Automated Environment Reduction for Debugging Robotic Systems

Meriel Von Stein[1] and Sebastian Elbaum[2]

***Abstract*— Robot failures often occur in complex environments. Identifying the key elements of the environment associated with such failures is critical for a faster fault isolation and, ultimately, debugging those failures. In this work we present the first automated approach for reducing the environment in which a robot failed. Similar to software debugging techniques, our approach systematically performs a partition of the environment space causing a failure, executes the robot in each partition containing a reduced environment, and further partitions reduced environments that still lead to a failure. The technique is particularly novel in the spatial-temporal partition strategies it employs, and in how it manages the potential different robot behaviors occurring under the same environments. Our study finds that environment reductions of over 95% are achievable within a 2-hour window.**

I. Introduction

When a robot fails in a complex environment the debugging process is challenging. There are usually large bags of time-stamped data, interweaving logical and physical states variables, multiple interconnected subsystems and processes, many unstated assumptions and unseen variables, and subsequently tons of potential hypotheses about what could have gone wrong [1], [11].

A critical step in this debugging process, and the focus of this paper, is the minimization of the environment where the robot failure was observed. Robots sense and act on their environment, yet it is usually the case that not all the elements in an environment are relevant to the failure. Simplifying the environment causing the failure can accelerate debugging by helping to communicate the essential issues associated with a failure and reducing the number of debugging hypotheses to consider [18].

Today, the reduction of the environment while debugging robot failures is largely a tedious, manual process [10]. We argue that a cost-effective path forward, borrowing from similar software debugging techniques [17], [18], is to re-execute the test that caused the robot to fail while systematically manipulating the environment. This cycle of environment reduction and test execution is repeated as long as a reduced environment retains the original failure and can be further manipulated.

In this work, we introduce the first automated approach to reduce a robot failure-inducing environment that leverages the principles of binary search employed in software debugging while accounting for the unique characteristics of the domain. Characteristics unique to robotics that do not manifest similarly in software include the spatial-temporal relationships between the elements of the environment and the nondeterminism associated with the robot operation. As we show, these characteristics provide new dimensions that significantly affect how to approach environment reduction.

Our study on a popular open source autonomous ground vehicle system in three distinct environments shows that this approach can reduce the number of elements in the environment by an average of 78% while retaining the original failure, and be applied with minimal developer involvement.

The contributions of this paper are as follows:

- an environment reduction technique that leverages nondeterminism, time series data, data with high interdependence, and variables that come from an unknown distribution.
- an automated framework that implements the technique. This framework is integrated with the Gazebo simulator for the manipulation of its worlds and incorporates three partitioning and prioritization schema pairings as well as a failure characterization based on the robot's pose that serves as an oracle. Additionally, it includes a configurable deflaking process to deal with nondeterministic executions.
- a study of 3 scenarios exerting different types of stress to induce actuation failures on a popular open source autonomous robot platform to which we apply our technique. Our findings show that this technique is able to reduce the size of the environment by a factor as large as 21 and by a factor of approximately 8 on average.

II. Background

Techniques to isolate failure inducing environments, and more in general debugging, can be organized into two groups, those focused on software systems, and those tailored to robotics.

Unlike those tailored to robotics, software debugging techniques were largely designed for hardware-independent programs. Zeller et al [18], [19] codified the fundamental delta debugging(*ddmin*). This approach formalized the logic that, given a set of changes to a program that conform to a set of properties ensuring monotonicity and validity, a subset of those changes is responsible for introducing a fault, and it used a variant of binary search to isolate the failure inducing changes. It implicitly assumes the program under consideration to be deterministic and without that assumption, the technique's effectiveness and efficiency can suffer dramatically. We base our approach and terminology on this work and follow up derivations [2], [6] but with a focus on physical environment in which the robot operates. These physical input types require specific handling and manipulation that recognizes physical constraints and ties to real-world processes. Our deflake process is similar in its "replay" aspect to [2]. More recently, Johnson et al [8]

[1]University of Virginia, USA, *meriel@virginia.edu*
[2]University of Virginia, USA, *selbaum@virginia.edu*

devised a manipulationist debugging approach by generating minimal difference pairs of passing and failing inputs. While the results were more correlative than causal, they served to highlight the importance of minimal difference sets in debugging practices. Wong et al [17] provide a and comprehensive survey of software fault localization techniques, ranging from logging and assertions to slicing and program spectrum based techniques. Among those, the slicing techniques that focus on removing irrelevant parts of a program for debugging are the closet to our approach.

Using a program state-based technique, Nie et al [13] involve a similar approach to ours to arrive at minimal failure causing configurations, focusing on the interactions of those parameter-value pairs in the configuration. However, their approach is based on combinatorial testing which limits the size of the set of parameter-value pairs through resource constraints. Most importantly, as a software-focused technique not geared towards cyberphysical systems, it does not involve physical aspects of the program, failure, or parameter-value pairs.

Because robot systems are built on distributed and noisy hardware and are prepared to operate on uncertain and not fully observable environments, debugging approaches must accommodate these characteristics. Khalastchi et al [10] provides a taxonomy of the techniques for fault detection and diagnosis in robotic systems. The major families of this taxonomy are model-based, knowledge-based, and data-driven approaches. One such model-based technique is that of Stavrou et al [16], which operates from an *a priori* model to detect actuator faults on differential-drive mobile robots as they operate in a controlled environment, similar to our approach and study. Popular knowledge-based approaches involve causal modeling or expert systems. Hamilton et al [7] use modeling in their RECOVERY fault diagnosis system for autonomous mobile robots that incorporates knowledge by way of robot design, sensor data, historical and mission information, and fault knowledge gathered from field experts. A data-driven technique is that outlined by Fagogensis et al [3], which uses a machine learning model to detect actuation failures and can modify its ability to detect online.

While software approaches to debugging have a great deal of support from the software engineering community, they do not directly translate to the context of debugging robotics failures. And while approaches from the robotics community have the advantage to be domain-specific, they seem to have overlooked the problem of automated environment reduction. We hope to merge the advantages of these two contexts in our environment reduction framework for robotics.

III. Approach

Given an environment E where robot R fails with failure f, the objective of our approach is to generate a minimal environment E' that still causes f where $E' \subset E$. We assume the process starts with a valid E, and a reduction operation $red(E)$ simply removes elements e_i from E to render E', a valid environment on which the robot can execute. Our algorithm then runs R on those E' which may induce the original failure f, a pass p, or cause a different failure f' (E' can cause R to fail in others ways).

Building on delta debugging, given C_E, the set of all potential reduction changes on E resulting in f, we define a minimal failure inducing environment change $C_{E'}$ as $C_{E'} \subseteq C_E, \forall e_i \in E', test(E' - e_i) \neq f$. That is, we seek to reduce the original failure inducing environment E to a minimal sub-environment E' where removing any single element e_i will not produce f. Note that there can be multiple minimal environments that cause failures other than f, and even multiple minimal environments E' that produce f. We do not seek to expend the resources necessary to find the global minimum environment but rather to obtain a meaningful environmental reduction within the resources available that still results in f.

Two unique aspects of reduction for robotic environments are worth highlighting. First, the elements of E in robotics are not just types in the cyber world; they are not just ints or floats, or parts of a grammar. Instead, they are entities in the real world that have physical properties, and spatial and temporal dependencies. When our approach partitions the environment, it is cognizant of such properties and dependencies. Second, due to the robot's inherent sensing, estimation, and actuation noise, test executions of robot systems can exhibit a high-degree of variability, with the corresponding variation in revealing a failure. We call such tests flaky, and they can severely limit existing fault isolation approaches, making them skip parts of the environment that matter. Our approach is parameterizable by the number of test re-executions to improve the chances to expose non-deterministic failures, mitigating this challenge.

A. Detailed Approach

Algorithm 1 takes in the robot under test R, the environment it is being tested in E', partitioning and prioritization schema as well as starting number of partitions (to be discussed later), and original failure f. We assume that there is some knowledge of how consistent f is on the part of the developer, as the sought-after failure must be deflakeable within the parameterized number of iterations, the parameterized timeout is reasonable, and the predicates on state variables that characterize f that distinguish a successful run from a failure are capturable.

As validity is not an available condition to prune the set of possible sub-environments, Algorithm 1 implements a depth-first search for the first minimal environment it finds. As shown in lines 2 and 3, we implement several schema for partitioning and prioritization of sub-environments generated from the original environment in order to better separate environments into equivalent parts. These are further explained in Table I and Algorithms 2 and 3.

After original environment E is partitioned and ordered, runs are conducted upon each E' in Algorithm 1, lines 5 to 10. The resulting artifacts are adjudicated as a similar failure to the failure f induced by the original environment, or a distinct failure f' or successful run p. The high degree of noise introduced by localization, the environment, and the

	Name	Description
Partitioning	model2model	Spatial-proximity of models based on k-means clustering.
	timeseg	Evenly subdivide trajectory by timestamp and length of execution.
	trajectoryseg	Trajectory-partitioning into segments based on changes in angular velocity $>$ delta.
	learner	Dynamic learner trained to detect failures from a feature vector.
Prioritization	random	Subenvironments prioritized in random order.
	sequential	The order in which subenvironment elements were added to the original environment.
	failure-proximal	Proximity of the centroid of partitions to the failure pose of the robot.
	avg. trajectory-proximal	Average proximity of the centroids of partitions to robot trajectory.
	min. trajectory-proximal	Minimum proximity of the centroids of partitions to robot trajectory.
	timestep	Timestep at which any model in the environment was sensed on the previous run.

TABLE I: Partitioning and prioritization schema.

simulator means that the system can produce a dissimilar run even on the original environment. Runs producing f' or p are considered flaky, rerun three times, and the results are evaluated against f. Our current approach to determine the result of each run is based on a simple *a priori* model of the system based on onboard sensors, navigation planning, and an oracle based on periodic transformation sampling.

Whether or not a new minimal environment has been found, partition granularity is incremented and a stopping condition is checked. The stopping condition for determining a minimal environment checks whether the current set of subenvironments produced only dissimilar results to the original and whether there was no further opportunity to increase granularity of the partitioning.

B. Partitioning and Prioritization Schemas

Partitioning and prioritization schemas leverage the spatial-temporal relations of robotic systems to more effectively prune environmental input. Algorithm 1 is supplemented by partitioning and prioritization schemas that integrate various aspects of the original and current run. Two of those schemas are used in the study and shown in greater detail in Algorithm 2 for partitioning and Algorithm 3 for ordering.

Algorithm 2 shows a k-means clustering of models' poses in relation to each other in the environment; effectively a model to model clustering. K-means is a predictable technique for vector quantization by a specified set of attributes [9], requires a relatively low amount of data to perform well and is established as being effective at clustering based on spatial and temporal distance.

A different partitioning scheme, the *timeseg* partitioning schema separates the robot trajectory evenly according to the value of *n_partitions* in order to separate the models that are closer in time to the failure. Assuming the models closer in time to the failure are more likely to have induced the failure, this should group together models with the greatest influence to induce the failure. Because k-means will not force data to fit to the specified number of clusters, this method will occasionally generate empty clusters or cluster all data together. To circumvent an early termination, we partition the environment into 1-model clusters on lines 3-6, which does not disrupt the clustering but rather serves to skip several iterations.

In Algorithm 3 the subenvironments are expressed as a set of environments defined by the presence or absence of models from the original environment. Distance is calculated in three dimensions due to many environments having a height component to them, such as hills and terraced surfaces, and for robots able to plan and actuate along the z-axis. The intuition here is to prioritize those models spatially proximal to the crash pose of the robot, as they could have greater correlation to inducing the failure.

Table I lists some other partitioning and prioritization schema that seek to structure environmental inputs to the system by leveraging temporal and spatial characteristics of robotics scenarios. Model2model partitioning is system-independent and partitions by the spatial relationships of the environment only. *Timeseg* and *trajectoryseg* partitioning leverage separate aspects of the system execution in relation to the environment and learner synthesizes all three. As per prioritization schema, random and sequential involve no information external to the environment. *Failure-proximal* ordering orders partitions by the minimum distance from the partition centroid to the failure pose of the robot. *Minimum trajectory-proximal* ordering finds the minimum distance from the partition centroid to any point on the trajectory and *average trajectory-proximal* ordering weights the trajectory according to the average distance of the centroid from the trajectory over the course of system execution. *Timestep* ordering leverages time series nature of the system execution to order by models by the first timestep in which they are sensed by the system.

C. Limitations

The deflake process assumes the failure to be present in a failure-inducing environment at least $\frac{1}{itertions}$ of the time or greater. Lower probabilities of f can cause $ddenv$ to ignore valuable partitions, leading to missed reduction opportunities. If an environment is configured with dynamic elements that have the ability to change pose, it cannot deduce pose dependency; i.e. if a plank leans against a wall, it may detect a point of contact between two models but cannot yet deduce that the plank's position depends on the presence of the wall.

Software input dependence is addressed in [12] but the process differs significantly from that of model dependence in a simulated environment in that the models are not directly

linked but rather can be correlated by collision boxes, pose, and their static or dynamic attributes.

D. Implementation

We implemented 3 instances of the approach to automatically simulate in Gazebo and collect test metrics. The Gazebo simulator [4] is a environment-building and simulation tool with hooks to Robot Operating System (ROS) [5] and for which models of many popular ROS robots are maintained and widely used. Environments are termed worlds that consist of compositions of discrete objects with configurable attributes, termed models.

These instances of the approach differ in their combination of partitioning and prioritization schema. *Model2model* partitioning was combined with two different prioritization schema, *failure-proximal* and *average trajectory-proximal.* The prioritization schema with the better performance, failure-proximal, was combined with the following timeseg partitioning schema. The reduction algorithm was automated by scripts that trigger test runs, partitioning and prioritization, world reduction, and an oracle to perform failure analysis.

Algorithm 1: *ddenv* algorithm for robotics

```
   Input: R, E, f, partition_schema, prior_schema, n_partitions,
          iterations, timeout
   Output: E'
1  while E' not reached do
2      subenvironments ← partition(E, f, n_partitions,
         partition_schema);
3      subenvironments ← prioritization(subenvironments,
         prior_schema);
4      for s in subenvironments do
5          result ←R(s,f,timeout)
6          if is_new_failure(result, f) or is_success(result) then
7              results ← deflake(s, R, iterations, timeout);
8              result ← find_similar_failure(results, f);
9          end
10         if is_similar_failure(result, f) then
11             E ← s;
12             break;
13     end
14     n_partitions ← n_partitions + 1;
15     if no similar failure produced by subenvironments and
         n_partitions == E.size then
16         E' ← E; break;
17     else if n_partitions > E.size then
18         n_partitions ← E.size;
19     end
20 end
21 return E'
```

IV. Study

We have applied our approach to the widely used Husky ground vehicle [15] and deployed it into three scenarios to assess the potential of our approach to simplify the environment associated with a failure. Our research questions are:

- **RQ1:** How cost-effective is our approach in reducing the size of the environment that led to a failure?
- **RQ2:** How do variations of partitioning and prioritization schema affect cost-effectiveness?

Algorithm 2: model2model partitioning algorithm

```
   Input: E, n_clusters
   Output: environment_clusters
1  model_poses ← [] for model in E.models do
2      model_poses ← model_poses += model.pose
3  end
4  environment_clusters ← kmeans.cluster(model_poses,
     n_clusters).labels;
5  if contains_empty_clusters(environment_clusters) then
6      for model in models do
7          environment_clusters ← new_cluster(model);
8      end
9  end
```

Algorithm 3: failure-proximal prioritization algorithm

```
   Input: environment_clusters, f
   Output: ordered_environments
1  crash_pose ← get_crash(f);
2  distances ← [];
3  for cluster in environment_clusters do
4      dist ← calc_distance_3D(cluster.centroid, crash_pose);
5      cluster.distance_from_crash ← dist;
6  end
7  ordered_environments ← sort_by_distance(environment_clusters);
```

A. Setup

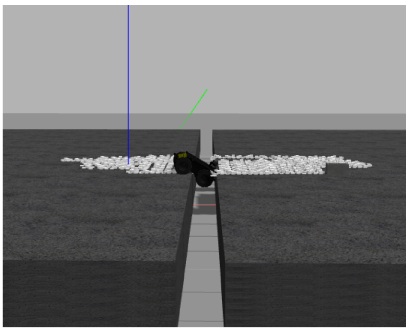
To answer these questions, we have designed three scenarios and configured the Husky to run within them. The Husky was chosen for its use as a generic ground vehicle suited for many environments and appendant open source navigation packages. This system and scenarios were evaluated under the Gazebo simulator. Three pairs of partitioning and prioritization schemas were chosen from Table I to test our minimal world reduction approach on three scenarios.

Similar to *ddmin*, we begin *ddenv* with n_partitions=2 and deflake() iterations=3. Enums for partitioning and prioritization schema are provided as parameters, conforming to three schema pairings, *model2model* and *failure-proximal*, *model2model* and *average trajectory-proximal*, and *timeseg* and *average trajectory-proximal*.

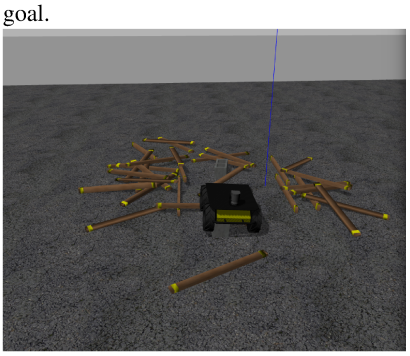
These scenarios were selected for being common occurrences in robotic environments. [14] They were designed to highlight variance of types of failures: a static failure in ditch, a dynamic failure in rubble, and a failure over time in friction. Each takes 45 seconds or less to reach its goal given a passing variation of the failing world.

The first scenario depicted in Figure 1a, **ditch**, consists of 43 models total with two asphalt planes with a 1m wide, 2m deep gap between them and patches of static rough terrain on each side. When approach at the right angle, the gap presents a high probability for the robot to get irretrievably stuck. The Husky uses just its compass, IMU, and odometry data to navigate from one plane to another.

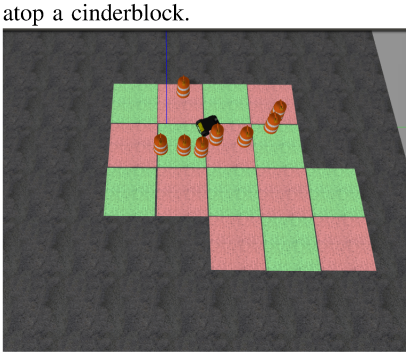
The second scenario depicted in Figure 1b, **rubble**, consists of 36 models total with a pile of dynamic 2x4 boards with a narrow, cluttered path through the rubble. There are two cinder blocks in the narrow path supporting two boards each. The robot is given a goal that forces it to navigate a



(a) Ditch scenario failure; husky stuck in ditch between spawning position and goal.



(b) Rubble scenario failure; husky caught atop a cinderblock.



(c) Friction scenario failure; husky cannot navigate precisely enough to pass between two barrels.

Fig. 1: Three failing scenarios

path through the rubble, but it often gets stuck on the hidden cinder blocks such that no wheel is touching the ground. The Husky uses its lms1xx laser scanner, compass, IMU, and odometry data to navigate from one plane to another.

The last scenario depicted in Figure 1c, **friction**, consists of 25 models total with a surface covered in patches of terrain with varying friction coefficients. The friction of the red patches is 1000 times greater than those of the green patches, and the friction of the asphalt plane that they are set into is in between. The robot is given a goal that forces it to navigate between narrow openings between construction barrels. due to the changes in friction of the patches it traverses before attempting to traverse the barrels, the control module might not able to line up the robot properly and it frequently gets stuck in the opening.

B. Results

Tables II-IV group results by scenario. The implementation to reproduce these results is available here.[1]

The **ditch** scenario results in Table II were comparable for techniques 1 and 3 and saw the greatest reduction in environment and runtime in technique 2. The *model2model* clustering and *averaged trajectory-proximal* ordering schemas' strong performance is attributable to the fact that it incorporates a greater amount of failure information about the failure and captures the rough terrain that perturbs actuation of the Husky before a catastrophic failure is induced by the ditch. Because this scenario prioritizes models by proximity to the trajectory, the raised plane and raised patches of rough terrain on the opposite side of the ditch are retained, whereas the first and third techniques respectively do not retain the asphalt plane because its center point is far from the crash, or because the robot spends most of its time stuck in the ditch which is, again, far from the center point of the asphalt plane.

Figure 2 shows the reduction in the world over the course of running our algorithm for technique 2. The steep dropoff in the first iteration indicates that a large partition was removed from the world and the world was still able to produce a failure. As the algorithm nears a minimal world, smaller partitions consisting of one model per partition are removed, causing a steady decline in the size of the reduced world shown in Figure 3.

The **rubble** scenario has comparable results using the first two schema pairs in Table III, with improved runtime and deflaking performance. The best technique in terms of runtime and reduction was *timeseg* partitioning. This schema was designed to leverage the time series aspect of the data collected from the run. The rubble scenario performed well with this segmentation because the Husky very quickly gets stuck on the rubble and remains stuck until timeout once the failure occurs, thus proportionately weighting the pieces of rubble that cause the failure. Because so much of the world surrounding the crash and trajectory is occupied by rubble, techniques 1 and 2 would include slightly more elements than necessary, showing smaller increments in world reduction.

Of all scenarios, **friction** saw the longest and second longest runtime of all experiments in the first and third techniques of Table IV, and second shortest runtime in technique 3. The *model2model* clustering and *failure-proximal* ordering performed exceptionally poorly on the friction scenario because the friction scenario was designed to exhibit ongoing system impedance that eventually led to a failure condition. The friction scenario environment exerted small perturbations on the Husky as it actuated over the in-ground patches of high friction differentials, creating small deviations in navigation over the course of the run. As a result, technique 1 weighted models solely in terms of what was near the crash in space and time, and thus its partitions were not directed along the trajectory but rather covered an equidistant cluster in space and time from the crash. Averaged trajectory-proximal ordering takes in the entire weighted trajectory

[1] github.com/Anon06160006/DDEnv

while the other partitioning schemas split it up, causing the algorithm to require higher numbers of iterations and deflaking to eliminate small partitions of models at a time.

Overall, the first technique did consistently poorly across all scenarios because these scenarios, while they do induce the failure in a specific place, tend to have some environmental input leading up to the failure that the robot cannot get out of. This technique discludes all of the information about the robot execution leading up to the failure. Conversely, for ditch and friction, the model2model partitioning and averaged trajectory-ordering technique offers the best reduction because it includes all of the execution information that the first technique ignores. For all scenarios and techniques, most of the reduction runtime was spent deflaking as evinced by the number of runs compared to size of the environment and percent reduction. Considered as a portfolio of reduction techniques, our approaches can offer an averaged 8x reduction across all scenarios.

Partitioning and Prioritization	**# Tests**	**% Reduction**	**Time (min)**
model2model, failure-proximal	268	84%	420
model2model, avg. trajectory-proximal	82	95%	108
timeseg, avg. trajectory-proximal	359	84%	487

TABLE II: Ditch scenario (original environment size of 43)

Partitioning and Prioritization	**# Tests**	**% Reduction**	**Time (min)**
model2model, failure-proximal	179	81%	274
model2model, avg. trajectory-proximal	178	78%	277
timeseg, avg. trajectory-proximal	162	89%	231

TABLE III: Rubble scenario (original environment size of 36)

Partitioning and Prioritization	**# Tests**	**% Reduction**	**Time (min)**
model2model, failure-proximal	727	44%	1098
model2model, avg. trajectory-proximal	116	76%	183
timeseg, avg. trajectory-proximal	430	68%	641

TABLE IV: Friction scenario (original environment size of 25)

V. Conclusions

This research introduces the first approach to address the problem of environment reduction with the use of physical and temporal knowledge unique to the environments of robotic systems. The study highlighted the potential value of the proposed approach to assist the robot debugging process. Our findings open up many avenues for continued study.

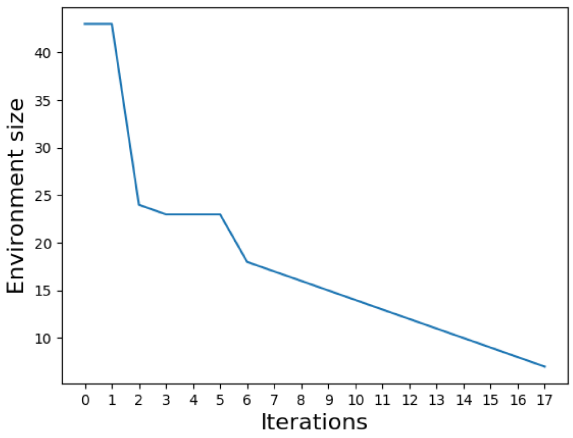


Fig. 2: Reduction in world size by number of models over iterations to find minimal world for ditch scenario.

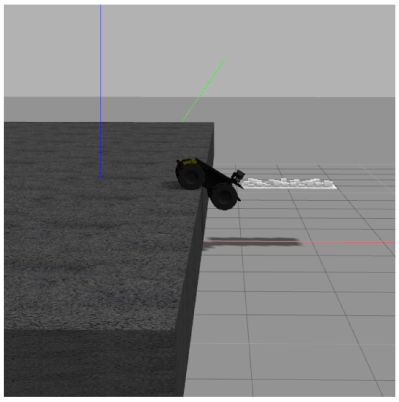


Fig. 3: Minimal world for ditch failure reached from *model2model* partitioning and *averaged trajectory-proximal* ordering.

Firstly, we would like to apply our approach to larger and more complex environments with thousands of elements to further advance its sophistication. Second, there are several other sources of information that we seek to exploit. For example, a likeness analysis of the other failures found by the analysis can provide further partition and prioritization insights according to the similarity of failures produced. Third, we can dramatically improve the efficiency of the approach by adding early cutoffs based on preconditions that environments must keep, such as the robot spawn location, in order to hasten the reproduction of the failure.

References

[1] Kevin Boos, Chien-Liang Fok, Christine Julien, and Miryung Kim. Brace: An assertion framework for debugging cyber-physical systems. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 1341–1344. IEEE Press, 2012.

[2] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, page 210–220, New York, NY, USA, 2002. Association for Computing Machinery.

[3] G. Fagogenis, V. De Carolis, and D. M. Lane. Online fault detection and model adaptation for underwater vehicles in the case of thruster failures. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2625–2630, 2016.

[4] Open Source Robotics Foundation. Gazebo robotics simulator. `http://gazebosim.org`, 2019.

[5] Open Source Robotics Foundation. Robot operating system. `https://www.ros.org/`, 2019.

[6] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: High coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 67–77, New York, NY, USA, 2012. Association for Computing Machinery.

[7] K. Hamilton, D. Lane, N. Taylor, and K. Brown. Fault diagnosis on autonomous robotic vehicles with recovery: an integrated heterogeneous-knowledge approach. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 4, pages 3232–3237 vol.4, 2001.

[8] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. http://people.cs.umass.edu/brun/pubs/pubs/Johnson20icse.pdfCausal Testing: Understanding Defects' Root Causes. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, Seoul, Republic of Korea, May 2020.

[9] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[10] Eliahu Khalastchi and Meir Kalech. On fault detection and diagnosis in robotic systems. *ACM Comput. Surv.*, 51(1), January 2018.

[11] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 425–442, USA, 2019. USENIX Association.

[12] Ghassan Misherghi and Zhendong Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 142–151, New York, NY, USA, 2006. Association for Computing Machinery.

[13] Changhai Nie and Hareton Leung. The minimal failure-causing schema of combinatorial testing. *ACM Trans. Softw. Eng. Methodol.*, 20(4), September 2011.

[14] Graz University of Technology. Husky ugv used for 3d mapping in mars simulation. `https://clearpathrobotics.com/husky-ugv-used-3d-mapping-mars-simulation/`, 2019.

[15] Clearpath Robotics. Husky unmanned ground vehicle. `https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/`, 2019.

[16] D. Stavrou, Demetrios G. Eliades, C. Panayiotou, and M. Polycarpou. Fault detection for service mobile robots using model-based method. *Autonomous Robots*, 40:383–394, 2016.

[17] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[18] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, October 1999.

[19] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.