Leveraging LLMs for Generating Document-Informed Hierarchical Planning Models: A Proposal

Morgan Fine-Morris[1,2], **Vincent Hsiao**[1,2], **Leslie Smith**[2], **Laura Hiatt**[2], **Mark Roberts**[2]

[1]NRC Postdoc, [2]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
morgan.f.fine-morris.ctr@us.navy.mil, vincent.hsiao.ctr@us.navy.mil, leslie.n.smith20.civ@us.navy.mil
laura.m.hiatt.civ@us.navy.mil mark.c.roberts20.civ@us.navy.mil

Abstract

Hierarchical planning facilitates faster planning using domain-specific decomposition methods, which require considerable knowledge engineering; automated generation of methods is an open problem. We propose a domain-independent LLM (Large Language Model) pipeline that uses Natural Language background information to generate methods. We describe a potential evaluation scheme for testing its performance against a classical planning baseline as well as an initial case study.

1 Introduction

Hierarchical network planning, which includes hierarchical task network (HTN) planning, hierarchical goal network (HGN) planning, and hierarchical problem network (HPN) planning performs planning via a divide-and-conquer strategy. HTNs have been show to be strictly more expressive than classical planning and faster (Erol, Hendler, and Nau 1994a,b). Additionally, Alford et al. (2016) have shown that HTNs can be translated into HGNs, indicating that HGNs have similar advantages. These improvements derive from the *decomposition methods* planners use to search the solution space. Decomposition methods describe, depending on the paradigm, how and when to decompose a task, goal, or problem into a series of simpler subcomponents, possibly with ordering constraints between them. Engineering the planning model is a major barrier to applying hierarchical network planning. Automated generation of decomposition methods is an open problem, which has mostly been tackled by analytic (Li et al. 2024; Hayes and Scassellati 2016; Hérail and Bit-Monnot 2023; Yang, Pan, and Pan 2007; Segura-Muros, Pérez, and Fernández-Olivares 2017; Lotinac and Jonsson 2016; Hogg, Muñoz-Avila, and Kuter 2016; Zhuo, Muñoz-Avila, and Yang 2014; Langley 2024) or NLP (Nguyen et al. 2017; Gopalakrishnan, Muñoz-Avila, and Kuter 2018; Fine-Morris et al. 2022) techniques, applied to solved problem examples.

But what to do in the absence of a library of plan traces? Recently, LLMs have been applied to the problems of generating classical planning models from NL (Xie et al. 2023; Oswald et al. 2024; Oates et al. 2024), to translating from NL commands to formal languages such as STL and TLT (Chen et al. 2023; Cosler et al. 2023; Liu et al. 2022), to NL-based decomposition and task planning (Zhu et al. 2023; Yuan et al. 2024), and to code and formal-language repair based on error information (Chen et al. 2024). We present a document-informed pipeline that combines the decomposition, translation, and repair capabilities of LLMs to generate decomposition methods in an HDDL-like language from a provided NL goal, NL domain information, and PDDL action model. Because the initial decomposition is formulated in NL, the pipeline can be repeatedly run on the subgoals produced by previous stages to create a hierarchical decomposition planning model for decomposing intermediate goals. We describe a potential evaluation scheme for comparing the performance of the generated methods against a classical-planning baseline and an ablation study to analyze the impact of the domain knowledge on the generated decompositions. The evaluation will make use of a decomposition planner which can plan with incomplete models by falling back on classical planning in the absence of methods. Our performance metrics will look at planning time, solution quality, and proportion of planning accomplish by decomposition planning model vs. classical planning. This last metric will measure the completeness of the generated model, measuring the ability of the pipelines to form a complete planning model for solving an initially provided goal. We suggest an initial domain for a case study, and in our concluding section suggest additions to the pipeline and other domains with which to expand the evaluation.

2 Related Work

PDDL Generation: Several works have investigated using LLMs to generate classical planning models. Oswald et al. (2024) and Oates et al. (2024) both look at generating PDDL actions on an action-by-action basis from NL text. Oswald et al. use NL descriptions of the actions and force the LLM to use provided predicates and types. Oates et al. allows the model to create its own predicates. Smirnov et al. (2024) generate a domain and problem definition in one pass, unlike the previous works which prompt on an action-by-action basis. They use external checking and goal-reachability analysis to identify errors in the domain and problem definitions and an error-correction loop to prompt the LLM to correct the errors. Guan et al. (2024) generate actions and new predicates, prompting the LLM to revise previously-generated actions with the new predicates. COWP (Ding et al. 2023) augments user-provided classical planning knowledge with common sense knowledge from the LLM, introducing new preconditions and effects to the actions of the action model. Liu et al. (2023) create a planning system which converts NL domain and problem descriptions into PDDL, constructs a plan using a classical planner, and then coverts the solution back into NL.

Document-Informed Planning: SPRING (Wu et al. 2023) generates plans using an LLM, NL domain context-information, and a DAG of chained queries guiding the LLM's reasoning, where the final node requests an action. Our system also uses NL domain information, but generates decomposition methods instead of selecting actions.

Task Decomposition: DECKARD (Nottingham et al. 2023) decomposes tasks into subgoals via LLM, then learns a policy for accomplishing each subgoal via gameplay while also correcting its decomposition. Our pipeline will handle subgoals by decomposing them, with no policy learning. Yuan et al. (2024) investigates Structured Complex Task Decomposition, generating ordered NL task decomposition graphs via LLM. Our pipeline will encode its goals and subgoals in formal, instead of natural, language. The CaStL framework Guo et al. (2024) generates a PDDL problem description and python scripts for constructing constraint expressions to solve TAMP problems from an input NL goal, PDDL domain definition, and initial-state scene graph. The expected end-products of our pipeline are decomposition methods and we are using NL domain information to inform the generation of the decompositions.

Translation: NL2TL (Chen et al. 2023) translates NL to temporal logics (TL); they LLM-generate a synthetic dataset of (NL, TL) pairs, correcting errors via human-in-the-loop, then fine-tune an LLM for future translations. Our pipeline does not generate synthetic data, or use fine-tuning or human-in-the-loop. Xie et al. (2023) translates NL goal descriptions to formal language (PDDL) given a PDDL domain and a one-shot translation example. They found that LLMs did this well, but struggled with required numeric and spatial reasoning. We translate not just the goal, but also generate (as NL) and translate the preconditions and subgoals. AutoTAMP (Chen et al. 2024) translates NL to Signal Temporal Logic (STL) combined with error-correction loops to catch syntax/semantic errors and refine LLM output. While there are significant similarities with our pipeline, we plan to comparing performance against a classical planning baseline instead of an LLM planner. DELTA (Liu et al. 2024) both converts a scene graph into formal domain and problem definitions via LLM and decomposes the task into a single layer of subtasks which are each solved autoregressively. Our proposed pipeline does not use scene graphs and can further decompose the subgoals by using them as new g_{NL}instances.

Minecraft and LLMs: A few works use LLMs to help agents to solve complex problems in the Minecraft domain. Voyager (Wang et al. 2023) generates curriculums and merges skill-implementations to make more complex skills. Plan4MC (Yuan et al. 2023) constructs skill graphs and finds paths through the graph to accomplish complex tasks. Ghost in the Minecraft decomposes complex NL tasks into easier NL subtasks to enable an LLM planner to construct NL plans, which are stored for re-use (Zhu et al. 2023). Although some of these generate code, none of them generate formal planning models as used by a traditional planner.

3 Preliminaries

We modify the definitions used by Höller et al. (2019) for HGNs, noting that all sets can be assumed to be finite unless otherwise specified: Let $\mathcal{L} = (P, T, V, C)$ where P is a set of lifted predicates, T is a set of types, V is a set of typed variables, and C is a set of typed constants. An HGN planning domain model, $\mathcal{D}^H$, is ($\mathcal{L}$, $\mathcal{G}$, M, O) where $\mathcal{L}$ is as previously defined, $\mathcal{G}$ is a set of lifted goal expressions, M is a set of decomposition methods describing how to decompose a goal into subgoals, and O is a set of operators. A method, $m \in M$, is ($Name(m)$, $Goal(m)$, $Par(m)$, $Pre(m)$, $Sub(m)$), respectively the method name, goal, parameters, preconditions, and subgoals. An action model, $\mathcal{D}^{CL}$, is ($\mathcal{L}$, O), where $\mathcal{L}$ is as previously defined and O is a set of operators. An operator, $o \in O$, is ($Name(o)$, $Par(o)$, $Pre(o)$, $Eff(o)$), respectively the operator name, parameters, lifted preconditions, and lifted effects. A state, s, is a set of fully ground (i.e., containing no variables) subset of predicates from P. An action, $a \in A$, is an operator fully-ground (all variables replaced with constants) to some state s, defined as ($Name(a)$, $Pre(a)$, $Eff(a)$), respectively the action name, ground preconditions, and ground effects.

4 Pipeline

We propose a domain-independent pipeline, with pseudocode in Algorithm 1 and a flow chart in Figure 1, that takes in (1) a NL goal description, g_{NL}, (2) an PDDL-based action model, $\mathcal{D}^{CL}$, annotated with descriptions of its types, predicates, and operators, and (3) hand-selected domain-knowledge necessary for accomplishing to the goal, $\mathcal{K}_{NL}$; it outputs a goal decomposition method, m, written in an HDDL-like language. We supply a PDDL model as input to ensure consistency of predicates and types in each generated method. While it would be valuable to allow the system to suggest symbols for the predicates and types needed for its decompositions, it requires a more complex pipeline to generate a cohesive set of symbols (as indicated by previous work on learning PDDL (Smirnov et al. 2024)) and it would require translating or re-generating the PDDL action model to use the new symbols, which would also be difficult and error-prone and introduce more points of failure to the pipeline. For now, we expect the pipeline to be provided with a correct expert-written or LLM-generated PDDL model. Hand-selecting $\mathcal{K}_{NL}$is a significant amount of work, which could be accomplished with RAG. This is not included in the current pipeline because it is not clear how best to incorporate it and how to detect missing knowledge.

The steps of the pipeline are as follows. The Decomposition Stage (Line 1) decomposes the user-provided target goal g_{NL} into a semi-structured NL goal-decomposition specification dg_{NL}, comprising target goal, preconditions, and subgoals of the target goal. The NL subgoals generated in this stage can be used as new target goals g_{NL}. The Translation Stage (Line 2) translates the components of the NL goal-decomposition specification dg_{NL} from NL into a predicate-based specification dg using the user-provided action model $\mathcal{D}^{CL}$. The Repair Stage (Lines 4-5) detects, describes, and repairs errors in the initial translation. The Methodization Stage (Line 8) inserts the formalized, corrected components of the dg into a template to construct a decomposition method m. Excepting the Repair Stage which calls multiple functions, each stage corresponds to one function. The Decomposition, Translation, and Repair functions all prepare the prompt according to a prompt template and the function inputs, submit the prompt to the LLM, and process the LLMs response to extract the result, which they return. The ErrorDetect and Convert functions are both hard-coded procedures which do not interact with an LLM.

The pipeline tackles the decomposition and translation as two separate stages for several reasons. The first reason is that previous works have already demonstrated that LLMs can do these tasks separately, therefore combining them into a single stage without first testing the performance of separate stages is risky for the second and third reasons. The second reason is that generating a decomposition in formal language forces the LLM to decompose and translate simultaneously, which preliminary testing suggested would increase errors. This could be remedied via chain-of-thought prompting which asks the LLM to do both tasks in sequence, but these prompts are generally more difficult to engineer. The third reason is information control: breaking the process into two stages allows more control over the information that the LLM has access to for a particular stage, preventing it from becoming 'distracted' by knowledge used in other stages. I.e., we can 'hide' $\mathcal{D}^{CL}$ from the Decomposition Stage and $\mathcal{K}_{NL}$ from the Translation Stage. Future work could compare this initial pipeline design with alternatives using chain-of-thought prompting to merge decomposition and translation and/or varying the levels of information restriction between stages. An additional benefit of the current design is that because the decomposition stage generates subgoals as NL, each subgoal in dg_{NL} can be extracted and directly used as a new g_{NL} input for the pipeline without any further prompting and with minimal processing.

At present, we will not attempt to generate multiple methods with different preconditions for the same goal. We anticipate this having minor impact, as planners skip trivial subgoals, ensuring that methods can be applied to states where some of their subgoals are accomplished, if the preconditions do not prevent it. Therefore, many minimal-precondition methods with subgoals that assume the agent "starts from scratch" (which is possible in the case-study domain) will be usable in many situations.

Decomposition [semistructured NL decompositions] (Line 1) generates a semi-structured goal-decomposition specification for a user-provided NL goal description. It takes as input (1) a NL goal description, g_{NL}, and (2) a collection of user-provided NL domain information pertaining to the goal, $\mathcal{K}_{NL}$. It outputs dg_{NL}, a python dict with three fields: (1) goal: the NL goal description provided by the user, g_{NL}, (2) precondition: an LLM-generated preconditions of the decomposition expressed in NL, and (3) subgoals: an LLM-generated sequence of subgoals for decomposing the goal expressed as NL. The following shows an example for the goal "player has crafttable".

```
{"goal": "player has crafttable",
 "precondition": "player can collect logs",
 "subgoals": "player has: logs, planks, crafttable"}
```

This stage comprises injecting the inputs into a provided prompt template to form the prompt, querying the LLM, and extracting the output from the response. Once extracted, each subgoal can be used as a new target goal, g_{NL}. This stage is loosely based on the decomposition process from the work of Zhu et al. (2023), which also generates decompositions as text, and the prompt is a modified version of a prompt used by their LLM Decomposer.

Translation [Predicates] (Line 2) translates the NL goal-decomposition specification into a predicate-based goal-decomposition specification. It takes as input (1) the semi-structured NL goal-decomposition specification dg_{NL} and (2) an action model annotated with descriptions of its types, predicates, and actions, $\mathcal{D}^{CL}$. It outputs the dg, a python dictionary with four fields: (1) goal, $Goal(g_{NL})$: the goal described as a predicate, (2) precondition, $Pre(Goal(g_{NL}))$: the precondition of the decomposition expressed as a list of predicates, (3) subgoals, $Sub(Goal(g_{NL}))$: the subgoal se-

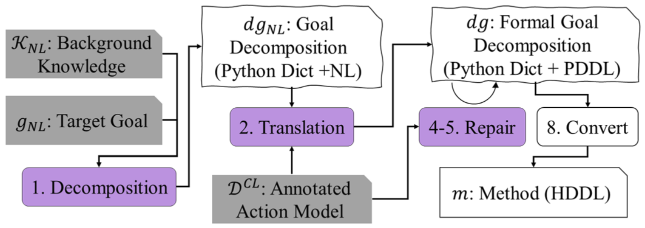
$\mathcal{K}_{NL}$: Background Knowledge
g_{NL}: Target Goal
1. Decomposition
dg_{NL}: Goal Decomposition (Python Dict +NL)
2. Translation
$\mathcal{D}^{CL}$: Annotated Action Model
dg: Formal Goal Decomposition (Python Dict + PDDL)
4-5. Repair
8. Convert
m: Method (HDDL)

Figure 1: The proposed pipeline showing LLM processes (lavender), provided content (gray), LLM output (wavy line boxes), the Convert procedure (rounded white) and the final HDDL output. Numbers in the figure reference the line of the Procedure 1.

Algorithm 1: Generate decomposition method m for target NL goal g_{NL}, using NL domain knowledge $\mathcal{K}_{NL}$, and action model ($\mathcal{D}^{CL}$).

```
Input: $g_{NL}, \mathcal{K}_{NL}, \mathcal{D}^{CL}$
1. $dg_{NL}$:= Decompose($g_{NL}, \mathcal{K}_{NL}$)
2. $dg$:=Translate($dg_{NL}, \mathcal{D}^{CL}$)
3. i := 0
4. while ($\mathcal{E}$ := ErrorDetect($dg, \mathcal{D}^{CL}$)) and $i \leq 5$ do
5.     $dg$:= Repair($dg, \mathcal{E}, \mathcal{D}^{CL}$)
6. if $\mathcal{E}$ then
7.     return false
8. $m$ := Convert($dg$)
9. return $m$
```

quence expressed as a list of predicates, and (4) a python list of (parameter, type) pairs containing each of the parameters, $Par(Goal(g_{NL}))$, used in the previous three fields. The following shows an example for the goal "player has crafttable".

```
{"goal": "agent_has(?table)",
"precondition": ["(object_located_at ?logs ?logs_loc)"],
"subgoals": ["(agent_has ?logs)", "(agent_has ?planks)",
           "(agent_has ?table)"}],
"parameters": [("?logs", "wood"), ("?logs_at","loc"),
            ("?planks","plank"), ("?table", "table")]}
```

In this stage, the pipeline will inject the inputs into a provided prompt template to form the prompt, prompt the LLM, and extract the output from its response.

Repair [revised specification] (Lines 4-5) detects errors produced in the translation stage, and prompts the LLM to re-generate dg based on the errors. It takes as input (1) the predicate-based goal decomposition, dg, (2) the annotated action-model, $\mathcal{D}^{CL}$. It outputs either a corrected dgor false if none was created. The stage comprises an error-detection substage which generates error messages for each error found, and a repair stage which re-prompts the LLM with instruction to repair dg according to the provided errors. The error detection substage incorporates HDDL parsers and a custom logic checker. It cannot ensure that the decomposition is correct, but it can identify python syntax errors, malformed predicates (due to wrong name, arity, or parameter type), and other formatting errors in dg, and generate error messages using error message templates. The error-detection loop will run some maximum number of times, tentatively 5, before returning failure.

Convert [HDDL] plugs the components of dg into the appropriate elements of the decomposition method, m. It is fully procedural and does not use an LLM.

5 Proposed Evaluation

We propose to evaluate the generated decomposition methods by comparing their performance during planning against that of a classical planning action model, $\mathcal{D}^{CL}$, using an HGN planner which can fallback on classical planning to solve a subproblem when the decomposition methods are insufficient. The fallback ability allows the planner to continue when provided incomplete methods.

We will measure performance as a metric of: (1) planning time, (2) problems solved within a timelimit, (3) plan length, (4) proportion of time spent classical planning, and (5) proportion of plan generated by decomposition, planning. We expect that as problem difficult/size increases, an effective HGN planning model will eventually require shorter planning times than the classical planning model. We can evaluate the performance of the hierarchical planning model as we add more methods to the pipeline, tracking its improvement or degradation as a function of (5) and (4).

We plan to test methods for Minecraft, which has a complex technology tree and significant existing corpus of domain information in user-authored wikis, making it a good domain for domain-knowledge-informed method generation. Additionally, the success of the work of Zhu et al. (2023), which also used Minecraft, in NL-based decomposition planning supports this as a reasonable domain.

We will be hand-selecting text from the major Minecraft wiki[1] to create $\mathcal{K}_{NL}$ for each goal.

We plan to use test-goals based on the pickaxe series of items of the Minecraft because it is central to the game. Initial states will be set at different levels of the technology tree, by changing what level of pickaxe the agent starts with (possibly none) and if the agent has access to the necessary craft stations to create its target pickaxe. Varying the level at which the agent start to climb the tech tree can help establish generalizability of methods. Assuming reasonable performance of the generated HGN, we will ablate $\mathcal{K}_{NL}$ to examine its role in producing a quality planning model.

6 Conclusion

We have described a pipeline for generating decomposition methods for goals provided as NL, using NL domain information and an NL annotated action model. We have presented a plan for evaluating the pipeline-generated decomposition methods against a classical planning baseline.

Expansions of the proposed work could include (1) evaluating the system on additional test domains to expand our case study into a full experiment set– some possibilities include Overcooked, Civilization, and Hex, as these have existing documentation; (2) adding a stage for document-informed construction of the action model or components; (3) adding a stage for extracting goal-pertinent domain knowledge from a larger corpus of domain knowledge; (4) adding a stage for prompting the LLM to imagine different situations in which the agent might want to achieve the goal, so we have multiple different decompositions with different precondition sets; (5) adding a stage for using planning, gameplay, or simulator interaction to refine the domain model; (6) adding a module for processing tables containing domain knowledge such that the LLM can use their information; (7) comparing the pipeline against an alternate version where decomposition and translation are accomplished in one prompt, possibly with C-o-T; and (8) varying what information each stage has access to, to determine if decomposition or translation perform better given access to other inputs in the pipeline.

Acknowledgements

We thank NRL for funding this research.

References

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. 3022–3029.

Chen, Y.; Arkin, J.; Dawson, C.; Zhang, Y.; Roy, N.; and Fan, C. 2024. Autotamp: Autoregressive Task and Motion Planning with Llms as Translators and Checkers. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 6695–6702. IEEE.

[1]https://minecraft.fandom.com/wiki/Minecraft_Wiki

Chen, Y.; Gandhi, R.; Zhang, Y.; and Fan, C. 2023. NL2TL: Transforming Natural Languages to Temporal Logics Using Large Language Models. arXiv:2305.07766.

Cosler, M.; Hahn, C.; Mendoza, D.; Schmitt, F.; and Trippel, C. 2023. Nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In Enea, C.; and Lal, A., eds., *Computer Aided Verification*, 383–396. Springer Nature Switzerland.

Ding, Y.; Zhang, X.; Amiri, S.; Cao, N.; Yang, H.; Kaminski, A.; Esselink, C.; and Zhang, S. 2023. Integrating Action Knowledge and LLMs for Task Planning and Situation Handling in Open Worlds. *Autonomous Robots*, 47(8): 981–997.

Erol, K.; Hendler, J.; and Nau, D. 1994a. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994b. Semantics for Hierarchical Task-Network Planning.

Fine-Morris, M.; Floyd, M. W.; Auslander, B.; Pennisi, G.; Gupta, K.; Roberts, M.; Heflin, J.; and Munoz-Avila, H. 2022. Learning Decomposition Methods with Numeric Landmarks and Numeric Preconditions. In *Proceedings of the 5th ICAPS Workshop on Hierarchical Planning (Hplan 2022)*, 29–37.

Gopalakrishnan, S.; Muñoz-Avila, H.; and Kuter, U. 2018. Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. *AI Communications*, 31(2): 167–180.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2024. Leveraging Pre-Trained Large Language Models to Construct and Utilize World Models for Model-Based Task Planning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*.

Guo, W.; Kingston, Z.; and Kavraki, L. E. 2024. CaStL: Constraints as Specifications through LLM Translation for Long-Horizon Task and Motion Planning. arXiv:2410.22225.

Hayes, B.; and Scassellati, B. 2016. Autonomously Constructing Hierarchical Task Networks for Planning and Human-Robot Collaboration. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 5469–5476.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2016. Learning Hierarchical Task Models from Input Traces. *Computational Intelligence*, 32(1): 3–48.

Hérail, P.; and Bit-Monnot, A. 2023. Leveraging Demonstrations for Learning the Structure and Parameters of Hierarchical Task Networks. *The International FLAIRS Conference Proceedings*, 36.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2019. HDDL–a Language to Describe Hierarchical Planning Problems. *HPlan 2019*, 6.

Langley, P. 2024. Learning Hierarchical Problem Networks for Knowledge-Based Planning. In Muggleton, S. H.; and Tamaddoni-Nezhad, A., eds., *Inductive Logic Programming*, volume 13779, 69–83. Springer Nature Switzerland.

Li, R.; Nau, D.; Roberts, M.; and Fine-Morris, M. 2024. Automatically Learning HTN Methods from Landmarks. *The International FLAIRS Conference Proceedings*, 37(1).

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. arxiv:2304.11477.

Liu, J. X.; Yang, Z.; Schornstein, B.; Liang, S.; Idrees, I.; Tellex, S.; and Shah, A. 2022. Lang2LTL: Translating Natural Language Commands to Temporal Specification with Large Language Models.

Liu, Y.; Palmieri, L.; Koch, S.; Georgievski, I.; and Aiello, M. 2024. DELTA: Decomposed Efficient Long-Term Robot Task Planning Using Large Language Models. arXiv:2404.03275.

Lotinac, D.; and Jonsson, A. 2016. Constructing Hierarchical Task Models Using Invariance Analysis. In *ECAI 2016*, 1274–1282. IOS Press.

Nguyen, C.; Reifsnyder, N.; Gopalakrishnan, S.; and Munoz-Avila, H. 2017. Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 226–231.

Nottingham, K.; Ammanabrolu, P.; Suhr, A.; Choi, Y.; Hajishirzi, H.; Singh, S.; and Fox, R. 2023. Do Embodied Agents Dream of Pixelated Sheep: Embodied Decision Making Using Language Guided World Modelling. arXiv:2301.12050.

Oates, T.; Alford, R.; Johnson, S.; and Hall, C. 2024. Using Large Language Models to Extract Planning Knowledge from Common Vulnerabilities and Exposures. In *Working notes of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS) at ICAPS 2024*.

Oswald, J.; Srinivas, K.; Kokel, H.; Lee, J.; Katz, M.; and Sohrabi, S. 2024. Large Language Models as Planning Domain Generators. *Proc. of the International Conference on Automated Planning and Scheduling*.

Segura-Muros, J. A.; Pérez, R.; and Fernández-Olivares, J. 2017. Learning Htn Domains Using Process Mining and Data Mining Techniques. In *ICAPS Workshop on Generalized Planning. Pittsburgh, United States*.

Smirnov, P.; Joublin, F.; Ceravola, A.; and Gienger, M. 2024. Generating Consistent PDDL Domains with Large Language Models. arxiv:2404.07751.

Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. arxiv:2305.16291.

Wu, Y.; Prabhumoye, S.; Min, S. Y.; Bisk, Y.; Salakhutdinov, R.; Azaria, A.; Mitchell, T.; and Li, Y. 2023. SPRING: GPT-4 Out-performs RL Algorithms by Studying Papers and Reasoning. arXiv:2305.15486.

Xie, Y.; Yu, C.; Zhu, T.; Bai, J.; Gong, Z.; and Soh, H. 2023. Translating Natural Language to Planning Goals with Large-Language Models. arXiv:2302.05128.

Yang, Q.; Pan, R.; and Pan, S. J. 2007. Learning Recursive HTN-Method Structures for Planning.

Yuan, H.; Zhang, C.; Wang, H.; Xie, F.; Cai, P.; Dong, H.; and Lu, Z. 2023. Plan4MC: Skill Reinforcement

Learning and Planning for Open-World Minecraft Tasks. arxiv:2303.16563.

Yuan, Q.; Kazemi, M.; Xu, X.; Noble, I.; Imbrasaite, V.; and Ramachandran, D. 2024. TaskLAMA: Probing the Complex Task Understanding of Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17): 19468–19476.

Zhu, X.; Chen, Y.; Tian, H.; Tao, C.; Su, W.; Yang, C.; Huang, G.; Li, B.; Lu, L.; Wang, X.; Qiao, Y.; Zhang, Z.; and Dai, J. 2023. Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via Large Language Models with Text-based Knowledge and Memory. arxiv:2305.17144.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning Hierarchical Task Network Domains from Partially Observed Plan Traces. *Artif. Intell.*, 212(1): 134–157.

A Prompts

Each prompt is formed by inserting the input components into the fields of the prompt. Fields are labels wrapped in curly braces,{}. Instance of double curly braces, {{}}, are not fields but part of the python dictionary syntax. After formatting the string so that the fields are replaced with their content, these double curly braces become single curly braces used by python to denote a dictionary.

All prompts have a "domain" field, which was always replaced with "the video game Minecraft".

A.1 Decomposition Prompt Template

To make the Decomposition Prompt from the template below, the {goal} field is replaced with g_{NL}, and the {bgknowledge} field is replaced with $\mathcal{K}_{NL}$.

~~~
You are an assistant for {domain}.
I will give you a goal and domain background knowledge pertaining to that goal. Please
    describe a way to decompose that goal into subgoals. Please write the decomposition in
    the standard form.
The standard form of the goal is as follows:
```
{{
    "goal": "the goal description",
    "subgoals": "a comma-separated series of subgoals into which the goal can be decomposed",
    "preconditions": "the minimal facts about the world that must be true before the subgoals
         of this goal can be achieved and which do not include the subgoals themselves"
}}
```

The information I will give you:
The goal: a description of the goal condition in natural language.
Background knowledge: some knowledge related to the goal.

Requirements:
1. You must generate the subgoals and preconditions based on the provided background
    knowledge instead of purely depending on your own knowledge.
2. The "subgoals" and "preconditions" should be as compact as possible, at most 3 sentences.
    The knowledge I give you may be raw texts from Wiki documents. Please extract and
    summarize important information instead of directly copying all the texts.
3. The "subgoals" should not include the work "or", i.e., the subgoals must be chosen such
    that they must be achieved in sequence to achieve the final goal.
4. The "preconditions" should not include the word "or" and they should not repeat the "
    subgoals" verbatim.

The goal:
{goal}

Background knowledge:
{bgknowledge}
~~~

A.2 Translation Prompt

To make the Translation Prompt from the template below, the {goaldecomp} field is replaced with dg_{NL}, and the {actionNL} field is replaced with $\mathcal{D}^{CL}$.

```
You are a planning-domain designer for {domain}. Your task is to translate a provided goal
    specification comprising natural language descriptions of a goal, subgoals, and
    preconditions into pddl goal, subgoals, and preconditions. Add a new "parameters" field
    to each translated specification dictionary which lists (variable, variable type) pairs
    for the variables used in the goals, subgoals, and preconditions. Use the predicates,
    types, objects, and action names in the action model I provide you to construct the pddl
    descriptions based on the natural-language descriptions in the goal specification. You
    may only use the action names for translating subgoals. If the provided predicates, types
    , objects, and actions are insufficient to express a component (goal, subgoals,
    preconditions) of the goal specification, write 'UNSUPORRTED' in the value for that
    component. I will provide natural-language descriptions of the predicates and actions and
     you should consider them when creating your translations.

Note that in pddl, variables are prefixed by "?". When writing a predicate, ommit the type
    annotations for its parameters as these are unnecessary. Predicates can be negated with
    the 'not' keyword, i.e. the negation of the predicate "(name ?param1 ?param2)" is "(not (
    name ?param1 ?param2))". No predicates that occur in the subgoals should be exactly
    replicated in the preconditions, and none of the preconditions should occur in the
    subgoals because preconditions are conditions which are already true in the world while
    subgoals are conditions which must become true in the future in order to achieve the goal
    . You may use operators such as "and" to indicate that a set of predicates must be true
    at the same time, using the format "(and predicate1 predicate2 ....)".

In your response, be sure to put each of your translated goal summaries in a python dict
    enclosed in ``` symbols.

Goal specification:
{goaldecomp}

Action Model & Natural-language Descriptions of Predicates and Actions:
{actionNL}

Translate the contents of each goal summary into a python dict with entries for 'goal', '
    subgoals', 'precondition', and 'parameters'. Provide the subgoals entry as a list of
    strings. Wrap python the dict inside ``` symbols.
```

A.3 Repair Prompt

To make the Repair Prompt from the template below, the {pddlgoalspec} field is replaced with dg, the {nlgoalspec} field is replaced with dg_{NL}, the {actionNL} field is replaced with $\mathcal{D}^{CL}$, and the {errors} field is replaced with $\mathcal{E}$.

```
You are a planning-domain designer for {domain}. Your task is to correct a predicate-based
    goal decomposition specification based on an error message.

I will give you the following information:
(1) the predicate-based goal decomposition (goal, parameters, precondition, and subgoals)
    formatted as a python dict,
(2) the natural language translation of the goal decomposition (goal, precondition, and
    subgoals)
(3) an action model describing the domain, with the actions, predicates, and types you may
    use to create the new goal-decomposition
(4) the error message describing the problem with the predicate-based goal decomposition

Remember that in pddl, variables are prefixed by "?". When writing a predicate, omit the type
     annotations for its parameters as these are unnecessary. Predicates can be negated with
    the 'not' keyword, i.e. the negation of the predicate "(name ?param1 ?param2)" is "(not (
    name ?param1 ?param2))". No predicates that occur in the subgoals should be exactly
    replicated in the preconditions, and none of the preconditions should occur in the
    subgoals because preconditions are conditions which are already true in the world while
    subgoals are conditions which must become true in the future in order to achieve the goal
    . You may use operators such as "and" to indicate that a set of predicates must be true
    at the same time, using the format "(and predicate1 predicate2 ....)".


PDDL goal specification:
{pddlgoalspec}

Natural language goal specification:
{nlgoalspec}

Action model & natural-language descriptions of predicates and actions:
{actionNL}

Error message:
{errors}

Please fix the PDDL goal decomposition specification to resolve the issues indicated by the
    error message, while also ensuring that your result is a reasonable translation of the
    natural-language verions of that decomposition specification. Remember to use proper PDDL
     syntax and consider the provided action model and predicates as well as and the original
     goal specification.
```