Self-Debugging Robots: Fault recovery through reasoning and planning

Christopher Thierauf
Human Robot Interaction Laboratory
Tufts University
Medford, MA
christopher.thierauf@tufts.edu

Matthias Scheutz
Human Robot Interaction Laboratory
Tufts University
Medford, MA
matthias.scheutz@tufts.edu

***Abstract*—Unexpected perturbations in an open-world task environment can cause various types of faults and failures which have to be dealt with to ensure long-term autonomous operation. We present an inference framework that enables an autonomous robot to generate and test fault hypotheses in open-world scenarios. The tests involve different types of introspective and overt behaviors based upon a derived fault hypothesis which informs behaviors which explore the failure. With suitable exploration, the robot can find the most sensible strategy to resolving the current failure condition if possible, allowing the task to be completed. We demonstrate the operation of our methods on a fully autonomous robot over a series of scenarios, ranging from a challenging yet solvable sensor occlusion case to mitigable or simultaneous failure cases.**

I. Introduction

As robots are being increasingly considered for deployment in unstructured and chaotic environments, they will inevitably experience different types of failures, either endogenously (e.g., a software crash or network failure) or exogenously (e.g., a jammed actuator or blocked sensor). Fig. 1a, for example, shows a robot rendered inoperable as a result of an obstruction occluding its head-mounted depth sensor: it perceives obstacles everywhere it turns, which prevents its navigation planner to find an obstacle-free path to its target location. As a result, the robot fails at its task.

Now suppose the robot had the ability to explore the nature of the failure, perhaps by checking whether it could see its own gripper to test if the camera was operational (Fig 1b), and by moving its gripper in front of its LIDAR unit to verify that LIDAR and arm are working correctly (Fig 1c). Performing these explorations would allow the robot to trace the fault to the camera. With a failure adequately explored, the robot can attempt to resolve it: for example, perhaps knowing the camera is obstructed in some way can be resolved by sweeping its arm close to the camera to remove the obstruction (Fig 1d), allowing it to resume its task.

We present such a system that utilizes the robot's existing sensing and planning capabilities to perform diagnostic tests and behaviors that allow it to track down potential faults and mitigate them. Importantly, the robot does not require predefined tests or mitigation behaviors, nor does it need to learn how to detect and recover from perturbations over time. Rather, our approach leverages explicit representations

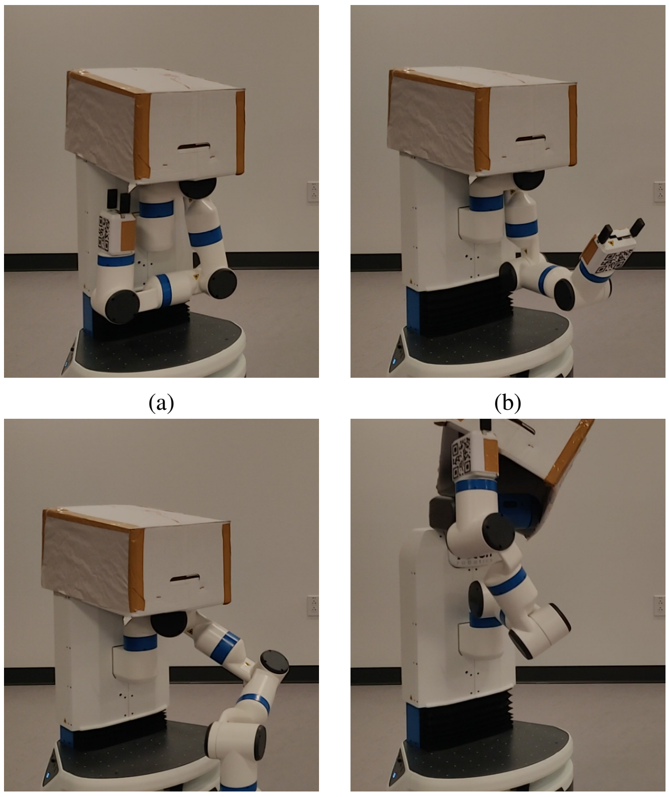

(a) (b)
(c) (d)

Fig. 1: Fetch robot with obscured camera (a) creates an observation problem, which is tested camera by looking at the gripper (b). This test does not pass, presenting ambiguity: is the camera or the arm broken? The arm is tested by blocking the LIDAR (c), showing it is operational. This testing has enabled the agent to infer something is blocking the camera, which it has an action to resolve (d).

of causal relations among different components in the robot's architecture (from sensors, to computational modules, to actuators) as a basis for introspective monitoring and the exploration of the potential effects of any detected discrepancies from nominal operation. Tracing the causes of the discrepancies to potential fault states, the robot can utilize its task and motion planners to perform self-testing behaviors in an effort to narrow down potential faults, ideally to a single cause which it can then attempt to mitigate, again using its task and motion planners. Our proposed approach allows the robot to find solutions, when they exist, even in situations the agent has never experienced (which is typically not possible for trained policies). In cases where solutions are unavailable (e.g., with permanent hardware failures), we will show that our proposed approach is still able to mitigate or self-assess to the potential benefit of an operator.

To present our approach, we start with a review of prior work on fault detection, and contrast it with our proposal of taking explicit actions in an attempt to acquire more information about the nature of the failure. We introduce the formal representation of our approach, and how it enables autonomous hypothesis generation and testing to provide a first-order logical representation of potential failure states which serves to define a planning problem for autonomous failure resolution. The implemented methods are then demonstrated on a fully autonomous robot which is able to explore and resolve its failure state. The conclusion summarizes our approach and provides an outlook for future work.

II. Background

Fault detection and diagnosis (FDD) has a long history in robotics, traditionally using model-based approaches to identify discrepencies, diagnose potential causes, and take appropriate action to mitigate them if possible (influentially surveyed in [17, 18] and more recently in [12, 16]). A survey of particular interest is [5] which explores the set of techniques for FDD through usage of Bayesian networks: these approaches use networks of states to make inferences of how a fault state may have come about. In many ways, our approach reflects a Bayesian net in that we will be utilizing a directed acyclic graph in a diagnostic context. It is easy to imagine our work benefiting from work in this area, such as approaches to efficiently generate Bayesian diagnostic graphs (e.g., [20, 26]).

However, while our approach is similar to a Bayesian network in that it uses a directed acylic graph (DAG), we allow for functional dependencies among nodes other than conditional probabilities. Most importantly, the key distinction between our work and any FDD work is that we are interested in active fault exploration, i.e., in utilizing robot behaviors that explore the agent's environment systematically to obtain useful information about the fault state which can form the basis for planning to mitigate it. Hence, our approach is more closely related to work in execution monitoring (surveyed in robotics specifically in [25]): specifically, the goal of integrating some form of task planning with an FDD system. IPEM (Integrated Planning, Execution, and Monitoring) [1] presents an early approach in this space by detecting and adapting to failure (though their approach uses a more direct mapping between observation and failure than ours). [8] also presents a system that maintains a logical representation of the world tightly integrated with robot behavior systems, enabling a real-world robot to self-monitor and constructing a knowledge and reasoning problem. Later, [33] presents FLUX, which leverages fluent calculus and a series of Constraint Handling Rules [11] to maintain an agent's understanding of its environment through (in part) failed actions. As with [35], they are interested in allowing the multi-step plans of robot agents to be more robust and adaptable (though the latter, as with [21], connects their task planning more tightly with grasp planning rather than general behavior).

More recently, [23] integrate their "observers" approach into linear temporal logic to produce a fault-tolerant robotics architecture. In [7], the authors enable explainability by connecting plan execution monitoring (more specifically, the failures observed in this process) to logical states for explainability and replanning. Presumably, as with our approach, a task planner could leverage the output of any of these systems for failure resolution (as demonstrated by [7]).

Task planning has also been explored in diagnostic contexts: XFRM [2] presents an early example, in which active monitoring is used to react to a variety of failure triggers. A more recent work in the same area is [15], in which open-world planning is also used to attempt to reason about and plan around potential failure. However, approaches of this nature are largely reactive, and are forced to make use of knowledge as it is observed, rather than discovering actions which may be able to reveal more about the open-world or failure context.

Despite this progress, none of these approaches intentionally take actions to probe the nature of a failure, generating fault hypotheses that can be explored and disproven, and arriving at a minimized set of possible failures which form a planning problem to resolve or mitigate it, as ours does. Doing so requires a tight integration between a knowledge and reasoning system, planning system, and robot behaviors.

Cognitive architectures meet this need, and are an area of research that we are particularly interested in. These approaches have been explored on humanoid-inspired platforms [4], and are frequently deployed on more traditional mobile robot platforms [14, 27]. These approaches have also been made platform-general, as explored in ACT-R/E [34], SOAR [22], ADAPT [3], and others. Similarly, symbolic planning systems have a long history in robotic architectures: Stemming from the seminal [24] is the previously-discussed [2] and [15], along with many other applications using the still widely-used PDDL [9] and POMDP (applications surveyed in [6]).

For the implementation, we leverage the DIARC cognitive architecture [30] for its ability to perform explicit robot action and action planning based on an understanding of robot actions and environment states [32] as well as the Robot Operating System (ROS) [28] for providing interfaces to the robot manipulator's navigation/kinematic stack [36]. Thus, our proposed system is characterized by the following five properties:

1) passive monitoring of ongoing task execution of an autonomous (robot) agent;
2) detecting failure and producing a reasonable set of potential causes for the failure;
3) producing a set of explicit robot actions which could reduce the set of detected potential causes by producing a task planning problem;
4) executing these robot actions, while continuing to mon-

itor, in an attempt to produce a more accurate understanding of the agent's environment;

5) making use of the collected information in an attempt to resolve the current failure.

1 and 2 have been previously explored in the literature, while 3-5 are our novel contributions described next.

III. Open-World Fault Mitigation through Active Hypothesis Testing

To illustrate our approach, we will use a Fetch robot performing a navigation task which it can complete autonomously and without difficulty thanks to a differentially-driven wheeled platform with odometry sources from wheels and two Inertial Measurement Units (IMUs). Additionally, both the on-board LIDAR unit in the robot's base and the head-mounted depth camera are used for obstacle avoidance. However, when the robot's head is obstructed (Fig. 1a), blocking one of the sensors used for obstacle avoidance, the navigation algorithm will (incorrectly) believe that there is an obstacle everywhere it turns. As a result, the robot cannot continue its trajectory and fails to reach its destination. We will in the following use this example to describe the robot's failure detection, testing, and resolution processes.

A. Requirements

We keep implementation requirements minimal to allow for integration into a wide variety of systems. These core requirements are an agent capable of making some base set of observations, and a symbolic agent planning system. Such functionality is common on robotic platforms.

We formalize the robot's operating environment as a Markov Decision Process (MDP) $M = \langle S, A, F, \gamma \rangle$, where S is the set of environmental states, A is the set of the robot's actions, F is a set of fluents numerically describing aspects of the robot or its environment, and $\gamma(s, a)$ is the transition function returning the distribution of possible states the robot can be in when executing action $a \in A$ in $s \in S$. This representation allows for both structured representations of the environmental states (e.g., "the gripper is holding an object") or robot-centric conditions (e.g., "the gripper fingers are currently 2mm apart") which enable the agent to compare expectations with measured data for self-evaluation (e.g., "when the gripper is holding an object, fingers should be more than 0mm apart").

This MDP M contains the pieces necessary to construct a planning domain. As discussed in much prior work[1], the planning domain $\Sigma = (S, A, \gamma)$ can be used to produce a planning problem where the agent starts from some $s_0 \in S$ and aims to achieve $s_g \in S$, producing a plan P (in the form of a sequence of actions $a \in A$).

Note that although usage of states often implies total domain observability, we do not make this assumption. As we will outline, a key piece of our approach is to acknowledge that there may be a discrepancy between expected and observed outcomes. Further, we consider a rolling window of size $k > 0$ for our fluents, allowing this data (e.g., sensor inputs) to be represented as $f_{t,k} = \{f_t, f_{t-1}, \ldots, f_{t-k}\}$. This is done to make it possible to address various forms of sensor noise collected on the robot.

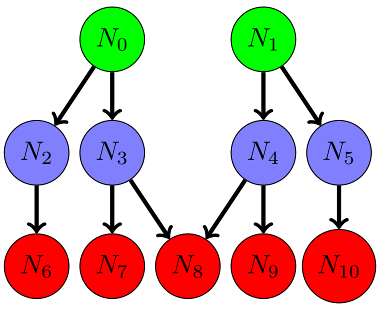
[1]See in particular Ghallab *et. al.* [13]

B. The Fault Discovery (FD) Graph

The core of our approach is to introduce a "Fault Discovery" (FD) layered directed acyclic graph $G = (\mathcal{N}, \mathcal{E})$. We draw a directed edge $E_{a,b} \in \mathcal{E}$ from node $N_a \in \mathcal{N}$ to node $N_b \in \mathcal{N}$ to indicate a causal dependence of the state or process represented by the node (e.g., how a wheel or navigation algorithm can be impacted by the state of having a jammed wheel). Nodes are of the form $N = (u, c, v, r)$ where u is a state in which this node is active and c is a self-evaluation strategy ("checker") which determines a value v storing the current evaluation of this node. We informally associate each node with some aspect of the agent for purposes of our own interpretation (e.g., saying that a node represents the agent's camera unit). As we will discuss, we place minimal restrictions on v to allow a wide variety of implementations within existing architectures. Finally, a node may or may not be marked as relevant ($r \in \{relevant, irrelevant\}$) to acknowledge that it may or may not be possible to check the value of a node in the current state.

At each layer, checkers associated with a node N are able to use the robot's perceptions as well as the values of other nodes in the graph to update a value for this particular node. For example, the robot may expect to see a high degree of similarity between similar sensors, between commanded versus actual motor output, etc. providing evidence that some subset of its operation is performing as expected: observing that three sources of odometry are in agreement is evidence that each of the three sources is functional; observing that progress towards a navigation goal is being made is evidence that both the odometry stack and wheels are operational, etc. Through propagation, we are then able to gain insight into processes which may not be directly observable.

Checkers associated with nodes having no parents are informed by observations about the world and the agent, providing the only input to the graph. If the state u of the node is met by the agent's current state, this node is marked as $r = relevant$ and the value of v is updated using information within F, as determined by the particular checker implementation. Checkers in other nodes use the values and operational status of their parents to determine their values and relevancy status. The relevancy status is set to "irrelevant" when all parents have a status of "irrelevant", and is "relevant" if the conditions set by the checker determine that the node is relevant. If all values $v \in G$ represent probabilities of normal operation of the respective nodes n, then the checker can calculate the conditional probability of $P(N|parents(N))$ given its parents. Alternatively, spreading activation (as implemented in neural networks) could be used, or logic-based implications of the form $\bigwedge parents(N) \rightarrow N$. In our implementation, we introduce some threshold level l,



	Node	Checker
🟢	N_0	Left wheel command = observation
🟢	N_1	Right wheel command = observation
	Node	Informal Description
🔵	N_2	Left wheel encoder
🔵	N_3	Left wheel motor
🔵	N_4	Right wheel encoder
🔵	N_5	Right wheel motor
	Node	Failure State
🔴	N_6	Left encoder failure
🔴	N_7	Left wheel jam
🔴	N_8	Power failure
🔴	N_9	Right wheel failure
🔴	N_{10}	Right encoder failure

Fig. 2: A simple graph for an imagined two-wheel robot, with encoders on each wheel capable of measuring the movement of that wheel. Observer nodes in green, system nodes in blue, problem nodes in red. N_0 and N_1 work to observe that when in the state of commanding wheel movement, wheel movement should be observed. This evidence of operation propogates to nodes representing each mechanism, allowing it to finally propogate to possible failure states. Note that despite the minimal number of inputs, a large amount of information can be inferred about the robot and its current operation, as well as the states impacting the robot's performance. While $N_7, N_8, N9$ may appear indistinguishable from each other, intentionally achieving the states associated with N_0 or N_1 may reveal useful information. N_6 is indistinguishable from N_7 (and N_9 from N_{10}), suggesting to a system designer that additional checker nodes would be valuable here.

which is a tunable parameter used to indicate a node is in an error state if $v < l$.

Edges between nodes then reflect the "flow" of evidence from observations to agent software processes and hardware, and possibly to representations of environmental states which may be impacting those processes and hardware. I.e., we draw an edge from node a to node b when node a is able to provide evidence that node b is functioning as expected. To make use of this representation, we will break nodes into three distinct types which continue to meet our existing definition of a node, but specialize within that definition: *observers*, *problems*, and *systems*. A minimal working example of such a graph is presented as Fig. 2.

1) Observer Nodes: Within the framework of a neural net, these are analogous to the nodes of an input layer. In our approach, these nodes aim to represent states within the environment which can be observed and evaluated, and so the checker for a node of this type must take in information relevant to this node's state u. For example, the state of moving forward may have a checker that observes current odometry data. In some cases these are observations that can always be made and should always be evaluated. For example, odometry sources should always have a high degree of similarity, and commanded motor outputs should be similar to measured motor outputs. We expect these to always be the case, and so the checkers associated with these nodes are always relevant: $u = \emptyset$ to indicate every state can satisfy the requirements for this checker.

2) Problem Nodes: These can be considered the output nodes, where the outputs are states which describe some failure (though as we will later explore, we will invert this graph to use Observer nodes as an output layer, producing tests). In our approach, these nodes aim to cover the case in which there is a state with the potential to cause a problem, but that cannot be explicitly determined. Nodes meeting this case will need to be assigned their respective values by following the chain of evidence through graph propagation. In this case, u is tied to some state causing failure: the aim is not to represent (for example) an IMU in failure, but rather to represent the state that has caused that IMU to fail. Thus, we restrict this state to representing things that may go wrong: jammed wheels, burned-out motors, networking errors, etc. In this way, the chain of evidence can be followed from an observation to a potential state in the world which needs to be resolved. While the state will be specific to this node, the checker will be the general propagation strategy.

With observer and problem nodes alone the robot can perform simple mappings of observations to causes of failure. In the more interesting cases, however, the relationship between a failure to meet expectation and the potential cause is not immediately clear. For example, is lack of similarity between IMU and wheel odometry due to a faulty IMU, broken wheel encoder, or wheel slippage? It is therefore useful to be able to represent the relationship between states and the robot's ability perceive or be impacted by these states, and so we introduce the final node classification.

3) System Nodes: These nodes are used to allow observations to propagate, and are analogous to hidden nodes. While we do not place formal restrictions on what they represent, it is often helpful to describe them as modeling the successful operation of a process or component. They may describe internal software processes which inform the checkers, or they may represent hardware components which may be impacted by a state in the environment. The checkers in these cases are always informed by propagation: a node representing an IMU (for example) should not self-evaluate, it should be evaluated based on what evidence is available to demonstrate that it is or is not operational. For these nodes, we set $u = \emptyset$ to state that they are always 'on' and evaluated through propagation.

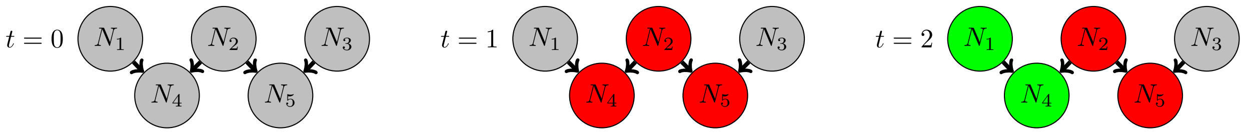


Fig. 3: Our algorithm on a simplistic graph. N_1,N_2,N_3 are Observer nodes, while N_4,N_5 are Problem nodes. At $t = 0$, the agent has no data input. At t_1, the agent reaches the state for N_2 and runs the associated checker, which fails. This failure propagates to N_4 and N_5. By explicitly performing the action associated with N_1, the agent can run another checker to rule out N_4 and conclude the problem is N_5. N_3 would have been an equally valid checker to select in this scenario.

C. Graph Construction

The graph used in our demonstrations is shown as Fig. 4. It is important to note that this one graph provides all simultaneous functionality. This broad functionality scope justifies the potential one-time graph construction cost. Additionally, it is reasonable to assume that a robot system has been configured prior to deployment (e.g., motion planning configurations, motor turning parameters, etc.). The existence of an expert capable of configuring these systems allows us to assume an expert capable of configuring this one. However, we observe that this is a cost already present in the previously-discussed one-to-one mapping approaches (which our approach outperforms, as we can replicate the functionality of one-to-one approaches while providing features they do not). We also observe that, as we now discuss, much of graph construction can be streamlined through automated processes.

1) Autonomous Graph Generation: First, observer nodes generally fall into two categories: one is data streams which, in some state of functional robot operation, ought to be highly similar. Checkers of this category can be trivially generated by recording a set of data during typical robot operation, creating a linear regression problem to determine appropriate correlations between sensors and/or robot commands. The second category involves states which are obtained from taking an action or from another state. In [10] and [19], the expected outcome of an action from a state is learned through past robot experience. It is easy to imagine the creation of checkers by using this data to form a base expectation. In our case, we collect a subset of data from successful operation, and run regression offline. In any case, the agent can produce a series of checkers where, if an action is taken or a state is obtained, the check will pass if the expected corresponding effect is detected. We exclusively use checkers of these two types, though more individually hand-crafted checkers may be useful in some applications.

For system nodes, we can again leverage DIARC's first-order symbolic logic approach to behavior. As discussed in [30], robot behaviors in DIARC take the form of "action scripts", which are themselves comprised of an ordered combination of one or more other action scripts or "primitive actions", with primitive actions being the coded functionality that the robot executes. Each primitive action is 'tagged' with the core system it depends upon (e.g., `closeGripper()` depends upon the gripper). Relations between checks evaluating actions and the processes they depend on can therefore be inferred as a fact of the architecture. Additionally, the system nodes of a given action script can therefore be found as the union of all systems of all actions within that script. Similarly, knowledge of what systems contribute to a given topic allow for checkers which make up that topic to be easily inferred. Further, the observation of System nodes being effectively hidden nodes would allow for existing training approaches to be used to generate these nodes.

Finally, problem nodes must be constructed. The agent, being symbolic or neuro-symbolic, has been provided with a set of states. Some of these states are examples of failures, and when a system or systems being non-operational provide evidence of this failure state, we connect those nodes. While the states themselves may also be dynamically constructed, dynamically determining how connections should be drawn requires insight which is challenging for autonomous systems. Further, the step of *solving* the problems represented by the states is challenging. We will discuss our work moving towards solving these issues in Section V, but overall we recognize it as a set of open problems well beyond the scope of this work: we are forced to assume that states describing failures, and the actions which resolve them, are already known to the agent.

When these approaches are combined in our specific setting, a very large set of checkers, and many system nodes, are produced autonomously. While this large set is not incorrect, it is again useful to leverage human intelligence in pruning it. For example, the Fetch actually has two IMU sensors: the graph appropriately detects that these should have a high degree of similarity, and we can reasonably infer that if they do not then one is broken. However, we are not interested in modeling this nuanced of a hardware system and so we consolidate their data. By pruning this (and other similar cases), we produce a graph that is largely equally functional while being more straightforward to understand and process. For purposes of a concise and easier-to-follow example, we will make use of a largely hand-crafted graph (visible in Figure 4).

D. Algorithms

Given an FD graph (e.g., the simplified graph in Figure 3), we can define the proposed general Fault Recovering Execution (FRE) algorithm (see Algorithm 1) for fault detection and discovery, as well as hypothesis generation and testing, which we now discuss.

At a high level, the algorithm takes an FD G, a set of already-known failures K (initially $K = \emptyset$), and a sequence of actions ($P = P_1, P_2, \ldots, P_k$) which it aims to perform. At

Algorithm 1 Fault-Recovering Execution (FRE) for failure-aware goal sequence execution

```
1: procedure FRE($P$, $G$, $K$)
2:     if $|P| = 0$ then
3:         return success, all steps met.
4:     Take action $P_0$ (where $P = \{P_0, P_1, \dots P_k\}$)
5:     Set of errors detected this timestep $D = \emptyset$
6:     for every node $N \in G$ do
7:         Run checker $c$ for $N$
8:         if $N$ is Problem Node, $r = relevant$, and $v < l$ then
9:             $D = D \cup \{N\}$
10:    Set of test states $T = \emptyset$
11:    if $D = \emptyset$ then
12:        Remove $P_0$ (where $\{P_0, P_1, \dots P_k\}$)
13:        FRE($P$, $G$, $K$)
14:    else
15:        if $|D \setminus K| > 0$ then          ▷ Narrowing down failure
16:            for Each $d \in D$ do
17:                $T_d = u$ for all ancestor nodes of $d$
18:            Remove unhelpful tests from $T$
19:            Get agent's current perceived state $s_0$
20:            if $T$ contains a non-empty set of states then
21:                $t$ = SelectExperiment($T$, $D$)       ▷ (Alg. 2)
22:                Prepend $(\Sigma, s_0, \{s_0 \wedge t\})$ to $P$   ▷ (Note 3)
23:            else                                ▷ Failure resolution
24:                Select state $s_d$ from some $d \in (D \setminus K)$
25:                Prepend $(\Sigma, s_0, \{s_0 \wedge \neg s_d\})$ to $P$
26:                $K = K \cup \{d\}$
27:            FRE($P$, $G$, $K$)
28:        else
29:            return failure, plan is obstructed.
```

every timestep, the agent recursively attempts to take the next action in the sequence (FRE:12) until there are no more actions to take (FRE:3). With each action taken, the agent updates the FD graph and uses it for failure detection (FRE:9).

If nodes are marked as being in a fault condition but have not yet been explored (FRE:15), the agent needs to find what set of states may be able to provide more insight as to what the specific cause of the error is. To do so, we make use of the fact that each node is associated with a state u: while many of these states will not be informative (as perhaps they have already been achieved), achieving one of these states will satisfy the conditions to run a checker that may provide more information to this node. We therefore collect all of the states for all ancestors of this potential failure node (FRE:17).

The output of this algorithm is collected with respect to each failure node which the agent believes may be responsible for the current fault, producing a set of test states to examine each failure node. However, not each state will be valuable: states in which we already know the outcome are not worth re-visiting, and states which relate to each failure node will not provide valuable information, either. For this reason, the agents performs a test state reduction step (FRE:18) before proceeding. This reduction step first removes tests which are relevant to every possible failure node, as these tests will be unable to provide any clarifying insights. The next step is to remove any tests in which the outcome is already known: recording the set of already-performed tests K allows the agent to avoid re-attempting tests that are unlikely to provide new information[2].

With the reduced sets of tests, we now attempt to find the most productive test, to the extent that this is possible (FRE:18). For simplicity, we will assume here that all fault states are equally likely (but nothing critical hinges on this assumption). For each node indicating a fault condition, the agent can find the number of times it appears in all fault conditions which indicates how many failure nodes will be impacted by the test at this node. Because the agent does not know the outcome of the test, the most effective strategy is one which impacts half of the failure nodes: if the test passes, the failure must be in the other half; if the test fails, the failure must be in this half.

Algorithm 2 Experiment selection

```
1: procedure SELECTEXPERIMENT($T$, $D$)
2:     Produce list of all potential test states $S_t = \bigcup T$
3:     for each test $t \in S_t$ do
4:         $C_t$ = # of occurrences of $t$ in all $T$
5:     Find midpoint $m = \lceil (max(C) - min(C)) \div 2 \rceil$
6:     for $n$ from $m$ to $max(C)$ do
7:         Attempt to select some $S_t$ with $n$ occurrences
8:         if selection is successful then
9:             return this $S_t$
```

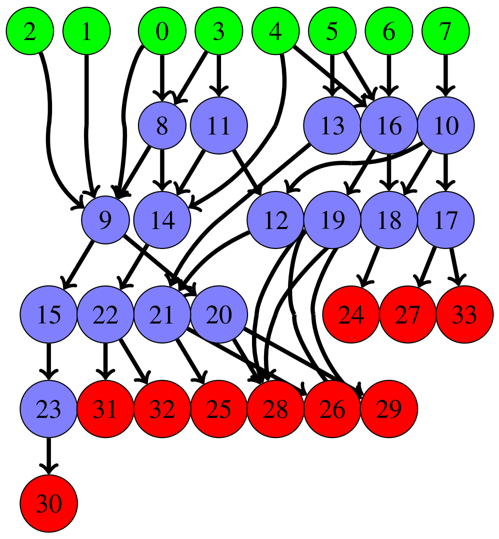
This arrives at a node which we are now interested in testing. The node has some state u, which can be provided as a goal state for the planner, producing a set of actions which can be prepended to P for execution (FRE:22)[3]. When the algorithm recursively continues (FRE:27), a new action is performed and the agent is provided with new insight into the possible failure. At some point, the number of possible tests will have been exhausted. This may be because there is one possible failure remaining, there are several failures, or there is total overlap between possible failures and their respective tests. In any of these cases, no further exploration can be performed and so the agent moves to failure resolution. The agent has some set of states which may be causing the failure, and which take the form of logical predicates. The negated form of these predicates can be provided as a planning problem and attempted (FRE:25).

IV. Addressing Failure

The goal of the demonstration is to evaluate the robot's ability to identify a problem, hypothesize a set of potential

[2]It is certainly possible for an error to be intermittent or for a test to fail. For our purposes, however, we do not consider these cases.

[3]Note that we use here the notation used in [13] to describe the solution to a planning problem as (Σ, s_0, S_g), where Σ is the planning domain, s_0 is the initial state, and S_g is set of states where the goal is satisfied.



N	Description	N	Description
0	Similar odom, command velocity	1	Similar fused odom, wheel odom
2	Similar fused odom, imu odom	3	Move base actions succeed
4	Gripper can obstruct LIDAR	5	Camera can see gripper
6	‘goToPose’ action succeeds	7	‘graspObject’ action succeeds
8	Platform mobility process	9	Odometry fusion process
10	Object manipulation process	11	Obstacle detection process
12	Depth camera unit	13	RGB camera unit
14	Laser data processing	15	IMU data processing
16	Arm movement process	17	Gripper unit
18	7-DoF arm unit	19	Torso unit
20	Differential drive unit	21	Head unit
22	LIDAR unit	23	IMU unit
24	Arm Broken	25	Camera Obstructed
26	Camera Broken	27	Gripper Broken
28	Hardware E-stop activated	29	Software E-stop activated
30	IMU broken	31	LIDAR obstructed
32	LIDAR broken	33	Gripper obstructed

Fig. 4: Recreation of the graph for the Fetch robot. Nodes in green highlight observer nodes; blue highlights system nodes, red highlights problem nodes. This one graph is used to model a variety of potential failure cases and their relationship with various observations and robot systems.

causes, test those hypotheses, and then resolve the problem. The demonstration is provided in video form[4]. Our first demonstration is summarized as Fig. 1. The algorithms presented in Section III have been fully integrated into the DIARC cognitive robotic architecture [30] running on ROS [28] and evaluated on an autonomous Fetch robot [36] performing navigation tasks.

Most critically, note that the same configuration is running across all demonstrations: we therefore show that our approach is able to correctly discern a failure case from a wide set of simultaneous possibilities. Our system is not arriving at a pre-configured conclusion or a “trial-and-error” approach. Rather, it is working through a logical and dynamic process of reducing the set of possible failure cases through intentional self-exploration.

A. Resolvable Failure

We first consider our motivating example: the condition where the head-mounted depth camera of the Fetch used for obstacle avoidance is obstructed (e.g., by fallen debris, see Fig 1a). Without fault detection and mitigation, the execution of the task fails while with our proposed approach, the robot can mitigate the fault. Specifically, the robot first observes that although wheel odometry and IMU odometry suggest it is moving as commanded by the navigation planner, progress towards the movement goal is not being made. As a result, while many of the checkers in the FD graph pass, the particular checker observing navigation progress fails, causing the robot to halt the current plan to enter the self-evaluation condition.

[4]Provided as supplementary material. Link in this footnote withheld for double-blind review.

Searching through the graph, the robot finds a set of problem nodes which are in failure states. Introspecting on the graph reveals a state `at_face(self,gripper)` with a checker, which links to some of these failure nodes. Based on the construction of the graph, the robot knows that it can achieve this state where it expects to be able to identify a fiducial marker mounted to the gripper. When the action is performed (Fig 1b), the test fails as a result of the obstructed camera.

However, after propagating the new evidence through the FD graph, it is unclear whether the failure is due to a camera problem or an arm problem. Hence, another test is necessary, and through the same test generation discovery process the state `at_laser(self,gripper)` is found. Performing this action (Fig 1c) and running the corresponding checker informs the robot that, as expected, placing the gripper in front of the LIDAR unit obstructs it. As before, the FD graph is updated.

The robot is now in a condition where, despite having more than one remaining potential failure node, there are no new tests it can perform because the potential problem states – `broken(self,camera)` and `obstructed(self,camera)` – both manifest themselves in the same way. While the robot does not have any plan to resolve a broken camera, it can find one to resolve the obstructed camera: the agent finds a plan which is identical to its current perception except that `not(obstructed(self,camera))` is satisfied. From this, the planner produces a face-sweeping action which is performed to resolve the current failure (Fig 1d). With the next recursive step, the FD graph is updated as the navigation is re-attempted, which occurs successfully as a result of the resolution of the fault.

B. *Mitigable Failure*

Some failure cases cannot simply be resolved. Instead, we are forced to mitigate their impact. Our approach additionally handles these cases thanks to the re-planning step: plans which might previously have been deprioritized can now be selected as the most viable strategy after the benefit of better understanding the current state of the agent and world.

One such set of cases includes sensor failures: if a critical sensor fails in the field, it is highly unlikely that it will be able to self-repair. Instead, it will be necessary for the agent to reduce the set of potential failure states as much as possible, and then find a new plan to resolve the goal while minimizing the impact of the ongoing failure. For example, consider a case where the robot has suffered damage to the LIDAR unit: this is a failure case the agent cannot resolve on its own, and the LIDAR data is a necessary for effective completion of the navigation task. While it is possible to use the head-mounted depth imaging, the robot is sensibly configured by the manufacturer to rely heavily on the high-rate LIDAR unit for obstacle avoidance.

Before beginning this demonstration, we have obscured the LIDAR unit to prevent it from providing sensible data. We have also provided the agent with a `reconfigureSensing(`x`,`y`)` action which, if activated, will reconfigure the sensing configuration such that the agent x will ignore data being produced by the sensor y. As before, the agent is then provided with a navigation goal, which fails as a result of the obstructed sensor.

The agent constructs a test to self-test its sensing. This time, however, `at_face` passes: the agent therefore infers that `obstructed(self, laser)` or `broken(self, laser)` must be true. A sweep in front of the LIDAR unit to satisfy `not(obstructed(self,laser))` does not resolve the problem, and so `broken(self,laser)` must be the current state. For the sake of mitigation, the `reconfigureSensing` action on the LIDAR unit is allowed to be a satisfactory resolution to the laser being broken.

Upon running this action, the set of navigation processes is reconfigured to operate at a much lower speed and to ignore the faulty LIDAR data. With the failure case being marked as 'resolved', goal progress resumes. With the navigation processes no longer perceiving obstacles from the LIDAR unit, it is able to complete the navigation task (though, at a less efficient speed than otherwise expected).

It is useful to note that it would be trivial to add a "go to the repair station" action, and to provide an ongoing goal of "be repaired if broken". Modifying the `reconfigureSensing` to trigger the state of being broken would then produce a behavior where the robot could return to a repair station after completing its goal (or hand off the goal to another agent and return immediately to the repair station, etc.).

C. *Simultaneous Mitigable Failure*

In some cases, failures will present simultaneously: for example, a power surge may damage both a sensor and a drive motor. Our approach is capable of handling several simultaneous failures through the same mechanisms as detecting any single failure: where we would previously narrow down to a series of indistinguishable failures and attempt solutions until we resolved the one, we now attempt solutions until we have resolved all of them.

To demonstrate this case, we imagine the Fetch having suffered a major collision: this one event has caused the gear drive of the left drive wheel to skip every other rotation, substantially reducing its rotation speed (which we simulate by modifying control velocities in real time, unknown to the rest of the architecture). In addition to this motor failure, the head unit is no longer sending appropriate data. Unlike previously, however, this failure case cannot be resolved by clearing debris. Instead, we imagine that it is a substantial damage which can only be resolved by long-term repairs. We model this damage by blocking the sensor with tape.

The agent first identifies failure thanks to the same process as the previous case, and the same debugging steps occur. This time, however, attempting to clear debris has not succeeded. With one of two possible solutions exhausted, the second is selected: the `reconfigureSensing` strategy is used to ignore the head sensing information going forward, allowing the agent to suboptimally complete its navigation task.

However, this alone has not resolved the agent's difficulties. The robot's movement still drifts sharply, which is observed by a checker attempting to assert a high degree of similarity between commanded and executed command velocities. Thanks to the extent of exploration already performed to resolve the prior issue, no further exploration is necessary to determine that an action to `reconfigureDriving` would be valuable. This action analyzes past driving data to apply a corrective counter-drift. With this action performed, driving behavior is able to proceed as normal.

D. *Simultaneous Non-Mitigable Failure*

Finally, some failures are such that any agent will be unable to resolve or even mitigate them. Consider a total motor power failure which prevents any movement, or a critical data cable that has frayed to the point of non-operation. Even in these cases, we find that our approach provides utility in being able to classify and identify potential failures. Although the agent may remain unable to self-resolve or even operate, a trivial "alert a human operator" action can be provided to the agent, providing it with the ability to update an operator with the information it has gathered about its failure case.

We demonstrate this case by again observing the navigation task. In this scenario, however, the emergency stop is triggered after navigation begins (cutting power to all motors but leaving sensing and computation online). As a result, the robot halts and navigation fails.

As with the demonstration of Section IV-A, the robot identifies that progress towards the navigation goal is not being made and triggers self-assessment. Again, introspection on the graph produces `at_face(self,gripper)`. However, when this state is planned to this time, the checker fails.

A series of other tests are similarly generated, all of which again fail due to an inability to perform any motion. As the agent has correctly identified, there is a problem with a wide variety of systems. As it further identifies, it has been unable to narrow down the source of the problem beyond a wide array of failures. However, of these, the most likely is one of the power failure cases.

The robot has exhausted its tests, observed that no actions can be taken, and that it therefore cannot resolve the problem. The remaining solution is a "call for help" action to alert a nearby human operator, which it performs.

V. Discussion and Future Work

Through the implementation of a novel performance monitoring system, we have enabled a robot agent to detect that a failure has occurred using the FD graph and take explicit actions based on the potential failures nodes in the graph to first isolate the fault and then attempt to mitigate the failure using the architecture's task planner. The core of the proposed method (the FD graph together with the recursive algorithms for determining faults and narrowing down potential causes) is general and can be integrated into any robotic architecture. It can be combined with machine learning approaches that learn how to detect faults and how faults can be mitigated (these models can be integrated as checkers and as a replacement for mitigation planners, respectively). It also allows for different mechanisms for propagating fault information in the FD graph that can be tailored to the available information (e.g., binary operational/non-operational values vs. probabilities of normal operation for nodes in the graph).

Future work focuses on improvements to generating graphs and on application areas. While we have shown portions of the graph can be auto-generated, more challenging is the relation of system nodes to failure nodes. We have shown that our system can autonomously detect failure, but classifying novel failures and relating them to states remains nontrivial. Further, this would require the generation of motor primitives capable of autonomous problem-solving, which falls far beyond the scope of this work. However, we are encouraged by the growth of creative problem-solving agents: such a system could be integrated into this system, using the exploration of the failure state as an effective tool to restrict the problem space. Similarly, we observe the success of natural language systems in cognitive architectures: in particular, there is much exciting work in using natural language as a tool to instruct new behavior to an agent. Our presented approach could be used as one valuable piece of a human-agent problem-solving dialogue, with the agent proposing potential failures and solutions, and a human partner instructing new methods to explore the failure or solve it.

We identify two key application areas for the methods in addition to simply increasing a robot's ability to finish its tasks under different perturbations: human-robot interaction and multi-agent tasks. The value of trust is well established in HRI, and similarly, perceptions of robot effectiveness are known to have a large impact on this metric[29]. Because our approach is based on explicitly represented sets of potential failure states, a more accurate description of failure can be provided (see again [7]), improving human trust in the robot even in cases where failures cannot be mitigated. In the multi-agent domain, one might imagine two identical robots which begin a task when one becomes damaged which turns a homogeneous multi-agent system into a heterogeneous multi-agent system where our methods could then be used to allow one agent to correct for the changed capacity of the other to maintain a high level of operation of the team (cp. to [31]).

VI. Conclusions

We proposed an integrated approach to fault detection and mitigation in a Fault Discovery (FD) graph based on representations of causal connections graph among different units, processes, and states in a robotic architecture. We showed tracing the dependencies of potential faults in graph allows a robot to produce experiments to isolate faults and then plan to mitigate them using the architecture's task and navigation planner. Future work will investigate more comprehensive methods for graph generation, with an emphasis on how failure states can be resolved once determined.

References

[1] Jose A Ambros-Ingerson, Sam Steel, et al. Integrating planning, execution and monitoring. In *AAAI*, volume 88, pages 21–26, 1988.

[2] Michael Beetz. Improving robot plans during their execution.

[3] D Paul Benjamin, Damian M Lyons, and Deryle W Lonsdale. Adapt: A cognitive architecture for robotics. In *ICCM*, pages 337–338, 2004.

[4] Catherina Burghart, Ralf Mikut, Rainer Stiefelhagen, Tamim Asfour, Hartwig Holzapfel, Peter Steinhaus, and Ruediger Dillmann. A cognitive architecture for a humanoid robot: A first approach. In *5th IEEE-RAS International Conference on Humanoid Robots, 2005.*, pages 357–362. IEEE, 2005.

[5] Baoping Cai, Lei Huang, and Min Xie. Bayesian networks in fault diagnosis. *IEEE Transactions on industrial informatics*, 13(5):2227–2240, 2017.

[6] Anthony R Cassandra. A survey of pomdp applications. In *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*, volume 1724, 1998.

[7] Gokay Coruhlu, Esra Erdem, and Volkan Patoglu. Explainable robotic plan execution monitoring under partial observability. *IEEE Transactions on Robotics*, 2021.

[8] Matthias Fichtner, Axel Großmann, and Michael Thielscher. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae*, 57(2-4):371–392, 2003.

[9] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[10] Tyler Frasca and Matthias Scheutz. A framework for robot self-assessment of expected task performance. *IEEE Robotics and Automation Letters*, 7(4):12523–12530, 2022.

[11] Thom Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37 (1-3):95–138, 1998.

[12] Zhiwei Gao, Carlo Cecati, and Steven X Ding. A survey of fault diagnosis and fault-tolerant techniques—part i: Fault diagnosis with model-based and signal-based approaches. *IEEE transactions on industrial electronics*, 62(6):3757–3767, 2015.

[13] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[14] Scott D Hanford, Oranuj Janrathitikarn, and Lyle N Long. Control of mobile robots using the soar cognitive architecture. *Journal of Aerospace Computing, Information, and Communication*, 6(2):69–91, 2009.

[15] Marc Hanheide, Moritz Göbelbecker, Graham S Horn, Andrzej Pronobis, Kristoffer Sjöö, Alper Aydemir, Patric Jensfelt, Charles Gretton, Richard Dearden, Miroslav Janicek, et al. Robot task planning and explanation in open and uncertain worlds. *Artificial Intelligence*, 247: 119–150, 2017.

[16] Inseok Hwang, Sungwan Kim, Youdan Kim, and Chze Eng Seah. A survey of fault detection, isolation, and reconfiguration methods. *IEEE transactions on control systems technology*, 18(3):636–653, 2009.

[17] Rolf Isermann. Supervision, fault-detection and fault-diagnosis methods—an introduction. *Control engineering practice*, 5(5):639–652, 1997.

[18] Rolf Isermann. Model-based fault-detection and diagnosis–status and applications. *Annual Reviews in control*, 29(1):71–85, 2005.

[19] Piyush Khandelwal, Fangkai Yang, Matteo Leonetti, Vladimir Lifschitz, and Peter Stone. Planning in action language bc while learning action costs for mobile robots. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

[20] Pieter Kraaijeveld, Marek Druzdzel, Agnieszka Onisko, and Hanna Wasyluk. Genierate: An interactive generator of diagnostic bayesian network models. In *Proc. 16th Int. Workshop Principles Diagnosis*, pages 175–180. Citeseer, 2005.

[21] Fabien Lagriffoul and Benjamin Andres. Combining task and motion planning: A culprit detection problem. *The International Journal of Robotics Research*, 35(8):890–927, 2016.

[22] John Edwin Laird, Keegan R Kinkade, Shiwali Mohan, and Joseph Z Xu. Cognitive robotics using the soar cognitive architecture. In *Workshops at the twenty-sixth AAAI conference on artificial intelligence*, 2012.

[23] Charles Lesire, Stéphanie Roussel, David Doose, and Christophe Grand. Synthesis of real-time observers from past-time linear temporal logic and timed specification. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 597–603. IEEE, 2019.

[24] Nils J Nilsson et al. Shakey the robot. 1984.

[25] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2), 2005.

[26] K Wojtek Przytula and Don Thompson. Construction of bayesian networks for diagnostics. In *2000 IEEE Aerospace Conference. Proceedings (Cat. No. 00TH8484)*, volume 5, pages 193–200. IEEE, 2000.

[27] Jordi-Ysard Puigbo, Albert Pumarola, Cecilio Angulo, and Ricardo Tellez. Using a cognitive architecture for general purpose service robot control. *Connection Science*, 27(2):105–117, 2015.

[28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, number 3.2, page 5. Kobe, Japan, 2009.

[29] Maha Salem, Gabriella Lakatos, Farshid Amirabdollahian, and Kerstin Dautenhahn. Would you trust a (faulty) robot? effects of error, task type and personality on human-robot cooperation and trust. In *2015 10th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 1–8. IEEE, 2015.

[30] Matthias Scheutz, Thomas Williams, Evan Krause, Bradley Oosterveld, Vasanth Sarathy, and Tyler Frasca. An overview of the distributed integrated cognition affect and reflection diarc architecture. *Cognitive architectures*, pages 165–193, 2019.

[31] James Staley and Matthias Scheutz. Evaluating task-general resilience mechanisms in a multi-robot team task. In *Proceedings of the Artificial Intelligence Applications and Innovations*, 2021.

[32] Kartik Talamadupula, Gordon Briggs, Tathagata Chakraborti, Matthias Scheutz, and Subbarao Kambhampati. Coordination in human-robot teams using mental modeling and plan recognition. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014.

[33] Michael Thielscher. Flux: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.

[34] J Gregory Trafton, Laura M Hiatt, Anthony M Harrison, Franklin P Tamborello, Sangeet S Khemlani, and Alan C Schultz. Act-r/e: An embodied cognitive architecture for human-robot interaction. *Journal of Human-Robot Interaction*, 2(1):30–55, 2013.

[35] Jan Winkler, Georg Bartels, Lorenz Mösenlechner, and Michael Beetz. Knowledge enabled high-level task abstraction and execution. In *First Annual Conference on Advances in Cognitive Systems*, volume 2, pages 131–148. Citeseer, 2012.

[36] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. Fetch and freight: Standard platforms for service robot applications. In *Workshop on autonomous mobile service robots*, 2016.