2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S) | 979-8-3503-9570-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/DSN-S60304.2024.00037

# Active Learning Omnivariate Decision Trees for Fault Diagnosis in Robotic Systems

Casidhe Hutchison
*National Robotics Engineering Center*, *Carnegie Mellon University*,
fhutchin@nrec.ri.cmu.edu

Milda Zizyte
*National Robotics Engineering Center*, *Carnegie Mellon University*,
milda@brown.edu

David Guttendorf
*National Robotics Engineering Center*, *Carnegie Mellon University*,
dguttendorf@nrec.ri.cmu.edu

Claire Le Goues
*National Robotics Engineering Center*, *Carnegie Mellon University*,
clegoues@cs.cmu.edu

Philip Koopman
*National Robotics Engineering Center*, *Carnegie Mellon University*,
koopman@cmu.edu

***Abstract*—Robotic systems are highly complex, and debugging failures in them can prove challenging. We propose a technique for using multivariate decision trees to create human interpretable descriptions of the input conditions that cause these failures in robotics systems. This approach uses active learning to efficiently create tests, and uses a multivariate decision tree that captures common boundary conditions in the software fault space. We provide an evaluation of this technique on a small set of faults from several robotics systems, and compare it against a previous technique in this space, Hierarchical Product Set Learning. Our proposed technique requires fewer tests, provides more accurate estimates of the fault conditions, and is more interpretable than the prior approach.**

***Index Terms*—robustness, autonomous systems, software quality, software testing**

## I. Introduction

As robotic software becomes increasingly powerful and complex, it becomes harder to identify the cause of faults. This is especially true in autonomy systems where, rather than simple function calls, the system evolves via the exchange of messages with fields that can number in the thousands. In this context, automated debugging tools become an important tool for developers to be able to identify where a faulty behavior comes from. Code analysis tools, like delta debugging [32] and static analysis [11] [30] are well established automated tools that assist with finding bugs in code, reducing the workload on both developers and testers.

Machine learning tool kits, now widely available in open-source libraries, allow computers to analyze data and efficiently generate models of the underlying behavior. Using machine learning, tools can create fault models [31], analyze code for abberant behavior [12] [18], and even suggest possible fixes [25]. Fault models are an important component of debugging, as well as a important inputs to additional debugging tools [17] [22]. However existing techniques, such as spectrum-based [1] and mutation-based [24] fault localization techniques describe likely lines of code for the fault, rather than the shape of the fault, or are hard to interpret [31].

NAVAIR Public Release 2022-445. Distribution A: approved for public release. This work was funded by the Test Resource Management Center (TRMC) and Test & Evaluation/Science & Technology (T&E/S&T) program and/or the U.S. Army Contracting Command Orlando (ACC-ORL-OPB) under contract W900KK-16-C-0006

The National Robotics Engineering Center (NREC) specializes in the development and maturation of robotics systems for industry and government sponsors. Our team at NREC specializes in the development of safety-focused testing and debugging tools for these sponsors. Our research context has given us access to more than 20 confidential robotics systems at various states of their software development life cycle, as well as access to robotics developers in these organizations and elsewhere.

The key insight of our work is that the faults that are being modeled are expressed via code, and we can more accurately model the structure of a fault by growing a decision tree where the splits are equations that are likely to appear as code snippets. This both increases the ability of the approach to capture these kinds of faults and improves the human interpretablilty of the fault. The contributions of this paper are as follows:

- A novel approach to generating test cases for fault diagnosis
- A novel approach to creating multivariate decision trees for fault diagnosis
- A comparison of our technique against a previous SoTA approach [31] to fault diagnosis in autonomy systems

## II. Background

Our research context focuses on the robustness testing of autonomy software. Software robustness testing evaluates a system's ability to maintain adherence to safety properties in the presence of unexpected or invalid inputs [21]. As autonomy systems are often cyber-physical systems [16] [3], these systems are often safety critical, in that a failure to meet the safety properties could result in damage to the system, the operating environment, or even the operators. Since sensor failures and user errors will eventually occur during a system's

979-8-3503-9570-9/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN-S60304.2024.00037



Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on May 12,2026 at 15:16:11 UTC from IEEE Xplore. Restrictions apply.

lifetime, it's important that these systems handle these invalid inputs safely and gracefully [9].

This has a number of implications in the design of testing tools. First, rather than expected input-output pairs that are found in traditional software testing [28], robustness testing allows for a more general oracle that checks the safety properties of the system [5]. This allows us to label the entire output space as correct or incorrect, without needing to construct explicit pairs.

The second implication for testing comes from the fact that autonomy systems are temporal, and inputs consist of messages that come in over time, rather than the single function call found in traditional software. This means that tests take a long time to run [2], and thus we are limited in the number of data points we can gather. This in turn affects the types of analysis we can do. To address this, we implemented an active learning technique that focuses on ensuring each test will maximize the amount of information, at the cost of increased computational complexity during test case generation. This trade-off is sensible, as the time to execute a test still dominates the total runtime [2].

**HPSL.** Hierarchical Product Set Learning (HPSL) [31] is a prior technique in this space, addressing speed and accuracy of bug description. HPSL uses a binary tree search to create bug descriptions by running exploratory trials.

This approach has some benefits for its application. First, it is an active learning technique, which ensures that we're gaining information on every trial. Second, by using a curated list of inputs, we can efficiently search the space of values quickly. Finally, the binary search approach is very efficient. In fact, it is so efficient at identifying relevant fields that we reuse this portion of the HPSL approach in preparing the data for Active Learning Omnivariate Decision Trees (ALO-Trees).

This approach does, however, have several drawbacks.

1) It assumes field independence, which is not always the case for a fault. This assumption impacts the fault description, which is simply a list of values for each field. It is assumed if all the fields take on any value from their list, the fault will be triggered; this is inaccurate when there is a relation between fields.
2) Using a fixed list means that the fault description inherently lacks precision. For example, we can know that $x \leq 64$ does not trigger the fault, while $x \geq 128$ does, and a list of specific value for $x$ does not express these ranges.
3) The final step in the HPSL process is exhaustive with respect to field values, making it somewhat costly to run.

## III. Method

Our approach, Active Learning Omnivariate Decision Trees (ALO-Trees), uses an active learning algorithm to generate test cases to be run, and a decision tree algorithm to model the data collected. The active learning tries to select samples that will be most informative when refining the existing tree (e.g. samples close to the decision boundary). The decision tree is retrained, and the active learning step can then be run again, using the updated model. The active learning component and the fit of the decision tree are independent, and thus can be considered separately.

We use the `find_active_fields` function from HPSL to derive the bug-relevant fields. In our experiments, we ignore the shared tree-based search step when evaluating HPSL against ALO-Trees.

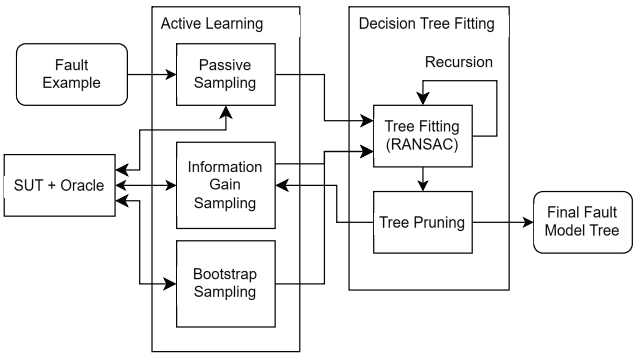


Fig. 1: The ALO-Trees architecture. The process starts with passive sampling, which produces initial data for the decision tree fitting. The tree is then used in active sampling to create new data points to refine the model.

### A. Active Learning

ALO-Trees uses active learning to create new test cases, aiming to maximize the information gained by running each test case. There are two potential avenues for increasing information:

**Exploitation** Create test cases that are near the decision boundaries of the model to refine the existing model.

**Exploration** Create test cases that are far away from the known data points to ensure that the model completely describes the behavior of the fault.

To balance the trade-off between exploration and exploitation, active sampling uses the supplied predictive model to label a potential trial, and then uses proximity to known trials to calculate an estimated uncertainty of the result if that trial were to be run. This uncertainty is used to calculate the potential information gain of running that trial. A point with high information gain is either very close to oppositely labeled points, or very far from similarly labeled points.

Estimating information gain requires a measure of uncertainty to minimize. However, predictor uncertainty is not terribly informative: it will be very confident in areas far away from the boundary, even when there is no data support. Instead, we calculate a heuristic uncertainty that the label assigned to a sample point by the predictor represents the true label based on two factors:

1) Uncertainty from lack of support, measured by the distance from known points with the same label.



Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on May 12,2026 at 15:16:11 UTC from IEEE Xplore. Restrictions apply.

2) Uncertainty from confusion: measured by distance to the set boundary (which can be approximated as the distance to known points that do not have that label).

By calculating these distances based on known points, this certainty estimation is tied closer to ground truth than if we were only using the predictor. The formula we use for estimating that a point will have the true label equivalent to its predicted label is:

$$P_{partition}(i|pred(i) = l) = \frac{1 + (1/d(i, T_l))}{2}$$

$$P_{boundary}(i|pred(i) = l) = \frac{d(i, \bar{T_L})}{d(i, T_L) + d(i, \bar{T_L})}$$

$$P = \sqrt{P_{partition} \cdot P_{boundary}}$$

Where $T_l$ is the set of inputs that have been tested and have the label $l$, $\bar{T_l}$ is the set of tested inputs that do not have the label $l$, and $d(i_1, i_2)$ is the euclidean distance in symmetric log space between two points $i$ ($d(i_0, I_1)$ represents the minimum distance to the set $I_1$; i.e. $min_{i_1 \in I_1} d(i_0, i_1)$). Our sampling approach is then to simply find the point with the highest entropy $H(x)$, based on this probability. As entropy is reduced to zero for a known point, the information gain at that point is equivalent to the entropy.

$$sample = \underset{\forall i \in I}{argmax}(H(i))$$

Figure 2 shows an example entropy map of the input space. Brighter areas correspond to higher entropy (those we'd like to sample). Note how that areas near the boundary are highly desirable, especially those close to oppositely labeled points. Note also that areas far from labels that support the prediction, such as the area to the top left of the prediction line, are likewise desirable to sample.

Creating test cases draws samples from an incredibly large input space. We therefore use symmetric log compression (an invertible operation) to shrink the sample space. While this decompression is lossy, representable floating point values are distributed logarithmically in Euclidean space, so the loss of precision is comparatively small.

Working with floating point numbers also means ALO-Trees needs to gracefully handle special values like INF and NaN. As this sampling approach is fundamentally geometric (computing distances) it cannot be used with such values. Removing such values from the data naively is undesirable, as it alters the fault characterization. We handle this using *domains*, or subsets of the input space with fixed special values. For example, consider the domain "the set of all inputs where $i[0]$ is NaN". All the points in this domain have a finite value at all indices except the first. We can create finite representations of all inputs in this space by excluding dimensions with special values, and then calculate information gain the usual way on the domain's samples. We reconstruct selected samples' full representation by reinserting special values.

Because information gain is calculated using only points in the domain, information gain values are not directly comparable. Additionally, domains suffer from state space explosion:

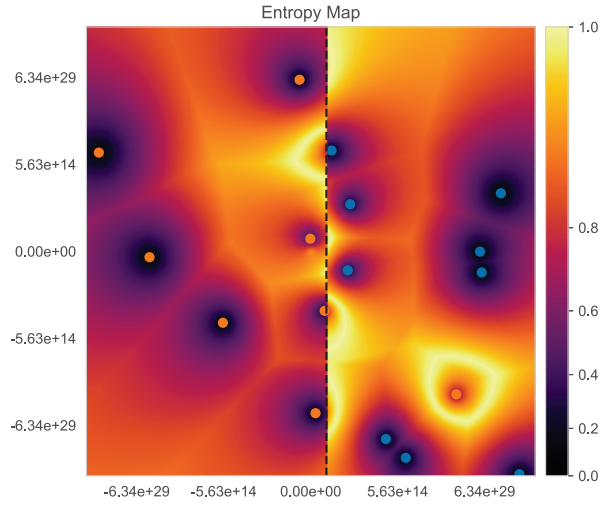


Fig. 2: A heat map of the calculated entropy of points in the input space. The areas with maximal entropy are those along the border, or close to a mislabeled point. The color map has been stretched at the top end to make it easier to distinguish changes in entropy close to 1, which are the important values when doing the $argmax$ optimization.

there are $4^k$ domains for an input space with $k$ floating point fields, so calculating information gain across all domains becomes expensive quickly. We also hypothesize, looking at bugs we have seen in the past, that bugs tend not to exist across many domains, often requiring one or two special values at most. We therefore use a *K-Armed Bandit* [4] to select which domain to sample from. A trial is considered a success if it increases diversity of population either in that domain or in the total data. This allows us to efficiently identify which domains to sample from (i.e. which special values to include in tests sent to the oracle).

*a) Batch Sampling:* In our research application (autonomy system testing), where running a test can take a very long time, it is necessary to run tests in parallel. Thus, we need to be able to draw multiple samples from the active learning algorithm at once. In order to discourage drawing multiple samples from the same area, we can artificially increase the certainty in the area of the sample by treating earlier samples in the batch as if they had already been run, and hallucinating a value from the predictor. This effectively decreases uncertainty in that area, decreasing the estimation of how much information can be gained near that point, which in turn encourages the sampling to search in other areas. These hallucinated values are discarded after the batch is drawn, and the true results from the SUT are incorporated before the next round of active sampling.

*1) Passive Sampling:* In order to calculate information gain, the algorithm requires a model to refine. For this reason, the first round of test case generation is carried out by a passive sampling algorithm. The passive sampler uses a Latin



Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on May 12,2026 at 15:16:11 UTC from IEEE Xplore. Restrictions apply.

Hypercube [23] to draw field values for trials. This sampling technique tries to get an even distribution of sample points across the sample space. This is done by partitioning all the possible values for each field in the test case into bins. The process of generating a test case is to, for each field in the test case, select a bin that has not been used in a previous test case, and then sample uniformly from within the range of that bin. The bins are of uniform size in the sample space of compressed values, so when decompressed will contain many more smaller values in test cases than larger ones. Again this is desirable, based on our previous experience on how the values that cause faults to be triggered are distributed throughout the value space [16] [20].

In order to appropriately handle special values for floats, one of the bins is reserved. When that bin is selected, the value returned is selected randomly from {NaN, INF, -INF}. The remainder of the bins are partitioned uniformly over the possible non-special values, as usual.

*2) Bootstrapping Sampling:* In cases where the populations of points with each label are severely unbalanced, it can be very difficult to create a predictive model. Without a good predictive model, active sampling no longer works. When this occurs, we use a Hypersphere sampler to explicitly search the area near trials from the underpopulated class. This helps create new trials in that class and balances out the populations.

Hypersphere sampling simply creates an $n$-dimensional sphere around a point from the under populated class, with radius $r$.

$$r = \frac{d(t, \bar{T}_l)}{2}$$

Where $\bar{T}_l$ is the set of all completed tests that do not have the same label as $t$.

This sampling aims to find more tests that are in the underpopulated class. If tests were not of the underpopulated class, then the next round of hypersphere sampling will have a smaller radius, due to the new tests in $\bar{T}_l$. Once the populations are more balanced, we return to regular active sampling.

### B. Fitting Decision Trees

Our approach is an extension of the C4.5 algorithm [27] for decision trees, with alterations to support various multivariate splitters. At each node in the decision tree, we calculate the split that provides the greatest Information Gain (decrease in weighted entropy). In C4.5, splits are calculated by finding a univariate inequalities. However, the calculation of multivariate splits is slightly more involved, and is detailed in Section III-C. When every node has reached a termination condition, we prune the tree using two pruning steps, described in Section III-D.

To avoid over fitting, the algorithm stops when any of the following termination conditions are met: (1) Population entirely one class, (2) Population size is less than 5, or (3) No split produced any Information Gain.

These roughly match the termination conditions of C4.5, but with the inclusion of a minimum population. This is a pre-pruning step [26] that we employ to focus more on providing a user interpretable explanation than on accuracy. Since we suspect that the underlying behavior of a fault is well described by our fault model (splitters), adding additional complexity to account for small populations is more likely to introduce noise and over-fitting than provide tangible benefit.

### C. Multivariate Splitters

Our overall goal is to find faults in code. We therefore have an expectation on the underlying shape of the portions of the input space corresponding to each label, as faults are typically caused by code defects. For example, a fault that occurs when a value is very large could trace to a memory allocation problem. However, in code, a fault could be caused by the interaction between two values, for example the dimensions of a 2d array could cause the same memory allocation fault. In this case the underlying function is described by $width * height > mem\_limit$. This product is hard to represent in a C4.5 Decision tree (or indeed by HPSL or Support Vector Machines), but is common in defects.

To make more descriptive decision trees, we incorporate these types of simple fault models into the splitters used for ALO-Trees. The models we chose to include come from our prior work on the robustness testing of autonomy system software [16]. These faults come from nearly a decade of testing over 20 autonomy software systems, and provide a reasonable (though definitely not exhaustive) representation of the underlying fault models for autonomy systems. As disjunctions and conjunctions are taken care of by the structure of the decision tree, we can break down our known faults to fault primitives, such as `field_x = some_value` or `field_x < field_y`. The primitives that we used are

- is INF
- is NaN
- $x = y$ (within $\epsilon$)
- $int(x) = int(y)$
- $x < p_0$ (the standard C4.5 split)
- $x + p_1 * y < p_0$
- $x * y < p_0$
- $x = p_0$ (within $\epsilon$)
- $x + p_1 * y = p_0$ (within $\epsilon$)
- $x * y = p_0$ (within $\epsilon$)

The $\epsilon$ that we use is $max(0.5, 0.01 * p_0)$ which performs reasonably well for both integers and floats.

In general, finding the optimal split is in NP [19] [15]. However, we can use heuristic approach of estimating boundary points, and then using RANSAC to find model parameters that approximately model those boundary points. RANSAC is ideal for this task, as it quickly find subsets of the data that are well explained by a model, by selecting the model parameters that provide the largest set of explained points, and discarding those points that are not explained. This means that if we have a conjunction of primitives (such as `(x < 0) and (y < 0)`), then RANSAC will select those boundary points that form a modelable subset, for example `x < 0`, while ignoring those boundary points that don't fit into this model. We estimate boundary points by finding the set of



Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on May 12,2026 at 15:16:11 UTC from IEEE Xplore. Restrictions apply.

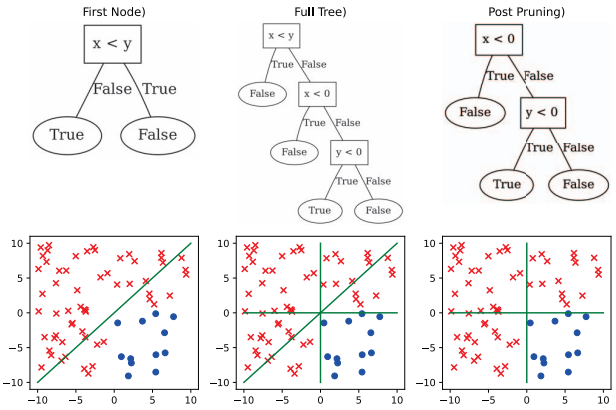


Fig. 3: The growth and subsequent pruning of a tree with a superfluous root node.

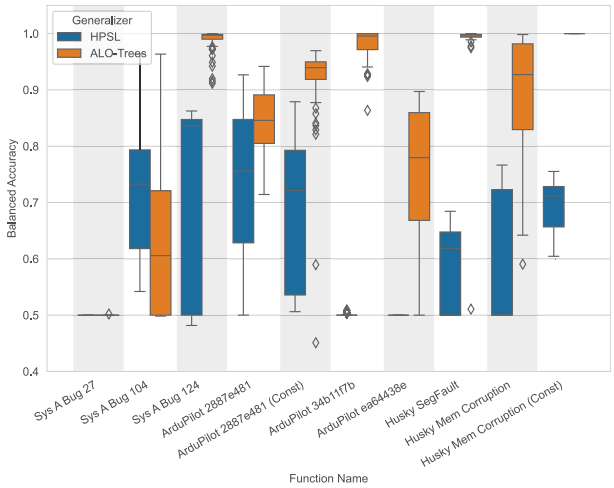


Fig. 4: Comparison of ALO-Trees vs HPSL in balanced accuracy of produced output. Distributions are over 120 trials of each approach. ALO-Trees shows substantial improvement in the majority of cases.

points in one class that are closest to the opposing class, then averaging each of these points with their nearest neighbor in the other class to create an approximate boundary point.

### D. Pruning

Pruning is a two-step process. The first step is the frequently used approach of Minimal-Cost-Complexity Pruning [8]. This pruning technique creates a family of trees by iteratively removing the minimum-cost complexity node. The final tree is the simplest tree within one standard deviation of the best performance. This well-known tree-simplification approach tends to increase performance, as simplified trees are less prone to over-fitting, and creates more readable trees.

The second pruning step is a top-down algorithm that replaces a node with one of its children. Because we use multivariate splits, tree splits are not orthogonal. This can lead to a situation where a split was optimal when it was calculated, but is rendered superfluous once its children are discovered. Figure 3 shows an example: "x $<$ y" provides high information gain when modeling the entire data set, but its children can ultimately correctly model the data without that split. This pruning creates new trees by replacing a given node by one of its children, grafting the other child on in either position, then evaluating all of resulting trees using cost complexity.

## IV. Experiments

We implemented ALO-Trees and HPSL to compare their performance. Using field values as features, we ran the algorithms on several bugs in robotic systems, which come from *System A* (a mature autonomy system, supplied by the project sponsor under NDA), *Clearpath Robotics Husky*, and *ArduPilot*. For each of the bugs in these systems, we created a mock of the bug based on inspection of the code that caused the fault. Then each algorithm was given a starting test case that demonstrated a violation, and tried to diagnose the failure.

Each algorithm was run 120 times, and the effectiveness of the algorithm was measured by the balanced accuracy of the generated fault description. This balanced accuracy is measured over a randomly generated test set of 1000 points for each of the 120 runs, which is biased to contain a minimum of 10% violating and non-violating trials (to avoid Balanced Accuracy being skewed by insufficient data for either class).

*a) System A:* System A is an automated platooning ground-based cargo transportation vehicle, under NDA, provided by the sponsor of this work. The bugs were taken from an internal database of bugs or the system, which described necessary conditions on the bug-triggering inputs. For System A, these are classified as `bug 27, 104, and 124`.

Bug 124 is a simple integer addition overflow. Bug 104 represents a case where bugs can occur under multiple disjoint conditions of two floats, in conjunction with requirements for three integer fields and four booleans, and is an example of the kinds of bug HPSL was designed for. Bug 27 occurred when an inconsistent state was set: when $x$, $y$ do not equal the sine and cosine of the same angle $\theta$.

### A. Evaluation

*a) ArduPilot:* Ardupilot [1] is an open source autopilot that provides autonomy capabilities for UAVs, UGVs, UUVs. For bugs in this system, we relied on the squaresLab ArduBugs repository[2] [29] bug reports. We examined the repository commit diffs for the bug fixes, and made the assumption that any changes on bounds checks on fields corresponded to values that would trigger bugs. In this case, we considered the fault to occur when the inputs matched the original conditional but not the updated conditional, or vice versa. We identified 10 relevant bugs, and randomly selected 3 of them. For one of these bugs (2887e481), it was unclear if a particular symbol was a variable or a constant, so both variants are included

[1] https://ardupilot.org/
[2] https://github.com/squaresLab/ArduBugs



Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on May 12,2026 at 15:16:11 UTC from IEEE Xplore. Restrictions apply.

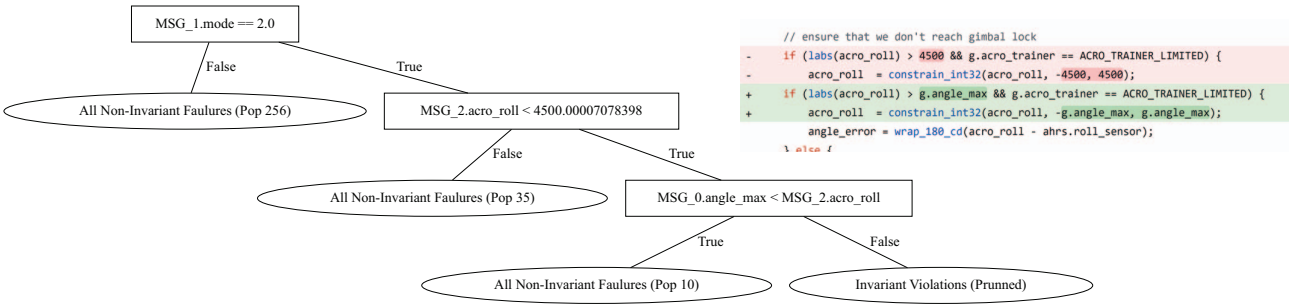


Fig. 5: The diff of bug 34b11f7b, and the corresponding ALO-Trees output

*b) Husky:* Husky[3] is a small UGV platform developed by Clearpath Robotics, which has some demonstration code available through ROS [4] that we used as our target system. The Husky bug is a documented [17] bug where an erroneous goal value causes the robot to exhibit a software crash. While the simplest version of this bug can cause a segfault, the code defect can also cause memory corruption. For this case, it was again unclear if a particular symbol was a variable or a constant. We include all three variants in the analysis.

These bugs are each reproduced as a python function call, for ease of integration with the active learning component and analysis. While we have made some assumptions about the shapes of these bugs based on the bug reports, we are confident that the input regions are representative of real bugs in real autonomy systems, and are sufficient for analysis.

Figure 4 compares the Balanced Accuracy of the results of ALO-Trees and HPSL. We use Balanced Accuracy, as the accuracy of a bug description should weight both the negative and positive class equally. This also removes the impact of population sizes when generating examples for evaluation, as described in IV.

In all but two cases, ALO-Trees outperforms HPSL. One of the cases, "bug 27," is a region where neither algorithm can identify the fault. In this case, the fault occurred any time that $sin^{-1}(x) \neq cos^{-1}(x)$, which proved too difficult for either algorithm to identify. "Bug 104" contains 9 variables, which may be too many for the ALO-Trees algorithm to work with. Fortunately, prior work [16] has shown that this type of fault is rare, and most faults in autonomy systems are low-dimensional, with often only one or two fields being relevant to a fault, even when the messages themselves contain thousands. In general, we find that ALO-Trees more accurately represents autonomy system bugs, while running 35-50% fewer tests.

The key goal for ALO-Trees is to produce outputs that are informative to a human debugger. Figure 5 shows a bug, in the form of a commit diff[5], and the ALO-Trees output for that bug. The tree representation of the bug closely matches the actual code, and we believe would be effective at guiding a human debugger to the correct section of code.

[3]https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/
[4]https://wiki.ros.org/Robots/Husky
[5]https://github.com/ArduPilot/ardupilot/commit/34b11f7b

## V. Related Work

Decision trees have a broad appeal as an interpretable machine learning technique, and there is a lot of work in extending decision trees beyond axis aligned splits as we have. [10] create appropriate split functions during training time to best fit the data while minimizing the number of splits. [14] optimizes features (transformations over the data) while learning trees to create shallow, interpretable trees. Both of these techniques are powerful solutions to a general machine learning problem, but the features they create may be non-intuitive. In contrast, our approach takes advantage of the narrow focus on modeling code faults to ensure that the splits are always readable combinations of variables.

Daikon [13], InvariMint [6] and CSight [7] are debugging tools that also infer system rules from black box system behavior, like ALO-Trees. These tools operate in a similar context to our work, but generally focus on system level behaviors and identify requirements/interface defects, rather than the faults in code that we aim to address in this work.

## VI. Conclusion

A key issue facing the field of robotics is the difficulty in debugging such complex systems. Our access to, and evaluation of, a diverse ecosystem of robotics systems from various industry and government sponsors has given us insight into the type of bugs that are common in robotics systems in practice, as well as the need for automated debugging tools that work in the robotics context.

ALO-Trees is a novel machine learning algorithm for generating bug descriptions. When compared against a prior technique, HPSL [31], we find that ALO-Trees does a better job accurately describing the fault triggering region of bugs in autonomy systems, while running fewer tests. Additionally, ALO-Trees produces a more interpretable output, to aid debuggers in identifying code defects. ALO-Trees achieves this performance by taking advantage of the constrained problem space of code defects, sacrificing the generalizability of other decision tree algorithms in order to make more succinct descriptions in the form of shallower trees. ALO-Trees also presents a novel active learning approach for searching large spaces (e.g. the entire float space), and creating meaningful tests to reduce the number of required trials.



Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on May 12,2026 at 15:16:11 UTC from IEEE Xplore. Restrictions apply.

## REFERENCES


[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[2] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107, 2020.

[3] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *Proceedings of the 13th International Conference on Software Testing, Validation, and Verification*. IEEE, 2020.

[4] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on learning theory*, pages 39–1. JMLR Workshop and Conference Proceedings, 2012.

[5] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[6] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering*, 41(4):408–428, April 2015.

[7] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *ICSE 2014, Proceedings of the 36th International Conference on Software Engineering*, pages 468–479, Hyderabad, India, June 2014.

[8] L Breiman, JH Friedman, R Olshen, and CJ Stone. Classification and regression trees. 1984.

[9] Domenico Cotroneo, Domenico Leo, Roberto Natella, and Roberto Pietrantuono. A case study on state-based robustness testing of an operating system for the avionic domain. pages 213–227, 01 2011.

[10] Yashesh Dhebar and Kalyanmoy Deb. Interpretable rule discovery through bilevel optimization of split-rules of nonlinear decision trees for classification problems. *IEEE Transactions on Cybernetics*, 51(11):5573–5584, 2021.

[11] Tobias Dürschmid, Christopher Timperley, David Garlan, and Claire Le Goues. Rosinfer: Statically inferring behavioral component models for ros-based robotics systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 928–928. IEEE Computer Society, 2024.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb 2001.

[13] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[14] Jack Henry Good, Torin Kovach, Kyle Miller, and Artur Dubrawski. Feature learning for interpretable, performant decision trees. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[15] David G. Heath, Simon Kasif, and Steven L. Salzberg. Induction of oblique decision trees. In *International Joint Conference on Artificial Intelligence*, 1993.

[16] Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. Robustness testing of autonomy software. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, page 276–285, 2018.

[17] Deborah Katz, Milda Zizyte, Casidhe Hutchison, David Guttendorf, Patrick E Lanigan, Eric Sample, Philip Koopman, Michael Wagner, and Claire Le Goues. Robustness inside out testing. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 1–4. IEEE, 2020.

[18] Deborah S. Katz, Casidhe Hutchison, Milda Zizyte, and Claire Le Goues. Detecting execution anomalies as an oracle for autonomy software robustness. In *International Conference on Robotics and Automation*, pages 9367–9373. IEEE, 2020.

[19] Caleb Koch, Carmen Strassle, and Li-Yang Tan. Properly learning decision trees with queries is np-hard, 2023.

[20] Philip Koopman, Kobey DeVale, and John DeVale. Interface robustness testing: Experience and lessons learned from the ballista project. *Dependability Benchmarking for Computer Systems*, pages 201–226, 2008.

[21] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, pages 230–239. IEEE, 1998.

[22] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[23] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[24] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162, 2014.

[25] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering*, 23(5):2980–3006, 2018.

[26] Nikita Patel and Saurabh Upadhyay. Study of various decision tree pruning methods with their empirical comparison in weka. *International Journal of Computers and Applications*, 60:20–25, 2012.

[27] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[28] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Siti Zaiton Mohd-Hashim. A comparative study on automated software test oracle methods. In *International Conference on Software Engineering Advances*, ICSEA '09, pages 140–145, 2009.

[29] C. Timperley, A. Afzal, D. S. Katz, J. Hernandez, and C. Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 331–342, 2018.

[30] Christopher S Timperley, Tobias Dürschmid, Bradley Schmerl, David Garlan, and Claire Le Goues. Rosdiscover: Statically detecting runtime architecture misconfigurations in robotics systems. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 112–123. IEEE, 2022.

[31] Paul Vernaza, David Guttendorf, Michael Wagner, and Philip Koopman. Learning product set models of fault triggers in high-dimensional software interfaces. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS '15, pages 3506–3511, Sept 2015.

[32] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Software Engineering-ESEC/FSE'99*, pages 253–267. Springer, 1999.




Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on May 12,2026 at 15:16:11 UTC from IEEE Xplore. Restrictions apply.