. The search proceeds by repeatedly selecting fault scenarios from the current population based on fitness. That is, high-performing structures can be chosen several times for replication, while poorly performing structures might not be chosen at all. Next, the algorithm constructs plausible new fault scenarios *(offspring)* by applying idealized *genetic search operators* to the selected structures. For example, *crossover* exchanges pieces of scenario representations to create new offspring, and *mutation* makes small random changes to the scenarios. The GA evaluates the new fault scenarios in the next iteration (generation).

The application

Figure 3 shows how we used GAs to test controller performance. Given a vehicle simulator and an intelligent controller, the GA automatically evaluates many scenarios and searches for interesting ones.

When applying a GA to a particular problem, it is often necessary to tailor the algorithm to a chosen representation language, and to develop new genetic operators that take advantage of available domain knowledge. Using the fault scenario representation discussed earlier, each member of the population represents a single fault scenario, and has the form

$$s\ s\ \ldots\ s\ r\ r\ \ldots\ r$$

where each s is a simulation parameter specifying an initial condition, and each r is a fault rule of the form *trigger* & *trigger* &...& *trigger* ⇒ *fault mode*.

GAs are often used to find a single near-optimal point in a search space. For this project, we were more interested in collecting a large number of interesting fault scenarios than in finding the single, most interesting one. In our initial experiments, we accomplished this by stopping the GA when convergence reached a predefined level. This meant that the final population still represented a widely diverse set of fault scenarios. We also tried having the system record the best individuals from each generation. It is also possible to keep a record of all scenarios tested, and then apply a postprocessor to search for diverse, interesting ones.

Another important consideration involved generating an initial population of fault scenarios. To have enough active faults, we had to force a large degree of generality in the triggers of the initial population's fault rules. We introduced a parameter that adjusted the percentage of triggers that were initially set to their full range. By tuning this parameter, we ensured that all the initial scenarios had at least some fault activity. This gave the GA sufficient information to construct more interesting scenarios over the course of the run.

The evaluation function. A GA's fitness function measures the usefulness of arbitrary points in the search space defined by the representation language. For these experiments, we defined a function that gives high ratings to scenarios that induce the controller to perform interesting behaviors. Maximizing the evaluation function searches for controller failures in the face of minimal aircraft problems, thus identifying interesting weaknesses in the controller. On the other hand, minimizing the function searches for controller successes in light of significant vehicle failures. This allows us to characterize the controller's robustness with respect to general classes of faults.

Fault activity measures the level of faults introduced over the entire mission. To calculate this, we normalize the absolute value of the active fault levels during a given time step so that they are between 1 and 10, and then take the product:

$$current\ fault\ activity = \prod_{active\ rules}((|\ fault\ level|\times 9.0)+1.0)$$

Then we calculate the average fault activity over the entire mission:

$$fault\ activity = \frac{\sum_{time} current\ fault\ activity}{time}$$

Using such factors as the distance from center line, the roll angle at touchdown, and the velocity of descent, the simulator returns a score based on the quality of the landing:

$$score = \begin{cases} 1 & \text{if crash landing} \\ 2 & \text{if abort} \\ 3 \rightarrow 10 & \text{if safe landing} \end{cases}$$

Thus, a score of 10 indicates a perfect landing. We now combine the fault activity and the score:

$$eval = 1\ /\ (fault\ activity \times score)$$

With no faults and a crash landing (actually, this is impossible), *eval* returns 1, the maximum possible value. With maximal fault levels throughout the mission and a perfect landing, *eval* returns 0.01, the minimum possible value.

Experimental results. In all our experiments, we used a population size of 100 and ran the GA for 100 generations, resulting in 10,000 evaluations. We first maximized the fitness function to find several minimum-fault, maximum-failure scenarios. Figure 4 shows a learning curve for a representative experiment: The x axis

represents the number of trials, and the y axis represents the average value returned by the function for all fault scenarios in that generation. The GA quickly homed in on scenarios with high fitness, that is, scenarios where minimal fault activity led to controller failure.

By examining the scenarios identified by the GA as interesting, we drew the following conclusions about the controller:

- Roll control is most critical as the aircraft starts to touch down.
- Sensor errors are much harder to recover from than are control errors.
- Even slight increases of drag cause the controller to behave poorly.

Next, we minimized the evaluation function to search for successful flights despite significant vehicle failures. We were able to characterize the controller's robustness with respect to some general classes of faults:

- The GA again found that the controller can recover from control faults, but that sensor faults are much harder to handle.
- Recovering from faults that affect aircraft pitch is easier than recovering from faults affecting its roll. This agrees with the earlier observation.
- The GA identified situations in which it is possible for some faults to "cancel" out the effects of other faults (for example, positive sensor errors might offset negative control errors).

In more of a qualitative affirmation of our method, we showed the designer of the AutoAce controller some of the interesting scenarios generated by the GA. The designer acknowledged that these scenarios provided some insight into parts of the intelligent controller that could be improved. In particular, the scenarios as a group tended to indicate classes of weaknesses, as opposed to highlighting single weaknesses. This allows the designer to improve the controller's robustness over a class as opposed to only patching specific instances of problems.

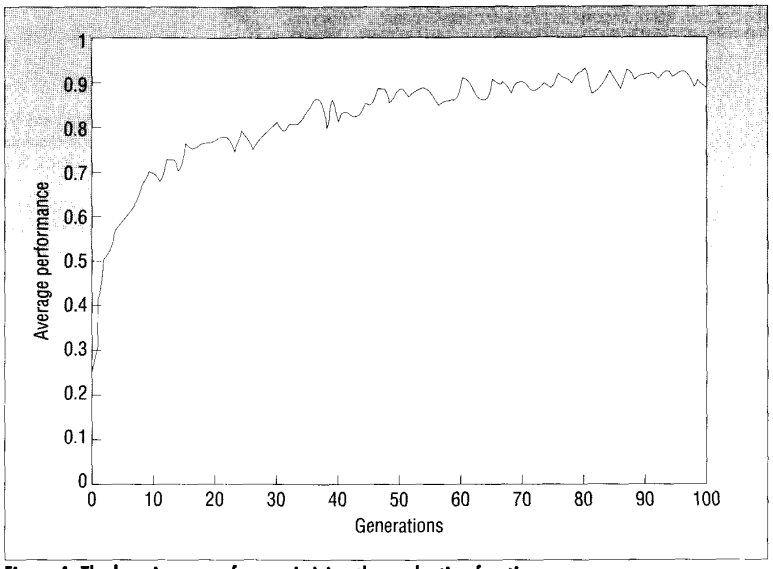


Figure 4. The learning curve for maximizing the evaluation function.

OUR APPROACH TO FITNESS, based on the extent to which fault activity influences mission performance, is promising. It offers advantages over manual testing of sophisticated software controllers, although it should supplement rather than replace other forms of software validation. The method can be applied to intelligent controllers for autonomous underwater, ground, or air vehicles; the basic approach stays the same.

We are also examining other evaluation functions to find more scenarios of interest to designers of vehicle controllers. We plan to apply these techniques to AUV controllers in the near future.

On another front, we are looking at changing the GA to improve its search for fault scenarios. One area we are examining is the use of *sharing functions* to help maintain diversity in the population.[8] These techniques should force the GA to cover some of the better solutions with a percentage of the population instead of converging to a single maximum, in essence making the population share the payoff. This is important in situations where the simulation time becomes excessively long.

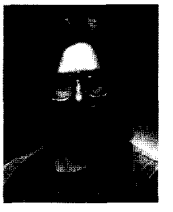
We have also begun to explore the use of heuristic mutation operators. For example, new operators will use information recorded during fault scenarios to trigger the generalization and specialization of fault rules. We expect this to result in more scenarios that are useful and good solutions that are identified more quickly.

References

1. B. Appleby, W. Bonnice, and N. Bedrossian, "Robustness Analysis Methods for Underwater Vehicle Control Systems," *Proc. Symp. on Autonomous Underwater Vehicle Technology*, IEEE, Piscataway, N.J., 1990, pp. 74-80.
2. R.L. Hartley and F.J. Pipitone, "Experiments with the Subsumption Architecture," *Proc. 1991 IEEE Int'l Conf. Robotics and Automation*, IEEE Comp. Soc. Press, Los Alamitos, Calif., 1991, pp. 1,652-1,659.
3. J.H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, 1975.
4. K.A. De Jong, "Adaptive System Design: A Genetic Approach," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-10, No. 9, 1980, pp. 566-574.
5. J.J. Grefenstette, C.L. Ramsey, and A.C. Schultz, "Learning Sequential Decision Rules Using Simulation Models and Competition," *Machine Learning*, Vol. 5, No. 4, Oct. 1990, pp. 355-381.
6. A.C. Schultz and J.J. Grefenstette, "Using a Genetic Algorithm to Learn Behaviors for Autonomous Vehicles," *Proc. AIAA Guidance, Navigation, and Control Conf.*, American Inst. for Aeronautics and Astronautics, Washington, D.C., 1992, pp. 739-749.
7. A.C. Schultz, "Using a Genetic Algorithm to Learn Strategies for Collision Avoidance

and Local Navigation," *Proc. Seventh Int'l Symp. Unmanned Untethered Submersible Technology*, Univ. of New Hampshire, Durham, N.H., 1991, pp. 213-225.

8. K. Deb and D.E. Goldberg, "An Investigation of Niche and Species Formation in Genetic Function Optimization," *Proc. Third Int'l Conf. Genetic Algorithms*, Lawrence Erlbaum Associates, Hillsdale, N.J., 1989, pp. 42-50.



Alan C. Schultz is a computer scientist in the Machine Learning Section at the Navy Center for Applied Research in Artificial Intelligence, which is part of the Naval Research Laboratory. He is also a doctoral candidate in information technology at George Mason University. His research involves genetic algorithms, learning in robotic systems, experience-based learning, and adaptive systems. He received his MS in computer science at George Mason University in 1988, and his BA in communications from American University in 1979. He is a member of IEEE, the IEEE Computer Society, AAAI, ACM, and the International Society for Genetic Algorithms.

John J. Grefenstette is head of the Machine Learning Section at the Navy Center for Applied Research in Artificial Intelligence. His research interests include machine learning, genetic algorithms, and autonomous systems. He serves on the editorial boards of *Machine Learning*, *Adaptive Behavior*, and *Evolutionary Computation*, and is a member of the IEEE Computer Society, AAAI, and the International Society for Genetic Algorithms. He received his BS in mathematics from Carnegie Mellon University in 1975 and his PhD in computer science from the University of Pittsburgh in 1980.

Kenneth A. De Jong is associate professor of computer science at George Mason University, and was with the Naval Research Laboratory when this research was performed. His research interests include adaptive systems, machine learning, expert systems, and knowledge representation. He is editor-in-chief of *Evolutionary Computation*. He received his BA in mathematics from Calvin College, and his MA in mathematics, MA in computer science, and PhD in computer science from the University of Michigan.

The authors can be reached in care of Alan Schultz at the Navy Center for Applied Research in Artificial Intelligence (Code 5514), Naval Research Laboratory, Washington, DC 20375-5000; e-mail, schultz@aic.nrl.navy.mil