2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

RECOVER: A Neuro-Symbolic Framework for Failure Detection and Recovery

Cristina Cornelio[1] and Mohammed Diab[2]

Abstract— **Recognizing failures during task execution and implementing recovery procedures is challenging in robotics. Traditional approaches rely on the availability of extensive data or a tight set of constraints, while more recent approaches leverage large language models (LLMs) to verify task steps and replan accordingly. However, these methods often operate offline, necessitating scene resets and incurring in high costs. This paper introduces RECOVER, a neuro-symbolic framework for online failure identification and recovery. By integrating ontologies, logical rules, and LLM-based planners, RECOVER exploits symbolic information to enhance the ability of LLMs to generate recovery plans and also to decrease the associated costs. In order to demonstrate the capabilities of our method in a simulated kitchen environment, we introduce ONTOTHOR, an ontology describing the AI2Thor simulator setting. Empirical evaluation shows that ONTOTHOR's logical rules accurately detect all failures in the analyzed tasks, and that RECOVER considerably outperforms, for both failure detection and recovery, a baseline method reliant solely on LLMs. Supplementary material, including the ONTOTHOR ontology, is available at: `https://recover-ontothor.github.io`.**

I. INTRODUCTION

With the increasing use of robots in tasks involving humans in the perception-action loop, understanding the reasons behind failures in both planning and execution is a significant challenge for enhancing the reliability, adaptability, and safety of autonomous systems. Robots need to comprehend why and when failures occur and devise appropriate solutions based on the current situation. To achieve this, robots should be equipped with robust planning, perception, and reasoning capabilities enabling them to analyze failures and propose recovery strategies in real time.

The standard approaches to autonomous robots are typically model-based or policy-based [1]. Model-based approaches can involve offline planning, where the robot considers the current state and utilizes its model to predict the next state and potential rewards, enabling it to plan a sequence of actions expected to maximize reward. In online model-based planning instead, the robot continuously re-plans based on the current state, adjusting its actions in response to changes in the environment. Policy-based approaches usually entail either open-loop policy, where the robot predicts a sequence of actions based on the initial state and goal, or closed-loop policy, where the robot predicts individual actions at each moment based on the current state and goal. These policies guide the robot's decision-making process, facilitating adaptive behavior in dynamic environments. In our approach, we integrate elements from both methodologies mentioned above: we employ an ontology and a set of rules to represent the environment, and utilize these representations to refine a policy determined by a large language model (LLM) acting as a planner. This combined strategy enables the model to guide the policy, especially in scenarios where limited data availability restricts the policy's exposure to relevant instances.

In this paper we introduce an innovative use of ontological knowledge bases and LLMs for failure recognition and recovery within robotic systems, thereby enhancing the overall reliability and efficiency of robotic task execution. Our framework, named RECOVER, leverages the available symbolic knowledge about an environment (e.g., the set of available objects and their properties) to efficiently detect failures when they occur, in an online fashion *during* the task execution. The symbolic representation, embedded within an ontology, enables the robot to map multi-modal data (e.g., video, images, and audio) to the same representation, allowing simultaneous reasoning across all data types. When a failure has been identified, an LLM is employed as a re-planner, producing a set of steps to perform to recover from the failure and complete the task.

Employing LLMs for planning is not novel and has been recently proposed in various contexts [2], [3], [4], [5], [6]. However, LLMs come with several limitations, including a tendency to "hallucinate" [7], [8]. By incorporating symbolic information, we can steer the LLM-planner towards generating fewer hallucinations, thereby ensuring that the system operates within the confines of available objects and actions.

Current approaches utilizing LLMs for failure recovery and re-planning typically function in an offline manner: initially, a plan is executed in full to solve a task; if a failure is detected, a revised plan is generated and the task execution restarts from the initial state. This necessitates resetting the scene before executing the revised plan. Performing this method iteratively enables the refinement of the plan until one that successfully leads to task completion is generated [9]. However, this approach is impractical in real-world scenarios where, for instance, the environment undergoes changes during task execution (such as an object being broken or moved), or when, as often happens in robotics, even a correct plan may result in failure during execution (such as when a robot struggles to perform a feasible action). In such scenarios, the ability to act in real-time is crucial. Our framework detects failures during task execution and generates a new plan based on the environment conditions at the moment of failure, without needing to observe the effects of the failure throughout the entire plan.

[1]Samsung AI, Cambridge, UK `c.cornelio@samsung.com`
[2]University of Plymouth, UK `mohammed.diab@plymouth.ac.uk`

Utilizing symbolic knowledge alongside natural language also confers explainability features to the framework, enabling transparent and understandable reasoning behind the robot's actions and responses to failures. This attribute not only enhances trust and transparency but also helps in debugging and optimizing the system.

Furthermore, the use of ontologies for storing the environmental description and reasoning enables the integration of specialized knowledge and personalization, thereby adapting the system's capabilities to particular tasks or environments, enhancing its effectiveness and versatility. As a contribution of our work, we extend established taxonomies [10] of failure categories in a human-robot interaction context, incorporating preferences such as allergies and dietary restrictions. We release an ontology named ONTOTHOR, specifically designed for the AI2Thor simulator, which we utilize in our experiments. ONTOTHOR describes kitchen environments and includes personalized features.

This kind of ontologies are particularly useful when interacting with human agents who may possess diverse preferences or requirements, or in scenarios involving technical and specialized equipment. Examples of such use cases are healthcare support (e.g., surgical robots or assistant bots), automated transportation (including spacecraft, planes, or cars), and household assistance (e.g., kitchen robots or cooking assistants).

A. State of the Art

Recent studies have investigated the utilization of pretrained LLMs for planning and executing actions in interactive environments, by using the priors of the LLM for plan generation [5], [11]. Usually, this involves converting multimodal observations into natural language, utilizing an LLM to generate domain-specific actions or plans, and then employing an agent for execution. However, these approaches are susceptible to hallucination and lack deep understanding and reasoning capabilities [3].

Reasoning, especially when grounded on concrete actions, is one of the most desired capability of LLMs, as it offers explainability and enables the generation of more meaningful solutions [2], [12]. However, despite substantial progress in the reasoning abilities of LLMs [13], the requirement to detect and correct errors in the reasoning process remains an open problem.

Some progress has been made in this regards: ReAct [14] is a prompting refinement technique that leverages the reasoning and planning capabilities of LLMs to iteratively refine a plan to solve a task. Although it introduces innovative features like iterative action-environment observations and LLM-policy refinement, it may face challenges in complex or real-world environments. Reflexion [15] offers a unique approach to reinforcing LLMs-based agents through verbal reflection on task feedback. By maintaining reflective text in an episodic long-term and short-term memory, it enhances decision-making in subsequent trials across diverse tasks. As an extension of Reflexion, ExpeL [16], SALAM [9], and SAMA [17] enhance their capabilities by analyzing patterns in failures and successes to extract valuable insights. These insights are then integrated into prompts, thereby improving decision-making and adaptive behavior. RetroFormer [18] introduces a framework to bolster LLMs-based agents by refining prompts based on environment feedback. Through policy gradient techniques, the system adjusts prompts iteratively, learning from rewards across various tasks and environments. This process enhances the LLM's performance by summarizing past failures and suggesting action plans for improvement.

The increasing use and advancements of LLMs in recent years have led to their widespread application across various research fields in AI, including robotics [19], [20], [21], [22]. Most relevant to our work is REFLECT [23] a framework that, similarly to Reflexion, utilizes LLMs in the context of robot failures to explain them and propose corrective plans. REFLECT demonstrates great promise in addressing robot failures and proposing corrective plans, however, it is currently unable to accommodate human preferences and personalization. Moreover, the necessity to observe the execution of the entire plan, reset the scene[1], and the cost of the extensive use of LLMs (both computational and monetary) are strong limitations.

II. THE METHOD: ONTOLOGY-BASED FAILURE IDENTIFICATION AND RECOVERY

RECOVER is a neuro-symbolic framework for failure identification and recovery that exploits symbolic knowledge to aid the execution of a task. The symbolic knowledge is provided in the form of an ontology and a set of logical rules that describe the environment, the actions of the robot, and the preferences of human agents. Moreover, it contains information about the possible failures that can occur in that particular environment and the corresponding recovery strategies. An overview of the RECOVER methodology is depicted in Figure 1.

To solve an input task, RECOVER takes as input a plan to execute and an ontology describing the environment. The plan comprises a symbolic, however interpretable, sequence of actions that the robot can perform on a given environment (e.g., *pick_up(mug)*). An ontology is a formal description of a domain or an environment, organizing concepts, classes, and entities along with the relationships between them. The knowledge is organized in the form of unary (representing the type/class of an entity) and binary (representing the relationship that might exist between two entities) predicates. Moreover, an ontology also has a set of rules that define properties and relations between the classes (e.g., all the objects of class Mug are also Fillable objects). In our scenario, the ontology describes the environment in which the robot is acting, the preferences of one or more human agents (e.g., dietary restrictions), and safety conditions (e.g.,

[1]In REFLECT, replanning necessitates completing the entire task first. After the failure is identified, a new plan is created to address it. The scene must then be reset, and the task restarted from the beginning using this updated plan. Although the failure will reoccur, the revised plan will include the necessary corrections leading to the successful execution of the task.

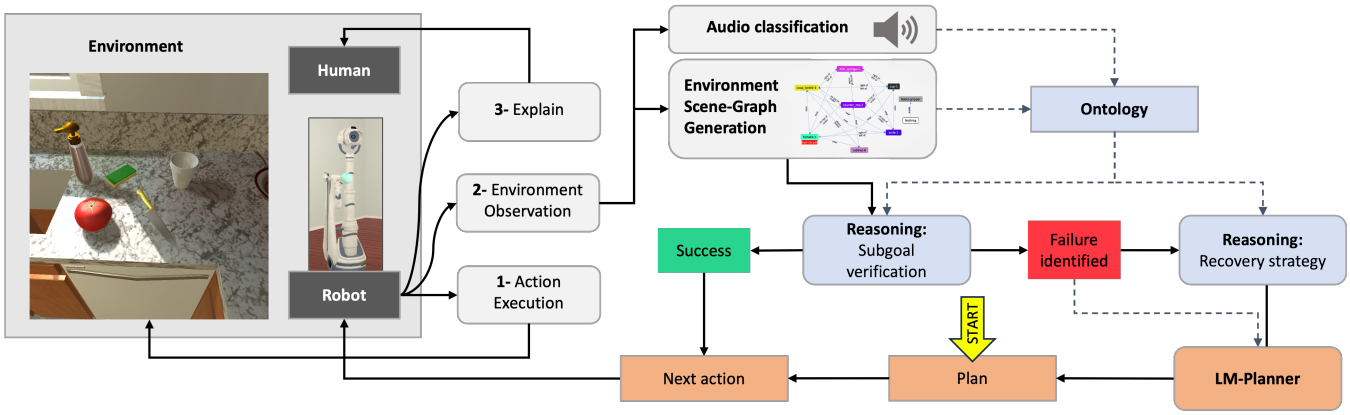


Fig. 1. Overview of the RECOVER framework. Starting with a plan the robot executes one action at a time over the environment. The outcome of each action conveyed through multi-modality information (audio and video) is then processed and converted into an audio label and a scene-graph. These are stored within the ontology and provided as input to the sub-goal verifier, which classifies the action execution as either a failure or a success. If the action is successful, the robot proceeds with the next step. In the event of failure, reasoning module will use the failure information to extract the recovery strategy from the ontology and will supply it to the LLM-planner. Subsequently, the LLM-planner generates a new plan to recover from the failure and accomplish the task. *Figure legend:* Blue elements represent model-based components, while orange elements denote policy-based components. Sharp-cornered shapes indicate input/output elements, whereas round-cornered shapes signify procedures. Dashed lines correspond to input sources, while solid lines indicate the procedural loop.

the stove must not be left on). During task execution, the ontology is populated by entities (also called instances), such as concrete objects, agents or events (e.g., mug-1, human-3, event_002). In what follows we will denote the ontological classes starting with a capital letter (e.g., class Apple) and instances with lower case letters (e.g., object apple-1).

The framework starts with the execution of the first action in the plan. The action is given to the robot that will (1) execute the action on the environment; (2) observe the status of the environment after the action has been performed; and (3) explain the action to the human using an LLM to translate the step into natural language.

While the action is being executed, the robot records any sounds that may occur, and after the action is completed, the robot observes the scene. The auditory information gathered by the robot is mapped by a classifier into a label s chosen within a set of available sound classes $\{s_1, \ldots, s_k\}$ (e.g., *ToggleOnMicrowaveSound* or *CloseFridgeSound*) and then stored in the ontology in the form of triples (e.g., (*event_001, has_sound, s*) and (*s, type, CloseFridgeSound*)).

The visual information gathered by the robot after the action execution is processed into a scene-graph. A scene-graph is a graph where each node corresponds to an element in an image. Each node is classified to belong to a certain class $c \in \{c_1, \ldots, c_n\}$ (e.g., Apple) representing the type information of the object. The edges between the nodes correspond to the binary relation r between two objects in the scene, and is chosen within a set of possible spatial relations $\{r_1, \ldots, r_m\}$. Each edge and the two corresponding nodes constitute a triple (e.g., (*apple-1*, *on-top-of*, *table-1*)). Additionally each object can have one or multiple states, that represent the condition of the object at a particular time step (e.g., a mug can be dirty or clean). Figure 2 shows three scene-graphs corresponding to the scenes observed by the robot before and after two action executions.

Once the environment observation results are stored in symbolic form within the ontological knowledge base, a set of logical rules, defining potential failures in the environment are applied. The application of the rules is done with a symbolic reasoner, such as a SPARQL reasoning engine, a RDF-reasoning engine (e.g., RDFOX [24]) or a more expressive reasoner (e.g., prolog [25] for first order logic). The result of the reasoning process is provided as either a success state, meaning that the action was performed successfully, or as a failure instance (e.g., *DroppingObjFailure*), meaning that the action was not performed successfully. In a success state, the robot will proceed with the next step of the plan. Instead, if a failure is inferred by the rules, the robot needs to replan before proceeding with the next action.

A set of recovery instructions are stored in the ontology for each failure type. These are provided as input to the LLM, along with the original plan, the goal of the task, the success condition, and the environment state in the form of natural language text.

The LLM then generates the new plan as a sequence of steps. This text is then mapped to a sequence of commands executable by the robot by computing a similarity measure between the embedding of each sentence and the elements in the pool of available actions that the robot can execute in a given environment.

Once the new plan is finalized, the robot resumes the loop by executing the next available action.

III. ONTOTHOR ONTOLOGY

We created an ontology, named ONTOTHOR, characterizing the kitchen environment of AI2Thor simulator [26]. AI2Thor is a 3D simulator designed for research in embodied artificial intelligence. It provides a realistic 3D environment where agents can navigate and interact with objects in a virtual home setting. AI2Thor offers various functionalities such as object manipulation, scene understanding, and task

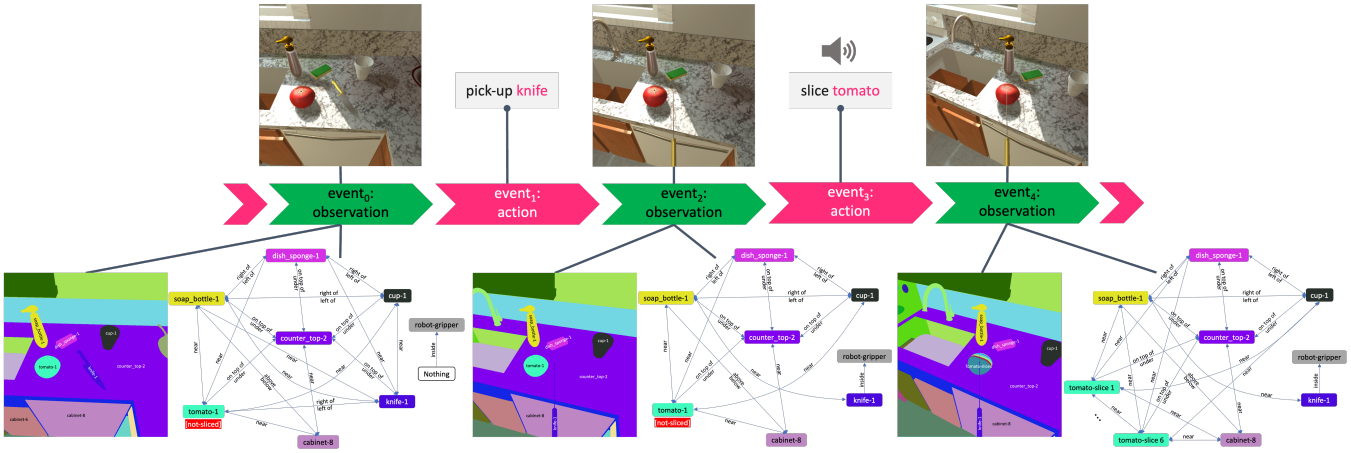


Fig. 2. Example of the information flow during plan execution, depicting the alternation of two event types: observation events (in green) and action events (in pink). Each action event can be associated with a sound recorded by the robot during the action's execution. Each observation event is linked to a scene-graph describing the frame captured by the robot. In the scene graph, each node represent an object identified in the scene and each edge represent the relation between two objects. The colors of the nodes in the scene graph correspond to those in the segmentation image (the adjacent image) where the objects have been identified.

execution and it serves as a valuable tool for studying navigation, object interaction, and spatial understanding within indoor environments.

ONTOTHOR contains the following classes describing the environment: **Action** (agent behaviors such as interactions with objects and surroundings), **Agent** (humans and robots, with humans categorized in different classes based on their preferences, e.g., Celiac or Vegan etc.), **PhysicalObject** (kitchen items categorized into consumables (e.g., fruits, vegetables), cookware (e.g., pots, pans), and appliances (e.g., ovens, coffee machines)), **PhysicalProperty** (attributes like breakable or fillable), **Sound** (types of kitchen sounds, e.g., opening fridges or dropping objects faucets), **SpatialRelation** (positional relationships between objects), **State** (current conditions of objects at specific times, like being broken or being empty), **Time** (event tracking with time stamps), and **Location** (home areas such as kitchen and living room).

Once the task execution is started, the ontology is populated by all the entities/individuals present and/or discovered during the task execution and by all the events that occurred. For example, if an object of class *Apple* is present in the scene, an entity called *apple-1* is created under that class. A number is always part of each object instance name to uniquely identify the object in the room. This is necessary since there might be multiple instances of the same object type (e.g. there are 2 apples in the kitchen). In what follows, for simplicity, we will omit the numbers when not necessary.

We defined the following classes in the ontology to facilitate the storage of events during the robot's task execution: **Event**: Describes both observation and action events, as described in Figure 2. *Action events* detail agent actions, including the involved source/target object, timing information and presence and type of sound that occurred during the action execution. *Observation events* record the states of physical objects and their spatial relations at a specific time in the form of scene-graph triples.

Triple: Employs second-order logic to describe the state of the environment. Each triple contains a subject, predicate, and object such as (*mug-1*, *near*, *cabinet-7*). These triples are linked only to observation events using the *hasTriple* relationship, enabling a comprehensive semantic description of the current event.

RecoveryStrategy: Guides the robot's actions in specific failure situations, considering factors such as object state and material. For example, if a cup breaks, the strategy may involve selecting a new mug; however, if it's plastic, the robot may opt instead to retrieve it. These strategies are encoded in natural language within the ontology by domain experts.

As described in Section II (see also Figure 2), during the task execution there is an alternation of *action events* and *observation events*. Thus, for each step in the plan, the ontology creates 2 events: $event_i$ with an associated action (e.g., *pick_up*) and $event_{i+1}$ with an associated observation-action. For example, in Figure 2, there are 5 events: $event_0$, $event_2$, and $event_4$ with action *observation*; $event_1$ with action *pick_up* and target object *knife*; and $event_1$ with action *slice* and target object *tomato*.

Each action event might also have sound information: for example in Figure 2 the event $event_1$ is not associated with any sound while $event_3$ (with action *slice*) has sound $sound_3$ of type *SliceVeggySound*.

As mentioned above, each observation event is connected to a set of triples that represent the scene-graph information. In Figure 2 the scene graph associated with the observation actions is depicted by the graph in the the bottom row of the image. For example, $event_2$ has 14 triples including (*knife-1*, *inside*, *robot-gripper*), (*tomato-1*, *near*, *soap-bottle-1*), and (*dish-sponge-1*, *on-top-of*, *counter_top-2*).

This approach permits the storage of the entire process within the ontology, allowing the utilization of reasoning to detect the presence of any failures.

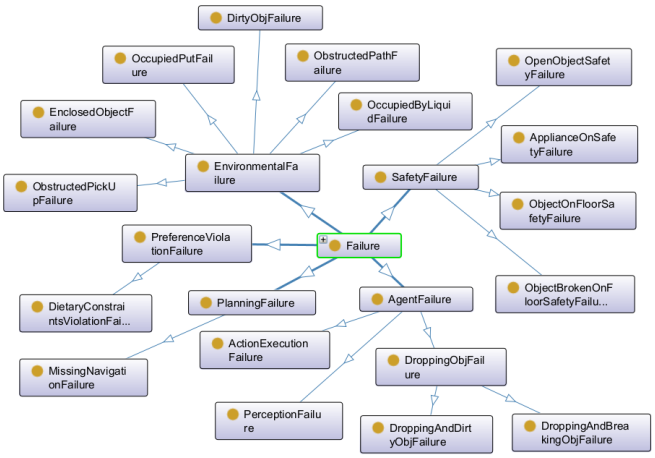


Fig. 3. Failures taxonomy in ONTOTHOR

A. Failures Classification and Detection Rules

The ontology categorizes also the potential types of failures that may arise in the AI2Thor environment, as depicted in Figure 3. The class **Failure** is a subclass of the **Event** class, as a failure can potentially be identified within each event. The classification is organized as follows:

AgentFailure: Comprises diverse sub-classes representing different failure types that occur during operation, including action execution failure, dropping object failure (potentially leading to breakage or soiling), and perception failure (e.g., misidentifying objects).

EnvironmentalFailure: Contains various types of failures related to environmental conditions. It covers situations such as dirty objects, inaccessible or obstructed paths, and receptacles filled with liquid or occupied placement regions.

PlanningFailure: Addresses cases where the original plan is incorrect, such as when steps are missing or unfeasible.

PreferenceViolationFailure: Identifies personalization failures related to human preferences, such as when individuals dislike certain ingredients or follow a specific diet.

SafetyFailure: Is used to maintain a secure environment. For example, a safety failure occurs if a glass breaks and shards scatter on the floor, or if the stove is left turned on.

Additionally, we defined a set of rules, each corresponding to a specific failure type that may arise within the AI2Thor kitchen environment. The rules are applied following each observation event to identify whether the executed action has resulted in a failure or not. For example, a rule to identify a failure generated by dropping an object (*DroppingObjFailure*) is as follows:

```
1  Event(e) ∧ hasAction(e,a)
2  ∧ (ActionWithHeldObject(a) ∨ NonInteractiveAction(a))
3  ∧ hasPreconditions(e,pre_c) ∧ hasTriple(pre_c,trp1)
4  ∧ hasSubject(trp1,held_obj1) ∧ ¬(Nothing(held_obj1))
5  ∧ hasObject(trp1,rg) ∧ RobotGripper(rg)
6  ∧ hasPostconditions(e,post_c) ∧ hasTriple(post_c,trp2)
7  ∧ hasSubject(trp2,held_obj2) ∧ Nothing(held_obj2)
8  ∧ hasObject(trp1,rg)          → DroppingObjFailure(e)
```

TABLE I
TASKS IMPLEMENTED IN THE EXPERIMENTS

	ID	Name	#steps	#objects
easy	T1	Serve wine	8	2
	T2	Make coffee	9	2
	T3	Boil water in a pot	10	2
	T4	Fry egg in a pan	11	2
	T5	Toast bread	12	3
	T6	Warm water (in microwave)	16	3
	T7	Cook potato slice (in microwave)	20	3
complex	T8	Simple salad	26	4
	T9	Clean/order kitchen	28	7
	T10	Vegetarian sandwich	30	5
	T11	Cook egg and potato slice	32	7
	T12	Complex salad	33	7

where the symbols $\wedge$, $\vee$ and $\neg$ are the standard AND, OR and NOT logic operators and $\rightarrow$ corresponds to the logic implication. This rule states that if the robot was holding an object (e.g., a knife) and performs an action with it (e.g., cutting an apple) and after the action it is not holding the object anymore, it means that the robot dropped the object during the execution of the action. More formally, the rule states that if there is an event e where (lines 1 and 2) the action a to be performed is not placing/picking-up an object or an empty-gripper action; (lines 3, 4 and 5) the robot was holding an object $held\_obj1$ before the action (since there is a triple $trp1$ in the preconditions pre_c – the environment status before the action execution – that indicate that something is in the robot gripper); and (lines 6, 7 and 8) the robot is not holding the object anymore after the action (since there is a triple $trp1$ in the post-conditions $post\_c$ – the environment status after the action execution – that indicate that nothing is in the robot gripper); then the event e is an instance of a *DroppingObjectFailure*.

IV. EXPERIMENTAL SETTING

For the experiments we implemented 12 tasks and 12 failure types in the kitchen scenario of the AI2THor simulator [26]. An overview of the different tasks is provided in Table I. The tasks are ordered based on the number of steps that are present in the original plan. Moreover, the tasks are divided in two categories: easy tasks with at most 20 steps, and complex tasks that have more than 20 steps and more than 3 objects.

The failures implemented in the experiments are a subset of the ones described in the ontology (due to the limitations of the simulator): (1) *EnclosedObjectFailure*; (2) *DroppingObjFailure*; (3) *DroppingAndDirtyObjFailure*; (4) *DroppingAndBreakingObjFailure*; (5) *DirtyObjFailure*; (6) *OccupiedPutFailure*; (7) *PlanningFailure* (missing step); (8) *ActionExecutionFailure*; (9) *DietaryConstraintsViolationFailure*; (10) *PlanningFailure* (wrong step); (11) *OccupiedByLiquidFailure*; and (12) *MissingNavigationFailure*. Not all the failures are possible in all the tasks: for example *DroppingAndBreakingObjFailure* is not available for the task "Boil water in a pot" since a pot is the only object that is picked up during the task and it is a non-breakable object.

All the failure-task combinations that are not available are indicated with a gray box in Table II.

A. Implementation details

We implemented the general pipeline in an entirely modular fashion, such that each component can be easily replaced with alternative methods.

Planning. For the initial plan, we used the ground-truth plan for each task, while the recovery plan was generated with GPT-4 acting as (LLM-based) planner. Alternatively, the initial/original plan can also be automatically generated using an LLM.

Steps mapping. The LLM's output is mapped to a set of feasible actions using a large pre-trained sentence embedding model (similar to [23] and [5]). Each sentence generated by the LLM (plan step) is matched to an executable action by maximizing cosine similarity between its embedding and those of feasible robot actions. We used a BERT-based model (from the *sentence_transformers* python library) for the embeddings.

Scene graph generation. Techniques to transform an image into a scene-graph have drastically improved in the last years. Scene graph generation can be done in different ways: for example by layering different models such as image segmentation, image classification and scene-graph predicate classification (as in [23]), or with an end-end model (a popular example is the work of Knyazev et al. [27]). In our experiments we follow the approach of Liu et al. [23]. This is a rule-based approach for predicate classification that, given the ground-truth bounding boxes, labels and state of the objects present in a scene, computes the relation label between each pair of objects based on the proximity and relative location of the items.

Sound classification. In the experiments we use the ground-truth labels for the detected sounds. An alternative neural-based sound classifier (such as [28]) can be used and is available in our code.

Failure detection. The failure rules are written in SPARQL, and their application is done with the SPARQL engine available in OwlReady2 python library [29]. We use the same library to store and query the ontology.

The experiments were performed on a Linux machine with 20 cores and an NVIDIA GeFOrce RTX 3090Ti GPU.

In the experiments we assume that there is at most one failure per task. A straightforward generalization of RECOVER would allow the identification and correction of nested failures (failures that happen during the recovery plan) as well.

V. EXPERIMENTAL RESULTS

The experimental results can be summarized as follows: 1) We demonstrated the capabilities of our rule-based subgoal-verifier, which achieved 100% on the selected tasks and failures; 2) We demonstrated the capabilities of our ontology-enhanced LLM re-planning pipeline on 90 task-failure pairs, obtaining a good success rate; 3) We show that the safety issues were correctly identified in all tasks and corrected in more than 90% of cases; 4) We compared with a purely neural online approach consisting of an LLM-based subgoal verifier and an LLM-based re-planner (more details below) and we showed that we significantly outperform both; 5) We demonstrated that with RECOVER we have a significant reduction of the costs.

Baseline model. Most of the failure identification and recovery methods available are designed for an offline setting, and are therefore not suitable for a direct comparison with RECOVER. One noticeable example is REFLECT [23]. However, despite the availability of their code, converting it to an online version would necessitate substantial effort. For this reason, we implemented as baseline a similar approach that is purely LLM-based and uses the same prompting developed in REFLECT. The method has two main components: 1) an LLM-based sub-goal verifier (**LM-SGV**) which is a binary classifier that assesses, at each step of the plan execution, whether an action has been executed successfully or not. It bases its evaluation on the environment's scene-graph before and after the action, the audio detected, and the list of available objects in the scene. 2) an LLM-based re-planner (**LM-RePl**) that receives as input the scene-graph of the current state of the scene, the list of available objects, the original plan, and the task goal description. It then generates a recovery plan as output.

We compare with the baseline model in two different settings: 1) Task-failure pairs where RECOVER successfully identified and corrected the failure (first two colored lines in Table III); 2) Task-failure pairs where RECOVER successfully identified the failure but failed to correct it (third and fourth colored lines in Table III).

A. Sub-goal verification

Our rule-based sub-goal verifier successfully detected failures in 100% of the analyzed cases. Conversely, the LLM-based sub-goal verifier (LLM-SGV, introduced above) achieves an accuracy of only[2] approximately 50% in both the analyzed scenarios (task-failure pairs where RECOVER successfully identified and corrected the failure, and task-failure pairs where RECOVER successfully identified the failure but failed to correct it). This implies that the failure was correctly identified at the correct execution step in only half of the cases. The results are provided in Table III under the category LLM-SGV.

B. Task re-planning

The failure recovery results for RECOVER are shown in Table II. The table is divided into two parts: the first block shows the results for easy tasks with at most 20 steps; and the second block shows the results for complex tasks with more than 20 steps.

[2]The poor performance of LLM-SGV may be attributed to the large volume of raw environmental information fed into the LLM (which is the exact same input used by the rule-based sub-goal verifier). Employing more advanced techniques, such as summarization, could enhance the performance of both, but this would result in a significant increase in computational cost.

TABLE II

SUCCESS RATE OF ONT-REPL REPLANNING WITH LLMS

The recovery rate is around 70% and it is consistent between easy tasks and complex tasks. However, the completion rate (whether the task is completed successfully after the recovery) is higher for easy tasks (59%) compared to complex tasks (33%) This is to be expected, since with a longer plan, the probability of the LLM to induce an error in the planning increases.

It is evident that certain failures are more straightforward to recover from, such as *MissingNavigationFailure*, while others, such as *OccupiedByLiquidFailure*, pose greater challenges. One contributing factor is the bias inherent in the LLM: for example, the LLM often generates re-plans assuming a robot with two arms, whereas the robot in the AI2Thor simulator is equipped with only one arm. As a result, the recovery plan may become infeasible because the robot cannot execute actions involving two objects simultaneously; instead, it must perform actions with one object at a time, sequentially. This discrepancy is particularly noticeable in tasks involving 2 objects that normally would be manipulated simultaneously (e.g., slicing food or pouring liquid). This issue persists even when explicitly specifying in the prompt that the robot has only one arm.

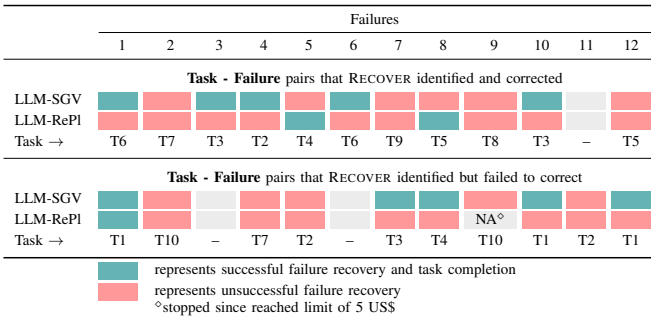
The results of the comparison with the baseline planner (LLM-RePl, introduced above) over easy tasks (when possible) are shown in Table III. We can see that in the first block of the table (containing tasks-failure pairs that RECOVER successfully identified and corrected), LLM-RePl was able to re-plan correctly only 18% of cases. In the second block (containing tasks-failure pairs that RECOVER successfully identified but failed to correct), LLM-RePl is able to re-plan correctly only 11%. We thus demonstrated that our method outperform significantly a pure LLM-based re-planner.

C. Additional Results

Safety. Our method identifies 100% of the safety issues that occurred during task executions, and it recovers successfully from 93% of them[3], restoring the safety of the environment. The LLM-based method instead only identifies and corrects 31% of the safety issues in the scenes analyzed.

TABLE III

SUCCESS RATE OF LLM-SGV AND LLM-REPL FOR EASY TASKS



Cost. Each API call to a LLM incurs a token-based cost, with different rates for input tokens (*prompt_tokens*) and output tokens (*completion_tokens*). In our case, the version of GPT used charges 30.0$ per million input tokens and 60.0$ per million output tokens. Consequently, the total cost is calculated by summing the costs for both input and output tokens: $cost = (30.0 * prompt\_tokens/1M) + (60.0 * completion\_tokens/1M)$. As indicated in Table IV, our approach demonstrates a significantly greater cost-effectiveness compared to an LLM-based method, resulting in a noticeable reduction in monetary costs. It is important to note that the value reported for the LLM-based method serves as a conservative estimate and may underestimate the actual costs. In our experiments, this value is heavily influenced by the step in the plan at which the failure occurs. The later the failure arises, the higher the associated cost, as the LLM-based sub-goal verifier queries the LLM at each step until a failure is detected. If the LLM fails to identify the failure, it continues to query the LLM at each subsequent step. This situation is exemplified by task T10 with failure 9, where the cost exceeded our predefined threshold of 5 US$.

VI. CONCLUSIONS AND FUTURE WORK

We presented RECOVER, a neuro-symbolic framework designed for online failure detection and recovery. Our approach exhibits good performance, surpassing a LLM-based baseline method. Moreover, we demonstrated it to be cost-effective. This is especially important in situations where failures are known or recurring, making the ability to identify them effectively and cost-efficiently crucial. Finally, given its personalization capability, RECOVER is able to identify and correct safety issues in an environment.

In future extensions, we plan to broaden the scope of this work by exploring simulator scenarios with human interaction. This entails studying how ontology-based failure detection and recovery mechanisms function when humans are actively engaged in the perception-action loop or when

[3]This percentage considers only the successful tasks. When failed tasks are also considered, the recovery rate is 86%.

TABLE IV
COST OF RECOVER VS LLM-BASED APPROACH (IN US$)

	Failures											
	1	2	3	4	5	6	7	8	9	10	11	12
RECOVER	**0.08**	**0.13**	**0.21**	0.28	**0.15**	**0.09**	**0.08**	**0.11**	**0.08**	**0.19**	–	**0.16**
LLM-based	0.14	0.16	0.91	**0.16**	1.04	0.82	0.21	0.16	1.00	0.51	–	0.36
Task →	T6	T7	T3	T2	T4	T6	T9	T5	T8	T3	–	T5
RECOVER	**0.13**	**0.14**	–	**0.06**	**0.08**	–	**0.18**	**0.16**	**0.07**	**0.09**	**0.08**	**0.06**
LLM-based	0.27	2.13	–	0.16	0.57	–	0.54	0.58	>5	0.42	0.57	0.27
Task →	T1	T10	–	T7	T2	–	T3	T4	T10	T1	T2	T1

specialized knowledge has crucial impact. A deeper integration of LLMs with ontologies [8] within a hybrid framework could significantly augment both reasoning capabilities and re-planning quality. Finally, further refinements in prompt engineering would undoubtedly enhance the system's performance.

REFERENCES

[1] K. Chatzilygeroudis, V. Vassiliades, F. Stulp, S. Calinon, and J.-B. Mouret, "A survey on policy search algorithms for learning robot controllers in a handful of trials," *IEEE Transactions on Robotics*, vol. 36, no. 2, pp. 328–347, 2019.

[2] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *CoRR*, vol. abs/2201.11903, 2022.

[3] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," *arXiv preprint arXiv:2204.01691*, 2022.

[4] S. Yao, R. Rao, M. Hausknecht, and K. Narasimhan, "Keep calm and explore: Language models for action generation in text-based games," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[5] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *International Conference on Machine Learning*. PMLR, 2022, pp. 9118–9147.

[6] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[7] O. Golovneva, M. P. Chen, S. Poff, M. Corredor, L. Zettlemoyer, M. Fazel-Zarandi, and A. Celikyilmaz, "Roscoe: A suite of metrics for scoring step-by-step reasoning," in *The Eleventh International Conference on Learning Representations*, 2022.

[8] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *IEEE Transactions on Knowledge & Data Engineering*, no. 01, pp. 1–20, 2024.

[9] D. Wang and L. Li, "Learning from mistakes via cooperative study assistant for large language models," in *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[10] M. Diab, M. Pomarlan, D. Beßler, A. Akbari, J. Rosell, J. Bateman, and M. Beetz, "An ontology for failure interpretation in automated planning and execution," in *Iberian Robotics conference*. Springer, 2019, pp. 381–390.

[11] J. Wu, R. Antonova, A. Kan, M. Lepert, A. Zeng, S. Song, J. Bohg, S. Rusinkiewicz, and T. Funkhouser, "Tidybot: Personalized robot assistance with large language models," *Autonomous Robots*, 2023.

[12] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[13] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022.

[14] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[15] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=vAElhFcKW6

[16] A. Zhao, D. Huang, Q. Xu, M. Lin, Y.-J. Liu, and G. Huang, "Expel: Llm agents are experiential learners," in *arXiv/2308.10144*, 2023.

[17] W. Li, D. Qiao, B. Wang, X. Wang, B. Jin, and H. Zha, "Semantically aligned task decomposition in multi-agent reinforcement learning," *arXiv preprint arXiv:2305.10865*, 2023.

[18] W. Yao, S. Heinecke, J. C. Niebles, Z. Liu, Y. Feng, L. Xue, R. R. N, Z. Chen, J. Zhang, D. Arpit, R. Xu, P. L. Mui, H. Wang, C. Xiong, and S. Savarese, "Retroformer: Retrospective large language agents with policy gradient optimization," in *The Twelfth International Conference on Learning Representations*, 2024.

[19] A. Z. Ren, A. Dixit, A. Bodrova, S. Singh, S. Tu, N. Brown, P. Xu, L. Takayama, F. Xia, J. Varley, *et al.*, "Robots that ask for help: Uncertainty alignment for large language model planners," in *7th Annual Conference on Robot Learning*, 2023.

[20] D. Das and S. Chernova, "Semantic-based explainable ai: Leveraging semantic scene graphs and pairwise ranking to explain robot failures," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 3034–3041.

[21] Y. Jin, D. Li, A. Yong, J. Shi, P. Hao, F. Sun, J. Zhang, and B. Fang, "Robotgpt: Robot manipulation learning from chatgpt," *IEEE Robotics and Automation Letters*, 2024.

[22] S. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor, "Chatgpt for robotics: Design principles and model abilities," *arXiv preprint arXiv:2306.17582*, 2023.

[23] Z. Liu, A. Bahety, and S. Song, "REFLECT: Summarizing robot experiences for failure explanation and correction," in *7th Annual Conference on Robot Learning*, 2023.

[24] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee, "Rdfox: A highly-scalable rdf store," in *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II 14*. Springer, 2015, pp. 3–20.

[25] "SWI-prolog," https://www.swi-prolog.org, version: 8.3.3.

[26] E. Kolve, R. Mottaghi, W. Han, E. VanderBilt, L. Weihs, A. Herrasti, D. Gordon, Y. Zhu, A. Gupta, and A. Farhadi, "AI2-THOR: An Interactive 3D Environment for Visual AI," *arXiv*, 2017.

[27] B. Knyazev, H. de Vries, C. Cangea, G. W. Taylor, A. C. Courville, and E. Belilovsky, "Graph density-aware losses for novel compositions in scene graph generation," in *BMVC*, 2020.

[28] A. Guzhov, F. Raue, J. Hees, and A. Dengel, "Audioclip: Extending clip to image, text and audio," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 976–980.

[29] J.-B. Lamy, "Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies," *Artificial intelligence in medicine*, vol. 80, pp. 11–28, 2017.