Detecting Execution Anomalies As an Oracle for Autonomy Software Robustness

Deborah S. Katz[1] and Casidhe Hutchison[2] and Milda Zizyte[3] and Claire Le Goues[1]

***Abstract*— We propose a method for detecting execution anomalies in robotics and autonomy software. The algorithm uses system monitoring techniques to obtain profiles of executions. It uses a clustering algorithm to create clusters of those executions, representing nominal execution. A distance metric determines whether additional execution profiles belong to the existing clusters or should be considered anomalies. The method is suitable for identifying faults in robotics and autonomy systems. We evaluate the technique in simulation on two robotics systems, one of which is a real-world industrial system. We find that our technique works well to detect possibly unsafe behavior in autonomous systems.**

I. Introduction

Robotics and autonomy systems are safety-critical, and their failures can be expensive and sometimes deadly. The risk of critical safety failures only increases as robotics applications have expanded into contact with the public [1]–[4], motivating increased testing, debugging, and Quality Assurance (QA) support.

Existing approaches to assuring safe behavior in these systems include formal verification and testing-based approaches, such as field testing [5], unit testing [6], robustness testing [7], [8], and fuzz testing [9]. Formally verifying real systems requires extraordinary investment of time and human effort [10]–[12], making it impractical to apply to entire systems [13], [14]. Bugs have been found in formally-verified systems due to invalid assumptions [15].

Many QA techniques based in testing, debugging, and program repair require an *oracle* [16] — a systematic approach for distinguishing between acceptable and unacceptable behaviors [17]. A human can act as an oracle, but human effort is expensive [18]. Often, these QA techniques use a simple automated oracle to detect obvious failures, such as software crashes and tasks that take too long [19]. A more sophisticated oracle can detect additional failures and unsafe behaviors, as is needed for safety-critical systems. For highly automated testing techniques – such as RIOT [8], [20] – the amount of effort required to generate testing oracles has a large impact on the overall testing effort. Automation of this part of the testing process can drastically reduce the needed user effort and expertise.

Detecting errors in robotics and autonomous systems may require a more involved process than in classical software [8], for various reasons. Cyber-physical systems, such as robots, must be safe at every point of execution, in contrast to more traditional software which often only must produce correct output. For evaluating system safety we focus on robustness testing, which measures the sensitivity of the system to environmental faults such as sensor failure or network message corruption. The distributed and cyber-physical nature of the systems can also mask errors when different components perform different functions: the failure of a perception subsystem to detect a pedestrian may be masked if the corresponding navigation subsystem had coincidentally designated a route that did not collide with the pedestrian. QA on autonomous and robotics systems is further complicated because source code is not guaranteed to be available for all parts of the system – for example, during third party test and evaluation. [7], [21].

We present a novel technique that automatically detects potentially unsafe behavior in robotics and autonomy system software by monitoring low-level execution behavior and performing anomaly detection on the results. Systems that interact with the real world are well-suited to execution monitoring because most of the overhead can be absorbed into time the system would have otherwise been spent waiting. Our work is based in the assumption that unusual behavior is more likely to exhibit errors than typical behavior and, therefore, that unusual behavior might represent unsafe behavior [22]. We present a semi-supervised machine learning approach based on clustering data points representing *nominal* executions; the resulting model serves as a test oracle for data points representing additional unknown executions. The approach uses low-level system monitoring and detects a significant portion of faults on two simulated robotics systems. We present initial experiments on robotics in simulation, demonstrating the use and effectiveness of our anomaly detection technique.

Section II outlines our method, combining low-level system characterization techniques, clustering, and anomaly detection. Section III discusses the evaluation of our technique, including the Systems Under Test (SUTs) and test inputs. Section III discusses experiment results. Section V highlights implications and use cases of our technique. Section VI fits our technique into context, and Section VII concludes.

This project was funded by the Test Resource Management Center (TRMC) Test and Evaluation / Science & Technology (T&E/S&T) Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) under Contract No. W900KK-16-C-0006, "Robustness Inside-Out Testing (RIOT)." DISTRIBUTION STATEMENT A - Approved for public release; distribution is unlimited. NAVAIR Case #2019-615.

[1]School of Computer Science, Carnegie Mellon University dskatz@cs.cmu.edu and clegoues@cs.cmu.edu

[2]National Robotics Engineering Center, Carnegie Mellon University fhutchin@nrec.ri.cmu.edu

[3]Dept. of Electrical and Computer Engineering, Carnegie Mellon University milda@cmu.edu

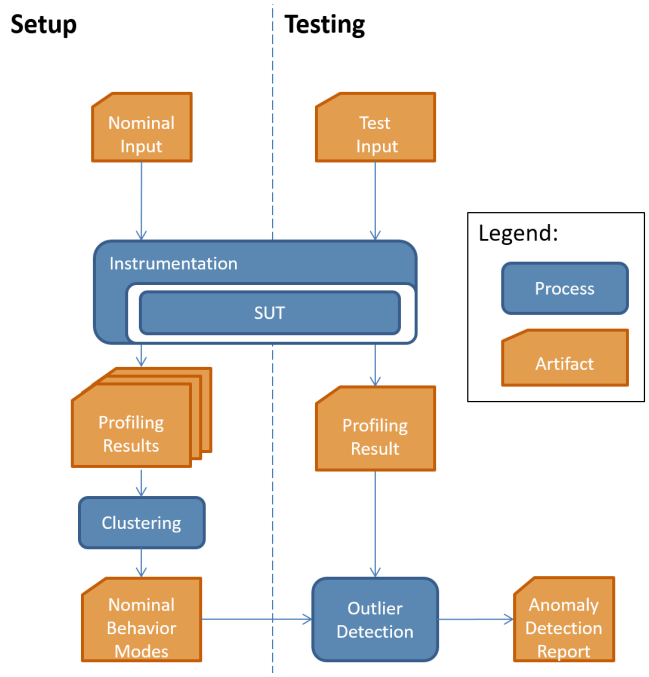

Fig. 1. The architecture diagram for this testing approach, outlining the setup and testing phases.

II. Method

Figure 1 illustrates our proposed technique for using anomaly detection to find unexpected behaviors in robotics systems in simulation. Our system assumes a System Under Test (SUT) – a robotics system that can be executed with a system characterization technique, either on real hardware or in simulation [23], [24]. We also assume at least one *nominal* system input – an input not known to cause any safety violations or unintended behavior. Given this input, the approach is comprised of two steps: setup, and testing.

Setup involves running an instrumented system on nominal inputs and collecting summaries of system execution (Section II-A). These summaries are — or are parsable to — a collection of values (either fixed-length or time series). We use the affinity propagation algorithm [25] to automatically cluster these representations, with clusters representing common modes, or statistical maxima, of nominal operation. Affinity propagation works especially well when the number of modes is not known *a priori*. The setup nominal data set is relatively small, so this step does not impact overall performance significantly.

In *testing*, we detect anomalies by comparing instrumentation-produced execution summaries of previously unseen inputs against the clusters of nominal behaviors. The comparison uses Local Density Cluster-Based Outlier Factor (LDCOF), a clustering-based anomaly detection technique (Section II-B). LDCOF produces an outlier score for a new input, representing how far the new execution is from a nominal operation. A higher outlier score signifies that a given execution may be indicative of unsafe system behavior.

A. System Characterization Techniques

We use three system profiling techniques: two off-the-shelf tools and one custom Valgrind tool.

1) PS Utility: The Python package psutil[1] collects information on running processes by interacting with underlying system services. The resulting data include: *User time* – the amount of CPU time the system has spent in the process, not the kernel; *System time* – the amount of time spent in kernel mode; *Resident set size* – the amount of RAM allocated to the process; and *Virtual memory size* – the amount of virtual memory the process has access to. We collect this data at a fixed frequency, resulting in a time series describing system behavior over the course of execution.

2) Valgrind Memcheck: A popular program profiling method is Valgrind,[2] an instrumentation framework for developing dynamic analysis tools. Valgrind's default tool, Memcheck, tracks a program's memory accesses, such as memory initialization and freeing resources. It wraps most instructions with instrumentation. We parse Memcheck's log file to extract values that summarize the system behavior.

3) Customized Valgrind Tool: SignalSeer*:* We designed a custom dynamic binary instrumentation tool on the Valgrind framework, to collect a broader set of low-level data about the executions of an SUT. We call this tool SignalSeer. It counts execution events and keeps track of minima and maxima for certain values, such as: number of machine instructions executed, number of memory stores, and minimum and maximum memory addresses for each.

The tool is designed to have low overhead, the choice of what to count inspired by Valgrind's instrumentation structure. We add an instrumentation point to each instruction and, at each of these points, only collect data that can be gathered efficiently. The tool minimizes calls to libraries and Valgrind's API at runtime. This tool provides a summary of execution behavior that is ready to use for our analysis.

B. Detecting Anomalies in System Execution

To detect execution anomalies, we need a way of measuring how anomalous a given execution is. We use Local Density Cluster-Based Outlier Factor (LDCOF), an anomaly detection algorithm that uses clustering to find outliers [26].

LDCOF works by applying a clustering algorithm to the data, separating the clusters into large and small clusters, then for each data point calculating an outlier score as follows:

$$LDCOF(p) = \min_{C_i \in \text{large clusters}} \frac{d(p, C_i)}{avg\_dist(C_i)}$$

where $d(p, C_i)$ is the distance between a point p and the center of cluster C_i, and

$$avg\_dist(C_i) = \frac{\sum_{p \in C_i} d(p, C_i)}{|C_i|}.$$

In our use case, we have separate training and target sets. We adapt the algorithm to consider *all* clusters of the nominal

[1] https://pypi.org/project/psutil/
[2] http://valgrind.org/

data to be "large clusters". This ensures that the outlier score is calculated against every mode of nominal operation.

LDCOF suits our use case well because it accounts for variance in operating modes. With LDCOF, a data point that is X distance away from a cluster with a lot of variance would be considered less anomalous than one that is the same distance away from a cluster with little variance. This behavior analogizes well to modes of system operation.

We use the estimate from $LDCOF(p)$ of how much of an outlier a given data point (and its corresponding execution) is, as an oracle. Any execution with an $LDCOF(p)$ greater than some threshold, is sufficiently far away that it is likely to represent an anomalous execution. We choose a threshold equal to one.

a) Dynamic Time Warping: For system characterization techniques – such as PSUTIL – that produce time series data, we adjust the clustering algorithm to compare time series data using Dynamic Time Warping (DTW) [27] instead of Euclidean distance. DTW calculates the disparity between two sequences under warping of the time axis by finding a sequence of index pairs (i, j) that minimizes

$$\sum_{k=1}^{end} |seq\_a[i_k] - seq\_a[i_k]|$$

subject to the following constraints.

$$(i_1, j_1) = (1, 1)$$

$$(i_{k+1}, j_{k+1}) \in \{(i_k + 1, j_k), (i_k, j_k + 1), (i_k + 1, j_k + 1)\}$$

$$(i_{end}, j_{end}) = (len(seq\_a), len(seq\_b))$$

.

In prose: it calculates the shortest distance between elements of two sequences, allowing adjustment of the relative ordering of elements between the sequences, as long as the ordering of elements within each sequence remains fixed.

DTW is not an actual distance function but instead a distance-like function, which means that we cannot calculate the center point of a cluster using DTW. Instead, we approximate the distance $d(p, C)$ as:

$$d(p, C) = \underset{q \in C_i, q \neq p}{\text{avg}} d(p, q)$$

III. Experiment Setup

Each experiment centers on one *SUT* (Section III-A) and one system characterization technique (Section II-A). In the *setup* phase, we repeatedly execute the SUT in simulation with a nominal input under the chosen system characterization technique.[3] For the *testing* phase, we use mutational fuzzing on the nominal input to generate test inputs. For these experiments the nominal input is an input file provided with the SUT. We run the SUT under the system characterization technique with each test input to generate a profiling result and classify the result using our LDCOF-based anomaly detection method. We compare detected anomalies against a set of manually written system invariants, a proxy (approximate stand-in) for true safety. Here, an invariant is a rule that must hold over the entire execution, covering anything from message transmission frequency to speed limits. We describe test and invariant construction in Section III-B and III-D.

A. Systems Under Test

We use two SUTs in these experiments: a research system (including common robotics libraries) and a commercial system. These systems were chosen for insight into real-world factory robotics systems and the behavior of some of the underlying libraries that are common in robotics development.

We drew nominal inputs from simulations provided with the robots, as demonstrations of intended system behavior. The inputs take the form of ROSBAGS, files that contain a series of messages of different types for a system. We replay the ROSBAGS to exercise the system with the same input.

We use the GAZEBO simulator for system execution, following the instructions in the SUT documentation. This gives us a consistent environment in which to exercise the SUTs while applying the nominal and test inputs.

1) BenchMark Bot (BMB): an artificial system our team designed for research on the Robot Operating System (ROS), common libraries, and interactions. It is a lightweight wrapper for key ROS functionalities, including path planning and switching between sources of control. The robot is designed to compute a path based on waypoints and what it senses in its environment. Because BMB is modular, it can be configured to use any of several planning algorithms.

We tested BMB using a recording of a simple exploration scenario. To do this, we only ran one node, rather than the entire system. The node we tested was *Global Planner* which takes in a point and outputs a likely path.

2) Fetch Robotics' Fetch: commercially-available "autonomous mobile robots that operate safely in commercial and industrial environments shared by people."[4] They pick up and transport heavy payloads, such as items in a warehouse environment. They have a mobile base and an arm for manipulation. The robots are built on ROS. Fetch Robotics provided simulation and the *disco dance* scenario, which we tested. Disco dance demonstrates the movement planners, taking the robot through a range of arm motions, with dummy "collision objects" for the arm to avoid.

B. Test Inputs

We use mutational fuzzing to create test inputs likely to induce safety failures. Changing values in inputs to exceptional values (such as MAX_INT, NAN, or -1) has been shown to effectively induce robotics systems failures [8], [19]. Specifically, for each selected message type in a nominal input file, we inject roughly ten exceptional values using a randomized value injection technique. We generate at least six test inputs per message type.

[3]We run all simulations on machines with the following configuration: Intel® Core™2 Duo CPU (E8500) 2 Cores @ 3.16GHz, 4 GB RAM

[4]http://fetchrobotics.com/automated-material-transport-v3/

```
1 if "torso_lift_joint" in joint:
2     if abs(joint_vel) > 0.1:
3         return True
4     if abs(joint_effort) > 450:
5         return True
6     if joint_pos < 0 or joint_pos > .4:
7         return True
```

Fig. 2. Code listing for a sample FETCH invariant.

These test inputs give a variety of input patterns, some of which cause safety violations, providing a data set on which to measure false positives and negatives. We determined whether each input contained faults by inputing it to the robot in simulation with no instrumentation and evaluating whether the simulation violated any invariants.

C. Metrics

To evaluate the effectiveness of our anomaly detection based oracle, we find the False Detection Rate (FDR) and sensitivity (a.k.a. recall). FDR is related to precision in that FDR equals one minus precision. We assign the positive label to safety violations, so the FDR represents the portion of detected anomalies that are not safety violations. Lower is better because false discovery results in unnecessary testing effort.

Sensitivity represents the proportion of test cases containing safety violations that were correctly detected. Ideally, this should be one, catching every safety-violating test input. Sensitivity is not influenced by the populations of safety violations in our test inputs, reducing the impact of our choice of testing method and parameters.

While we lack an absolute ground truth for safety violations, we evaluate anomaly detection performance by calculating FDR and sensitivity against a proxy for ground truth: a set of invariants, as described in Section III-D. To provide context, we also evaluate a reference oracle — detection of crashes via core dump.

D. Invariants

The evaluation uses an explicitly-written set of invariants as a proxy for ground truth. *Invariants* are rules which must hold true over the execution of the program. Here, the invariants represent safety constraints: a violation means that the system has become unsafe.

Specifically, the invariants in this experiment are PYTHON functions, human coded from portions of the system documentation. These functions are evaluated over the output log from a test input to determine if the rules were violated. An example of a subset of the invariants in FETCH is in Figure 2. This subset of invariants represents restrictions on the allowable values for properties of the TORSO_LIFT_JOINT.

Invariants provide a more in-depth analysis of system behavior and can help find faults beyond those that cause core dumps [8]. Invariants can closely approximate the real world safety of the system, but they require substantial effort and expertise to create a comprehensive set.

E. Core Dumps

For additional context, we evaluate a simple *core dumps* oracle — whether or not an execution crashes and produces a core dump. This very basic restriction on system behavior (that it should not crash) provides a bare minimum of fault detection in testing scenarios. Crash rate is often used as a simple oracle for system safety in robustness testing [28], [29].

IV. EXPERIMENT RESULTS

For evaluation, we compare our technique's detected violations against the ground truth proxy. A positive label indicates a safety violation (detected behavioral anomaly or invariant violation). A negative label means no violation is found (no anomaly is detected or invariant violated, respectively). We calculate accuracy metrics by comparing labels from our technique against our ground truth proxy.

A. Results

Experiment results are in Table I. Each row represents one of the SUTs. For each system, the table gives FDR and Sensitivity against manually written invariants for each system characterization technique. The table also reports FDR and Sensitivity of core dumps against manually written invariants. The FDR of core dumps against manually written invariants is zero by definition because a check for core dumps is included in the manually written invariants.

1) False Detection Rate: FDR highlights the number of test inputs identified as anomalies that were not actual safety violations. These false alarms can consume testing budgets because they lead to investigation of test inputs that are not actual faults. The false detection rate varied from 0.00 to 0.50, over two systems and three system characterization techniques: up to half of detected anomalies were not safety violations. SIGNALSEER performed best, with a maximum FDR of 0.25, meaning that 1 in 4 detection is incorrect.

2) Sensitivity: The sensitivity metric captures the proportion of safety violations that were detected as anomalies. Our sensitivity was high, reaching 100% when analyzing the behavior of BenchMark Bot with SIGNALSEER. For Fetch, a commercially available test system, the worst case sensitivity of the anomaly detection oracles (PSMON: 0.11) is double that of the reference oracle (0.06). In the best case, SIGNALSEER gives a nearly 6 fold improvement.

B. Experiment Discussion

The anomaly detection oracles achieved sensitivity that was as good as or, more often, better than that of the reference oracle. The reference oracle only detects system crashes, but there are many other kinds of potential safety violations, such as the example invariant for Fetch discussed in Section III-D. The higher sensitivity represents the ability to detect safety violations beyond system crashes.

However, sensitivity is also far from perfect, never reaching above 50% on Fetch. We believe that one reason is that

TABLE I

FALSE DETECTION RATE ("FDR") AND SENSITIVITY ("SENS.") VS. MANUALLY-WRITTEN INVARIANTS USING PROCESS MONITORING ("PS"), MEMCHECK, AND SIGNALSEER. (CORE DUMPS ARE ALSO EVALUATED VS. INVARIANTS, FOR CONTEXT. CORE DUMPS FDR IS 0 BY DEFINITION.)

		PS		MEMCHECK		SIGNALSEER		Core	
System	Scenario	FDR	Sens.	FDR	Sens.	FDR	Sens.	FDR*	Sens.
BMB	Global	0.00	0.44	0.45	0.66	0.00	1.00	0.00	0.44
Fetch	Disco	0.50	0.11	0.25	0.35	0.25	0.50	0.00	0.06

we used monitoring tools that do not track data values, only execution behavior. It cannot detect a safety violation that only manifests in data, such as a speed limit violation. Given that limitation, we think the results are quite promising.

Likewise, we consider that 0.50 and below to be excellent results for FDR given that this technique is highly automated. It does not require the domain-specific knowledge, such as that needed to write invariants.

The effect of using a proxy for system safety: Invariants are an imperfect proxy for ground truth system safety. The expressiveness of the invariant checker is limited, and there can be human error in encoding. A larger issue is that it is difficult to ensure a set of invariants is complete. Human operators are very bad at writing accurate and complete invariants, as we have observed in our experience with testing robots and is noted in the literature [30]. For this reason, we expect the invariants to incorrectly label some executions negative (in which there is a safety violation not found by the invariants).

Because we are using invariants as a proxy for real-world safety, it is possible that some *False Detections* — in which anomaly detection finds an issue not found by the invariants — are actually valid real-world safety violations, in which case the real-world FDR of these oracles would be lower.

The effect of this uncertain ground truth proxy on our sensitivity results is much more difficult to reason about. We do not know which trials the proxy mislabels, so we are unsure whether they were detected as anomalous. False positives from the proxy translate to uncertainty in sensitivity.

V. DISCUSSION

In this section, we discuss several interesting features of the approach and outline threats to the validity of our study.

A. Monitoring Techniques Can Be Used Together

We suspect that designing a different custom monitoring tool or combining the outputs of different monitoring tools may allow the technique to detect violations the current setup misses. In our experiments, we observed that a portion of the execution anomalies were detected when using one monitoring approach and not others. Because the overall technique is general, it is possible to create composite inputs to the anomaly detection algorithm that incorporate the outputs (profiling results) from more than one monitoring technique. Such an approach may lead to improved detection. We also suspect that adding additional low-level elements to be tracked by the customized tool — such as elements that capture data values — may result in finding additional anomalies.

B. Manually-Written Invariants are an Imperfect Proxy for Real-world System Safety

While we use explicit invariants taken from system documentation as ground truth, these invariants are far from perfect. In fact, they were even sometimes violated by normal system behavior. For example, one of the designer-provided invariants for BMB specified a minimum transmit frequency of 5Hz, while the code (and associated comments) set the *target* transmit frequency at 1Hz. In such cases, we chose to modify our invariants because labeling the nominal behavior of the system as faulty would make further analysis difficult.

These conflicts between documented and implemented behavior are not uncommon [30] and are one of the difficulties of creating explicit testing oracles.

C. Case Study — "False Positive" May Reveal Actual Fault

Because invariants represent an imperfect ground truth, it is possible for our technique to detect an anomaly that we erroneously define as a false positive, when the invariants are incapable of identifying the anomaly.

In fact, we found such a case, in BMB, for executions for which anomaly detection with MEMCHECK identified anomalies but there were no invariant violations. By manually examining the mutated inputs corresponding to these executions, we found that each perturbed the /PERCEPTION/MAP field. Manual inspection revealed that improper values in this field may lead to memory corruption, even when they do not cause a core dump. The invariants only detected a problem when there was a core dump. The anomalies detected using our technique with MEMCHECK are genuine faults not found by invariants, even though our analysis labels them as false positives.

D. Use in Debugging Techniques

One primary area of application for this work is in use with other software testing and debugging tools. Anomaly detection techniques, such as those described here, can serve as an automated oracle for these tools.

Automated testing and debugging tools can provide important information to the developer, but they typically require an oracle that describes if a system behaved correctly or not during a test. Using the approach described in this paper increases the amount of automation provided by these tools by removing the need for users to write oracles.

For highly automated tool chains such as RIOT [20] – a testing framework for the robotics and autonomous systems robustness domain – where many testing features are already automated, an automated oracle drastically reduces the amount of user involvement and expertise required.

E. Threats to Validity

Certain anomalies may occur in simulation that would not occur on actual robotics system hardware and vice versa. However, testing in simulation is a valid approach to discovering real bugs in autonomy systems [23], [24]. Additionally, the faults detected typically trace to code defects that exist regardless of platform, such as memory faults due to lack of bounds checking or CPU spikes due to busy loops.

In addition, the technique may not generalize beyond the systems we tested, or beyond the context of mutational input testing; the oracles are thus specific to this context and less general than, e.g., explicit invariants. However, the math behind our approach does not depend on this setup and could work with input data generated in different ways, without any requirement for labeling which executions exhibit correct behavior. We also evaluate on more than one system to provide evidence of potential generalizability.

Another threat is that system monitoring may introduce errors, such as timing errors, due to overhead; this would be exacerbated for testing on hardware, as real-world robotics processors have limited capacity. We advocate that the oracles created using this technique be used primarily in testing, rather than deployment. This threat is partially mitigated because we measure core dumps and explicit invariants on uninstrumented systems; thus, any problem in the SUT behavior detected by those techniques cannot be due to instrumentation, and instrumentation-induced failures will manifest as false positives. Finally, we observe that distributed systems that operate in real time, such as our target systems, spend a lot of time waiting. In practice, we have observed that much of the monitoring overhead can often be absorbed into this wait time, with little observable overhead.

VI. Related Work

This approach fits in a body of software testing work on the problem of determining whether a program behaves as intended, known as the *oracle problem*. There are several useful summaries of oracle problem research [31], [32].

Our technique is in the category of anomaly, novelty, and outlier detection techniques that identify data points that are unusual, appear to deviate markedly from, or be inconsistent with patterns in the rest of the data [33]–[35]. It falls in a subgroup, analogous to one-class classification in machine learning [33], in which normal (nominal) data provides the basis for a model which is used to determine whether new data fit [34]. Our approach extends these techniques to robotics and autonomous systems, using monitoring techniques that are particularly well-suited to these domains.

Clustering algorithms, which arrange similar data points into groups [26], are the basis for many anomaly, novelty, and outlier-detection approaches. Work has applied clustering to identification of software failures in various contexts [36]–[41]. We use off-the-shelf clustering algorithms [25], which create the models on which we build our anomaly detection.

Other work on complex automated oracles automatically detects *invariants* — which hold true over all correct executions — and identifies bugs by finding invariant violations. These techniques automatically generate inferences about a program's semantics [22]. These techniques have various limitations, such as source code requirements, restrictions to particular languages or to detecting particular kinds of properties and can struggle to scale [42]–[46]. By contrast, our approach can be language- and semantics-independent and is not constrained by many of the factors that restrict scalability in automated invariant detection, such as restrictions on the number of invariants because of memory limitations.

Formal verification of cyber-physical systems, such as robotics systems, is another approach for avoiding software faults. However, there are many gaps in practical application to entire real systems [10]–[12]. Because of these and other challenges, formally-verified systems can still contain bugs [15]. A significant drawback of formal verification is that it requires extraordinary time and human effort [10], [13], [14]. In contrast to formal techniques, our technique does not require specialized expertise and human effort.

Testing autonomous vehicles and robotics systems presents problems unique to those domains [8], [21], [47]–[50]. Prior work has demonstrated the usefulness of testing these systems in simulation [24], [51], [52].

A related technique involving clustering analysis of system execution in autonomous vehicles is Range Adversarial Planning Tool (RAPT) [53]. RAPT is an active sampling approach for guiding testing. RAPT demonstrates the feasibility and utility of clustering behavior in autonomy systems.

VII. Conclusions

We have presented a method that uses anomaly detection to detect unsafe behavior in robotics systems. The algorithm uses system monitoring techniques to obtain profiles of executions. It uses a clustering algorithm to create clusters of those executions, representing nominal execution. A distance metric (LDCOF) determines whether additional execution profiles belong to the existing clusters or should be considered anomalies. The method is suitable for identifying faults in robotics and autonomy systems.

In future work, we would like to evaluate our technique on situations in which the initial training data is derived from more diverse executions. We evaluated our system by building a model based on data derived from executions not known to have errors, data for which no core dumps or invariant violations are detected. In theory and with small modifications, this technique could derive a model from executions that are not all nominal, without necessarily needing labels identifying nominal executions. We would like to try the approach on this kind of data. We would also like to extend the approach to situations in which the input data is derived from more varied execution behavior.

REFERENCES

[1] A. Glaser, "Watch Amazon's Prime Air make its first public U.S. drone delivery," 2017, accessed Mar. 1, 2018. [Online]. Available: https://www.recode.net/2017/3/24/15054884/amazon-prime-air-public-us-drone-delivery

[2] A. Levin, "Alphabet drones will deliver burritos in australia test," 2017, accessed Mar. 1, 2018. [Online]. Available: https://www.bloomberg.com/news/articles/2017-10-16/drones-to-deliver-burritos-in-australia-as-alphabet-begins-tests

[3] "Uber 'not criminally liable' for self-driving death," Mar. 2019, accessed July 12, 2019. [Online]. Available: https://www.bbc.com/news/technology-47468391

[4] "Tesla's latest autopilot death looks just like a prior crash," May 2019, accessed July 12, 2019. [Online]. Available: https://www.wired.com/story/teslas-latest-autopilot-death-looks-like-prior-crash/

[5] D. Kohanbash, "Why test robots? – Field testing series – Part 1," Jan 2014, accessed July 12, 2019. [Online]. Available: http://robotsforroboticists.com/field-testing-series-part1-why-test/

[6] A. Paikan, S. Traversaro, F. Nori, and L. Natale, "A generic testing framework for test driven development of robotic systems," in *Modelling and Simulation for Autonomous Systems*, 2015, pp. 216–225.

[7] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Fault-Tolerant Computing*, ser. FTCS '98, June 1998, pp. 230–239.

[8] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman, "Robustness testing of autonomy software," in *International Conference on Software Engineering - Software Engineering in Practice*, ser. ICSE-SEIP '18, 2018, pp. 276–285.

[9] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, pp. 32–44, 1990.

[10] X. Zheng and C. Julien, "Verification and validation in cyber physical systems: Research challenges and a way forward," in *Software Engineering for Smart Cyber-Physical Systems*, 2015, pp. 15–18.

[11] X. Zheng, C. Julien, M. Kim, and S. Khurshid, "Perceptions on the state of the art in verification and validation in cyber-physical systems," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2614–2627, Dec 2017.

[12] M. U. Sanwal and O. Hasan, "Formal verification of cyber-physical systems: Coping with continuous elements," in *Computational Science and Its Applications*, ser. ICCSA '13, 2013, pp. 358–371.

[13] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, Jun 2010.

[14] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–2:70, Feb 2014.

[15] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *European Conference on Computer Systems*, ser. EuroSys '17, 2017, pp. 328–343.

[16] A. Bertolino, "Software testing research and practice," in *Abstract State Machines*, ser. ASM '03, 2003, pp. 1–21.

[17] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[18] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander, "Trade-off between automated and manual software testing," *International Journal of System Assurance Engineering and Management*, vol. 2, no. 2, pp. 114–125, Jun 2011.

[19] P. Koopman, K. Devale, and J. Devale, *Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project*, 2008, ch. 11, pp. 201–226.

[20] P. Koopman and M. Wagner, "Robustness Inside-Out Testing (RIOT) Project," Carnegie Mellon University, Robotics Institute, Tech. Rep. CMU-RI-TR-XXX, 2019.

[21] S. Forrest and W. Weimer, "The challenges of sensing and repairing software defects in autonomous systems," Regents of the University of New Mexico, Tech. Rep., 2014.

[22] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Symposium on Operating Systems Principles*, ser. SOSP '01, 2001, pp. 57–72.

[23] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues, "Crashing simulated planes is cheap: Can simulation detect robotics bugs early?" in *International Conference on Software Testing, Validation and Verification*, ser. ICST '18, April 2018, pp. 331–342.

[24] T. Sotiropoulos, H. Waeselynck, and J. Guiochet, "Can robot navigation bugs be found in simulation? an exploratory study," in *Software Quality, Reliability and Security*, ser. QRS '17, 2017, pp. 150–159.

[25] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, 2007.

[26] M. Amer and M. Goldstein, "Nearest-neighbor and clustering based anomaly detection algorithms for RapidMiner," in *RapidMiner Community Meeting and Conference*, ser. RCOMM '12, 2012, pp. 1–12.

[27] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Knowledge Discovery and Data Mining Workshop*, ser. KDD Workshop '94, 1994, pp. 359–370.

[28] J. DeVale and P. Koopman, "Robust software – no more excuses," in *Dependable Systems and Networks*, ser. DSN '02, 2002, pp. 145–154.

[29] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux kernel behavior under errors," in *Dependable Systems and Networks*, ser. DSN '03, 2003, pp. 459–468.

[30] R. R. Lutz, "Analyzing software requirements errors in safety-critical embedded systems," in *International Symposium on Requirements Engineering*, ser. RE '93, 1993, pp. 126–133.

[31] M. Pezze and C. Zhang, "Automated test oracles: A survey," *Advances in Computers*, vol. 95, pp. 1–48, 2014.

[32] 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*, vol. 41, no. 5, pp. 507–525, 2015.

[33] M. A. F. Pimentel, D. A. Clifton, L. Clifton, and L. Tarassenko, "A review of novelty detection," *Signal Processing*, vol. 99, pp. 215–249, 2014.

[34] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, 2004.

[35] I. Ben-Gal, "Outlier detection," in *Data mining and knowledge discovery handbook*. Springer, 2010, pp. 117–130.

[36] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *International Conference on Software Engineering*, ser. ICSE '01, 2001, pp. 339–348.

[37] C. Mao and Y. Lu, "Extracting the representative failure executions via clustering analysis based on Markov profile model," in *Advanced Data Mining and Applications*, ser. ADMA '05, 2005, pp. 217–224.

[38] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouche, "Techniques for classifying executions of deployed software to support software engineering tasks," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 287–304, 2007.

[39] S. Lu and R. Lysecky, "Analysis of control flow events for timing-based runtime anomaly detection," in *Workshop on Embedded Systems Security*, ser. WESS '15, 2015, pp. 3:1–3:8.

[40] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, "Applying classification techniques to remotely-collected program execution data," *SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 146–155, 2005.

[41] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing failure: The distribution of program failures in a profile space," in *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '01, 2001, pp. 246–255.

[42] 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*, vol. 27, no. 2, pp. 99–123, 2001.

[43] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, pp. 35–45, 2007.

[44] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *International Conference on Software Engineering*, ser. ICSE '02, 2002, pp. 291–301.

[45] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Design Automation Conference*, ser. DAC '05, 2005, pp. 775–778.

[46] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, and G. Sullivan, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 87–102.

[47] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Queue*, vol. 14, no. 2, pp. 91–110, 2016.

[48] L. Fraade-Blanar, M. S. Blumenthal, J. M. Anderson, and N. Kalra, "Measuring automated vehicle safety: Forging a framework," RAND Corporation, Tech. Rep., 2018. [Online]. Available: https://www.rand.org/pubs/research_reports/RR2662.html

[49] P. Koopman and M. Wagner, "Autonomous vehicle safety: An interdisciplinary challenge," *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 1, pp. 90–96, 2017.

[50] ——, "Toward a framework for highly automated vehicle safety validation," in *WCX World Congress Experience*, ser. WCX '18. SAE International, 2018. [Online]. Available: https://doi.org/10.4271/2018-01-1071

[51] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues, "Crashing simulated planes is cheap: Can simulation detect robotics bugs early?" in *International Conference on Software Testing, Validation, and Verification*, ser. ICST '18, 2018, pp. 331–342.

[52] A. C. Madrigal, "Inside Waymo's secret world for training self-driving cars," *The Atlantic*, August 2017.

[53] G. Mullins, P. Stankiewicz, R. Hawthorne, J. Appler, M. Biggins, K. Chiou, M. Huntley, J. Stewart, and A. Watkins, "Delivering test and evaluation tools for autonomous unmanned vehicles to the fleet," *Johns Hopkins APL Technical Digest (Applied Physics Laboratory)*, vol. 33, pp. 279–288, April 2017.