Using Large Language Models to Generate and Apply Contingency Handling Procedures in Collaborative Assembly Applications

Jeon Ho Kang[1], Neel Dhanaraj[1], Siddhant Wadaskar[1], and Satyandra K. Gupta[1]

***Abstract*— In manufacturing, minimizing operational delays is crucial for efficiency and resilience. Therefore, efficiently handling contingencies is essential in human-robot teams working on assembly (i.e., collaborative assembly) applications. This paper introduces a novel approach to generating contingency handling procedures by leveraging recent advances in Large Language Models (LMMs). Our approach uses LLMs to update the required tasks in hierarchical task networks (HTNs) to handle contingencies. The results demonstrate that our approach can handle various contingencies in assembly applications and minimize the impact on the assembly completion time.**

I. Introduction

There is a significant interest in automating assembly operations to improve human productivity and reduce the need for humans to perform ergonomically challenging tasks. Many assembly operations are complex and, therefore, cannot be completely automated. Humans and robots have complementary strengths. One solution to improve automation on assembly tasks is to deploy human and robot teams. Humans can work on tasks that require a high level of dexterity, and robots can perform routine tasks.

Recently, several advances have been made in robotic manipulators that make them safer for humans. These advances are enabling the development of collaborative work cells, where humans and robots can work in close proximity. Such collaborative cells are gaining popularity for assembly applications.

Hierarchical Task Networks (HTN) can be used to represent complex plans. Mixed Integer Linear Programming (MILPs) can be used to generate the optimal task sequencing and allocations for multi-agent teams using the information available in HTNs. In mass-production applications, robots are currently utilized for simple assembly tasks. Extensive testing and custom fixtures are used to reduce the uncertainty to the maximum possible extent and eliminate the possibility of task execution failures. Unfortunately, these strategies cannot be utilized in high-mix applications where assemblies change frequently. Contingency situations created by task execution failures occur more frequently in high-mix manufacturing because of the high variability in the tasks carried out in such applications. Figure 1 shows an example of a contingency where a screw-driving operation has failed and will require an intervention.

Addressing contingencies requires adjusting HTNs by altering tasks and changing available resources. Once these modifications are done, a new task allocation and schedule

[1]Viterbi School of Engineering, University of Southern California, CA USA {jeonhoka, dhanaraj, wadaskar, guptask}@usc.edu

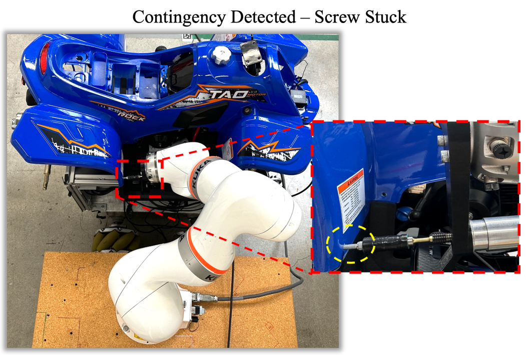


Fig. 1: Example of an ATV assembly cell. The image depicts a contingency scenario where the screwing operation fails due to a jammed screw.

can be generated. Only relying on humans to identify and handle tasks and adjusting HTNs may lead to long delays in the makespan for the tasks for the following reasons. Firstly, human experts are not always available for intervention within the cell. Having a human constantly idle, waiting for contingencies, is an inefficient use of resources. Furthermore, expecting every available human to possess the expertise to modify the HTN may not be realistic. Hence, it is important to reduce the amount of expertise needed for humans and automate handling contingencies that are commonly encountered in manufacturing so that only a limited number of contingencies will require human intervention.

In this paper, we focus on both efficiency and autonomy when addressing contingencies within multi-agent manufacturing assembly cells. Our key contributions are threefold: 1) introduce a formulation of contingency handling procedures that refines tasks and update the HTN to recover from contingencies, 2) use a Large Language Model (LLM) to generate procedures for handling contingencies that are similar to contingencies seen before, and 3) use LLM to convert human expert directives into contingency handling procedures.

II. Related Works

Contingency-Aware Task Planning Extensive literature exists on optimal task planning for multi-robot cells [1]–[4]. These approaches generally solve the task planning problem using methods like heuristic search [5]–[9], hierarchical task planning [10]–[12], mixed integer programs [13]–[17], genetic algorithms [18], and auction-based methods [19], [20] to find optimal task allocations. Recently, mixed integer linear program formulations have been used for manufacturing domains due to the availability of powerful open-source solvers [15], motivating our work to take advantage of these new methods. New advances in machine learning techniques further speed up these approaches, making them strong candidates for optimizing assembly makespan [21]. However, the aforementioned methods require significant engineering and implementation of domain knowledge.

Uncertainty in the robots, tasks, and environment [22], [23], especially in high-mix, low-volume manufacturing applications [24], [25], can lead to failures/contingencies. If these contingencies can be modeled, then they can be proactively managed [26]–[30]. This area of research is being investigated by the multi-agent task allocation [31], stochastic programming [32], and multi-agent reinforcement community [33]. Contingencies that cannot be modeled and predicted require reactive contingency management or failure recovery. This is a new field of research in both low-level task execution correction and high-level plan repair [34], [35]. High-level plan repair is particularly difficult to address due to the need for the robot to understand and reason over how to address the failure. This has spurred research into developing an ontology for failure representation in manufacturing environments and utilizing semantic information behind robot task failures [36], [37].

LLM Based Planning Recent advancements in LLM have fostered a significant interest in the robotics community [38], [39]. Its application for solving planning problems has proven to be useful. However, one of the major challenges is prompting the LLM to yield the desired task plan, a topic that has garnered significant attention across various domains [40]–[46]. A key focus area has been the reasoning and abstraction of high-level tasks into sub-tasks. [47]–[54].

Using LLM in isolation presents difficulties when directing the generated tasks to agents. Despite refined prompting techniques, the generated tasks may be infeasible. This challenge has led to investigations into the significance of environment feedback and awareness for failure-aware planning, as highlighted by [52]. As a result, numerous researchers have integrated LLM with diverse approaches. One notable method involves language-conditioned reinforcement learning, which aims to bifurcate skill learning from language grounding through an intermediary semantic representation, as discussed by [55].

Additionally, with success in generating code with LLMs [56], many works explore utilizing LLM for generating code tailored for agent task execution. For instance, [52], [54] employs APIs to yield a series of actions for routine tasks within a virtual home environment. Close to our work is [57], which harnesses the hierarchical tasks to decompose tasks into multiple steps. However, manufacturing scenarios necessitate optimal task sequencing. Directly generating alterations to these task sequences via LLM does not guarantee an optimal outcome. Therefore, we employ LLM to modify the task representation within the HTN, and subsequently, an optimizer determines the optimal task schedule.

III. Problem Formulation

Background: We use an HTN representation that accommodates sequential, independent, and parallel task relationships as formulated in [14]. The task network is instantiated with the root node τ_{root} representing the overall task set. The leaf nodes τ_{atomic} are the atomic tasks that need to be executed in the manufacturing task. Between τ_{root} and τ_{atomic} are subtasks, τ_{sub}, which serve to describe a relationship constraint among child nodes. $\tau_{sub\_sequential}$ dictates that task child nodes are executed sequentially from left to right. This mirrors the ordering constraints commonly found in assembly processes. $\tau_{sub\_independent}$ allows for individual execution of tasks without any specific order. Lastly, $\tau_{sub\_parallel}$ allows agents to perform tasks concurrently.

Furthermore, we define a resource model describing the state of both the parts and agents within the cell. Each task has agent and/or part requirements. These resources are characterized by specific states, such as when an agent is unavailable or a part is missing. We define contingency as any alteration in resource model states and possible action-level disruptions like motion execution contingencies.

Let $\mathcal{H}$ denote a Hierarchical Task Network and $\mathcal{R}$ represent the associated resource model. We define a task planner, $\mathcal{O}(\mathcal{H},\mathcal{R})$ which generates a sequence S of task agent allocations $\mathcal{A}_j \leftrightarrow \tau_i$. This assignment links a task $\tau_i \in \mathcal{H}$ to an agent $\mathcal{A}_i \in \mathcal{R}$. The outcome of this assignment has an associated total duration to complete all tasks, or makespan t. The overarching goal is to construct a framework that addresses a variety of contingencies, $\mathcal{F}$, and generate a computationally interpretable contingency handling procedure, α.

Problem Statement: Consider a scenario where $\mathcal{F}$ requires modifications within a system. For example, $\mathcal{F}$ could result in a change of $\mathcal{H}$ or state for $\mathcal{R}$. These changes might arise from the altered availability of an agent, $\mathcal{A}_i$, a defect in a specific part, $\mathcal{P}$, or a failure in executing τ_{atomic}. To resolve these disruptions, we need a contingency handling procedure, α in order to return to normal operation. The new recovery task set $\tau_\alpha \in \alpha$ is designed to revise $\mathcal{H}$ according to the updated status of $\mathcal{R}$ and τ_i. When our system applies α to our initial $\mathcal{H}$, we get a new HTN, $\mathcal{H}'$, resource model, $\mathcal{R}'$, and an adjusted task set, $(\tau,\tau_\alpha) \in \tau$'. Then, $\mathcal{O}'(\mathcal{H}',\mathcal{R}')$ generates a new task sequence with an associated makespan t'.

We also consider t^{α}_{gen}, the time required to generate α, and t^{α}_{exec}, the time needed to incorporate α by adjusting $\mathcal{H}$ such that the new recovery tasks are attached to the best τ_{sub}. Therefore, the efficacy of α depends $t_\alpha = \sum duration(\tau_{\alpha_i})$ which directly impacts the new total makespan t'. The objective of this work is to narrow the gap between t' and t by generating a near-optimal α that balances efficiency and effectiveness. In addition, we need to generate a α that leads to minimizing both t^{α}_{exec} and t^{α}_{gen}.

Overview of Approach: We present a methodology that employs a contingency manager to modify and supplement tasks in the existing $\mathcal{H}$. This system, illustrated in Figure 2, begins with an executor that directs robots to carry out a predetermined task sequence, S.

During execution, disruptions from $\mathcal{F}$ can arise. If detected by the manufacturing cell monitor, the contingency process is initiated. We initially attempt to use LLM for new α generation by using existing contingency handling procedures as prompts, with the specifics detailed in Section V. A human expert then reviews these edits for feasibility. Once

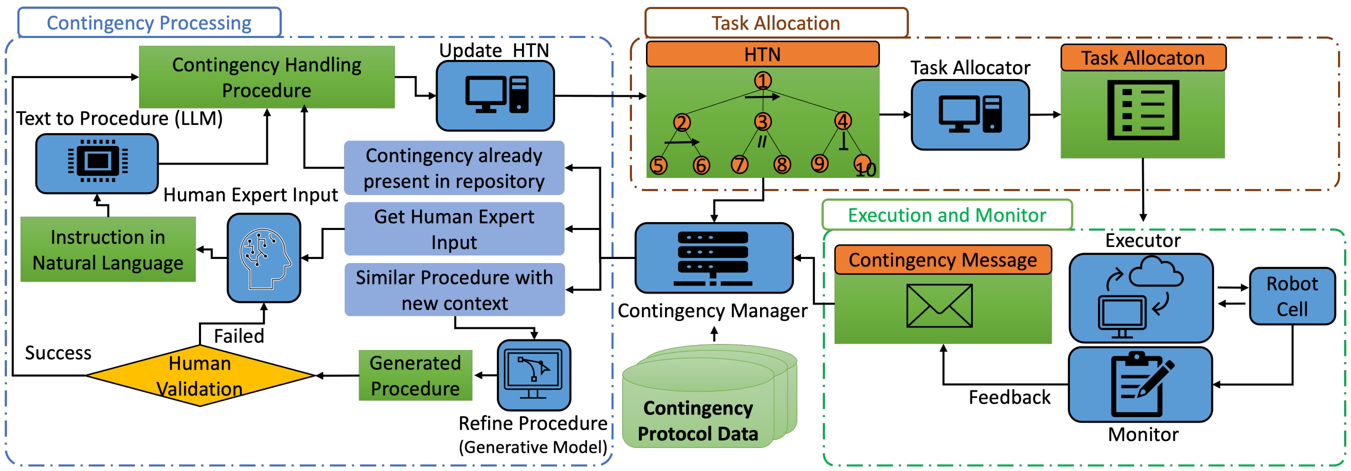


Fig. 2: Overview of the system architecture

validated, the edits are assimilated into the system and saved in a handling procedure repository. The repository enhances the predetermined handling procedure and the example set, reducing future human interventions.

There are instances where the code-to-code LLM's suggested α might be infeasible. In these cases, we lean on natural language instructions from human experts. The language-to-code LLM then translates these instructions into a computational α, avoiding the need for further programming. Details of the system can be found in Section VI. Also, integration of α into $\mathcal{H}$ using established procedures is found in Section IV. After the integration, S' is computed by the task scheduler, aiming to minimize makespan, preserving optimally while incorporating τ_i, with τ_α.

IV. Reformulating HTN and Incorporating Tasks to Recover from Contingencies

We start with essential primitive to edit $\mathcal{H}$ and give the LLM and humans a way to formulate α. We use these class functions to enable the code-to-code LLM to define α as discussed in Section V, as well as to make a fair comparison of t^{α}_{gen} between humans and LLMs. Let us assume we have an optimal contingency handling procedure, α. How our system generates α will be discussed in Section V and VI. To incorporate α while minimizing t^{α}_{exec}, we need steps to restructure $\mathcal{H}$ so that α can be integrated with other existing tasks. Our method identifies the least number of the required disassembly tasks across multiple handling procedures. Furthermore, our method determines α's optimal placement in $\mathcal{H}$ and identifies the best sequence of recovery tasks. Further explanations are provided in Section IV-B.

A. *Primitives to Define Recovery Tasks*

In the AlterTree class, we define the following three primitives to construct the expression α: *add_node(task_parent, task_child, Agent)*, *add_agent_model(Task, Agent, Duration)*, *set_agent_state(Agent, Availability)*. The first two primitives enable users to add task nodes and associated agent-task duration cost models to the task nodes. Additionally, we have a function to set the agent's operational state as available

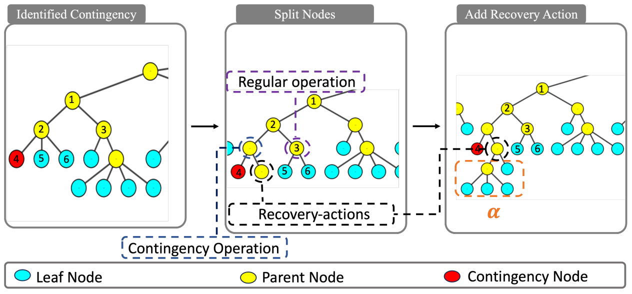


Fig. 3: Step by step strategy for incorporating α into HTN.

or unavailable. Upon setting the agent state unavailable, the program sets the tasks and their sequential dependencies linked to inactive agents as infeasible. Subsequently, when the solver generates a schedule, it will disregard those tasks, allowing functioning agents to resume other tasks while the assigned agents handle contingency. This function is necessary because, in manufacturing assembly, it is common for agents to become non-functional. Such situations can cause an assembly cell to come to a standstill, leading other functioning agents to be sidelined, extending t. For example, suppose a robot specialized in screw-driving is currently under maintenance. In that case, other agents can still perform tasks like picking up and inserting other wheels until the screw-driving robot is operational.

B. *Incorporation of Recovery Tasks in HTN*

Figure 3 illustrates the procedures for implementing these steps. Before incorporating the α, the contingency manager first splits the parent of the contingency task $\tau_{\mathcal{F}}$ into 1) the contingency operation, which includes the failed task node and the corresponding recovery tasks and 2) the regular operation which are the remaining tasks in τ_{sub}. It then appends the failed task and its associated α to the created τ_{sub} so that we allow the scheduler to incorporate recovery tasks and contingency tasks.

Additionally, addressing a contingency may necessitate a disassembly process. Therefore, the contingency manager undertakes the following steps to generate the necessary disassembly procedures for the malfunctioning component.

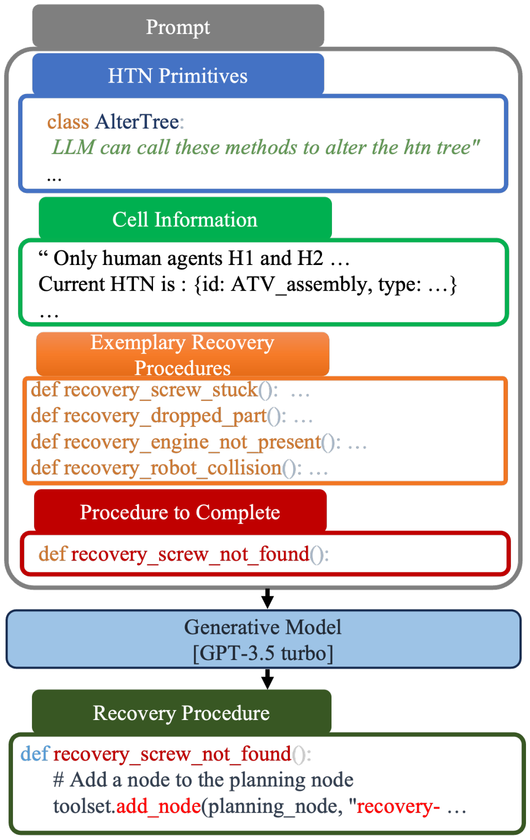
1) Check for tasks that have been completed. 2) Determine which completed tasks are sequentially bound within the HTN. 3) Invert the operation sequence to identify the counterpart disassembly process.4) Add these disassembly processes to the disassembly task in α. 5) Set these operations incomplete to be redone after contingency is resolved so that it is rescheduled in the new plan.

When multiple $\mathcal{F}_i$ occurs, and two or more contingencies require a similar disassembly process, generating an efficient contingency handling procedure becomes essential. Consider a case in ATV assembly where an engine leak and a crack in the outer main body frame coincide during assembly. A naive approach might generate and schedule the two α sequentially. This approach may lead the system first to replace the outer frame and disassemble the engine, leading to more wasted time. In contrast, we design our system to prioritize the engine recovery task by following the steps: 1) Determine if there are redundant tasks in the set of contingency handling procedures. 2) Examine the task sequence by first analyzing which tasks are executed in the original $\mathcal{H}$. 3)Eliminate the disassembly tasks associated with later task assembly steps.

V. Utilizing Existing Contingency Handling Procedures to Generate New Contingency Handling Procedures

To reduce reliance on human experts, we generate α using a code-to-code LLM (i.e., LLMs that can generate new code based on the existing code) by providing the context to formulate effective α. We employ the GPT 3.5 turbo as our backbone architecture to generate the contingency handling procedure α. We configure the GPT with the code for three HTN primitives under the Pythonic class AlterTree, as discussed in Section IV-A. We annotate each primitive with comments describing its functionality. We further supplement GPT with the current HTN's tree data structure, failed task and agent, details about the agents in the cell, and inherent task constraints in the text. For instance, constraints like ”only human agents H1 and H2 and mobile platform agents m1 and m2 can move” ensure that the handling procedure excludes infeasible task assignments. We also feed the model exemplary handling procedure set $\{\alpha_1, \alpha_2, \alpha_3 ...\}$. The system architecture is given in Figure 4. The final input to GPT is a contingency definition such as ”recovery_screw_not_found” structured as an empty function, prompting it to construct the corresponding α for the specified $\mathcal{F}$.

Figure 5 shows the example output for this approach. In the example α, the LLM uses the HTN primitives by referencing the comments and the functions within the AlterTree class. This approach ensures that GPT applies the correct syntax and function arguments to the new α. It further references the specific information about the cell, allowing it to assess the capabilities of agents and evaluate the current status of the $\mathcal{H}$. Moreover, it draws from the past scenarios in the exemplary α, which informs the appropriate recovery tasks for different contingencies. When presented with an empty function, the model adeptly fills it in. An example of this is the 'screw not found' scenario. In this



```
class AlterTree:
 LLM can call these methods to alter the htn tree"
...
```

```
" Only human agents H1 and H2 …
Current HTN is : {id: ATV_assembly, type: …}
…
```

```
def recovery_screw_stuck():  …
def recovery_dropped_part(): …
def recovery_engine_not_present(): …
def recovery_robot_collision(): …
```

```
def recovery_screw_not_found():
```

```
def recovery_screw_not_found():
    # Add a node to the planning node
    toolset.add_node(planning_node, "recovery- …
```

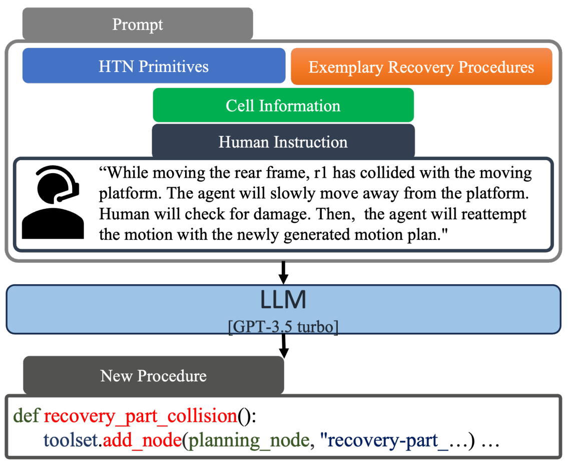
Fig. 4: The system diagram provides a detailed view of the prompting technique to generate a contingency handling procedure.

instance, the model modifies ’recovery_engine_not_found’ to craft ’recovery_screw_not_found,’ highlighting that both tasks revolve around notifying the cell's monitor to source replacements for absent components.

Importantly, beyond the provided prompt, GPT leverages its broad world knowledge and manufacturing data from its pre-trained data set to devise tasks tailored for managing particular contingencies. As showcased in Figure 5, there is a nuanced distinction in its approach between the engine and screw contingencies. While the contingency handling procedure provided in the example for the engine missing scenario directly notifies the absence, the screw missing situation triggers a more intricate process. Here, GPT introduces two additional tasks to search for the screw and scan the workspace for any missing screws. This capability emerges from GPT's understanding of screw-related textual information and its inherent physical properties. Such knowledge equips the model to grasp potential issues stemming from a missing screw in the workspace. Drawing from real-world contexts, an engine, being a more conspicuous component, might not necessitate a workspace search when missing. Usually, the appropriate response involves requesting the part and then waiting for its arrival. Conversely, for a missing screw, the model instructs the agent to scan the workspace to determine if a stock of screws is available to refill the screw feeder. Furthermore, GPT exhibits a capability for organizing tasks. It automatically groups two atomic tasks of the same parent task under a shared parent task.

```
def recovery_screw_not_found():
    # Add a node to the planning node
    toolset.add_node(planning_node, "recovery-screw_not_found", "sequential")
    toolset.add_node("recovery-screw_not_found", "recovery-search_screw",
                     "sequential")
   toolset.add_node("recovery-search_screw", "recovery-scan_workspace",
                     "atomic", agent='H', duration=6)
   toolset.add_node("recovery-search_screw", "recovery-notify_monitor",
                    "atomic", agent='H',  duration=6)
   toolset.add_node("recovery-screw_not_found", "recovery
                    -pick_new_screw", "atomic", agent='r1', duration=8)
```

Fig. 5: Example generation of contingency handling procedure prompted in Figure 4



"While moving the rear frame, r1 has collided with the moving platform. The agent will slowly move away from the platform. Human will check for damage. Then, the agent will reattempt the motion with the newly generated motion plan."

```
def recovery_part_collision():
    toolset.add_node(planning_node, "recovery-part_...) ...
```

Fig. 6: For this module, human instruction in natural language is added to convert it to program-level modification to HTN.

VI. Translating Human Expert Instructions into Contingency Handling Procedures

When the method described in Section V fails to generate α, we seek human help. The system architecture of our implementation is depicted in Figure 6. As done in Section V, we supply the GPT 3.5 turbo with a similar prompt. The distinguishing factor is the final prompt: humans describe the contingency handling procedure in natural language, which the language-to-code LLM then translates into the executable α.

Figure 7 presents a contingency handling procedure generated using the discussed method. As illustrated in Section V, GPT derives insights from the primitives, cell information, and example α. However, there is a notable difference in its method for identifying node names. It leverages human-provided natural language instructions to reconstruct α in code format. A clear demonstration of this is the comparison between task id, "recovery_move_away", and the directive "r1 will slowly move away from the platform.".

A closer examination of Figure 7 reveals GPT's ability to make inferences. It determines which agent undertakes which tasks, even without explicit indications. For instance, while the natural language refers to a 'human' agent, GPT translates it, adhering to the correct syntax, and uses the identifier "H." As illustrated, all designated agents are correctly assigned to execute the recovery tasks, especially those contending with the contingency, r1. Additionally, in the absence of specific data on the duration of recovery tasks,

```
def recovery_part_collision():
    toolset.add_node(planning_node, "recovery-part_collision", "sequential")
    toolset.add_node("recovery-part_collision", "recovery_move_away",
                     "atomic", agent="r1", duration=10)
    toolset.add_node("recovery-part_collision", "recovery-check_damage",
                     "atomic", agent="H", duration=6)
    toolset.add_node("recovery-part_collision", "recovery-reattempt_motion",
                     "atomic", agent="r1", duration=12)
```

Fig. 7: Example conversion from natural language prompt in Figure 6 to code

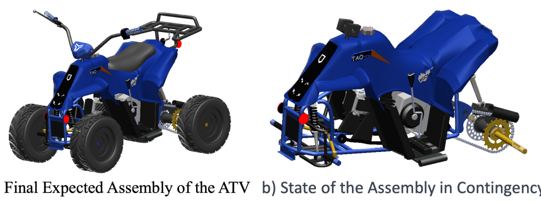

a) Final Expected Assembly of the ATV
b) State of the Assembly in Contingency

Fig. 8: State of ATV assembly composed of 17 parts. Contingency is detected in the 6th part of the assembly.

GPT references the HTN and example α to determine and fill in the probable time steps needed.

VII. Results

A. *Testbed*

We assess our method in the All-Terrain-Vehicle (ATV) assembly cell producing one ATV. The t for initial task allocation without contingency is 800 seconds. There are 17 assembly components in the overall cell. The cell comprises 12 stations, each tasked with affixing different parts to the main chassis and one contingency cell for disassembly. The contingency cell aims to dismantle essential components within the assembly, allowing access to the faulty part.

Our HTN includes 24 agents, comprising eight humans, 15 robots, and a single mobile platform. Within this network, there are 148 task nodes and 94 leaf nodes. We classify contingencies into five distinct groups: agent-related, task-specific, motion planning, part-specific, and miscellaneous. For each category, we present ten scenarios, serving as test cases to evaluate the efficacy of our proposed approach.

B. *Large Language Model for Adaptive Contingency Handling Procedures*

We compare our method with a benchmark that directly generates tree data structure in Python dictionary format. During our experiment, we maintain consistency between the two prompting techniques, ensuring they share the same context regarding the cell and the HTN. We initially provide the same set of exemplary α for nine identical contingencies but presented in two distinct formats.

Figure 9 shows differences in success rates between the two methods. As detailed in Section VII-A, we produce α for ten distinct contingencies in each category. Our evaluation metric defines success based on the accuracy and executability of the generated α. Any procedure that cannot be utilized or executed is deemed unsuccessful. We then calculate the success rate as a percentage.

From the comparison, we identified that generating raw tree data structure is more prone to making syntactical errors because the data structure is structurally more complex with various parenthesis and spaces, which often leads to a more complicated interpretation of the context of the assembly cell, thereby reducing the likelihood of producing executable α.

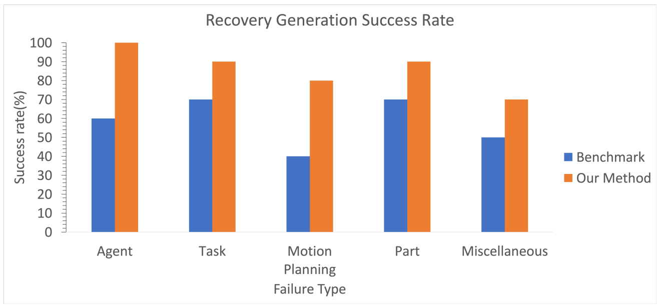


Fig. 9: Figure illustrates a comparison of success rates in generating handling procedures between two methods. It provides a summary of their relative efficacy, highlighting differences in their ability to create actionable and accurate recovery tasks.

C. *Contingency Handling Procedure Generation Time Comparison*

We benchmark our approach against the traditional method where α is formulated through manual programming by humans. We assume that human starts from the skeleton code and tries to minimize t^{α}_{gen}. If the code-to-code LLM generates an infeasible task, we introduce humans to provide natural language instruction to devise α as discussed in Section VI and add it to the delay introduced.

Table I shows the comparative analysis of t^{α}_{gen} between LLM generated and the benchmark. Our method had a distinct advantage in addressing the first four types of contingencies. There was a minimal difference in miscellaneous contingencies because LLM, in this context faced challenges in generating an accurate α on its initial attempt.

This observation aligns with our findings from Section VII-B. Miscellaneous contingencies generally had the lowest success rates due to their inherent uncertainties and the wide range of contexts. However, our method still showed some advantages even for those contingencies that required human intervention because, when the LLM relied on expert natural language instructions, it was generally more accessible for experts to articulate contingency handling tasks in natural language. Humans typically took longer in programming handling procedures, resulting in comparable t^{α}_{gen}.

Introduced t^{α}_{gen} compared to t		
Category	Human generated	LLM Generated
Agent	$+0.10t \pm 0.03t$	$+0.01t \pm 0.01t$
Task	$+0.10t \pm 0.03t$	$+0.02t \pm 0.03t$
Motion	$+0.12t \pm 0.03t$	$+0.03t \pm 0.04t$
Part	$+0.12t \pm 0.03t$	$+0.03t \pm 0.03t$
Miscellaneous	$+0.08t \pm 0.02t$	$+0.05t \pm 0.04t$

TABLE I: Our analysis shows variability in t^{α}_{gen}, which stems from the differences between simple and complex contingencies. This observation holds regardless of the method.

D. *Makespan Improvement Analysis*

Figure 10 compares t' between human-generated α and our method. The new makespan comprises t^{α}_{gen} and the execution

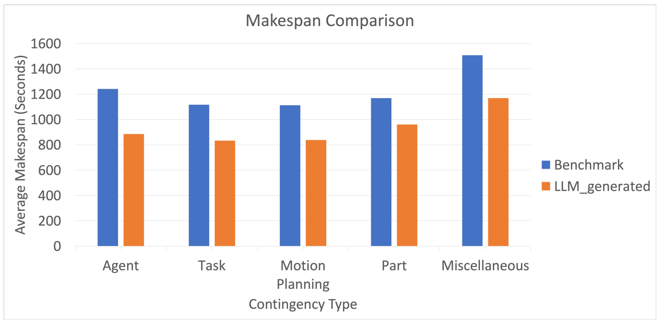


Fig. 10: The figure illustrates the disparity in makespan between the conventional human-programmed approach and the LLM method. A lognormal distribution is employed to account for potential delays—up to a maximum of 10 minutes—owing to the availability of human programmers.

time of the recovery tasks. Consistent with our earlier results, the miscellaneous contingency category exhibits a minimal difference in makespan between the two approaches and heightened makespan because of the considerable delay time in t^{α}_{gen}, coupled with idle time stemming from unanticipated circumstances. Furthermore, we incorporate delay models to account for the time required for programmers to be available, assuming a maximum delay of 10 minutes.

The makespan differences are narrower for part contingencies, mainly due to the consistent waiting time for replacement parts, irrespective of the method used. However, our method significantly reduces the makespan for other types of contingencies because our handling procedures often sidestep the need for human intervention, leveraging agent-level actions like ”retreat” and ”re-plan.”

E. *Case Study for Complex Contingency Recovery*

Figure 8 b) depicts a state of ATV during contingency within an ATV assembly. As the upper cover assembly progresses, an agent detects a defective engine and notifies the cell monitor. The contingency handling procedure generated by the LLM described in Section V includes 11 nodes: three parent nodes, five undo tasks, and three re-do tasks. Utilizing the generated α, the total time to complete the assembly amounted to 915 seconds, resulting in a 14% increase from the nominal makespan. If we were to wait for a human to modify the HTN, then this time would be a lot longer.

VIII. Conclusions

We have demonstrated that contingency handling procedures can be automatically generated using LLMs, drawing information from procedures for similar contingencies. LLMs also efficiently translate human instructions in natural language into contingency handling procedures. LLMs encapsulate a common understanding of assembly parts and tasks, enabling them to fill in missing information and establish sequencing constraints. Upon formulating the contingency procedure, the LLM modifies the task representation. This modification in representation space ensures that when the task scheduler sequences tasks, it addresses the issue and maintains sequence optimality. As a future direction, we aim to delve into prompt efficiency, exploring the minimal prompts necessary for LLMs to yield comparable results.

References

[1] H. L. Kwa, J. Leong Kit, and R. Bouffanais, "Balancing collective exploration and exploitation in multi-agent and multi-robot systems: A review," *Frontiers in Robotics and AI*, vol. 8, p. 771520, 2022.

[2] G. A. Korsah, A. Stentz, and M. B. Dias, "A comprehensive taxonomy for multi-robot task allocation," *The International Journal of Robotics Research*, vol. 32, no. 12, pp. 1495–1512, 2013.

[3] A. Hentout, M. Aouache, A. Maoudj, and I. Akli, "Human–robot interaction in industrial collaborative robotics: a literature review of the decade 2008–2017," *Advanced Robotics*, vol. 33, no. 15-16, pp. 764–799, 2019.

[4] H. Chakraa, F. Guérin, E. Leclercq, and D. Lefebvre, "Optimization techniques for multi-robot task allocation problems: Review on the state-of-the-art," *Robotics and Autonomous Systems*, p. 104492, 2023.

[5] J. Hoffmann, "Ff: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, pp. 57–57, 2001.

[6] D. Bryce and S. Kambhampati, "A tutorial on planning graph based reachability heuristics," *AI Magazine*, vol. 28, no. 1, pp. 47–47, 2007.

[7] Z. Li, A. V. Barenji, J. Jiang, R. Y. Zhong, and G. Xu, "A mechanism for scheduling multi robot intelligent warehouse system face with dynamic demand," *Journal of Intelligent Manufacturing*, vol. 31, pp. 469–480, 2020.

[8] V. Tereshchuk, N. Bykov, S. Pedigo, S. Devasia, and A. G. Banerjee, "A scheduling method for multi-robot assembly of aircraft structures with soft task precedence constraints," *Robotics and Computer-Integrated Manufacturing*, vol. 71, p. 102154, 2021.

[9] Y. Zhang and L. E. Parker, "Multi-robot task scheduling," in *2013 IEEE international conference on robotics and automation*. IEEE, 2013, pp. 2992–2998.

[10] A. R. Mosteo and L. Montano, "Simulated annealing for multi-robot hierarchical task allocation with flexible constraints and objective functions," in *Workshop on network robot systems: toward intelligent robotic systems integrated with environments. int. conf. on intelligent robots and systems*. Citeseer, 2006.

[11] S. Bae, S. Joo, J. Choi, J. Pyo, H. Park, and T. Kuc, "Semantic knowledge-based hierarchical planning approach for multi-robot systems," *Electronics*, vol. 12, no. 9, p. 2131, 2023.

[12] J. Munoz-Morera, F. Alarcon, I. Maza, and A. Ollero, "Combining a hierarchical task network planner with a constraint satisfaction solver for assembly operations involving routing problems in a multi-robot context," *International Journal of Advanced Robotic Systems*, vol. 15, no. 3, p. 1729881418782088, 2018.

[13] M. Gombolay, R. Wilcox, and J. Shah, "Fast scheduling of multi-robot teams with temporospatial constraints," in *Robotics: Science and Systems Foundation*, 2013.

[14] Y. Cheng, L. Sun, and M. Tomizuka, "Human-aware robot task planning based on a hierarchical task model," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1136–1143, 2021.

[15] A. Ham and M.-J. Park, "Human–robot task allocation and scheduling: Boeing 777 case study," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1256–1263, 2021.

[16] S. Fatemi-Anaraki, R. Tavakkoli-Moghaddam, M. Foumani, and B. Vahedi-Nouri, "Scheduling of multi-robot job shop systems in dynamic environments: mixed-integer linear programming and constraint programming approaches," *Omega*, vol. 115, p. 102770, 2023.

[17] D. Guo, "Fast scheduling of human-robot teams collaboration on synchronised production-logistics tasks in aircraft assembly," *Robotics and Computer-Integrated Manufacturing*, vol. 85, p. 102620, 2024.

[18] C. Liu and A. Kroll, "A centralized multi-robot task allocation for industrial plant inspection by using a* and genetic algorithms," in *Artificial Intelligence and Soft Computing: 11th International Conference, ICAISC 2012, Zakopane, Poland, April 29-May 3, 2012, Proceedings, Part II 11*. Springer, 2012, pp. 466–474.

[19] M. Badreldin, A. Hussein, and A. Khamis, "A comparative study between optimization and market-based approaches to multi-robot task allocation," *Advances in Artificial Intelligence*, vol. 2013, pp. 12–12, 2013.

[20] J. G. Martin, J. R. D. Frejo, R. A. García, and E. F. Camacho, "Multi-robot task allocation problem with multiple nonlinear criteria using branch and bound and genetic algorithms," *Intelligent Service Robotics*, vol. 14, no. 5, pp. 707–727, 2021.

[21] H. Aziz, A. Pal, A. Pourmiri, F. Ramezani, and B. Sims, "Task allocation using a team of robots," *Current Robotics Reports*, vol. 3, no. 4, pp. 227–238, 2022.

[22] M. J. Matarić, G. S. Sukhatme, and E. H. Østergaard, "Multi-robot task allocation in uncertain environments," *Autonomous Robots*, vol. 14, pp. 255–263, 2003.

[23] L. Zhou and P. Tokekar, "Multi-robot coordination and planning in uncertain and adversarial environments," *Current Robotics Reports*, vol. 2, pp. 147–157, 2021.

[24] N. Dhanaraj, R. Malhan, H. Nemlekar, S. Nikolaidis, and S. K. Gupta, "Human-guided goal assignment to effectively manage workload for a smart robotic assistant," in *2022 31st IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. IEEE, 2022, pp. 1305–1312.

[25] J. Falco, K. Van Wyk, and K. Kimble, "Advances in robot technology supporting low-volume/high-mix small part assembly operations," 2022.

[26] N. Dhanaraj, S. V. Narayan, S. Nikolaidis, and S. K. Gupta, "Contingency-aware task assignment and scheduling for human-robot teams," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 5765–5771.

[27] S. Shriyam and S. K. Gupta, "Incorporation of contingency tasks in task allocation for multirobot teams," *IEEE Transactions on Automation Science and Engineering*, vol. 17, no. 2, pp. 809–822, 2020.

[28] S. Shriyam, "Contingency handling in mission planning for multi-robot teams," Ph.D. dissertation, University of Southern California, 2019.

[29] S. Shriyam and S. K. Gupta, "Incorporating potential contingency tasks in multi-robot mission planning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 3709–3715.

[30] ——, "Task assignment and scheduling for mobile robot teams," in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 51807. American Society of Mechanical Engineers, 2018, p. V05AT07A075.

[31] S. Choudhury, J. K. Gupta, M. J. Kochenderfer, D. Sadigh, and J. Bohg, "Dynamic multi-robot task allocation under uncertainty and temporal constraints," *Autonomous Robots*, vol. 46, no. 1, pp. 231–247, 2022.

[32] A. Messing, J. Banfi, M. Stadler, E. Stump, H. Ravichandar, N. Roy, and S. Hutchinson, "A sampling-based approach for heterogeneous coalition scheduling with temporal uncertainty."

[33] A. Wong, T. Bäck, A. V. Kononova, and A. Plaat, "Deep multiagent reinforcement learning: Challenges and directions," *Artificial Intelligence Review*, vol. 56, no. 6, pp. 5023–5056, 2023.

[34] S. Reig, E. J. Carter, T. Fong, J. Forlizzi, and A. Steinfeld, "Flailing, hailing, prevailing: Perceptions of multi-robot failure recovery strategies," in *Proceedings of the 2021 ACM/IEEE International Conference on Human-Robot Interaction*, 2021, pp. 158–167.

[35] D. Das, S. Banerjee, and S. Chernova, "Explainable ai for robot failures: Generating explanations that improve user assistance in fault recovery," in *Proceedings of the 2021 ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 351–360. [Online]. Available: https://doi.org/10.1145/3434073.3444657

[36] K. Skarzynski, M. Stepniak, W. Bartyna, and S. Ambroszkiewicz, "A generic ontology and recovery protocols for human–robot collaboration systems," in *Communication and Intelligent Systems: Proceedings of ICCIS 2021*. Springer, 2022, pp. 973–987.

[37] S. Matsuoka and T. Sawaragi, "Recovery planning of industrial robots based on semantic information of failures and time-dependent utility," *Advanced Engineering Informatics*, vol. 51, p. 101507, 2022.

[38] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, 2021.

[39] S. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor, "Chatgpt for robotics: Design principles and model abilities," *Microsoft Auton. Syst. Robot. Res*, vol. 2, p. 20, 2023.

[40] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[41] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[42] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[43] A. Patel, B. Li, M. S. Rasooli, N. Constant, C. Raffel, and C. Callison-Burch, “Bidirectional language models are also few-shot learners,” *arXiv preprint arXiv:2209.14500*, 2022.

[44] M. Skreta, N. Yoshikawa, S. Arellano-Rubach, Z. Ji, L. B. Kristensen, K. Darvish, A. Aspuru-Guzik, F. Shkurti, and A. Garg, “Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting,” *arXiv preprint arXiv:2303.14100*, 2023.

[45] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback, 2022,” *URL https://arxiv. org/abs/2203.02155*, vol. 13, 2022.

[46] L. Reynolds and K. McDonell, “Prompt programming for large language models: Beyond the few-shot paradigm,” in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

[47] 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.

[48] 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.

[49] P. A. Jansen, “Visually-grounded planning without vision: Language models infer detailed plans from high-level instructions,” *arXiv preprint arXiv:2009.14259*, 2020.

[50] S. Li, X. Puig, C. Paxton, Y. Du, C. Wang, L. Fan, T. Chen, D.-A. Huang, E. Akyürek, A. Anandkumar *et al.*, “Pre-trained language models for interactive decision-making,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 31 199–31 212, 2022.

[51] R. Patel and E. Pavlick, “Mapping language models to grounded conceptual spaces,” in *International Conference on Learning Representations*, 2021.

[52] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, “Progprompt: program generation for situated robot task planning using large language models,” *Autonomous Robots*, pp. 1–14, 2023.

[53] Y. Xie, C. Yu, T. Zhu, J. Bai, Z. Gong, and H. Soh, “Translating natural language to planning goals with large-language models,” *arXiv preprint arXiv:2302.05128*, 2023.

[54] A. Zeng, M. Attarian, B. Ichter, K. Choromanski, A. Wong, S. Welker, F. Tombari, A. Purohit, M. Ryoo, V. Sindhwani *et al.*, “Socratic models: Composing zero-shot multimodal reasoning with language,” *arXiv preprint arXiv:2204.00598*, 2022.

[55] A. Akakzia, C. Colas, P.-Y. Oudeyer, M. Chetouani, and O. Sigaud, “Grounding language to autonomously-acquired skills via goal generation,” *arXiv preprint arXiv:2006.07185*, 2020.

[56] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[57] Y. Cao and C. Lee, “Robot behavior-tree-based task generation with large language models,” *arXiv preprint arXiv:2302.12927*, 2023.