Mithra: Anomaly Detection as an Oracle for Cyberphysical Systems

Afsoon Afzal, Claire Le Goues, *Member, IEEE*, and Christopher Steven Timperley

Abstract—Testing plays an essential role in ensuring the safety and quality of cyberphysical systems (CPSs). One of the main challenges in automated and software-in-the-loop simulation testing of CPSs is defining effective oracles that can check that a given system conforms to expectations of desired behavior. Manually specifying such oracles can be tedious, complex, and error-prone, and so techniques for automatically learning oracles are attractive. Characteristics of CPSs, such as limited or no access to source code, behavior that is non-deterministic and sensitive to noise, and that the system may respond differently to input based on its context introduce considerable challenges for automated oracle learning. We present Mithra, a novel, unsupervised oracle learning technique for CPSs that operates on existing telemetry data. It uses a three-step multivariate time series clustering to discover the set of unique, correct behaviors for a CPS, which it uses to construct robust oracles. We instantiate our proposed technique for ArduPilot, a popular, open-source autopilot software. On a set of 24 bugs, we show that Mithra effectively identifies buggy executions with few false positives and outperforms AR-SI, a state-of-the-art CPS oracle learning technique. We demonstrate Mithra's wider applicability by applying it to an autonomous racer built for the Robot Operating System.

Index Terms—Robotics and autonomous systems, cyberphysical systems testing, anomaly detection, oracle learning, clustering, Mithra

1 Introduction

Cyberphysical systems (CPSs) integrate physical (e.g., sensors, actuators) and computational components (e.g., monitoring, perception, planning, control) to tackle problems that neither physical or computational parts alone could solve [91]. CPSs are an important part of our everyday lives and have many safety-critical applications in avionics, medical operations, and transportation. Failures in these systems can be extremely expensive [2], [22] and even deadly [75], [86] As a result, quality assurance is an essential part of development for these systems.

Alongside formal verification [48], [69], testing plays an essential role in ensuring the safety and quality of CPSs. Field testing importantly serves to assess whole-system behavior in realistic environments and is critical in identifying potentially catastrophic failures before deployment. However, field testing is inherently manual and time-consuming, and is potentially expensive, dangerous, and prone to errors of human judgment [8]. Automated testing, via software-in-the-loop (SITL) simulation, presents a promising alternative to testing whole-system behavior that is substantially faster, cheaper, and safer than manual field testing [7], [36], [39], [79], [95], [96], [104], [106].

• *The authors are with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA. E-mail: afsoona@alumni.cmu.edu, clegoues@cs.cmu.edu, ctimperley@cmu.edu.*

Manuscript received 15 September 2020; revised 29 August 2021; accepted 7 October 2021. Date of publication 15 October 2021; date of current version 14 November 2022.
This work was partially supported by AFRL (#FA8750- 15-2-0075), DARPA (#FA8750-16-2-0042), NSF-1563797, AFRL 19-PAF00747, and the NSF (#CCF-1750116).
(Corresponding author: Afsoon Afzal.)
Recommended for acceptance by C. Wang.
Digital Object Identifier no. 10.1109/TSE.2021.3120680

Fully automated whole-system testing requires oracles that can determine whether a given CPS behaves correctly for a given set of inputs [15]. In this paper, we tackle the problem of providing such an oracle for mature systems; systems that are in the final stages of development, and are ready to be tested at scale ahead of their deployment (e.g., technology readiness level of 4 or higher [72]). In research and practice, domain experts manually provide CPS oracles in the form of a set of partial specifications, or assertions to automatically evaluate the correctness of the system's behavior [6], [56], [68], [71], [89], [120]. However, manually writing such specifications is tedious, complex, and error-prone [39], [69], [76].

An attractive alternative to manual oracle specification is to automatically learn them, such as from CPS traces (e.g., [3], [27], [45], [50], [77]). Automatically learning oracles for CPSs presents several key challenges: (1) CPSs often contain proprietary third-party components (such as cameras or other sensors) for which source code is unavailable, and so techniques should minimize or avoid relying on source code access [20], [62]. (2) CPSs are inherently non-deterministic due to noise in both their physical (e.g., sensors, actuators, feedback loops) and cyber components (e.g., timing, thread interleaving, random algorithms) and may react to a given command in a potentially infinite number of subtly different ways that are considered to be acceptable [63], [117], as illustrated in Fig. 1. That is, for a given input and operating environment, there is no single, discrete response that is correct, but rather an envelope of responses that are deemed correct. And so techniques should be robust to small, inherent deviations in behavior. Finally, (3) the CPS may respond differently to a given instruction based on its environment, configuration, and other factors (i.e., its operating context) [21]. For example, in the scenario illustrated in Fig. 1, the copter may refuse to fly to the specified point if

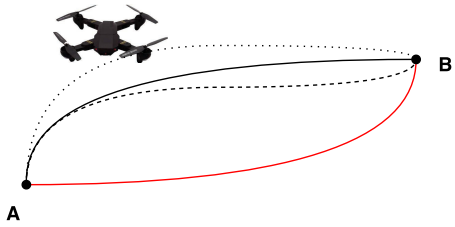


Fig. 1. Different example trajectories a quadcopter might take to perform the same set of instructions (flying from point A to point B). The black trajectories show acceptable behavior with respect to the instructions; the red trajectory shows erroneous behavior.

its battery is depleted. And so, techniques must be capable of capturing contextual behaviors for a given command. While these challenges individually are not unique to CPSs, their combination is rarely observed in other systems, making automated oracle inference for CPSs extremely challenging.

We present Mithra,[1] a novel oracle learning approach, based on anomaly detection, that tackles all of the above challenges. By observing many executions, Mithra identifies common behaviors, which it uses to construct its oracle. Without the need for source code access, Mithra accepts existing unlabeled telemetry logs as its input, which are typically produced by CPSs over the course of their operation. Mithra applies unsupervised multi-step clustering to its input data (i.e., *traces*) to construct a set of *behavioral clusters*, each representing a unique contextual behavior. Mithra determines whether a given execution trace (i.e., telemetry log) exhibits anomalous behavior, which is treated as erroneous [32], based on its similarity to identified behavioral clusters. Mithra tackles all three above-mentioned challenges as it does not require source code access, avoids overgeneralization and is robust to small deviations from expected behavior, and identifies contextual behaviors.

We evaluate Mithra on a dataset of 24 real bugs from the popular ArduPilot system. Our results show that Mithra effectively generates CPS oracles, and is more successful at doing so than prior work: Mithra correctly labels 69.3% of execution traces, outperforming AR-SI [45] (the previous state-of-the-art) by 11.5%. To demonstrate the wider applicability of Mithra, we evaluate it on a dataset of 153 artificial bugs in F1/10, an autonomous racer built on top of the popular Robot Operating System [92].

To the best of our knowledge, none of the prior work effectively tackles all three of the key challenges to CPS oracle learning. Some are not fully automated [27], [77], [116], or require access to source code [25], [33], [83]; others are affected by noise and non-deterministic behavior [11], [40], [65], [114], or cannot capture contextual behaviors [3], [12], [45]. One of the most recent techniques introduced by Chen *et al.* [25] tackles challenges 2 and 3 by learning behavioral models of the CPS, but requires source code access to generate labeled data for their supervised learning approach. AR-SI [45] tackles challenges 1 and 2 by checking the smoothness of the CPS's execution in terms of its sensor values. We provide a more detailed discussion of related work in Section 8.

This paper makes the following contributions:

- We present a novel oracle generation approach for CPSs, based on anomaly detection via multi-step multivariate time series clustering, that does not assume source code access, is robust to imperfect traces [94], and can be applied to any system that logs telemetry data, as is standard for CPSs.
- We evaluate our technique on ArduPilot, a popular, open-source autopilot. We collect a dataset of 24 bugs, and thousands of traces for ArduPilot. The dataset is publicly available to be used by researchers in the future.
- We demonstrate the wider applicability of our technique by applying it to a dataset of 153 artificial bugs for F1/10 system, a ROS-based autonomous racer. We make the dataset publicly available as part of our replication package.
- We evaluate Mithra's effectiveness by comparing against AR-SI [45], a state-of-the-art technique. We show that our technique performs significantly better in predicting correctness of traces.
- We provide a replication package for our study, complete with evaluation datasets, and source code for our prototype implementation: https://doi.org/10.6084/m9.figshare.14619177.

2 Case Study

As a running example, we describe ArduPilot, an autopilot software for a diversity of vehicles, including conventional airplanes, multirotor helicopters, and submarines, that is used by over a million vehicles across the world. At the time of writing, the ArduPilot codebase is primarily written in C++ and contains over 300,000 lines of code (measured using SLOC). ArduPilot has been widely used in studies on CPSs as it represents a fairly complex open-source CPS [5], [45], [66], [109], [120].

2.1 Motivating Scenario

ArduPilot is a mature autopilot software for CPSs that is used in a wide variety of vehicles and environments that are either in, or approaching, deployment. Although ArduPilot is functionally stable and used by over one million vehicles [1], it continues to evolve, and new issues and erroneous behaviors are continually discovered and reported over time. In 2019 alone, 722 new issues were filed on ArduPilot's issue tracker, of which 130 were labeled as bugs. Many of these bugs, such as the one described in Issue #9657, occur only under specific conditions and may result in behavioral changes that may have not been considered by ArduPilot's testing team.[2] In the case of Issue #9657, the vehicle misbehaves when instructed to navigate a series of waypoints that includes a spline path. By default, the vehicle will travel along a straight line between waypoints. However, operators may also instruct the vehicle to traverse a smooth path between waypoints along a spline. In the relatively rare event that a series of

1. Zoroastrian divinity of contracts, who is undeceivable, infallible, and eternally watchful: https://en.wikipedia.org/wiki/Mithra

2. Issue: https://github.com/ArduPilot/ardupilot/issues/9657 fixed by pull request https://github.com/ArduPilot/ardupilot/pull/10338 [Date Accessed: September 2, 2020].

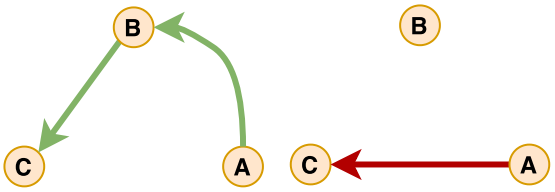


Fig. 2. A simplified depiction of the motivating example (ArduPilot's Issue #9657). The left figure illustrates the intended path of the vehicle from A to C via B. The vehicle is instructed to travel along a spline between A and B, before continuing along a straight line between B and C. The right figure illustrates the actual, erroneous path of the vehicle. The issue causes the vehicle to skip the spline waypoint B, and travel directly from A to C along a straight line.

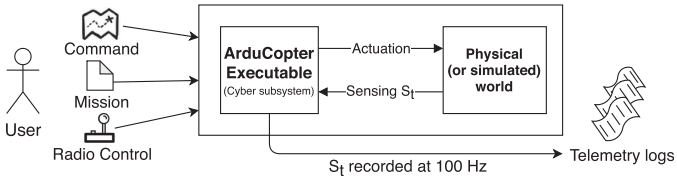


Fig. 3. A simplified view of the ArduCopter communications architecture. Input is provided by the user to the cyber component in the form of discrete commands and missions, or as a continuous radio control signal. The cyber component sends signals to actuate the physical component of the system, and reads sensor values. The state of the system, reported by the sensors, is periodically written to a telemetry log.

waypoints includes a spline path, the vehicle will erroneously skip the first waypoint along a spline path (Fig. 2).

Identifying such bugs requires both a means of *triggering* the bug (i.e., subjecting the system to a particular scenario and environment), and *detecting* that a failure has occurred (i.e., the system behaves in an unintended manner). Numerous studies on automated test input generation have focused on addressing the *triggering* problem [39], [44], [74], [78], [105], [106], [110]. Using artifacts and models of the system, or a search-based approach, these studies propose ideas on generating test inputs, scenarios, and environments that trigger and expose different behaviors of the system. In this work, we assume a means of triggering bugs and focus our attention on the problem of automatically *detecting* failures.

The example of ArduPilot and Issue #9657 motivates our approach in creating oracles for mature systems. As mentioned, a mature software (e.g., ArduPilot) performs common scenarios as expected. For example, when a vehicle is instructed to navigate to a target location, it performs as expected under most conditions. Note that if such common behavior becomes faulty in a mature system, the maintainers and testers would be alerted quickly, as it affects many users and scenarios. However, in circumstances involving behaviors that are less commonly used, such as scenarios that involve spline waypoints, the vehicle may misbehave and perform not exactly as expected.

In this work, we use a novel clustering approach to automatically identify the common behaviors of the system, which we use to form an oracle that can distinguish between expected and unexpected executions. In Section 4, we describe Mithra's approach for constructing such oracles.

2.2 ArduCopter's Architecture

For our running example, we use ArduPilot (version Copter-3.6.9) as the controller for a simulated quadcopter. Fig. 3 provides a simplified view of the cyber and physical components of ArduCopter. The user provides input to ArduCopter's cyber component in one of three forms: (1) as a discrete *command* from a ground control station, such as TAKEOFF, along with a set of parameters (e.g., desired altitude); (2) as a precomputed sequence of such commands, known as a *mission*; or (3) in the form of a continuous sequence of radio control signals. The cyber component of ArduCopter interacts with the physical component by polling its sensors at a fixed interval (e.g., once every 10ms) to determine its *extrinsic state* and sending signals to its actuators based on its extrinsic state and the user-provided input. The extrinsic state $\boldsymbol{s}_t$ of the system at time t describes the values of its *state variables*, each representing the value of a particular sensor, and is composed of both continuous (e.g., VELOCITY) and categorical values (e.g., STATUS). The cyber component of the system periodically logs its extrinsic state to a *telemetry log* at a fixed rate (e.g., 10 Hz). From the telemetry log, we extract an *execution trace* S for each command execution that records the sequence of extrinsic states logged during execution. We use the execution trace as input to our technique.

Fig. 4 provides a simplified example of two execution traces for the TAKEOFF command. Each execution trace can be represented as a *heterogeneous multivariate time series*: time series data consisting of multiple dimensions that include both continuous and nominal data. Since the time taken to complete an execution may vary, traces are variable in length and may consist of thousands of recorded states. For example, a 30-second execution of a single command results in a trace with 300 state observations if telemetry is recorded at 10 Hz. On another execution, the same command may take 50 seconds to complete and result in 500 state observations.

In this work, we restrict our attention to command-based user inputs and leave an application of our approach on continuous inputs to future work. We consider 10 out of 25 commands supported by the ArduCopter mission planner, shown in Table 1,[3] and 18 associated state variables describing properties like orientation, position, and velocity.[4] The 15 excluded commands consist of 10 commands specific to particular hardware (e.g., Do-Digicam-Control triggers the camera shutter if the copter is mounted with a camera), 4 commands controlling the mission planner itself and having little to no impact on the behavior of the system (e.g., Do-Jump skips commands in the mission), and 1 command, Loiter-Unlimited, that halts the execution of the mission planner, as the system loiters above a location indefinitely.

3 Clustering Multivariate Time Series

Our oracle learning approach builds oracles by clustering telemetry logs represented by multivariate time series (MTS). In this section, we provide the necessary background in MTS clustering to understand the techniques underlying our approach. Time series clustering has widely been used

3. http://ardupilot.org/copter/docs/mission-command-list.html [Date Accessed: September 2, 2020].

4. The full list of these 18 state variables are included as an appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2021.3120680.

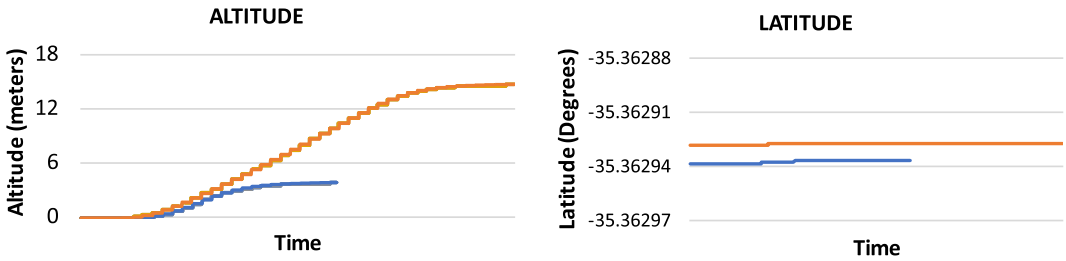


Fig. 4. An example of two execution traces for the ArduCopter's TAKEOFF command with respect to its ALTITUDE and LATITUDE state variables. In the bottom trace (blue), TAKEOFF(ALT:4.0), the copter elevates 4 meters above the ground. In the top trace (orange), TAKEOFF(ALT:14.7), the copter elevates 14.7 meters above the ground. In both cases, the LATITUDE remains roughly fixed.

to find common patterns in streams of data in a variety of domains including bioinformatics and biology, genetics, finance, air quality control, and meterology [13], [24], [28], [101]. In this paper, we present a novel formulation of MTS clustering that effectively and concisely encodes correct CPS behavior, and which can be used as an efficient oracle for the purpose of simulation-based testing.

k-Medoids: The k-medoids algorithm [58] is a clustering technique that uses a given distance metric to partition a given dataset into k clusters such that the distance between the points within a cluster and the center of that cluster (i.e., the *centroid*) is minimized. Unlike the well-known k-means, in which the center of a cluster is the average between its points, k-medoids uses an existing representative point within the cluster as its center. By using an existing point to represent the centroids of each cluster, k-medoids avoids the difficulties of computing a mean time series from a set of variable-length MTS, which may not be physically meaningful. Furthermore, k-medoids is attractive for clustering MTS datasets because it does not introduce additional, expensive distance calculations (e.g., measuring the distance between a given point and the mean of a cluster, as in k-means). k-medoids only compares existing points to one another, and so a distance matrix can be efficiently precomputed. When accounting for the multiple clustering runs that are necessary to determine a suitable k, k-medoids requires $O(n \cdot (n-1))$ unique distance calculations as opposed to $O(k^2n^2)$ required by k-means.

Distance Metrics: Any clustering approach requires a suitable distance metric. A common distance metric is euclidean distance (i.e., L2 norm), which is inexpensive to compute. However, euclidean distance can only be used for *same-length* MTS (i.e., time series of an equal duration and number of observations). In our case, where this assumption does not hold, we require an alternative distance metric. We discuss two alternative metrics that can compare variable-length MTS: Dynamic Time Warping [16] and Eros [115]. Table 2 provides a high-level comparison of these distance metrics in terms of their cost and associated qualities.

TABLE 1
A List of the Command Types Supported by ArduCopter's Mission Planner that are Considered in this Work, Their Number of Parameters, and a Brief Description of Their Function

Command Types	# of Params	Description
WAYPOINT	4	Straight navigation to waypoint.
SPLINE_WAYPOINT	4	Spline navigation to waypoint.
TAKEOFF	1	Takeoff from the ground.
LAND	2	Land on the ground.
LOITER_TURNS	4	Loiter & turn above a location.
LOITER_TIME	4	Loiter at a location for set time.
RETURN_TO_LAUNCH	0	Return to home location.
CHANGE_SPEED	2	Set the target horizontal speed.
SET_HOME	4	Set home location.
PARACHUTE	1	Trigger a parachute.

Dynamic Time Warping (DTW) [16], [41], [54], [55] is a similarity measure[5] that compares temporal sequences (i.e., traces) in terms of their "shape". DTW accounts for variations in duration, length, speed, and amplitude between two traces by mapping points from one trace to another trace via a non-linear process of "warping", illustrated in Fig. 5. DTW computes the optimal mapping between A and B such that every point in A is mapped to at least one point in B and vice versa in such a way that the order of points is retained, and the sum of distances between mapped points is minimized.

Although DTW provides a powerful means of comparing variable-length k-dimensional time series, it comes at the cost of a considerably $O(kmn)$ higher runtime complexity compared to $O(kn)$ complexity of the L2 norm, where m and n are the lengths of two time series. This can be reduced using a DTW approximation or lower bound such as FastDTW [99] or LB_Keogh [59].

The Extended Frobenius norm [115], or Eros, is a cheaper alternative to DTW that uses Principal Component Analysis (PCA) [4], [47], [90] to measure the distance between two variable-length MTS. Instead of measuring similarity between them by aggregating similarities between their individual variables, Eros treats each MTS as a matrix and uses the principal components to measure similarity.

Given an MTS dataset, Eros first determines the eigenvectors and eigenvalues of the covariance matrices of each MTS within the dataset. Eros then aggregates the eigenvalues to obtain *weights* for the dataset. Finally, Eros uses those weights to measure the similarity between two MTS in terms of their associated eigenvectors.

Eros is considerably cheaper to compute than DTW with an amortized runtime complexity that is linear in the number of variables in the MTS, and unlike euclidean distance, can be applied to variable-length MTS. Eros can account for differences in shape and is capable of handling shifts in time, but unlike DTW, it does not account for scaling over time.

5. Note that although DTW measures a distance-like quantity, it is not a *true* distance metric since it violates the triangle inequality: $d(\boldsymbol{x}, \boldsymbol{z}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z})$.

TABLE 2
A High-Level Comparison of Several Distance Metrics for Clustering Multivariate Time Series

	Euclidean Distance	Eros	DTW
Cost	Low	Low	High
Variable length MTS	✗	✓	✓
Agnostic to shift in time	✗	✓	✓
Agnostic to scaling over time	✗	✗	✓

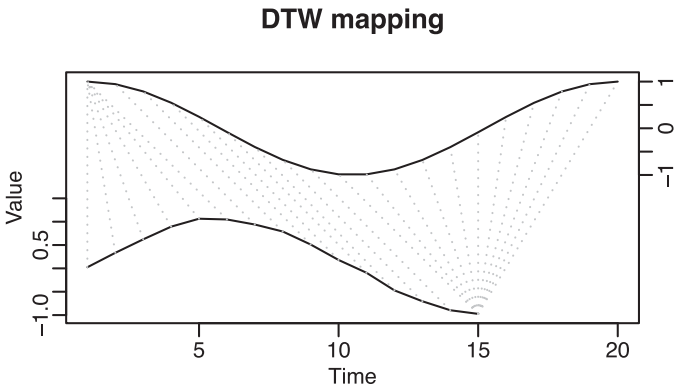


Fig. 5. Dynamic time warping measures the distance between two time series of unequal length by mapping the points in each time series onto the other via warping. The solid lines represent individual time series and the dashed lines represent the warping that maps one onto the other. By warping, DTW allows similarity between time series to be computed based on their *shape*.

4 Approach

In this section, we describe Mithra, our proposed unsupervised oracle learning approach, based on anomaly detection for mature cyberphysical systems: Section 4.1 presents an overview, Section 4.2 describes the preprocessing of the training data, Section 4.3 presents how Mithra learns oracles, and Section 4.4 describes how Mithra's oracles are queried. Finally, we discuss implementation details in Section 4.5.

4.1 Overview

Mithra learns oracles for CPSs that accept a vocabulary of discrete commands, and produce telemetry logs (e.g., ArduCopter). As Fig. 6 presents, Mithra first decomposes the telemetry logs (i.e., execution traces) consisting of sequential execution of multiple commands, and aggregates all command traces that represent the same command type together to create the training data for each command type.[6] Mithra uses the training data to identify clusters representing the different behaviors for each command type. For example, based on traces such as those in Fig. 4, Mithra detects one such common behavior, TAKEOFF(ALT:$<p_{alt}>$), in which ALTITUDE gradually increases until reaching a specified altitude p_{alt} while LATITUDE remains constant. Building oracles for each command type rather than individual test cases (i.e., missions), allows Mithra to derive oracle for a limited number of commands that can create thousands of different test cases. A test case is only considered *passing* when all commands in the mission perform as expected.

Approaches for clustering and classifying time series that are based on comparing differences in shape are often superior in terms of performance than those that compare differences in time [9], [93]. Unfortunately, clustering strictly with a DTW distance measure does not scale to large datasets. As a result, Mithra clusters execution traces based on overall shape using a three-step approach inspired by Aghabozorgi *et al.*'s method for clustering large time-series data [10]. Fig. 7 provides a high-level overview:

1) *Preclustering*: A low-resolution version of the training data is clustered into *preclusters* to reduce the search space.
2) *Purifying*: As the low-resolution preclusters are insufficiently accurate, Mithra next creates a set of *subclusters* for each precluster using high-resolution data.
3) *Merging*: Similar subclusters are merged to obtain a set of *behavioral clusters*, producing a simpler model that is cheap to query.

Using its learned *behavioral clusters*, Mithra constructs an oracle for each command based on anomaly detection, that marks execution traces as either CORRECT or ERRONEOUS based upon their similarity to the contextual behaviors represented by those clusters.

Note that although the structure of our technique draws inspiration from the prior work, Aghabozorgi *et al.*'s approach [10] can only be applied to datasets of time series with fixed length, and thus is not suitable off-the-shelf for our problem domain. Transforming traces to a fixed length would either require scaling and downsampling longer traces, losing important information, or padding shorter traces with nominal data that would introduce inaccuracies and reduce the effectiveness of clustering. Our novel contribution is to use k-medoids in a multi-step clustering process using Eros and FastDTW on a combination of both high and low-resolution data that carefully balances accuracy and efficiency to effectively discover clusters for variable-length data without introducing artifacts. Our multi-step clustering process carefully overcomes the considerable computational costs of simply using k-medoids and DTW on high-resolution trace data in a single clustering step, which cannot be applied to large number of traces in the dataset. In Section 5.5, we evaluate the effect of each step on overall performance.

Overall, Mithra's approach is designed to tackle all three challenges of testing CPSs outlined in Section 1:

1) Mithra does not require access to the source code or any other artifacts of the system. It is a blackbox approach that relies on readily available telemetry data to identify and distinguish between behaviors.
2) Mithra is, by design, tolerant to noise and non-determinism since it (a) builds its oracle using many observations of system behaviors, and (b) uses an acceptance threshold to account for noisy and non-deterministic data.
3) Mithra is capable of capturing context-dependent behavior while the context is captured by observable

6. From this point forward, we simply refer to command traces of a command type as *traces*.

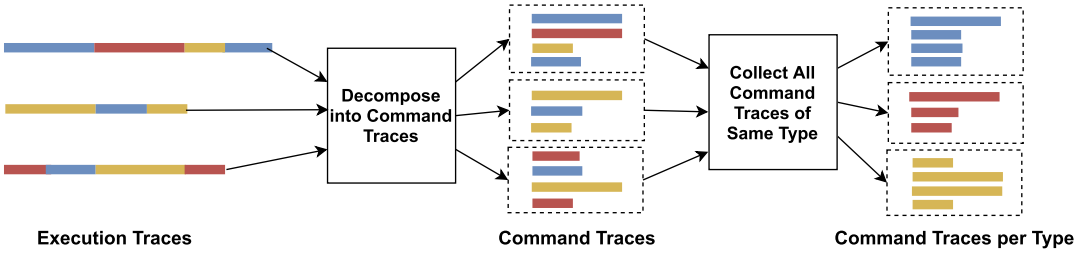


Fig. 6. An overview of preparing relevant traces to each command type from the mission execution traces that consist of sequential execution of multiple commands. Each color represents a different command type.

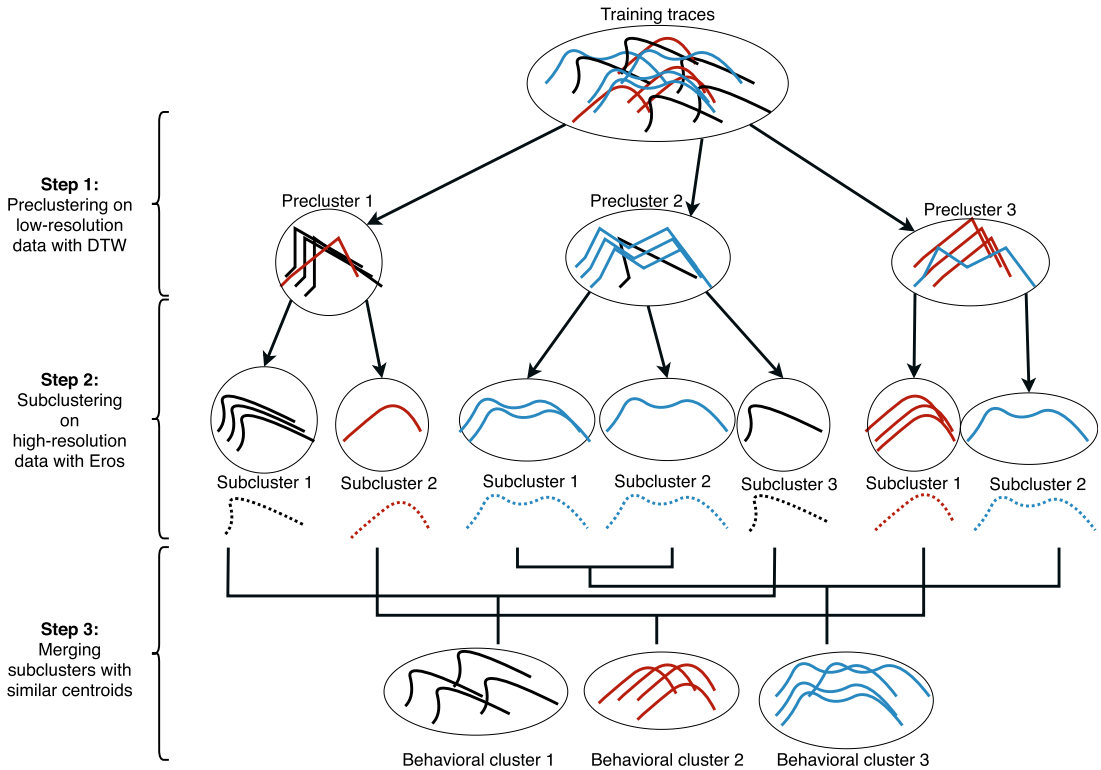


Fig. 7. An overview of Mithra's three-step clustering approach of preclustering, subclustering, and merging. Solid lines represent individual traces, and dashed lines represent cluster centroids.

variables. For instance, the differences in behavior due to battery level can be captured by Mithra when the effects of differing battery levels (e.g., decreased speed) is captured by the training data.

4.2 Training Data

In the training phase, Mithra takes, as input, a set of telemetry logs. Ideally, the set should contain logs that exercise all functionality of the system, covering a diversity of possible scenarios, though this is not a strict requirement. Mithra constructs an individual training set for each command type within the vocabulary of the CPS by extracting the relevant execution traces for that command type from the provided set of telemetry logs, shown in Fig. 6.

Note that, like most other techniques [3], [12], [45], [50], [81], Mithra is unsupervised. Thus, these training logs are not labeled in terms of whether they correspond to correct or erroneous behaviors. Similar to the prior techniques [11], [25], [33], [50], [65], [85], [116], Mithra makes the assumption that most programs behave correctly most of the time, and erroneous behavior is typically rare [32].

Mithra preprocesses training data in three ways:

1) *Converting Categorical Data.* Categorical variables (e.g., ArduCopter's MODE, which takes values such as `STABILIZE`, `AUTO`, and `GUIDED`), complicate distance measures, as the distance between two categorical datapoints can only be measured by whether they take the same value. Mithra converts categorical data to numerical data using one-hot encoding [29], where each category is turned into a dimension with binary value.
2) *Normalization.* Since it may not be meaningful to compare different state variables (e.g., VELOCITY and LATITUDE) due to differing ranges and units, we standardize [42] the data to ensure that differences in each state variable are treated with equal importance. Each dimension (i.e., state variable) within a time series is scaled to resemble a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$. Note that, by default and for the experiments reported in this paper, we scale dimensions according to a

normal distribution since we do not assume knowledge of the underlying distributions for each dimension. However, if such knowledge is available, an informed user may adjust this normalization step to use an alternative distribution (e.g., Poisson) in lieu of a normal distribution.

3) *Feature Selection*. Clustering techniques can suffer from the *curse of dimensionality* on datasets with many dimensions [23]. Therefore, Mithra accepts an option to select N_{FEATURES} dimensions in the training data with the highest entropy [30] as a preprocessing step. The entropy of a dimension X is defined as $H(X) = \sum_x P(x) log P(x)$ where $P(x)$ the probability of observing a particular value $x \in X$ High entropy in a dimension indicates that it can be informative in distinguishing different behaviors. We leave investigation of other feature selection approaches to future work.

4.3 Oracle Learning

Given a training set of traces for a command, Mithra attempts to identify the set of contextual (i.e., disjunctive) behaviors for that command. Mithra uses a three-step time series clustering approach that allows clustering to scale to a large number of detailed traces:

Step 1: Preclustering: Mithra first downsamples the training execution traces to produce a set of low-resolution traces to be clustered. By reducing the resolution of the data, Mithra can more efficiently compute DTW distance on an approximation of its input traces. The goal of this step is to reduce the search space for the subsequent, more computationally intensive steps.

To lower trace resolution, Mithra uniformly drops data points from each time series. For example, trace $t = [S_0, S_1, \ldots, S_{100}]$ with 101 data points can be downsampled to a lower-resolution trace $t' = [S_0, S_5, S_{10}, \ldots, S_{95}, S_{100}]$ with 21 data points. Even though t' does not represent the exact behavior of trace t, it approximates t's shape and can be used to create an initial set of *preclusters*.

To obtain the set of preclusters, Mithra applies k-medoids clustering to the low-resolution data using FastDTW [99] as its distance metric. The number of clusters k is obtained dynamically by finding the $1 < k < k_{max}$ that maximizes the silhouette score [97].

Step 2: Purifying: Since low-resolution data is used to obtain the set of preclusters, those preclusters may represent spurious patterns that do not hold on the original, high-resolution data. Therefore, in the second step, Mithra divides the contents of each precluster into multiple subclusters based on their Eros similarity. Although Eros is a less effective means of measuring similarity between traces than DTW (i.e., scale information is lost), it is inexpensive to compute and provides useful partial information about similarities in shape. To calculate the subclusters, we apply k-medoids clustering within each precluster, and find the *medoid* that is most representative of all traces within a subcluster.

Step 3: Merging: Finally, Mithra uses FastDTW to merge subclusters that share a similar shape based on the original, high-resolution data. This step prevents representation of the same contextual behavior by multiple subclusters, which leads to a simpler model that is cheaper to query. To do so, Mithra first computes the DTW distance among the medoids of subclusters using the original, high-resolution traces for those medoids. Although the time series are more detailed than those used during preclustering, the total number of time series, and, by extension, distance calculations, is far smaller, ensuring this step is scalable.

Mithra then uses the computed medoid distances to reduce the set of subclusters into a set of *behavioral clusters* by merging subclusters that share highly similar medoids. Mithra uses hierarchical clustering [53] to find the sets of similar subclusters. For every set of similar subclusters, Mithra constructs a new behavioral cluster that includes all their traces, and applies DTW averaging with uniform scaling [35] to the medoids of those subclusters to produce a centroid that best represents all traces in the new behavioral cluster.

Finally, Mithra uses FastDTW to compute μ_β and σ_β for each behavioral cluster β based on the distance from the traces within β to the centroid of that cluster $\boldsymbol{c}_\beta$, which Mithra uses to construct the decision boundary for β.

4.4 Oracle Querying

The behavioral clusters for each command represent qualitatively different modes of behavior observed for that command. These may include both behaviors that are frequently observed and assumed to be correct (e.g., clusters with more than one hundred traces), as well as behaviors that are rarely observed and suspected to be erroneous (e.g., clusters with fewer than five traces).

Mithra uses the behavioral clusters to predict whether a *new* trace is CORRECT or ERRONEOUS by comparing it to the centroid of its best-fit cluster. More formally, given a previously unseen execution trace τ for a command, Mithra first finds the behavioral cluster $\beta^*_\tau \in \text{BC}$ that most closely resembles τ based on the DTW distance between τ and the centroid of each cluster

$$\beta^*_\tau = \underset{\beta \in \text{BC}}{\arg\min}\, DTW(\tau, \boldsymbol{c}_\beta)$$

Mithra then uses β^*_τ to predict the label ℓ_τ for that trace as

$$\ell_\tau = \begin{cases} \text{ERRONEOUS} & \text{if } |\beta^*_\tau| < \rho \\ \text{ERRONEOUS} & \text{if } DTW(\tau, \boldsymbol{c}_{\beta^*_\tau}) > \mu_{\beta^*_\tau} + \theta\sigma_{\beta^*_\tau} \\ \text{CORRECT} & \text{otherwise} \end{cases}$$

where $|\beta|$ is the number of traces within β, $\rho \in \mathbb{Z}^+$ is the *rarity threshold*, and $\theta \in \mathbb{R}^+$ is the *acceptance rate*. If β^*_τ contains fewer than ρ traces, it is assumed to represent a rare, and thus, erroneous behavior, and so, τ is marked as ERRONEOUS. The rarity threshold allows Mithra to be more robust towards erroneous traces in the training data. In the more likely case where β^*_τ contains at least ρ traces, then β^*_τ itself is assumed to represent a common, and thus, correct behavior. In that case, Mithra uses the precomputed DTW distance to determine whether τ lies within the decision boundary of β^*_τ, and if so, labels it as CORRECT. The acceptance rate θ is used to alter the extent of the decision boundary and provides the user with a means of controlling the precision-recall tradeoff of the classifier to their preferences. We investigate and discuss the effects of θ in Section 5.3.

4.5 Implementation

Our implementation of Mithra, which we release as part of our replication package, allows tuning of parameters to our approach, such as resolution used during *Preclustering* and N_{FEATURES} (Section 4.2).

Derived Variables. One additional optional argument that can improve Mithra's performance is parameter handling. The behavior of a CPS with respect to a certain command often depends upon the parameters provided to that command. In the example of Fig. 4, if the copter flies to altitude of 10 meters instead of 4 when instructed to TAKEOFF (ALT:4.0), the trace should be marked as ERRONEOUS. However, by default, Mithra cannot connect two relevant dimensions in the traces (in this case, the ALTITUDE of the copter and the parameter passed to the command p_{alt}).

To account for parameter values, we can add new dimensions to input traces that are dynamically computed, and derived from other dimensions (i.e., values of parameters and state variables). For example, for the command TAKEOFF(ALT:$<p_{alt}>$), Mithra derives a new variable DIST_ALT as $(p_{alt} - \text{ALTITUDE})$, adds this variable to the time series and performs the rest of the approach on both the new and original variables. With this new dimension, Mithra's learned clusters represent that, for example, in CORRECT TAKEOFF ($<p_{alt}>$) traces, the value of DIST_ALT always converges to zero; we can mark ERRONEOUS cases where it does not (e.g., flying to 2 meters altitude when 5 is given as the parameter). Note that this added dimension does not specify the correct or expected behavior; it merely expresses a meaningful connection between parameters and state variables.

The definitions for derived dimensions are presently user-provided. As the number of command parameters is usually very limited and many commands share the same set of parameters, specifying these definitions is fairly simple. For example, many of the commands in ArduPilot take parameters related to location,[7] for which providing the definitions only once would be sufficient. For our case study of ArduPilot, we specify definitions for 4 derived dimensions that are shared among 7 of 10 commands. The definitions for these added dimensions are provided as part of our replication package. Note that Mithra *can* operate without these additional dimensions, but it will be less accurate. We anticipate that such dimensions are likely automatically discoverable, a prospect that we leave to future work.

Telemetry Sampling Rate. As its input, Mithra expects a set of telemetry traces that describe the state at approximately the same fixed time interval. For systems that use sensors with heterogeneous polling rates that are managed by different processes (e.g., in ROS), telemetry data for individual variables may be logged at different frequencies and offsets. In a preprocessing step, we produce an appropriate trace for Mithra by stepping through the telemetry at a fixed time interval, determined by the telemetry sampling rate, and using the most recently reported observation for each variable at each discrete time step.

The telemetry sampling rate can be increased to allow Mithra to better discriminate between traces. However, this will lead to longer training times and diminishing returns.

5 Evaluation

To determine whether our technique is an effective oracle learning method for mature cyberphysical systems, we conduct experiments, outlined in Section 5.2, on the case study system described in Section 2. We compare Mithra to the state-of-the-art [45] (AR-SI, described in Section 5.1). We answer the following research questions:

RQ1 (Accuracy) How accurately does our clustering method distinguish between correct and erroneous traces? (Section 5.3)

RQ2 (Comparison) How does the labeling accuracy of Mithra compare to AR-SI [45], a state-of-the-art oracle learning approach for cyberphysical systems? (Section 5.4)

RQ3 (Conceptual Validation) How do Mithra's individual steps influence its overall labeling accuracy? (Section 5.5)

RQ4 (Time) How long does it take to train and query Mithra, and how does it compare to AR-SI? (Section 5.6)

Finally, we evaluate Mithra on an autonomous racing CPS in Section 5.7 to show its applicability to systems beyond ArduPilot. In Section 5.8, we discuss threats to the validity of this evaluation.

5.1 Baseline

To compare our approach with the state-of-the-art, we reimplement He *et al.*'s approach for creating autoregressive system identification (AR-SI) oracles for CPSs [45].[8] Like our approach, AR-SI targets CPSs, does not assume source code access, does not require training on a bug-free, ground-truth version of the CPS, and operates on a multi-variate time series. Based on the assumption that many CPSs are designed to run *smoothly* when noise is under control, AR-SI determines whether a trace is erroneous or correct by checking the smoothness of the system's behavior. Let $Y_i \in \mathbb{R}^m$ represent the state of the system at time i with m state variables, and $U \in \mathbb{R}^q$ represent user input (i.e., command parameters). AR-SI models the relationship between U and Y_i as follows

$$Y_i = \left(\sum_{j=1}^{p} A_j Y_{i-j} \right) + BU + \xi_i \tag{1}$$

and optimizes model parameters $A_1, A_2, \ldots, A_p \in \mathbb{R}^{m \times m}$ and $B \in \mathbb{R}^{m \times q}$ so that the runtime accumulated SI error energy ξ_i is minimized. Then, AR-SI uses the optimal model parameters ($A_1^*, A_2^*, \ldots, A_p^*$ and B^*) to predict the next state of the system Y_{i+1}

$$\hat{Y}_{i+1} = \left(\sum_{j=1}^{p} A_j^* Y_{i+1-j} \right) + B^* U \tag{2}$$

and collects the prediction error as $e_{i+1} = \hat{Y}_{i+1} - Y_{i+1}$.

In other words, AR-SI uses the past p observed states of the system to predict its next state with the assumption that state changes tend to be smooth and the prediction error should be low. When the prediction error for all states in the trace is computed, AR-SI checks whether they contain

7. https://ardupilot.org/planner/docs/common-mavlink-mission-command-messages-mav_cmd.html#frames-of-reference

8. The source code of AR-SI is not publicly available, and we were unable to gain access via private email correspondence.

an outlier prediction error. If so, the trace is marked as ERRONEOUS, otherwise it is marked as CORRECT. Any prediction error outside of $\mu \pm 6\sigma$ is considered an outlier.

AR-SI was originally evaluated on our case study CPS; we discuss methodology next.

5.2 Experimental Methodology

We construct a benchmark for our case study, ArduPilot, which we use to evaluate our research questions. This benchmark consists of a *training dataset* and an *evaluation dataset*. The training dataset consists of unlabeled traces for 2500 randomly generated missions; it is used to train Mithra. The evaluation dataset provides a labeled, balanced set of 233 erroneous and 233 correct traces. We use it as ground truth when measuring the accuracy of Mithra and AR-SI (i.e., the ability to discriminate between erroneous and correct traces). Note that the labels of the evaluation dataset are not provided to either approach. Each mission that is generated for these datasets consists of between 1 to 8 commands from types presented in Table 1 (with repetition). Each command accepts between zero and four parameters as input.

To ensure reproducibility and avoid physical harm, we use software-in-the-loop (SITL) simulation to obtain traces in lieu of traces from real-world field testing. We sample state at a rate of 10 Hz according to the simulation clock rather than the wall clock, retaining the same information as a corresponding field trace. The collected mission traces in this experiment contain between 90 to 5400 sample data points (with median of 1640 data points). We use a 10 Hz sampling rate as the baseline approach, AR-SI, cannot handle a sampling rate higher than 10 Hz [45]. The practice of using simulation to obtain traces for this type of evaluation is common [11], [25], [45]. Below, we provide key details about benchmark construction.

Training Dataset. As a source of training data for our technique, we record traces for 2500 randomly generated missions in simulation; To accelerate data collection, we spread the process across 30 cores and use 40X simulation speedup. In total, we took roughly 15 hours to collect training traces. The generated missions of this dataset contain an average of 6.75 commands, with an average of 1.85 parameters per command.

Evaluation Dataset. We construct our evaluation dataset by first identifying 11 historical bugs via manual investigation of issues and bug-fixing commits on the ArduPilot repository.[9] We specifically target issues that impact the autonomous mission executor of the Copter vehicle: issues that are tagged as *bug*, and directed to Copter subsystem or all Ardu vehicles, and are executed by the `auto` mode (mission controller) of the system. Additionally, we only considered the issues that result in observable changes in the system behavior based on the description provided on the issue-tracker by the users, since both Mithra and AR-SI are only capable of identifying anomalies in the system's behavior. Therefore, configuration bugs and issues due to variation in equipment and software are out of scope for both approaches.

We transform each historical bug into a controlled *bug scenario* by manually grafting the bug onto the ground-truth version of ArduPilot, COPTER-3.6.9. By individually grafting the bugs onto the ground-truth version, rather than using those historical versions directly, we ensure that the only differences in behavior are due to a particular bug and not from an unrelated change to the program. We generate an additional 13 bugs by applying the same historical faults to other parts of the code, raising the total number of bug scenarios to 24.

For each bug scenario, we use a hand-written mission template, tailored to that scenario, to randomly generate 10 missions that trigger and manifest the bug. We create multiple missions for a single bug scenario since different set of commands and parameters may have different behaviors on the specified bug scenario, and 10 randomly generated missions can cover more variety of these changes in behavior. After running each mission, we collect line coverage of the execution to ensure that the executed mission does in fact execute the lines of interest (i.e., faulty lines).

Finally, we use the generated missions to construct an evaluation set of correct and erroneous traces. We obtain 240 erroneous traces by executing each bug scenario against its associated bug-triggering missions. However, we exclude traces resulting in software crashes (e.g., segmentation faults) from our dataset since those traces can simply be labeled as ERRONEOUS and no oracle is required. We exclude 7 out of 240 traces due to system malfunction. On average, the generated missions contain 5.58 commands and 2.36 parameters per command. We then obtain 233 correct traces by executing all 233 bug-triggering missions against the ground-truth version of the program, and create a balanced set of evaluation traces.

Comparison to AR-SI's Methodology. AR-SI was originally evaluated on a dataset of 8 historical ArduPilot bugs and 17 artificial bugs created by fault injection [45]. Similar to our approach, He *et al.* collect a set of traces, which are considered ERRONEOUS if they execute the faulty lines, and CORRECT otherwise. However, the AR-SI dataset of bugs and traces is not available publicly, and we have been unable to gain access via private correspondence. Therefore, we created a dataset of 24 real-life bugs and 466 traces, and release it as a benchmark to be used by studies in the future.

To evaluate the effectiveness of AR-SI, He *et al.* compared AR-SI against a "human oracle" devised by CPS experts. The human oracle consists of three rules that check that the velocity and angular velocity of the copter are within certain bounds (e.g., "velocity shall never exceed $\pm$20m/s"). He *et al.* found that AR-SI produced fewer false positives and false negatives than the human oracle. Approximately 70% of traces that were identified as erroneous by the human oracle were, in fact, correct. We choose not to evaluate against a human oracle since its performance is dependent upon the knowledge and skills of the experts, and therefore any comparison to such an oracle would not yield meaningful insights on the performance of Mithra or AR-SI.

Setup. To account for nondeterminism, we run each experiment on 20 different seeds. For all experiments, we run Mithra with maximum number of clusters $k_{max} = 15$, and feature selection $N_{\text{FEATURES}} = 10$. We have selected these options as a high (and safe) upper bound based on the number of different behaviors that can arise in a command according to the system's documentation, and our understanding of the number of features that can have significant

9. https://github.com/ArduPilot/ArduPilot

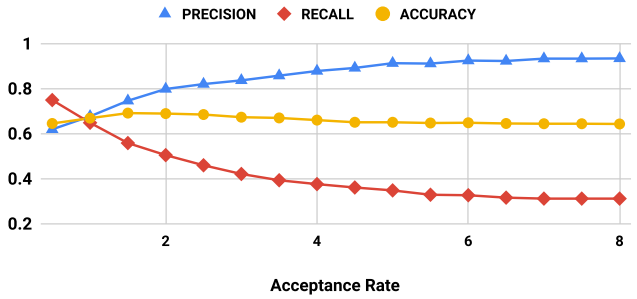


Fig. 8. Relationship between Mithra's median precision (blue triangles), recall (red diamonds) and accuracy (yellow circles) and acceptance rate used to classify outliers.

impact on the system's behavior with respect to a command. We discuss selection of rarity threshold ρ, and acceptance rate θ in Section 5.3. We run AR-SI with $p = 10$, which is the best performing parameter for this approach on ArduCopter reported by the original paper [45].

We conduct our experiments on a single machine, running Ubuntu 18.04, with the following specifications: TR 2990WX (32 cores), 64GB RAM, GTX 1080 Ti, and a 1 TB NVMe SSD. We used Python 3.6.2, TensorFlow 1.14.0, Docker 18.06.1-ce, PyClustering 0.9.0 [84].

Replication. We provide our source code, raw results, scripts to analyze those results, and benchmark traces as part of our replication package: https://doi.org/10.6084/m9.figshare.14619177.

Evaluation Metrics. To evaluate a candidate model (i.e., the output of our technique), we iterate over each trace in the evaluation set and check whether the label predicted by the model (i.e., CORRECT or ERRONEOUS) matches the expected label. Note that a trace is labeled as CORRECT if and only if all the command traces of that trace are labeled CORRECT. We then compute the number of true positives TP (erroneous traces marked as ERRONEOUS), false positives FP (correct traces marked as ERRONEOUS), true negatives TN (correct traces marked as CORRECT), and false negatives FN (erroneous traces marked as CORRECT). From those values, we obtain a summary of model performance:

Precision: fraction of traces reported as erroneous that are truly erroneous ($\frac{TP}{TP+FP}$).

Recall: fraction of erroneous traces reported as such ($\frac{TP}{TP+FN}$).

Accuracy: fraction of correctly labeled traces ($\frac{TP+TN}{TP+FP+TN+FP}$).

Note that we use accuracy rather than F1-score, defined as the harmonic mean of recall and precision, as an overall measure of performance as the F1-score places little weight on false positives and is best suited to imbalanced datasets. Below, we answer our research questions presented in Section 5.

5.3 RQ1: Accuracy

Using the training dataset, Mithra identifies a set of behavioral clusters for every command type in Table 1. On average, Mithra identifies 5.56 clusters per command type, ranging from 3 to 14 clusters per command type with median of 5. Among all identified clusters over all 20 seeds, 4% of clusters contain fewer than 5 traces, which can represent rare behavior in the training dataset.

TABLE 3
The Impact of Rarity Threshold ρ on Mithra's Performance With Respect to the Number of Traces that are Marked as ERRONEOUS Due to Rarity Over All 20 Seeds, and the Median Accuracy Reached by Mithra ($\theta = 1.5$)

Rarity threshold ρ	# of ERRONEOUS traces by rarity (20 seeds)	Median accuracy
0	0	69.3%
5	8	69.3%
10	78	68.8%

Fig. 8 illustrates the median performance of Mithra with different acceptance rates θ (with $\rho = 5$). As the acceptance rate increases, recall decreases and precision increases, resulting in a more conservative model that detects fewer erroneous traces overall, but ensures that traces marked as erroneous are more likely to be truly erroneous. Overall accuracy remains fairly steady as the acceptance rate is increased, demonstrating the tradeoff between false negatives and false positives. By modifying the acceptance rate, users can customize Mithra to their preferences [52], [98].

Overall, Mithra achieves a median accuracy of 66.5% across all seeds, and reaches its highest accuracy of 69.3% when its acceptance rate $\theta = 1.5$ (marking 74.7% of truly correct traces, correct). We therefore use acceptance rate $\theta = 1.5$ for the rest of our experiments.

To study the impact of rarity threshold ρ on Mithra's performance, we set its value to 0, 5, and 10, and compute the number of traces that are marked as ERRONEOUS duo to matching with a rare behavior cluster, and measuring the median accuracy of Mithra presented by Table 3. As expected, when $\rho = 0$ no trace considered as presenting rare behavior since no cluster has fewer than 0 traces in it to be considered rare. Compared to $\rho = 0$, $\rho = 5$ and 10 mark higher number of traces as ERRONEOUS duo to rarity (8 and 78 respectively). $\rho = 5$ reaches the same median accuracy of 69.3%, while $\rho = 10$ results in slightly lower median accuracy of 68.8%. We use the rarity threshold of $\rho = 5$ for the rest of our experiments.

As an example of a correctly detected behavior for ArduCopter, we take a look at the behavioral clusters for the LOITER_TIME(TIME, LAT, LON, ALT) command. According to the ArduCopter documentation,[10] the behavior of LOITER_TIME is described as "The vehicle will fly to and then wait at the specified location for the specified number of seconds." However, as stated in the documentation, if the given latitude and longitude are both set to zero, the copter should hold at its current location. Fig. 9 illustrates the behavioral clusters that were identified by Mithra for LOITER_TIME. Cluster 1 captures traces where the latitude of the copter changes drastically, whereas in Cluster 2, the latitude of the copter remains constant. In this example, we can see that Mithra automatically identifies the two correct behaviors of LOITER_TIME as stated in the documentation.

The motivating example described in Section 2.1 illustrates the case where the copter misbehaves on SPLINE_WAYPOINT command that is followed by another navigation

10. http://ardupilot.org/copter/docs/mission-command-list.html#loiter-time

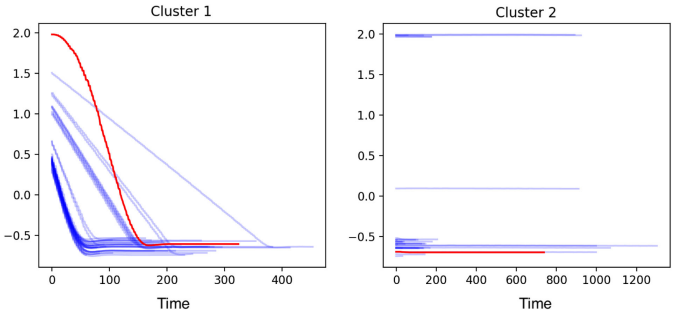


Fig. 9. Two behavioral clusters for LOITER_TIME that were learned by Mithra, plotted with respect to normalized LATITUDE (y-axis) over time (x-axis). Each blue line represents a single trace in the cluster, and the red lines represent the centroid of the cluster. The left cluster captures the behavior of the copter moving to a specified location before loitering, whereas the right cluster shows the behavior of remaining at its current location and loitering.

command. On 20 evaluation traces (10 correct and 10 erroneous) generated for this issue, Mithra reaches median accuracy, recall and precision of 70% , 90%, and 66.6%, respectively with $\theta = 0.5$, and 65%, 50%, and 71%, respectively with $\theta = 1.5$. Intuitively, this demonstrates that when Mithra is provided traces that trigger this issue, Mithra can correctly mark those traces as ERRONEOUS 90% of the time ($\theta = 0.5$).

5.4 RQ2: State-of-the-Art Comparison

Fig. 10 presents a comparison of the performance of Mithra against AR-SI. The median precision, recall, and accuracy of AR-SI are 62.2%, 39.0%, and 57.8% respectively, compared to Mithra's 74.7%, 56.0%, and 69.3%. Using a Mann-Whitney U test ($\alpha = 0.05$) we demonstrate that Mithra achieves significantly higher precision, recall, and accuracy compared to AR-SI. That is, Mithra detects a greater number of erroneous traces and does so with higher confidence.

We additionally use the intra-class correlation coefficient $\mathrm{ICC}(3,1)$ [61] to measure the reliability of Mithra and AR-SI across 20 seeds. This metric measures the consistency of a model in assigning the same label to a given trace across different seeds, and takes a value between zero and one; one being perfect reliability, and zero the complete absence of reliability. We find that Mithra demonstrates a "good" reliability of 0.840, whereas AR-SI exhibits a "poor" reliability of 0.349. Intuitively, this result shows that Mithra is more likely to assign the same label to a given trace regardless of the seed used during training.

Table 4 presents the performance of Mithra and AR-SI on each bug in our evaluation dataset over all 20 seeds. As presented, there are 10 bugs that Mithra labels with more than 75% accuracy, out of which 5 are historical bugs. However, AR-SI is only capable of labeling traces related to bug #8 with more than 75% accuracy. This table also shows that Mithra can label traces of four bugs with zero false negatives, meaning that it marks all faulty traces for those bugs as ERRONEOUS over all 20 seeds, and the accuracy on those bugs are all above 85%, which shows it is also accurate in labeling the correct traces. AR-SI is not able to label traces for any of the bugs with zero false negatives. In fact, the lowest number of false negatives it reaches is 60 traces for bug #8. If we only consider the 11 historical bugs, Mithra reaches mean accuracy, precision, and recall of 70.6%, 75.0%, and 58.6%, while AR-SI reaches 58.6%, 62.4%, and 39.2% respectively.

TABLE 4
For Each Bug, the Total Number of Traces that are Marked True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) Over All 20 Seeds Per Approach, and the Overall Accuracy (Acc.) on Labeling Traces for Each Bug

	Mithra					AR-SI				
Bug	TP	TN	FP	FN	Acc.	TP	TN	FP	FN	Acc.
★1	53	166	34	147	54.7	63	145	55	137	52.0
▲2	70	130	50	110	55.5	53	132	48	127	51.4
▲3	72	63	77	68	48.2	42	121	19	98	58.2
★4	48	118	82	152	41.5	58	151	49	142	52.2
★5	106	160	40	94	66.5	43	168	32	157	52.7
★6	54	171	29	146	56.2	48	160	40	152	52.0
★7	200	176	24	0	94.0	119	156	44	81	68.7
★8	200	158	42	0	89.5	140	163	37	60	75.7
★9	58	171	29	142	57.2	56	159	41	144	53.7
★10	85	190	10	115	68.7	66	171	29	134	59.2
▲11	185	165	35	15	87.5	124	162	38	76	71.5
▲12	67	165	35	133	58.0	44	162	38	156	51.5
★13	149	171	29	51	80.0	107	142	58	93	62.2
▲14	179	179	21	21	89.5	117	163	37	83	70.0
▲15	99	177	23	101	69.0	64	150	50	136	53.5
★16	151	154	46	49	76.2	102	126	74	98	57.0
▲17	200	145	55	0	86.2	104	146	54	96	62.5
▲18	105	148	52	95	63.2	86	133	67	114	54.7
▲19	140	117	23	0	91.8	73	109	31	67	65.0
▲20	148	169	31	52	79.2	82	156	44	118	59.5
▲21	63	152	48	137	53.7	60	142	58	140	50.5
★22	121	183	17	79	76.0	46	164	36	154	52.5
▲23	29	181	19	171	52.5	43	147	53	157	47.5
▲24	63	149	51	137	53.0	68	154	46	132	55.5

The bugs are either historical (★) or inspired by historical (▲) bug.

5.5 RQ3: Conceptual Validation

Each of the three steps of Mithra's clustering approach is designed to improve the accuracy of its detected clusters

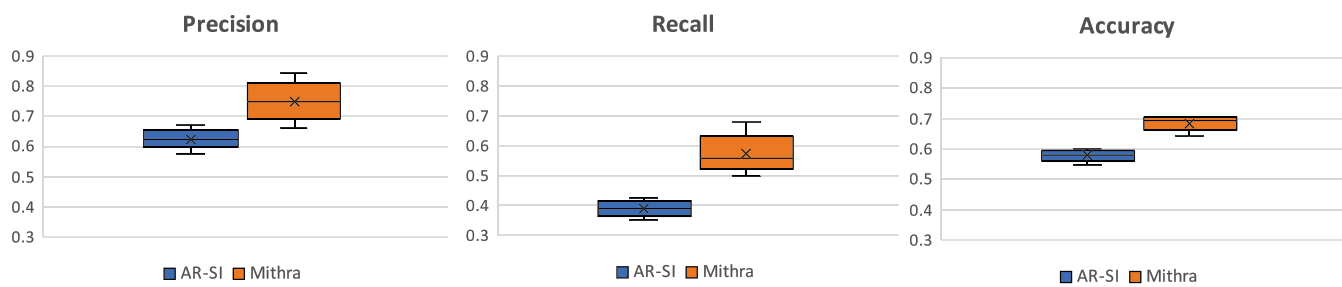
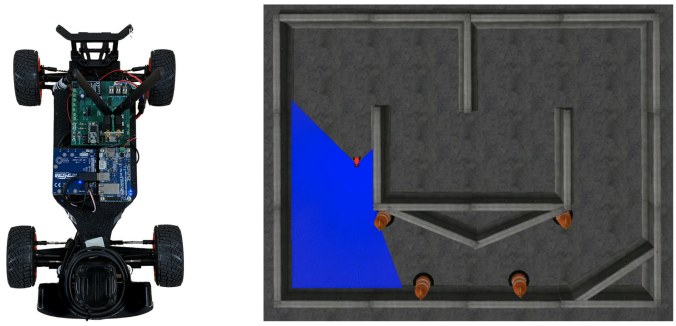


Fig. 10. A performance comparison between AR-SI and Mithra. Using a one-sided Mann-Whitney U test, we show that Mithra outperforms AR-SI significantly ($\alpha = 0.05$) in terms of precision, recall, and accuracy.

TABLE 5
A Comparison of Mithra's Performance When the Output Clusters of One of its Steps is Used to Construct the Classifier in Terms of Precision, Recall, and Accuracy, Reported by Their Median and Interquartile Range (IQR) Measuring the Difference Between 75th and 25th Percentiles Across 20 Seeds

	Preclusters		Subclusters		Beh. Clusters	
	Median	IQR	Median	IQR	Median	IQR
Precision	0.52	0.01	0.72	0.06	0.75	0.06
Recall	0.96	0.03	0.59	0.04	0.56	0.04
Accuracy	0.54	0.02	0.68	0.02	0.69	0.02

Using a one-sided Mann-Whitney U test [73], we show that both Behavioral Clusters and Subclusters have significantly higher precision and accuracy, and lower recall than Preclusters ($\alpha = 0.01$). We are unable to find a significant difference between Subclusters and Behavioral Clusters.



(a) (b)

Fig. 11. (a) The F1/10 vehicle; one tenth of the size of a real Formula 1 race car. (b) A simulated race track with four obstacle cones (orange), the F1/10 vehicle (red), and the range covered by the vehicle's sensors (blue). The vehicle follows the inside or outside walls to navigate through the track counter-clockwise, and avoids obstacles.

while supporting scalability. To evaluate the individual impact of those steps, we use the output produced by each step (i.e., preclusters, subclusters, and behavioral clusters) as input to oracle querying, which we then use to measure the performance of each step (Table 5).

Using the outputs of either the second or third step of our approach (i.e., subclusters and behavioral clusters) to produce a classifier results in significantly higher precision and accuracy ($\alpha = 0.05$) than a classifier constructed using the output of only the first step (i.e., preclusters). This finding demonstrates that solely using Dynamic Time Warping on low-resolution data is insufficient on its own for precisely detecting behavioral patterns.

We are unable to show a significant difference in performance between using subclusters and behavioral clusters. Recall, however, that the intention behind Mithra's third step is not to improve functional performance, but rather to effectively reduce the number of reported clusters by combining clusters that represent the same behavior. On average, Mithra identifies 21 subclusters for each command, which it reduces to an average of 5.5 behavioral clusters after its merging step. By merging non-unique clusters, we reduce the cost of oracle querying by decreasing the number of expensive DTW distance calculations. Furthermore, reporting fewer clusters may ultimately aid user comprehension of the discovered behaviors and thus provide higher confidence in the output of the technique. However, non-unique clusters do not impact Mithra's performance since oracle querying is independent of cluster uniqueness. Our results provide empirical evidence that the process of merging clusters is indeed effective at reducing the number of clusters, and does not have any significant impact on overall performance, thereby indicating that information is preserved.

To investigate the importance of preclustering, we apply step 2 of Mithra's approach in isolation to the original training traces. The resulting classifier obtains a median precision, recall, and accuracy of 53.2%, 90.9% and 55.4%, respectively, providing evidence that preclustering of low-resolution traces with DTW before subclustering results in significantly higher precision and accuracy ($\alpha = 0.01$).

5.6 RQ4: Time

Our approach for automatically generating CPS oracles requires an up-front training stage, whereas AR-SI can simply be applied to evaluation traces without training. Although Mithra's training can take several hours to complete, depending on the size of the training data, that cost only needs to be paid once and can be amortized. For our experiments, Mithra's training took 4 hours and 45 minutes to complete and was spread across 30 threads. However, by storing and reusing computed distance matrices, Mithra's training time for subsequent seeds was reduced to an average of 29.59 minutes. AR-SI's cost of labeling a single query trace is relatively expensive, since it repeatedly optimizes a number of parameters for every datapoint in the trace. In our experiments, on average, it took 27.65 minutes for AR-SI to label all evaluation traces using 30 threads (i.e., each trace took approximately 107 thread-seconds). For Mithra, it took an average of 2.79 minutes to label all evaluation traces using 30 threads (i.e., each trace took approximately 11 thread-seconds). Using an independent samples t-test, we show that querying Mithra is significantly ($p < 0.001$) faster than AR-SI.

Overall, Mithra does require an upfront training cost that AR-SI does not; given a trained model, oracle querying for Mithra is approximately 10X faster than AR-SI.

5.7 Wider Applicability

To show that Mithra is not limited to a single system (i.e., ArduPilot), we demonstrate Mithra on the F1/10 platform [87], shown in Fig. 11. F1/10 is an open-source, autonomous racing cyber-physical platform, one tenth of the size of a real Formula 1 racing car, that is designed to be used as a testbed for research and education. We chose F1/10 as an additional case study to demonstrate the applicability of Mithra to a system built on top of the Robot Operating System [92], the most popular robotics development platform, sometimes referred to as the "Linux of Robotics" [111].

In this experiment, we use Mithra to learn an oracle for the wall-following command of F1/10,[11] in which the vehicle uses its sensors to complete laps around the race track without crashing. The wall-following command takes a single parameter that specifies whether the vehicle should follow the inside or outside walls of the track. The vehicle will indefinitely complete laps around the track in a counter-clockwise direction, remaining close to desired wall, until

11. https://github.com/linklab-uva/f1tenth_gtc_tutorial

instructed to stop by the user. Since "missions" for this system consist of a single command of indefinite duration, we impose a wall-clock time limit when collecting traces. These traces consist of seven state variables, describing the vehicle's position and orientation at each point of observation.

We assess Mithra on F1/10 using a similar approach to our evaluation on ArduPilot, outlined in Section 5.2, by constructing a benchmark. We use simulation to construct a training dataset of 75 unlabeled traces, covering both inside and outside wall-following behaviors. Note that we collect substantially fewer training traces for F1/10 compared to ArduPilot since the latter has a greater set of commands and parameters. To construct an evaluation dataset, we first automatically inject 234 faults into the F1/10 source code using four mutation operators: Wrong Arithmetic Operation, Wrong Value Assigned to a Variable, Missing Parentheses, and Wrong Logic Clause. We use Comby [107], a tool for searching and changing code structure, to apply the mutations to the code. We use artificial faults for evaluation since F1/10 does not have a rich enough development history to extract historical faults. After running the command with both parameters on the syntactically valid, non-crashing bugs and collecting the traces, we manually identify the mutants that led to failure (i.e., crashing into obstacles). We identify 153 mutants and produce 261 faulty traces. To ensure a balanced evaluation dataset, we collect an additional 261 traces using the unmodified F1/10 system.

We run Mithra with rarity threshold $\rho = 5$, maximum number of clusters $k_{max} = 15$, and without feature selection, and repeat the experiment with 20 seeds. Mithra reaches its highest median accuracy (81.0%) when $\theta = 1$, with median precision and recall of 84.6% and 74.9%, respectively. By comparison, AR-SI achieves its highest median accuracy (51.3%), with a median precision and recall of 51.7% and 37.1%, respectively, when $p = 10$.

The high performance of Mithra on F1/10 may be explained by how erroneous behaviors in this system manifest. In most cases, the vehicle misbehaves smoothly, and does not necessarily show sudden, unexpected changes; rather, it slowly navigates along the wrong path. Mithra detects that the behavior does not match previously identified behavioral clusters. In contrast, AR-SI only detects erroneous behaviors that involve abrupt changes, which may explain why AR-SI performs poorly on F1/10.

Overall, these results demonstrate the wider applicability of Mithra by showing that Mithra can be successfully applied to another system (i.e., F1/10).

5.8 Threats to Validity

Construct Are we asking the right questions? We assess Mithra's effectiveness as an oracle learning technique by measuring how accurately its generated oracles discriminate between correct and erroneous system behavior. We use precision, recall, accuracy, and querying time as metrics of performance, and compare to AR-SI, a state-of-the-art oracle learning approach that operates under the same assumptions, as a baseline. To gain a deeper insight into how Mithra works, we evaluate how each of its individual steps contribute to the overall effectiveness.

Internal Did we skew the accuracy of our results with how we collected and analyzed information? In many CPSs, executing faulty lines and triggering a bug does not guarantee that the bug will manifest. However, many of our bugs are associated with a bug report on ArduPilot's issue tracker and describe missions that manifest the bug. We create mission templates based on the bug reports and our own understanding of the bugs, and generate random missions from those templates. The mission templates are a source of internal validity.

As the source code of AR-SI was not available to us, we implemented our own version of AR-SI based on the description provided in the paper [45]. Our implementation of AR-SI represents a potential threat to internal validity. We release our implementation of AR-SI as part of our replication package.

External Do our results generalize? In theory, our approach is applicable to any CPS that logs its telemetry data. However, we only evaluate on two instances of such systems. We pick ArduPilot as a fairly complex and highly popular system that is widely used as a representative of real CPSs in prior work [5], [45], [66], [109], [120], and we pick F1/10 as system built on top of the popular Robot Operating System [92].

In this paper we only evaluated Mithra on traces collected over command-based mission executions, since commands trigger autonomous control of the system that is expected to perform a set of actions. In other words, using the mission planner and providing a set of commands, we focus on the system's behavior in autonomous mode (i.e., `auto` mode in ArduPilot) rather than manual control. In theory, Mithra can be adapted to handle continuous inputs (e.g., controlling the quadcopter with joystick), using techniques such as sliding windows [118], and meta-featuring [49]. However, we leave investigating applicability of Mithra on continuous commands to future work.

Although Mithra is agnostic to the source of its traces and can be applied to field traces, we do not evaluate Mithra on field traces and leave that for future work.

Replicability Can others replicate our results? To allow others to inspect, replicate, and extend our experiments, we provide a replication package for our study, containing our evaluation datasets and the source code for Mithra and our implementation of AR-SI.

Conclusion Did we draw correct conclusions from our data? From our experiments, we conclude that Mithra outperforms AR-SI according to several important measures of performance: precision, recall, accuracy, and querying time. We measure performance on a balanced set of 466 execution traces, representing a variety of operating conditions (e.g., mission, particular bug), and use an appropriate one-sided non-parametric test (Mann-Whitney U) to demonstrate statistically significant improvement.

6 Assumptions and Limitations

In this section, we discuss the assumptions made by Mithra, and the limitations that affect our approach as a result of making these assumptions.

Anomalous-Yet-Correct Behavior. Our approach, like others, treats anomalous behavior as erroneous, and common erroneous behavior as correct [11], [25], [33], [50], [65], [85], [116]. However, anomalous behavior also includes corner cases

and rare behaviors that are not observed during training, which are not necessarily erroneous, and erroneous behavior can be observed in the training data. Even though reporting the anomalous-yet-correct behaviors as erroneous is not ideal, and can result in false-positives, it can inform the developers of under-tested functionality. In addition, most systems typically perform as expected [32], as it is easier for developers to detect and debug an erroneous behavior that is observed frequently.

Dependency on Training Data. Overall, the performance of our approach depends on its training data, a limitation it shares with other dynamic model learning techniques [11], [33], [65], [85]. If the provided traces do not provide sufficient coverage of the unique behaviors of the robot, our approach will fail to identify those behaviors. However, generating a diverse set of training data is an orthogonal problem we leave to future work. Additionally, even though Mithra works on any CPSs that generates telemetry logs, it is limited by the variables that are recorded in these logs, and the extent of which these variables truly represent the system's behaviors. For example, if the telemetry logs of a smart thermostat does not include data on the environment's temperature, Mithra is not capable of truly capturing the system's behavior.

Sequential, Synchronous Execution. Our approach assumes sequential execution of commands and cannot handle asynchronous or concurrent executions. This assumption limits the applicability of Mithra on systems that require multi-process, asynchronous operation, which is prevalent among CPSs. Taking ideas from testing distributed systems [17], [18], we can include such traces in our approach in the future.

7 Future Work

In this section, we discuss opportunities for future work, including potential improvements to the approach and opportunities for additional application and further evaluation.

Application, Usability, and Further Evaluation. In this paper, we applied Mithra to two real-world, exemplar systems, and evaluated Mithra's ability to accurately distinguish between CORRECT and ERRONEOUS traces. However, we did not evaluate or explore the usability of our approach in terms of its ability to to help developers to identify, locate, and address faults as part of a larger quality assurance approach. For example, to aid in debugging an observed failure, Mithra could be adapted to provide the user with information about the variables within trace that contribute most to its ERRONEOUS label (e.g., an altitude is abnormally large). As such an investigation would require human studies that are beyond the scope of this paper, we leave exploration of this possibility to future work, along with an exploration of how Mithra can be integrated into fault localization and automated program repair approaches.

In this work, we limited the evaluation of our approach to a single configuration of the system. Mithra can naively handle different configurations by learning a different set of clusters under each configuration. However, such an approach would almost certainly be prohibitive in terms of the amount of required training data and the time taken by clustering. Alternatively, configuration variables could be injected into the traces themselves during clustering, allowing meaningful differences due to configuration to be identified during feature selection and incorporated into the learned clusters. Given the considerable challenges involved in efficiently exploring configuration spaces [57], we leave an exploration of how Mithra can be adapted to account for such variability to future work.

Another important factor in the input space of CPSs is the operating environment. However, capturing and compactly encoding relevant environment features within a trace is a difficult research problem that requires novel ideas. In this work, we evaluated Mithra without directly including any information about the simulated environment (e.g., weather, obstacles, terrain). Some of this information is easier to include (e.g., barometric pressure, wind speed), while other pieces of information are trickier to represent (e.g., the position and heading of nearby aircraft). This limits Mithra's ability to relate certain learned behaviors to specific environment factors. In other words, when Mithra is applied to training traces that has been collected in different environments, it can recognize those different behaviors (e.g., behavior A that is only performed in presence of strong winds), but Mithra is unable to relate these behaviors to correlated environmental factors. If, for example, behavior A incorrectly occurs in a non-windy environment, Mithra would not be able to mark it as ERRONEOUS. Additionally, effectively exploring the environment space, and automatically evolving the simulated environments in such a way that they expose the system to more diverse scenarios is an important area that needs to be investigated in the future [34], [60], [70], [80], [112].

Improving Mithra. Throughout the paper, we have highlighted several opportunities to further improve various aspects of Mithra, including alternative feature selection and normalization approaches, and automating the discovery of derived variables. Below, we briefly discuss two additional opportunities for improvement: incremental training and developer-assisted cluster validation.

In its current form, the clusters found by Mithra cannot be evolved to incorporate newly collected data: The training process must be repeated from scratch with an updated dataset to recompute the clusters such that they accurately reflect any new, modified, or removed behaviors of the system. This limitation can make Mithra expensive to practically deploy on frequently evolving systems. In future studies, we intend to explore whether Mithra can efficiently reuse and evolve previously identified clusters to account for newly collected data.

The accuracy and utility of Mithra could be improved by using a semi-automated approach that uses feedback from the developer to confirm the correctness of discovered clusters and identify small clusters that correspond to rare, erroneous behavior exhibited in the training set. This could take place by, for example, presenting several representative traces from each of the identified clusters to the user and asking the user whether those traces are indeed CORRECT or ERRONEOUS. This additional step would help to overcome Mithra's limitations in identifying frequent-but-erroneous and rare-but-correct behaviors during training.

8 Related Work

Mithra is most closely related to prior work on anomaly detection in cyberphysical systems [26], [37], [43], [46], [81], [88], [108], [120]. We have already positioned Mithra with respect to AR-SI [45] in detail in Section 5.1. Chen *et al.* [25] build models by combining mutation testing and machine learning: they generate faulty versions (mutants) of the tested system and then learn SVM-based models using supervised learning over the resultant data traces corresponding to system execution. They evaluate on a model of a physical water sanitation plant. Our system improves on this prior work by obviating the expensive mutant-generation step by virtue of making use of unsupervised learning techniques. Ghafouri *et al.* [38] show that common supervised approaches in this context are vulnerable to stealthy attacks. An unsupervised technique [50] evaluated on the same treatment plant model trains a Deep Neural Net (DNN) to identify outliers (similar in spirit to our approach), but cannot be applied to time series data, a key concern in many CPSs. Ye *et al.* [116] use a multivariate quality control technique to detect intrusions by building a long-term profile of normal activities in information systems and using the norm profile to detect anomalies. However, it is a parametric technique and is not fully automated.

To generate test oracles for CPSs, Menghi *et al.* [77] propose an approach that automatically translates CPS requirements specified in a logic-based language into test oracles specified in Simulink. However, writing the specifications for requirements of a CPS is difficult and error-prone [39]. Other approaches target the detection of particular attack classes specifically. Choi *et al.* [27] present a technique that infers control invariants to identify external physical attacks against robotic vehicles; its models combine knowledge about a vehicle's physical properties and control algorithms, as well as the laws of physics. Like AR-SI, it uses system identification (SI) to detect malicious attacks on CPSs; however, it requires a training step. Alippi *et al.* [12] learn Hidden Markov Models of highly correlated sensor data that are then used to find sensor faults. Abbaspour *et al.* [3] train adaptive neural networks over faults injected into sensor data to detect fault data injection attacks in an unmanned aerial vehicle. Our approach does not target a particular class of failures, and can be used to detect both sensor faults and attacks on the system.

Other techniques infer invariants or finite state models describing correct software, which is known as dynamic specification mining [11], [17], [19], [31], [33], [40], [51], [64], [65], [82], [83], [85], [100], [114]. Most require source code access or instrumentation, and none are suitable for time series data. Techniques like Daikon [33] and its numerous successors [31], [40], [82], [83] learn source- or method-level data invariants rather than models of correct execution behavior. Jiang *et al.* [51] use Daikon on messages that are passed between different processes in ROS systems to learn invariants that apply to the messages. Techniques like Texada [65] and Perracotta [114] do learn temporal properties between events but do not model or learn temporal data properties, a key primitive in CPS execution (Artinali [11] comes closest to this goal, learning event ordering and data properties *within* an event). Other techniques use console logs generated by the system as the source for mining invariants and detecting anomalies [14], [67], [102], [113]. Overall, such techniques target orthogonal use cases and systems as compared to our context.

As another way of approaching the oracle problem for CPSs, studies have used metamorphic testing to observe the relations between the inputs and outputs of multiple executions of a CPS [66], [103], [119]. Lindvall at al. [66] exploit tests with same expected output according to a given model to test autonomous systems. Zhou and Sun [119] use metamorphic testing to specifically detect software errors from the LiDAR sensor of autonomous vehicles. Tian *et al.* [103] introduce DeepTest, a testing tool for automatically detecting erroneous behaviors of DNN-driven vehicles. As an oracle, they use metamorphic testing by checking that properties like steering angle of an autonomous vehicle remain unchanged in different conditions such as different weather or lighting.

9 Conclusion

In this paper, we introduce Mithra, an automated tool that demonstrates a three-step multivariate time series clustering approach as an effective means of generating oracles for cyberphysical systems. As part of our evaluation on a widely used robotics platform, we show that Mithra identifies a higher number of faulty executions than AR-SI, a state-of-the-art oracle generation technique for CPSs, and does so with a higher level of confidence. We show that Mithra is generally more reliable and may be used to provide an oracle for automated, simulation-based testing as part of a continuous integration and deployment workflow.

Acknowledgments

The authors are grateful for their support.

References

[1] About ArduPilot, Accessed: Sep. 02, 2020.
[2] Schiaparelli landing investigation makes progress, 2016. Accessed: Sep. 02, 2020. [Online]. Available: http://www.esa.int/Our_Activities/Human_and_Robotic_Exploration/Exploration/ExoMars/Schiaparelli_landing_investigation_makes_progress
[3] A. Abbaspour, K. K. Yen, S. Noei, and A. Sargolzaei, "Detection of fault data injection attack on UAV using adaptive neural network," *Procedia Comput. Sci.*, vol. 95, pp. 193–200, 2016.
[4] H. Abdi and L. J. Williams, "Principal component analysis," *WIREs Comput. Statist.*, vol. 2, no. 4, pp. 433–459, 2010.
[5] M. Afanasov, A. Iavorskii, and L. Mottola, "Programming support for time-sensitive adaptation in cyberphysical systems," *ACM SIGBED Rev.*, vol. 14, no. 4, pp. 27–32, 2018.
[6] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proc. Int. Conf. Softw. Testing Verification Valid.*, 2013, pp. 352–361.
[7] A. Afzal, D. S. Katz, C. Le Goues , and C. S. Timperley, "Simulation for robotics test automation: Developer perspectives," in *Proc. Int. Conf. Softw. Testing Valid. Verification*, 2021, pp. 263–274.
[8] A. Afzal, C. Le Goues , M. Hilton, and C. S. Timperley, "A study on challenges of testing robotic systems," in *Proc. Int. Conf. Softw. Testing Valid. Verification*, 2020, pp. 96–107.
[9] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah, "Time-series clustering–a decade review," *Inf. Syst.*, vol. 53, pp. 16–38, 2015.
[10] S. Aghabozorgi and T. Y. Wah, "Clustering of large time series datasets," *Intell. Data Anal.*, vol. 18, no. 5, pp. 793–817, 2014.

[11] M. R. Aliabadi, A. A. Kamath, J. Gascon-Samson , and K. Pattabiraman, "ARTINALI: Dynamic invariant detection for cyber-physical system security," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2017, pp. 349–361.

[12] C. Alippi, S. Ntalampiras, and M. Roveri, "Model-free fault detection and isolation in large-scale cyber-physical systems," *Trans. Emerg. Top. Comput. Intell.*, vol. 1, no. 1, pp. 61–71, 2016.

[13] A. Bagnall and G. Janacek, "Clustering time series with clipped data," *Mach. Learn.*, vol. 58, no. 2/3, pp. 151–178, 2005.

[14] L. Bao, Q. Li, P. Lu, J. Lu, T. Ruan, and K. Zhang, "Execution anomaly detection in large-scale systems through console log analysis," *J. Syst. Softw.*, vol. 143, pp. 172–186, 2018.

[15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[16] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proc. Workshop Knowl. Discov. Databases*, 1994, pp. 359–370.

[17] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 468–479.

[18] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Commun. ACM*, vol. 59, no. 8, pp. 32–37, 2016.

[19] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *Trans. Comput.*, vol. 100, no. 6, pp. 592–597, 1972.

[20] L. C. Briand, Y. Labiche, and M. M. Sówka, "Automated, contract-based user testing of commercial-off-the-shelf components," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 92–101.

[21] M. Broy, "Engineering cyber-physical systems: Challenges and foundations," in *Complex Systems Design & Management*. Berlin, Germany: Springer, 2013, pp. 1–13.

[22] R. N. Charette, "Nissan recalls nearly 1 million cars for air bag software fix," *IEEE Spectrum*. Piscataway, NJ, USA: IEEE Press, 2014.

[23] E. Chávez, G. Navarro, R. Baeza-Yates , and J. L. Marroquín, "Searching in metric spaces," *Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

[24] Y. Chen *et al.*, "Air quality data clustering using EPLS method," *Inf. Fusion*, vol. 36, pp. 225–232, 2017.

[25] Y. Chen, C. M. Poskitt, and J. Sun, "Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system," in *Proc. Symp. Secur. Privacy*, 2018, pp. 648–660.

[26] L. Cheng, K. Tian, and D. D. Yao, "Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2017, pp. 315–326.

[27] H. Choi *et al.*, "Detecting attacks against robotic vehicles: A control invariant approach," in *Proc. Conf. Comput. Commun. Secur.*, 2018, pp. 801–816.

[28] M. S. Coelho, *Patterns in financial markets: Dynamic time warping*, PhD dissertation, NSBE-UNL, 2012.

[29] P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Hove, East Sussex, U.K.: Psychol. Press, 2014.

[30] T. M. Cover, *Elements of Information Theory*. Hoboken, NJ, USA: Wiley, 1999.

[31] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 281–290.

[32] 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," *Operating Syst. Rev.*, vol. 35, no. 5, pp. 57–72, 2001.

[33] M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1/3, pp. 35–45, 2007.

[34] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli , and S. A. Seshia, "Scenic: A language for scenario specification and scene generation," in *Proc. Conf. Program. Lang. Des. Implementation*, 2019, pp. 63–78.

[35] A. W.-C. Fu, E. Keogh, L. Y. Lau, C. A. Ratanamahatana, and R. C.-W. Wong, "Scaling and time warping in time series querying," *Int. J. Very Large Data Bases*, vol. 17, no. 4, pp. 899–921, 2008.

[36] A. Gambi, M. Mueller, and G. Fraser, "Automatically testing self-driving cars with search-based procedural content generation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2019, pp. 318–328.

[37] A. Ghafouri, A. Laszka, A. Dubey, and X. Koutsoukos, "Optimal detection of faulty traffic sensors used in route planning," in *Proc. Int. Workshop Sci. Smart City Operations Platforms Eng.*, 2017, pp. 1–6.

[38] A. Ghafouri, Y. Vorobeychik, and X. Koutsoukos, "Adversarial regression for detecting attacks in cyber-physical systems," in *Proc. Int. Conf. Artif. Intell.*, 2018, pp. 3769–3775.

[39] C. Gladisch, T. Heinz, C. Heinzemann, J. Oehlerking, A. von Vietinghoff, and T. Pfitzer, "Experience paper: Search-based testing in automated driving control applications," in *Proc. Int. Conf. Automated Softw. Eng.*, 2019, pp. 26–37.

[40] S. Grant, H. Cech, and I. Beschastnikh, "Inferring and asserting distributed system invariants," in *Proc. Int. Conf. Softw. Eng.*, 2018, pp. 1149–1159.

[41] J. Gu and X. Jin, "A simple approximation for dynamic time warping search in large time series database," in *Proc. Int. Conf. Intell. Data Eng. Automated Learn.*, 2006, pp. 841–848.

[42] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[43] Y. Harada, Y. Yamagata, O. Mizuno, and E.-H. Choi, "Log-based anomaly detection of CPS using a statistical method," in *Proc. Int. Workshop Empir. Softw. Eng. Pract.*, 2017, pp. 1–6.

[44] F. Hauer, A. Pretschner, M. Schmitt, and M. Groetsch, "Industrial evaluation of search-based test generation techniques for control systems," in *Proc. Int. Symp. Softw. Rel. Eng. Workshops*, 2017, pp. 5–8.

[45] Z. He, Y. Chen, E. Huang, Q. Wang, Y. Pei, and H. Yuan, "A system identification based oracle for control-CPS software fault localization," in *Proc. Int. Conf. Softw. Eng.*, 2019, pp. 116–127.

[46] M. W. Hofbaur and B. C. Williams, "Mode estimation of probabilistic hybrid systems," in *Proc. Int. Workshop Hybrid Syst. Comput. Control*, 2002, pp. 253–266.

[47] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *J. Educ. Psychol.*, vol. 24, no. 6, pp. 417–441, 1933.

[48] B. Hoxha, H. Bach, H. Abbas, A. Dokhanchi, Y. Kobayashi, and G. Fainekos, "Towards formal specification visualization for testing and monitoring of cyber-physical systems," in *Proc. Int. Workshop Des. Implementation Formal Tools Syst.*, 2014.

[49] M. Hu *et al.*, "Detecting anomalies in time series data via a meta-feature based approach," *IEEE Access*, vol. 6, pp. 27760–27776, 2018.

[50] J. Inoue, Y. Yamagata, Y. Chen, C. M. Poskitt, and J. Sun, "Anomaly detection for a water treatment system using unsupervised machine learning," in *Proc. Int. Conf. Data Mining Workshops*, 2017, pp. 1058–1065.

[51] H. Jiang, S. Elbaum, and C. Detweiler, "Inferring and monitoring invariants in robotic systems," *Auton. Robots*, vol. 41, no. 4, pp. 1027–1046, Apr. 2017.

[52] B. Johnson, Y. Song, E. Murphy-Hill , and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 672–681.

[53] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[54] T. Kahveci and A. Singh, "Variable length queries for time series data," in *Proc. Int. Conf. Data Eng.*, 2001, pp. 273–282.

[55] T. Kahveci, A. Singh, and A. Gurel, "Similarity searching for multi-attribute sequences," in *Proc. Int. Conf. Sci. Statist. Database Manage.*, 2002, pp. 175–184.

[56] A. Kane, T. Fuhrman, and P. Koopman, "Monitor based oracles for cyber-physical system testing: Practical experience report," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2014, pp. 148–155.

[57] C. Kästner *et al.*, "Toward variability-aware testing," in *Proc. Int. Workshop Feature-Oriented Softw. Develop.*, 2012, pp. 1–8.

[58] L. Kaufmann and P. Rousseeuw, "Clustering by means of medoids," *Data Analysis based on the L1-Norm and Related Methods*. Amsterdam, The Netherlands: North Holland, 1987, pp. 405–416.

[59] E. Keogh, "Exact indexing of dynamic time warping," in *Proc. Int. Conf. Very Large Data Bases*, 2002, pp. 406–417.

[60] F. Klück, Y. Li, M. Nica, J. Tao, and F. Wotawa, "Using ontologies for test suites generation for automated and autonomous driving functions," in *Proc. Int. Symp. Softw. Rel. Eng. Workshops*, 2018, pp. 118–123.

[61] T. K. Koo and M. Y. Li, "A guideline of selecting and reporting intraclass correlation coefficients for reliability research," *J. Chiropractic Med.*, vol. 15, no. 2, pp. 155–163, 2016.

[62] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Proc. Int. Symp. Fault-Tolerant Comput.*, 1998, pp. 230–239.
[63] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. Int. Symp. Object Compon.-Oriented Real-Time Distrib. Comput.*, 2008, pp. 363–369.
[64] C. Lemieux, "Mining temporal properties of data invariants," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 751–753.
[65] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining," in *Proc. Int. Conf. Automated Softw. Eng.*, 2015, pp. 81–92.
[66] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze, "Metamorphic model-based testing of autonomous systems," in *Proc. Int. Workshop Metamorphic Testing*, 2017, pp. 35–41.
[67] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *Proc. USENIX Annu. Tech. Conf.*, 2010, pp. 1–14.
[68] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, "Dodona: Automated oracle data set selection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 193–203.
[69] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *ACM Comput. Surv.*, vol. 52, no. 5, pp. 1–41, 2019.
[70] R. Majumdar, A. Mathur, M. Pirron, L. Stegner, and D. Zufferey, "Paracosm: A language and tool for testing autonomous driving systems," 2019, *arXiv:1902.01084*.
[71] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 1012–1021.
[72] J. C. Mankins, "Technology readiness levels," *White Paper*, vol. 6, no. 1995, 1995, Art. no. 1995.
[73] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 1947.
[74] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous simulink models," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 595–606.
[75] P. McCausland, "Self-driving Uber car that hit and killed woman did not recognize that pedestrians jaywalk," *NBC News*.
[76] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human Oracle costs associated with automatically generated test data," in *Proc. Int. Workshop Softw. Test Output Valid.*, 2010, pp. 1–4.
[77] C. Menghi, S. Nejati, K. Gaaloul, and L. C. Briand, "Generating automated and online test Oracles for simulink models with continuous and uncertain behaviors," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 27–38.
[78] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh, "Automatic test case generation from simulink/stateflow models using model checking," in *Proc. Int. Conf. Softw. Testing Verification Rel.*, vol. 24, no. 2, pp. 155–180, 2014.
[79] G. E. Mullins, P. G. Stankiewicz, and S. K. Gupta, "Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles," in *Proc. Int. Conf. Robot. Autom.*, 2017, pp. 1443–1450.
[80] G. E. Mullins, P. G. Stankiewicz, R. C. Hawthorne, and S. K. Gupta, "Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles," *J. Syst. Softw.*, vol. 137, pp. 197–215, 2018.
[81] P. Nader, P. Honeine, and P. Beauseroy, "l_p-norms in one-class classification for intrusion detection in SCADA systems," *Trans. Ind. Inform.*, vol. 10, no. 4, pp. 2308–2317, 2014.
[82] T. Nguyen, M. B. Dwyer, and W. Visser, "SymInfer: Inferring program invariants using symbolic states," in *Proc. Int. Conf. Automated Softw. Eng.*, 2017, pp. 804–814.
[83] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to generate disjunctive invariants," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 608–619.
[84] A. Novikov, "PyClustering: Data mining library," *J. Open Source Softw.*, vol. 4, no. 36, 2019, Art. no. 1230.
[85] T. Ohmann *et al.*, "Behavioral resource-aware model inference," in *Proc. Int. Conf. Automated Softw. Eng.*, 2014, pp. 19–30.
[86] S. O'Kane, "Boeing finds another software problem on the 737 max," *The Verge*, 2020.
[87] M. O'Kelly *et al.*, "F1/10: An open-source autonomous cyber-physical platform, 2019.
[88] F. Pasqualetti, F. Dörfler, and F. Bullo, "Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design," in *Proc. Conf. Decis. Control*, 2011, pp. 2195–2201.
[89] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *Proc. Int. Conf. Softw. Testing Verification Valid.*, 2013, pp. 342–351.
[90] K. Pearson, "On lines and planes of closest fit to systems of points in space," *London Edinburgh Dublin Philos. Mag. J. Sci.*, vol. 2, no. 11, pp. 559–572, 1901.
[91] A. Platzer, *Logical Foundations of Cyber-Physical Systems*. Berlin, Germany: Springer, 2018.
[92] M. Quigley *et al.*, "Ros: An open-source robot operating system," in *Proc. Int. Conf. Robot. Autmation*, 2009, Art. no. 5.
[93] C. A. Ratanamahatana and E. Keogh, "Three myths about dynamic time warping data mining," in *Proc. Int. Conf. Data Mining*, 2005, pp. 506–510.
[94] G. Reger, H. Barringer, and D. Rydeheard, "Automata-based pattern mining from imperfect traces," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 1–8, 2015.
[95] C. Robert, T. Sotiropoulos, H. Waeselynck, J. Guiochet, and S. Vernhes, "The virtual lands of Oz: Testing an agribot in simulation," *Empir. Softw. Eng.*, vol. 25, pages 2025–2054, 2020.
[96] E. Rocklage, H. Kraft, A. Karatas, and J. Seewig, "Automated scenario generation for regression testing of autonomous vehicles," in *Proc. Int. Conf. Intell. Transp. Syst.*, 2017, pp. 476–483.
[97] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, pp. 53–65, 1987.
[98] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, pages 58–66, 2018.
[99] S. Salvador and P. Chan, "Toward accurate dynamic time warping in linear time and space," *Intell. Data Anal.*, vol. 11, no. 5, pp. 561–580, 2007.
[100] L. Schmidt, A. Narayan, and S. Fischmeister, "TREM: A tool for mining timed regular specifications from system traces," in *Proc. Int. Conf. Automated Softw. Eng.*, 2017, pp. 901–906.
[101] H. Skutkova, M. Vitek, P. Babula, R. Kizek, and I. Provaznik, "Classification of genomic signals using dynamic time warping," *BMC Bioinf.*, vol. 14, no. 10, 2013, Art. no. S1.
[102] Y. Thounaojam, W. Setiawan, and A. Narayan, "MA2DF: A multi-agent anomaly detection framework," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, 2020, pp. 30–36.
[103] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proc. Int. Conf. Softw. Eng.*, 2018, pp. 303–314.
[104] 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 *Proc. Int. Conf. Softw. Testing Verification Valid.*, 2018, pp. 331–342.
[105] C. E. Tuncali," *Search-based Test Generation for Automated Driving Systems: From Perception to Control Logic*, "PhD thesis, Arizona State Univ., Tempe, AZ, USA, 2019.
[106] C. E. Tuncali, T. P. Pavlic, and G. Fainekos, "Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles," in *Proc. Int. Conf. Intell. Trans. Syst.*, 2016, pp. 1470–1475.
[107] R. van Tonder and C. Le Goues , "Lightweight multi-language syntax transformation with parser parser combinators," in *Proc. Conf. Program. Lang. Des. Implementation*, 2019, pp. 363–378.
[108] V. Verma, G. Gordon, R. Simmons, and S. Thrun, "Real-time fault diagnosis [robot fault diagnosis]," *Robot. Autom. Mag.*, vol. 11, no. 2, pp. 56–66, 2004.
[109] W. Weimer, S. Forrest, C. L. Goues, and M. Kim, "Cooperative, trusted software repair for cyber physical system resiliency," Univ. Virginia Charlottesville, Charlottesville, VA, USA, Tech. Rep. AFRL-RI-RS-TR-2018–182, 2018.
[110] A. Windisch, "Search-based testing of complex simulink models containing stateflow diagrams," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 395–398.
[111] K. Wyrobek, "The origin story of ROS, the Linux of robotics," 2017. Accessed: Sep. 02, 2020.
[112] Q. Xia, J. Duan, F. Gao, Q. Hu, and Y. He, "Test scenario design for intelligent driving system ensuring coverage and effectiveness," *Int. J. Automot. Technol.*, vol. 19, no. 4, pp. 751–758, 2018.
[113] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. Symp. Operating Syst. Princ.*, 2009, pp. 117–132.

[114] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 282–291.
[115] K. Yang and C. Shahabi, "A PCA-based similarity measure for multivariate time series," in *Proc. Int. Workshop Multimedia Databases*, 2004, pp. 65–74.
[116] N. Ye, S. M. Emran, Q. Chen, and S. Vilbert, "Multivariate statistical analysis of audit trails for host-based intrusion detection," *Trans. Comput.*, vol. 51, no. 7, pp. 810–820, 2002.
[117] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, "Understanding uncertainty in cyber-physical systems: A conceptual model," in *Modelling Foundations and Applications*. Berlin, Germany: Springer, 2016, pages 247–264.
[118] T. Zhang, D. Yue, Y. Gu, Y. Wang, and G. Yu, "Adaptive correlation analysis in stream time series with sliding windows," *Comput. Math. Appl.*, vol. 57, no. 6, pp. 937–948, 2009.
[119] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, pp. 61–67, 2019.
[120] E. Zibaei, S. Banescu, and A. Pretschner, "Diagnosis of safety incidents for cyber-physical systems: A UAV example," in *Proc. Int. Conf. Syst. Rel. Saf.*, 2018, pp. 120–129.

Afsoon Afzal received the PhD degree in software engineering from Carnegie Mellon University, Pittsburgh, Pennsylvania, in 2021. She is a software engineer with Nuro Inc. She is interested in applying automated quality assurance methods, including automated testing, and repair to evolving and autonomous systems. More information is available at: http://www.cs.cmu.edu/ afsoona.

Claire Le Goues (Member, IEEE) received the BA degree in computer science from Harvard University, Cambridge, Massachusetts, and the MS and PhD degrees from the University of Virginia, Charlottesville, Virginia. She is an associate professor with the School of Computer Science, Carnegie Mellon University, where she is primarily affiliated with the Institute for Software Research. She received an NSF CAREER award. She is interested in constructing high-quality systems in the face of continuous software evolution, with a particular interest in automatic error repair. More information is available at: http://www.cs.cmu.edu/ clegoues.

Christopher Steven Timperley received the MEng and PhD degrees in computer science from the University of York, York, U.K. He is a systems scientist with the School of Computer Science, Carnegie Mellon University. He is interested in automated techniques that help developers to build, test, and deploy high-quality software for robotic and autonomous systems. More information is available at: http://www.christimperley.co.uk.