Automated Environment Reduction for Debugging Robotic Systems

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

***Abstract*— Complex environments can cause robots to fail. Identifying the key elements of the environment associated with such failures is critical for 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 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 of a ground robot on three failure scenarios 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 many potential hypotheses about what could have gone wrong [1], [12].

A critical step in this debugging process, and the focus of this paper, is the reduction 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 executing 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 [19].

Today, even with advanced simulation capabilities that allow for the reproduction of failure causing tests, the reduction of the environment while debugging robot failures is largely a tedious, manual process [11]. We argue that a cost-effective path forward, borrowing from similar software debugging techniques [18], [19], is to re-execute the test that caused the robot to fail while automatically and systematically reducing the environment. This cycle of environment reduction and test re-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 failure-inducing environment for robots that: 1) leverages the principles of binary search employed in software debugging, and 2) accounts for the unique characteristics of the robotics domain such as the spatial-temporal relationships between the elements of the environment and the non-determinism associated with the robot operation. As we show, these unique characteristics significantly affect how to partition the elements in an environment and how the reduction is prioritized according to the influence on those elements on the robot operation.

The contributions of this paper are as follows:

- an environment reduction technique for robot systems based on the physical properties of robots and their environments. Our technique leverages these physical properties to perform temporally and spatially aware partitioning of the environment, and handles nondeterminism through the repeated execution of tests to match the likelihood of the failure.
- an automated framework that implements the technique. It incorporates three partitioning and prioritization schema pairings as well as a failure characterization based on the robot's pose that serves as an oracle. It also includes a configurable deflaking process to deal with non-deterministic executions. This framework is integrated with the Gazebo simulator for the manipulation of environments.
- a study of 3 scenarios on a popular open source autonomous ground vehicle system in three distinct failure-causing environments. It shows that the technique can reduce the number of elements in the environment by an average of 78% and a maximum of 95% while retaining the original failure, and be applied with minimal developer involvement.

II. Background

We present an overview of the techniques to isolate failure inducing environments, and more in general debugging, 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. Wong et al. [18] provide an extensive survey of software fault localization techniques, ranging from traditional debugging techniques (e.g., logging, assertions, breakpoints) to more sophisticated static and dynamic analysis techniques (e.g., slicing, program-spectrum, statistical). Among the many software debugging techniques, one of the most popular is known as delta debugging, codified by Zeller et al. [19] in their algorithm(*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

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

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

The authors thank the progenitors of the Husky project for making it open-source and well maintained. This work was funded in part by NSF #1924777 and #1853374.

introducing a fault. Delta debugging used a variant of binary search to systematically explore the change space and isolate the failure inducing changes. It implicitly assumed the program under consideration to be deterministic and without that assumption, the technique's effectiveness and efficiency suffer. The approach was later applied to explore the space of failure-inducing inputs attempting to identify the key inputs that cause a program to fail [20]. This work provides the bases for the environment reduction problem that we tackle. Now, many variants of delta debugging have been proposed, often integrated with other approaches (e.g., [2], [3], [7], [8], [13], [17]), and when put in the hands of software engineers they have shown to accelerate the debugging process [10]. We base our approach and terminology on this body of work around delta debugging but with a focus on physical environments in which a robot operates. These environments require special manipulation that recognizes physical constraints and ties to real-world processes.

Different from delta-debugging, Nie et al. [14] explored input space reduction using coverage arrays. Their focus is on the reduction of well-defined configuration spaces, for which coverage arrays are particularly well suited to guarantee a target level of coverage among the configuration parameters. Although we imagine such an approach could be adapted to the robotics domain with the redefinition of configurations in terms of, for example, motion planning, it seems like a poor match for the reduction of environments in which mobile robots operate given their size and complexity.

Because robot systems are susceptible to noise and are prepared to operate on uncertain and not fully observable environments, debugging approaches emerging from the robotics community have focused in accommodating these characteristics. Khalastchi et al [11] provides a taxonomy of the techniques for fault detection and diagnosis in robotic systems that includes: 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.Popular knowledge-based approaches involve causal modeling or expert systems. Hamilton et al. [9], for example, 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. Among the data-driven techniques, Fagogensis et al. [4] use a machine learning model to detect actuation failures. The domain-specificity makes these techniques powerful for robotics, or at least for the kind of robots they target. Yet, none of them focus on environment reduction. The technique we proposed is thus orthogonal and complementary to this body of approaches.

Overall, we find that while software approaches to debugging have seen dramatic leaps towards automated in the last decades, 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 aim to merge the advantages of these two contexts in our environment reduction framework for robotics.

III. Approach

Given an environment E comprised of elements $\{e_0, ..., e_n\}$ where robot R testing fails with failure f, the objective of our approach is to reduce E to E' such that:

$$E' \subset E \wedge test(R, E') = f \wedge E' \text{ is 1-minimal} \quad (1)$$

The $test$ operation may induce the original failure f, a pass p, or cause a different failure f' (E' can cause R to fail in other ways). The 1-minimal failure inducing environment E' is one where:

$$\forall e_i \in E', test(R, E' - e_i) \neq f \quad (2)$$

That is, we seek to reduce the original failure inducing environment E to a 1-minimal environment $E' \subset E$ where removing any single element e_i will not result in f. Just like delta debugging, we aim to detect a 1-minimal since finding a global minimal E' can be prohibitively expensive.

Central to such reduction is the operation $red(E)$, which removes elements $e_i ... e_j$ from E to render E'. Two unique aspects of $red(E)$ 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 and prioritizes what partitions to remove, 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 failure exposure. Borrowing from the software testing literature we call these flaky tests, 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, *ddenv* for robotics, takes in the robot under test R, the environment it is being tested in E, a partitioning and prioritization schema as well as starting number of partitions (to be discussed later), and the original failure f. The algorithm also takes two parameters, timeout (*dflk-to*) and iterations (*dflk-it*), that define the deflaking scope.

The algorithm implements a depth-first search for the first 1-minimal environment it finds. As shown in lines 3 and 4, the partitioning and prioritization of sub-environments are abstracted into their own functions. We cover some of them in Table I and Algorithms *env_partitioning* and *prioritization* by failure-proximal.

After the original environment E is partitioned and ordered, the robot is tested under each subenvironment E'. If

	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.

required, the result is deflaked between lines 5 to 10. The resulting artifacts are adjudicated as being the same as failure f induced by the original environment, a distinct failure f', or successful run p. This process assumes that test data enables such determination. For example, in our study, the reported failures include the final pose and time of the robot in E' which allows us to determine whether they are f or f'.

Algorithm 1: *ddenv* algorithm for robotics

```
   Input: R, E, f, part_schema, prior_schema, n_partitions, dflk-it,
          dflk-to
   Output: E'
1  1-minimal ← False;
2  while ¬ 1-minimal do
3  |   Subenvironments ← partition(E, f, n_partitions, part_schema);
4  |   Subenvironments ← prioritize(Subenvironments,
   |     prior_schema);
5  |   for s in Subenvironments do
6  |   |   result ← test(R, s, dflk-to);
7  |   |   if is_new_failure(result, f) or is_success(result) then
8  |   |   |   distribution ← deflake(s, R, dflk-it, dflk-to);
9  |   |   |   result ← is_similar_failure(distribution, f);
10 |   |   end
11 |   |   if result == f then
12 |   |   |   E ← s;
13 |   |   |   break;
14 |   end
15 |   n_partitions ← n_partitions + 1;
16 |   if result != f and n_partitions == E.size then
17 |   |   E' ← E;
18 |   |   1-minimal ← True;
19 |   else if n_partitions > E.size then
20 |   |   n_partitions ← E.size;
21 |   end
22 end
23 return E';
```

As discussed earlier, the uncertainty introduced by the robot sensors, algorithms, actuators, the environment, and the simulator means that the system can produce a dissimilar run even on the original environment. Thus, the algorithm conservatively assumes that tests producing f' or p are potentially flaky, reruns them *dflk-it* times, and the results are re-evaluated against f.

Whether or not a new 1-minimal environment has been found, the partition granularity is incremented (line 15) and a stopping condition for whether f is produced by a subenvironment is checked (line 16). The stopping condition for determining a 1-minimal environment checks whether the current set of sub-environments 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

Our partitioning and prioritization schemas leverage the spatial-temporal relations of robotic systems to more effectively prune the environment. We have developed a family of schemas, but due to space constraints we only provide two algorithms as examples. Algorithm *env_partitioning* for partitioning the environment based on the model and trace physical attributes, and their relationship to each other, and Algorithm *prioritization* by failure-proximal for prioritization based on time proximity to failure.

Algorithm 2: *env_partitioning*

```
   Input: E, n_clusters, trace, attrib_type
   Output: environment_clusters
1  for model in E do
2  |   if attrib_type == pose then
3  |   |   model.attrib ← model.pose;
4  |   else if attrib_type == time_first_sensed then
5  |   |   model.attrib ← compute_timestep(model.pose, trace);
6  |   else if attrib_type == distance_to_trajectory then
7  |   |   model.attrib ← compute_dist(model.pose, trace);
8  |   else if attrib_type == ... then
9  |   |   model.attrib ← ...;
10 |   end
11 end
12 environment_clusters ← kmeans(E, n_clusters, axis=attrib_type);
13 return environment_clusters;
```

Algorithm *env_partitioning* shows a k-means clustering of $e \in E$ based on their attributes. The attributes of the chosen type are first collected from each model appearing in the test trace. Since the elements in the environment are often referred to as models in popular simulators, we will use that label for this partitioning. Intuitively, clustering is meant to group elements or models in the environment that are close to each other. In our implementation we use the $pose$ of the models as the $attribute\_type$, but the algorithm could support other types like clustering by the $velocity$ for example when working with multiple dynamic models. We elected to use k-means because it is an explainable technique, requires a relatively low amount of data to perform well, and is effective at clustering based on spatial attributes. We also developed a temporal partitioning scheme based on k-means that we call *timeseg*. This partitioning schema splits the robot

trajectory, from the start to the failure, based on the temporal distance to the failure. The intuition is that elements closer in time to the failure are more likely to have induced the failure. The elements are clustered according to the timestep at which they are first sensed by the robot.

In terms of prioritization, Algorithm *prioritization* by failure-proximal is meant to order the elements spatially proximal to the crash pose of the robot, as they are expected to have a greater influence in inducing the failure. 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.

Algorithm 3: *prioritization* by failure-proximal

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

Table I lists some of the partitioning and prioritization schema we developed. As we have seen, *model2model* partitions the environment based on the spatial relationships between elements in the environment. *Timeseg* and *trajectoryseg* partition aspects of the robot execution in relation to the time they occurred or the proximity to the trajectory of the robot. *Learner* synthesizes all three by applying feature learning to a vector containing the attributes of the previous three partitioning schema. As per prioritization schema, random and sequential involve no information from the environment. *Failure-proximal* 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 the time series nature of the system execution to sequentially order elements by the first timestep in which they are sensed by the system.

C. Limitations

The application of the approach requires for the failing test scenario to provide access to its models so they can be manipulated as part of the debugging process. As the accessibility to the models is diminished, so is the approach capability to reduce the environment. The reduction also relies on repeated test executions, making the approach particularly suitable for debugging in simulation.

The deflaking process assumes that the failure is present in a failure-inducing environment at least $\frac{1}{dflk-it}$ of the time. Lower probabilities of f can cause $ddenv$ to ignore valuable partitions, leading to missed reduction opportunities.

Some of the partition and prioritization algorithms build upon the assumption that the elements in the environment are static (constant pose) to derive spatio-temporal relationships. Environments with dynamic objects would render those algorithms inadequate. Small adjustments to such algorithms to compute, for example, the minimal distance from an element to the robot during a test could help to overcome such limitations. Still, more sophisticated notions of spatial dependencies such as those capturing whether two robots are moving towards or away from each other could render even better information for partitioning and sorting and will be the focus of our future work.

D. Implementation

We implemented 3 instances of the approach to automatically simulate the reduction and generation of environments for the Gazebo simulator [5] and collect test metrics. The Gazebo simulator is an environment-building and simulation tool with hooks to Robot Operating System (ROS) [6] and for which models of many popular ROS robots are maintained and widely used. In the Gazebo parlance, environments 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 best performance, *failure-proximal*, was combined with the *timeseg* partitioning schema. The reduction was automated through scripts that trigger test runs, partitioning and prioritization, environment reduction, and an oracle to perform failure analysis.

IV. Study

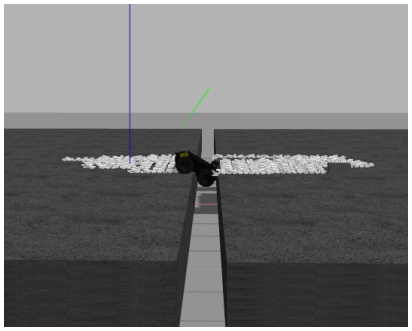
We have applied our approach to the Husky ground vehicle [15] and deployed it into three scenarios to explore the potential of our approach to simplify the environment associated with a failure. Our research questions are:

- **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?

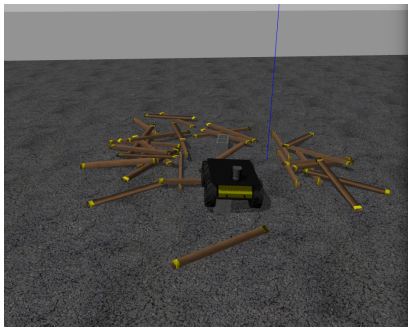
We measure cost-effectiveness as the tradeoff between environment reduction and the number of and wall-time for tests executed.

A. Setup

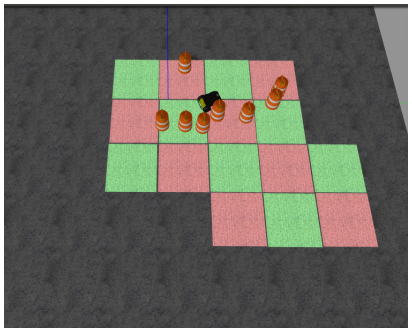
To answer these questions, we have designed three scenarios and configured the Husky to run within them. The Husky was chosen for its use as a generic ground vehicle suited for many environments and appendant open source navigation packages. This system and scenarios were evaluated under the Gazebo simulator. Three pairs of partitioning and prioritization schemas were chosen from Table I to test our 1-minimal environment reduction approach on three scenarios.



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



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



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

Fig. 1: Three failing scenarios

We begin *ddenv* with *n_partitions*=2 and *dflk-it*=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*.

We selected three scenarios that bring distinct contexts and failure occurrences in robotic environments. Each test scenario has a timeout of 45 seconds to reach its goal. The first scenario depicted in Figure 1a, labeled as **ditch**, consists of 43 models with two asphalt planes with a 1m wide, 2m deep gap between them and patches of static rough terrain on each plane. 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.

The second scenario depicted in Figure 1b, labelled as **rubble**, consists of 36 models, 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 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, labelled as **friction**, consists of 25 models with a surface covered in patches of terrain with varying friction coefficients. The friction coefficient of the red patches is 1000 times greater than those of the green patches which are set to μ=1, and the friction coefficient of the asphalt plane that they are set into is μ=100. The robot is given a goal that forces it to navigate between narrow openings between the construction barrels. Due to the changes in friction of the patches it traverses before attempting to going through the barrels, the control module might not able to line up the robot properly and it frequently gets stuck in the opening.

	Partitioning and Prioritization	**# Tests**	**%Reduction**	**Time (min)**
ditch	model2model, failure-proximal	268	84%	420
	model2model, avg. trajectory-proximal	82	95%	108
	timeseg, avg. trajectory-proximal	359	84%	487
rubble	model2model, failure-proximal	179	81%	274
	model2model, avg. trajectory-proximal	178	78%	277
	timeseg, avg. trajectory-proximal	162	89%	231
friction	model2model, failure-proximal	727	44%	1098
	model2model, avg. trajectory-proximal	116	76%	183
	timeseg, avg. trajectory-proximal	430	68%	641

TABLE II: Scenario reduction results by schema.

B. Results

Table II groups results by scenario.[1]

The **ditch** scenario results show all techniques provide a reduction of at least 84%, meaning only 7 of the 43 models are retained. Results in Table II were comparable for the first and third techniques and saw the greatest reduction in environment and runtime in the second technique. 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 schema 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 *model2model+failure-proximal* and *timeseg+averaged trajectory-proximal* 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,

[1]Access the implementation at: `github.com/MissMeriel/DDEnv`

far from the center point of the asphalt plane.

Figure 2 shows the reduction in the environment size for all three techniques. The steep dropoff in the first iterations indicates that large partitions are removed from the environment and the environment was still able to produce a failure. The techniques' different partitioning and prioritization schema lead them to remove different parts of the environment, reaching their 1-minimal E' at different stages in their exploration. Still, all three reduced the environment to less than 25% in just 8 iterations. The resulting 1-minimal reduction of the ditch environment for *model2model+failure-proximal* is shown in Figure 3.

The **rubble** scenario has comparable results using the first two schema pairs in Table II, 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 temporal proximity to the failure. The rubble scenario performed well with this segmentation because the Husky quickly gets stuck on the rubble and, after the failure occurs, remains stuck until timeout, thus proportionately weighting the pieces of rubble that cause the failure. Because so much of the environment surrounding the crash and trajectory is occupied by rubble, *model2model+failure-proximal* and *model2model+averaged trajectory-proximal* would include slightly more elements than necessary, showing smaller increments in environment reduction.

The three techniques provided significant gains in the **friction** scenario. We also note that Table II shows the *model2model+failure-proximal* technique required the longest time, but the second shortest time for *model2model+averaged trajectory-proximal* which converged quickly. This variation is explained by the mismatch between the failure scenario and some of the schemas. The *failure-proximal* ordering caused a poor performance because the friction scenario is designed to exhibit ongoing system impedance that eventually leads to a failure condition. That is, the issues leading to the failure arise earlier in the test. Instead, the *averaged trajectory-proximal* ordering takes in the entire weighted trajectory, which is a better match for this scenario. Also note that *model2model* is better suited for this scenario than *timeseg*, which generates model partitions based on time because the failure is associated with events that happen through the test. This scenario also required the most deflaking of them all, which is not surprising given its complexity.

Considered as a portfolio of reduction techniques, our approaches can offer 78% reduction on average across all scenarios. In practice, this means that the engineer debugging the robot system can focus on a much smaller portion of the environment, leading to more precise hypothesis about the source of the failure, with shorter bags of data to analyze, resulting likely in an accelerated debugging process.

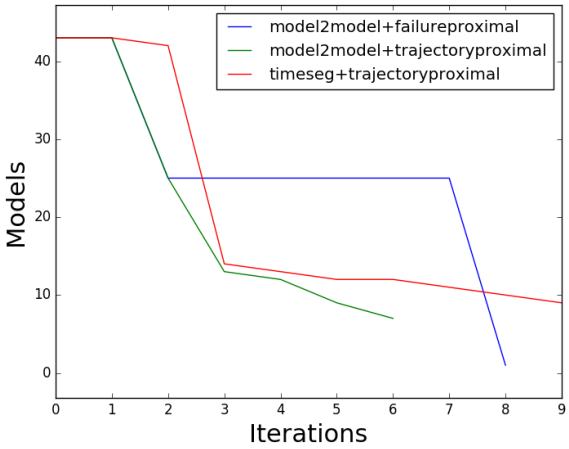


Fig. 2: Reduction in number of models per environment over iterations of three techniques for ditch scenario.

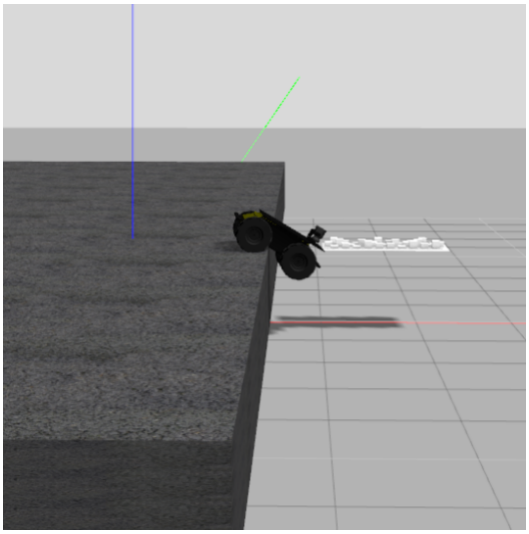


Fig. 3: 1-minimal environment for ditch failure reached from *model2model* partitioning and *averaged trajectory-proximal* ordering.

V. Conclusions

This research introduces the first approach for environment reduction with the use of physical and temporal knowledge unique to 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. Firstly, we would like to apply our approach to larger and more complex environments that include dynamic obstacles to better understand the tradeoffs and optimizations between reduction, runtime, and number of tests. Second, there are several other sources of information that we seek to exploit. For example, an analysis of the likeliness of multiple failures found by the approach can provide further partition and prioritization insights. 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, 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] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 342–351, 2005.

[4] 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.

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

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

[7] 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.

[8] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 263–272, New York, NY, USA, 2005. Association for Computing Machinery.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[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] William N. Sumner and Xiangyu Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 272–281, 2013.

[18] 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.

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

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