CLMASP: COUPLING LARGE LANGUAGE MODELS WITH ANSWER SET PROGRAMMING FOR ROBOTIC TASK PLANNING

Xinrui Lin* **Yangfan Wu*** **Huanyu Yang** **Yu Zhang** **Yanyong Zhang** **Jianmin Ji†**

University of Science and Technology of China
{xinruilin, uuyf, yanghuanyu}@mail.ustc.edu.cn, {yuzhang, yanyongz, jianmin}@ustc.edu.cn

June 6, 2024

ABSTRACT

Large Language Models (LLMs) possess extensive foundational knowledge and moderate reasoning abilities, making them suitable for general task planning in open-world scenarios. However, it is challenging to ground a LLM-generated plan to be executable for the specified robot with certain restrictions. This paper introduces CLMASP, an approach that couples LLMs with Answer Set Programming (ASP) to overcome the limitations, where ASP is a non-monotonic logic programming formalism renowned for its capacity to represent and reason about a robot's action knowledge. CLMASP initiates with a LLM generating a basic skeleton plan, which is subsequently tailored to the specific scenario using a vector database. This plan is then refined by an ASP program with a robot's action knowledge, which integrates implementation details into the skeleton, grounding the LLM's abstract outputs in practical robot contexts. Our experiments conducted on the VirtualHome platform demonstrate CLMASP's efficacy. Compared to the baseline executable rate of under 2% with LLM approaches, CLMASP significantly improves this to over 90%.

1 Introduction

In the field of intelligent agents, particularly in robotics [1], one of the primary challenges is parsing brief human instructions to locate corresponding items in complex environments and formulate executable task plans while adhering to action constraints. For example, the instruction "wash clothes in the washing machine" should be decomposed into tasks like retrieving detergent from the cupboard, taking clothes out of the basket, placing both in the washing machine, and then starting it. Effective task planning requires the agent to understand verbal instructions and the user's underlying intent, combining this with common sense and scene-specific items to generate the executable sequence of actions. Considerable work has been attempted using Large Language Models (LLMs), like GPT4 [2], with common sense and some reasoning abilities for task planning, yet we observe that the aforementioned scenario still poses significant challenges for LLM-generated executable plans [3].

On the one hand, placing a large amount of scene content in the limited context window of a language model hinders its ability to process complex scenes [4]. Consider a real household environment where relationships between thousands of items might exist: inputting all of these into the language model is impractical and puts undue stress on the model. On the other hand, numerous constraints or preferences in a scene may not be fully adhered to by a language model [5]. For instance, a detail like plugging in a socket before switching on the TV can be key to the successful execution of a task plan, yet LLMs tend to overlook such checks. These constraints, some common sense and some not, cannot ensure that the language model's task planning will always satisfy them. Explicitly listing and repeatedly reminding the model to adhere to these constraints is also a taxing and not necessarily effective approach.

*These authors are equal contributors to this work and designated as co-first authors.
†The corresponding author.

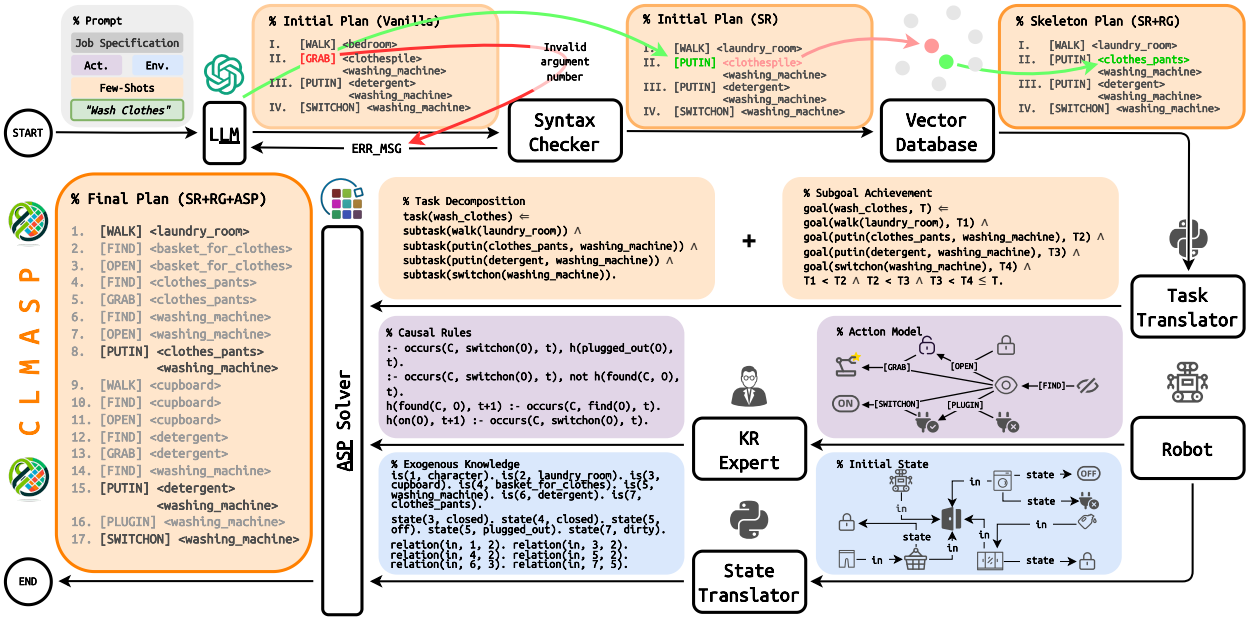

Figure 1: The flowchart of **CLMASP** applied to the "*Wash Clothes*" task. In the flowchart, program modules are represented by boxes, with arrows indicating the direction of data flow. Different types of data are distinguished by colored boxes above the arrows: flesh color for plans, purple for the action model, blue for the robot's observed states, and gray for prompts. The methods represented are LLM original output (Vanilla), Self-Refinement (SR), Referring-Grounding (RG), and ASP Programming (ASP), with CLMASP integrating all three advanced methods (SR, RG, and ASP) for enhanced processing. Following the data flow in CLMASP, the initial plan is generated and self-corrected for verb errors via LLM prompts, with incorrect nouns replaced through nearest vector search. In the ASP segment, while human experts are still required to extract the Causal Model and translate it into ASP rules, the translation of the skeleton plan and robot observations is fully automated. Remarkably, the execution rate of the plan developed by CLMASP can exceed 90%.

Therefore, we need a method to explicitly list the constraints between scene items and automatically plan action sequences, thereby offloading the planning burden from the language model to an external system. Naturally, we realized that Answer Set Programming (ASP) [6] can fulfill this requirement well. We introduced ASP into LLM planning, abstracting the action planning into two levels based on these two elements. The higher level is to construct the skeleton of the plan, generated by the LLM based on natural language instructions, which does not have to be executable but must correctly understand the task and be grammatically correct. The lower level is to obtain an executable plan following the skeleton, solved based on the scene described by ASP and the robot's action model.

We call this method CLMASP, a two-level planning approach provided by the LLM's framework and completed by ASP, for generating robot natural language tasks into atomic action sequences. As shown in Figure 1, this method leverages the LLM's general planning ability to generate key steps in robot task planning, rather than relying on expensive training fine-tuning to generalize on open instruction sets.

Our contributions are summarized as follows:

- We present CLMASP, a two-tiered planning method supported by a framework provided by LLM and completed by ASP, which requires no training, fine-tuning. This combination enhances the implementation of external knowledge and automates the refinement of planning outcomes, overcoming limitations of both LLMs and ASP.

- We demonstrate the efficacy of CLMASP on VirtualHome [7], a challenging test environment for task planning on complex household activities. The executable rate of generated plans exceeding 90%, which implicates the potential of coupling LLM with KR methods for implementing an effective cognitive user interface on device control.

2 Related Work

2.1 LLM-based Task Planning

LLMs are trained on massive offline data and embody internalized common sense knowledge [8], exhibiting surprising zero-shot generalization capabilities [9]. These models are extensively used in robotic planning tasks. Most existing approaches utilizing LLMs for planning problems include:

- Viewing planning problems as sequence generation tasks. [10] encodes action primitives into specific tokens, utilizing a sequence generation method to predict and produce action tokens tailored for particular scenarios and tasks, thereby effectively planning out a coherent sequence of actions.
- Modeling planning tasks akin to reinforcement learning problems. [11] utilizes a value function to receive environmental feedback, which, when combined with the probabilities of atomic actions, outputs a strategically planned sequence of tasks.
- Treating planning problems as 'code' generation tasks. Here, 'code' broadly refers to any structured representation, leveraging the code generation ability of language models to convert all or part of a planning problem into a structured representation. [12] engages in planning through internal monologue within the language model, while [13, 14] assist planning by establishing a structured world model. [15, 16, 17] excel in generating Pythonic code, wherein the language model retrieves operation primitives of atomic actions and then generates executable code or pseudocode, showcasing one of the model's strong suits. [18] specializes in transforming problems into planning domain definition language (PDDL) code for effective planning.
- Viewing planning problems as reordering tasks. [19] approaches planning problems by viewing them as tasks reordering, adeptly sorting an eligible set of instructions based on semantic similarity.

A large number of methods focus on parameter-frozen LLMs and the perspective of 'code' generation tasks. The advantage of these methods lies in their ability to generate interpretable intermediate representations, making the 'thought' process of the language model more controllable and operational. However, it is challenging for these methods to handle numerous objects and constraints in complex scenarios, thereby the complexity of their testing scenarios is often limited [18]. Moreover, the intermediate representations provided by these works usually lack a strict formal definition and are less scalable. Therefore, we focus more on formalizing the representation of robotic planning problems as ASP rules, offloading the specific planning pressure to external solvers through a two-stage planning process, enabling the LLM agent to provide executable planning sequences in complex scenarios with numerous constraints.

2.2 KR-based Task Planning

Various KR-based planning approaches, like situation calculus [20] and non-monotonic causal theories [21, 22], can be used to formalize task planning problems and generate possible solutions through logical reasoning [23].

Compared with action reasoning formalisms based on classical logic, non-monotonic causal theories allow for convenient formalization of many challenging phenomena such as the frame problem, indirect effects of actions (ramifications), implied action preconditions, concurrent interacting effects of actions, and things that change by themselves [23]. These features make the language of causal theories suitable for formalizing task planning problems in open environments [24]. For instance, if the user prefers to keep the fridge door closed, then such non-monotonic causal rule can be directly added to the robot's knowledge base, while keeping other rules unchanged. However, it is challenging to manually encode all of these knowledge in the knowledge base, which limits the application of knowledge-based task planning approaches. For instance, the success rate of solving open task planning problems in [24] is less than 25%.

These non-monotonic causal theories can be further translated into ASP programs [25] and solved by efficient ASP solvers, like clingo[3]. ASP has been successfully applied in the action planning of service robots [26], autonomous driving [27], and multi-agent path finding [28].

In this paper, we combine both advantages of LLM-based and KR-based approaches, using LLM to generate the skeleton of the plan and using the ASP program to refine the skeleton into an executable plan.

[3] `https://potassco.org/clingo/`

3 Preliminaries

3.1 LLMs for Planning

Service robots in open-world environments need to understand various natural language instructions and execute corresponding actions, which requires natural language understanding, semantic parsing, and effective planning. LLMs are well-equipped for this, thanks to their extensive knowledge base, nuanced understanding, and moderate reasoning abilities.

In this work, we use LLMs to generate initial sequences of actions for user tasks, termed "skeleton plans", where LLMs also exhibit some commonsense reasoning abilities. However, LLM-generated skeleton plans often necessitate refinement to align with the robot's operational capabilities and constraints. Hence, a module is required to enhance these skeletons with missing details, rendering them executable.

3.2 ASP Programs for Planning

To examine and complete the skeleton plans, we realized that KR-based action reasoning approaches are well-suited for the requirement. Among these approaches, answer set programming (ASP) programs for planning are preferred, as ASP is a non-monotonic logic programming formalism which can effectively handle the nature of non-monotonic causality in action reasoning [29] and can be solved by efficient ASP solvers [30]. Moreover, we can consider these skeleton plans as special cases of Golog programs [31], which can be further specified by a set of ASP rules and reason with an action theory specified by an ASP program [32, 24].

Here, we briefly review the necessary concepts about ASP and ASP planning. Due to the requirement of computational efficiency, we consider only finite normal logic programs. Following the formalization in the Potassco clingo document[4], an *answer set program* (ASP program) is a finite set of ASP rules of the form:

$$A_0 \texttt{ :- } A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n.$$

where $n \geq m \geq 1$ and $A_0, A_1, \ldots, A_n$ are atoms. An atom is either a simple predicate $\texttt{p}$ or a predicate with arguments $\texttt{p}(\texttt{t}_1, \ldots, \texttt{t}_\texttt{o})$, where each argument $\texttt{t}_\texttt{i}$ ($1 \leq i \leq o$) is a *term*, which can be a number, a *constant* (starting with a lowercase letter), or a *variable* (starting with an uppercase letter).

We also call A_0 the *head* of the ASP rule and $\{A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n\}$ the *body* of the rule. With a slight abuse of the notion, a formula of the form

$$\texttt{ :- } A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n.$$

is considered as an abbreviation o the rule

$$F \texttt{ :- } \neg F, A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n.$$

where F is a new atom that does not appear in other rules.

The *answer sets* of an ASP program are defined in [33]. Given an ASP program P and a set S of atoms, the GL-transformation of P on S, written P^S, is obtained from P by deleting:

1. each rule that has $\neg A$ in its body with $A \in S$, and
2. all $\neg A$ in the bodies of the remaining rules.

For any S, P^S is a normal logic program without any literals of the form $\neg A$, then P^S has only one minimal model. Now a set S of atoms is an *answer set* of P iff S is the minimal model of P^S. An ASP program may have zero, one or multiple answer sets as solution.

ASP programs have been widely used for planning problems [34]. We follow the action language $\mathcal{C}+$ [22] to specify the action theory of an agent, which can also be represented in non-monotonic causal logic introduced by McCain and Turner [21]. Then we can convert these causal rules to corresponding ASP rules [25] and use ASP solvers, like clingo, to compute answer sets of the resulting ASP program. Notice that, each answer set contains an executable plan for the agent.

In the following, we take the action `wash` in the simulator VirtualHome [7] as an example. The related notations are:

- `occurs(C, wash(O))`: the action of washing the object `O` that is performed by the character `C`.

[4] https://potassco.org/doc/

- `clean(O)`: the fluent that the object `O` is clean.
- `holds_lh(C, O)` (resp. `holds_rh(C, O)`): the fluent that the left (resp. right) hand of the character `C` is holding the object `O`.
- `unempty_lh(C)` (resp. `unempty_rh(C)`): the fluent that the left (resp. right) hand of the character `C` is not empty.
- `empty_lh(C)` (resp. `empty_rh(C)`): the fluent that the left (resp. right) hand of the character `C` is empty.

The effect of executing `occurs(C, wash(O))` is `clean(O)`, that can be described in $\mathcal{C}+$:

caused `clean(O)` **if** $\top$ **after** `occurs(C, wash(O))`.

The corresponding causal rule is:

$$\texttt{occurs(C, wash(O))}_t \Rightarrow \texttt{clean(O)}_{t+1}.$$

The definition of causal theory is reviewed in the next section. Here, the atom f_t denotes that the fluent f is true at the time step t. Then the resulting ASP rule is:

```
h(clean(O), t+1) :- occurs(C, wash(O), t).
```

With a slight abuse of the notion, we use `h(clean(O), t+1)` to denote that the fluent `clean(O)` holds at the time step $t+1$, and `occurs(C, wash(O), t)` to denote that the action `occurs(C, wash(O))` occurs at the time step t.

One of the preconditions of `occurs(C, wash(O))` is either `empty_lh(C)` or `empty_rh(C)`, that can also be described in $\mathcal{C}+$:

nonexecutable `occurs(C, wash(O))` **if** `unempty_lh(C)` $\wedge$ `unempty_rh(C)`.

The corresponding causal rule is:

$$\texttt{occurs(C, wash(O))}_t \wedge \texttt{unempty\_lh(C)}_t \wedge \texttt{unempty\_rh(C)}_t \Rightarrow \bot.$$

Then the resulting ASP rule is:

```
:- occurs(C, wash(O), t), h(unempty_lh(C), t), h(unempty_rh(C), t).
```

In specific, we use fluents `unempty_lh(C)` and `unempty_rh(C)` instead of fluents `holds_lh(C, O)` and `holds_rh(C, O)` in the above ASP rule is to reduce the number of variables appearing in the rule, which can significantly improve the computing efficiency for ASP solvers.

There are static causal laws between fluents, i.e., `holds_lh(C, O)` causes `unempty_lh(C)`. Then in $\mathcal{C}+$:

caused `unempty_lh(C)` **if** `holds_lh(C, O)`.

The corresponding causal rule is:

$$\texttt{holds\_lh(C, O)}_t \Rightarrow \texttt{unempty\_lh(C)}_t.$$

The resulting ASP rule is:

```
h(unempty_lh(C), t) :- h(holds_lh(C, O), t).
```

There are also inertial laws of fluents for the frame problem. Take the fluent `empty_lh(C)` as an example, in $\mathcal{C}+$:

inertial `empty_lh(C)`.

The corresponding causal rule is:

$$\texttt{empty\_lh(C)}_t \wedge \texttt{empty\_lh(C)}_{t+1} \Rightarrow \texttt{empty\_lh(C)}_{t+1}.$$
$$\neg\texttt{empty\_lh(C)}_t \wedge \neg\texttt{empty\_lh(C)}_{t+1} \Rightarrow \neg\texttt{empty\_lh(C)}_{t+1}.$$

By considering the efficiency of the ASP program[5], we refine the translation and construct the resulting ASP rule as:

```
h(empty_lh(C), t+1) :- h(empty_lh(C), t), ¬h(unempty_lh(C), t+1).
```

[5]Note that, involving classical negation in ASP programs would reduce the computational efficiency of ASP solvers.

Moreover, the relation between `empty_lh(C)` and `unempty_lh(C)` is:

```
:- h(empty_lh(C), t), h(unempty_lh(C), t).
```

We can specify other actions and fluents similarly. Then we construct an ASP program to specify the action model of an agent, which can be used to compute executable plans for the agent to achieve the required goal state and solve the classical planning problem. In the next section, we show how to specify the skeleton plans in the ASP program and how to compute executable plans that follow these skeletons and fill in missing details.

The task planning problem for a robot with its specified action model given skeleton plans can be represented by an ASP program, whose answer sets correspond to the executable plans for the problem [32]. This answer set planning approach has been implemented as the task planning component of the domestic service robot KeJia [24, 35, 36], who used to be the champion of Robocup@Home 2014[6] and the top runner in the (Enhanced) General Purpose Service Robot test for many years [37].

As discussed in the previous subsection, LLMs are well-equipped for generating proper skeleton plans. Later, the above ASP planning approach with skeleton plans can further refine these skeletons to be executable for the specified agent. However, there are still multiple challenges for implementing the process, especially for complex problems, like task planning problems in VirtualHome. Note that, CLMASP needs to handle complex task planning problems in VirtualHome, which often involve thousands of objects and thousands of relations between objects. Then the ASP encoding needs to be well-designed to ensure effectiveness. More details can be found in our supplementary material.

4 Method

CLMASP integrates the general planning capabilities of LLMs with the logical reasoning capabilities of ASP. This section details the CLMASP methodology, a two-stage approach for robotic task planning. We first specify the semantics of the action model, then define the planning problem with skeleton plans in Subsection 4.1. The processes of the two stages in CLMASP are then detailed in Subsections 4.2 and 4.3, respectively.

4.1 Problem Definition

Notice that, the action model specified by action language $\mathcal{C}+$ can be converted to a *causal theory* [22], which can be translated to an ASP program [25] and solved by ASP solvers. Here we specify the semantics of the action model based on the notions of causal theory [21].

A *causal theory* is a set of *causal rules* of the form: $\phi \Rightarrow \psi$, where ϕ and ψ are formulas without variables. Intuitively, the causal rule reads as "ψ is caused if ϕ is true". An *interpretation* I is a set of literals such that for each atom a in the language, either $a \in I$ or $\neg a \in I$ but not both. Given a causal theory T and an interpretation I, the *reduction*

$$T^I = \{\psi \mid \text{for some } \phi \Rightarrow \psi \in T \text{ and } I \models \phi\}.$$

T^I is a propositional theory. We say that I is a *causal model* of T if I is the unique model of T^I.

Given an action model specified by the causal theory T, we define a *state* s for time t as a set of fluent-atoms with the time t. Intuitively, s denotes a world specified by the fluents that are true at a time step. Let the time names in T be $\{0, 1, \ldots, n\}$, we can define a *trajectory* as a sequence $\langle s_0, a_0, s_1, \ldots, a_{n-1}, s_n\rangle$, where s_i is a state for time i $(0 \leq i \leq n)$, and a_j is an action-atom $(0 \leq j < n)$. Note that, a causal model of the causal theory T for the action model contains exactly a trajectory of the above form, i.e., $s_0 \cup \{a_0\} \cup \cdots \cup \{a_{n-1}\} \cup s_n$ is a causal model of T.

Given a description of the goals to be completed, we can use fluent-formula ψ_n to specify the requirements of the goal states. Then we can add the causal theory T with the causal rule $\neg\psi_n \Rightarrow \bot$. Clearly, a causal model or a trajectory of such causal theory corresponds to a solution of the planning problem.

Before we define the planning problem with skeleton plans, we first provide the definition of skeleton plans. The underlying signature is consisted with three pairwise-disjoint sets: a set of *action* names, a set of *fluent* names, and a set of *subtask* names, where each subtask is also defined by a skeleton plan and circular references between subtasks are not allowed. A *fluent-specification* is formed from fluent names using propositional connective. A *skeleton plan* is defined recursively as follows:

- an action name a is a skeleton plan,
- a fluent-specification φ is a skeleton plan,

[6] `https://athome.robocup.org/`

- a subtask name p is a skeleton plan,
- if P_i $(1 \leq i \leq m)$ are skeleton plans, then $P_1; \ldots; P_m$ is a skeleton plan.

Intuitively, the skeleton plan $P_1; \ldots; P_m$ represents a procedure executed from P_1 to P_m.

Notice that, compared to Golog [31] and procedures in [32, 24], we simplified the definition of skeleton plans by considering only sequential structures and omitting "if-then", "while-do", and "non-deterministic choices". This mainly due to the consideration of the computational efficiency. On one side, LLMs are good at generated these sequential skeleton plans. On the other side, task planning problems in VirtualHome often involve thousands of objects and relations between objects, which requires the ASP encoding to be simple and efficient.

The *planning problem with a skeleton plan* is a pair (T, P), where T is a causal theory specify the action model of the agent and P is a skeleton plan with the same signature of T.

Let $\tau = \langle s_0, a_1, s_1, \ldots, a_{n-1}, s_n \rangle$ be a trajectory of T, we define τ *satisfies* a fluent-specification φ if for some $0 \leq i \leq n$, $s_i \models \varphi_i$ where φ_i is the fluent-formula obtained from φ by replacing each occurred fluent name f by the fluent-atom f_i for the time step i. τ *satisfies* a skeleton plan P is defined recursively as follows:

- If $P = a$, where a is an action name, then a is the action name occurred in the action-atom a_0;
- If $P = \varphi$, where φ is a fluent-specification, then $s_0 \models \varphi_0$;
- If $P = p$, where p is a subtask name, then τ satisfies a skeleton plan for the subtask p;
- If $P = P_1; \ldots; P_m$, where P_i $(1 \leq i \leq m)$ are skeleton plans, then there exists $0 \leq n^1 \leq n^2 \leq \cdots \leq n^{m-1} \leq n$ such that:
  - the trajectory $\langle s_0, a_0, \ldots, s_{n^1} \rangle$ satisfies P_1;
  - the trajectory $\langle s_{n^1}, a_{n^1}, \ldots, s_{n^2} \rangle$ satisfies P_2;
    $\ldots$
  - the trajectory $\langle s_{n^{m-1}}, a_{n^{m-1}}, \ldots, s_n \rangle$ satisfies P_n.

At last, a trajectory τ is a *solution* of the planning problem with a skeleton plan (T, P), if τ is a trajectory of the action model T and satisfies the skeleton plan P.

The requirements for satisfying the skeleton plan can also be encoded in ASP, then we can compute the solutions of (T, P) by computing the answer sets of the corresponding ASP program. The ASP encoding needs to be well-designed to ensure effectiveness. More details can be found in our supplementary material.

4.2 Generating Skeleton plans by LLMs

This stage of the CLMASP focuses on interpreting task instructions and developing initial skeleton plans using LLMs. Algorithm 1 also details this stage.

4.2.1 Initial Plan Generation

The initial skeleton plan is produced using a Chain-of-Thought (CoT) prompting approach [38]. The process starts by supplying the LLM with specific environmental information and primitive actions through a structured prompt, incorporating an example task planning in a verb-object format.

Figure 1 showcases our prompt, which integrates task-related elements such as verbs $\mathcal{V}$ and object categories $\mathcal{C}$. This integration enhances the relevance and reproducibility of CLMASP by ensuring the LLM considers context-specific details, such as environmental observations, basic actions and objects. The designed prompt aims to guides the LLM to produce a structured response, beginning with a natural language description of the intended actions. This leads into a detailed, formatted breakdown, featuring a 'thoughts' section that narrates the sequence of the plan and an 'actions' section that clearly lists each step. As shown in Algorithm 1, we generate a sequence of actions τ_s^0, where each action is in the form of a verb-object pair.

4.2.2 Syntactic Self-Refinement (SR)

Following the generation of the initial plan, a syntactic self-refinement process corrects emerging syntactic errors during open-loop generation [39], such as deviations from the expected format or non-conformities with the VirtualHome execution syntax. This involves re-prompting the LLM with error feedback iteratively to enhance the plan's syntactic precision.

Algorithm 1 Skeleton Planning Algorithm

Input: Task Instruction $task$, Verb Set $\mathcal{V}$, Category Set $\mathcal{C}$
Parameter: CoT Template T_s, Max Iterations $k_{\max}$, Language Model LM As General Planner, Embedding Model EM
Output: Skeleton Plan τ_s

$\tau_s^0 \leftarrow \mathrm{LM}(\mathrm{T}_s(task, \mathcal{V}, \mathcal{C}))$
for $k = 1$ **to** $k_{\max}$ **do**
  if grammar_verifier(τ_s^{k-1}) **then**
    break
  else
    error_info $\leftarrow$ get_error(grammar_verifier, τ_s^{k-1})
    $\tau_s^k \leftarrow \mathrm{LM}(\mathrm{T}_s(task, \mathcal{V}, \mathcal{C}, \tau_s^{k-1}, \text{error\_info}))$
  end if
end for
for all c_q of objects in τ_s^k **do**
  if $c_q \notin \mathcal{C}$ **then**
    $q \leftarrow \mathrm{EM}(c_q)$
    $c_m \leftarrow \arg\min_{c_i \in \mathcal{C}} \mathrm{Sim}(q, \mathrm{EM}(c_i))$
    Replace c_q with c_m in τ_s^k
  end if
end for
return τ_s^k

Algorithm 1: Skeleton Planning Algorithm

The refinement employs a rule-based grammar verifier to detect syntax errors, refining the plan until it achieves the desired syntactic accuracy or reaches the maximum iteration limit, $k_{\max}$. As shown in Algorithm 1, we can provide the LLM a prompt each round, assembled from the allowed verb set $\mathcal{V}$, object categories $\mathcal{C}$, the answers from the previous round τ_s^{k-1}, and the corresponding error prompts error_info.

4.2.3 Semantic Referring-Grounding (RG)

Following the syntactic self-refinement, this phase corrects semantic inaccuracies such as the plan's reference to non-existent objects like "clothespile" , an example shown in Figure 1. For accurately mapping expressions to the correct environmental objects, the RG process involves constructing a vector database for the scene with Milvus [40] and embedding all scene categories using the model *text-embedding-ada-002*.

As shown in Algorithm 1, each object category c_q in the initial plan τ_s^k is embedded and compared against actual scene objects c_m through cosine similarity searches. The cosine similarity between two vectors $\vec{a}$ (embedding of c_q) and $\vec{b}$ (embedding of c_m) is calculated as follows:

$$\cos(\theta) = \frac{\sum_{i=0}^{n-1} a_i \cdot b_i}{\sqrt{\sum_{i=0}^{n-1} a_i^2} \cdot \sqrt{\sum_{i=0}^{n-1} b_i^2}} \tag{1}$$

where a_i and b_i are components of vectors $\vec{a}$ and $\vec{b}$ respectively. If τ_s^k includes object categories not present in the scene, we find the closest matching category c_m via cosine similarity and replace it.

This replacement ensures all object references are contextually appropriate and verifiably present, aligning the plan accurately with the real-world environment and enhancing both the semantic integrity and practical applicability of the generated skeleton plan.

By employing a dual-phase refinement process(SR and RG), we hope that the generated skeleton plan is both syntactically sound and semantically accurate, tailored specifically to the task instructions. Then we can encode τ_s^k to a skeleton plan P as specified in the previous subsection.

4.3 Fine-Grained Planning by ASP

We developed a series of Python scripts that convert action descriptions into logic programs with answer set semantics, enabling planning through ASP.

As introduced in Subsection 4.1, we can specify the ASP program for the action model of the agent, which encompasses knowledge of an agent's actions and environmental changes, crucial for the an agent's built-in knowledge. After encoding the initial state and the requirements of the skeleton plan, the resulting ASP program can compute the solutions of the planning problem with the skeleton plan.

We implement the planning problem with skeleton plans in ASP for task planning problems in VirtualHome. As task planning problems in VirtualHome often involve thousands of objects and relations between objects, based on the above encoding, we also introduced a number of auxiliary fluents to reduce the number of variables appeared in each ASP rule and reduce the the number of grounding instances that are related for solutions. More details can be found in our supplementary material.

5 A Case Study

In this section, we illustrate the CLMASP approach for planning on the running example "*Wash Clothes*", which is already simply presented in Figure 1.

Given the task "Wash Clothes", we first need to guide the LLM to generate a specific plan for this task, replacing the traditional steps of writing a robot instruction recipe [41] or task procedure [32]. Specifically, we use the following four-step *job specification* to explain the task and some constraints to the LLM:

```
SYSTEM:
You serve as an AI task planner. 1. Your task is to create a plan to achieve
a goal by converting it into a sequence of actions. Each action shouldfollow the format
"[verb] <target1> <target2>", where 'verb' represents the action, and 'target1' and 'target2'
are optional arguments.You are limited to the following action verbs:
- [find] <arg1>: Find 'arg1'.
- [open] <arg1>: Open 'arg1'.
- [putin] <arg1> <arg2>: Put 'arg1' inside 'arg2'.
- [switchon] <arg1>: Turn 'arg1' on.
... (omitted) ...
2. You can only use the following values as arguments:
Permissible Scenes: home_office, laundry_room, bedroom, ... (omitted) ...
Permissible Objects: detergent, clothes_pants, cupboard, ... (omitted) ...
3. You must describe your plan in natural language at the beginning. After that,
you should list all the actions together. The response should follow the format:
{
  "thoughts":"Your plan description ... step by step",
  "actions":[
    "action1",  "action2", "action3",
    ...
  ]
}
4. Here is an example plan to achieve a goal for reference: ... (omitted) ...)
```

Subsequently, we provide an example to demonstrate, aiming to stimulate its few-shot learning ability. At the end of the prompt, we follow with the task that needs to be output this time, which is:

```
USER:
The goal is to "wash clothes". Begin your plan. Your response should be formatted
as a JSON object that can be successfully parsed by Python's json.loads() function.
```

Here, we require the output in JSON format, which helps us parse the reply and check for errors.

We conduct simple syntax and constraint checks in the *Syntax Checker* module. If these checks are not passed, a predefined exception is thrown, and a re-output is requested:

```
USER:
Revise your plan. Your plan ... (omitted) ... failed.
Because: Invalid argument number. Please check action format of "grab".
```

Usually, within a few rounds, it can correct some simple errors, such as output format mistakes or improper verb usage. However, it typically does not strictly adhere to noun constraints, which is a typical shortcoming of LLMs.

Therefore, we use a vector database for referring-grounding. We map all items in the scene to vectors in a high-dimensional vector space using the *text-embedding-ada2 model*, and then perform nearest neighbor search (e.g., based on cosine distance) to obtain a semantically closest vector and retrieve its semantic label. For example, "apple" and "fruit" are close, as are "book" and "novel"; in this example, "clothespile" is replaced with "clothes_pants". The vectorization or embedding process is based on pretrained models, and its effectiveness largely depends on the performance of these language models and the "semantic distance" learned during training. We call the resulting plans skeleton plans, as they usually have a lower executability but indeed contain essential steps. For instance, the four steps shown in the example are necessary but not executable, as LLM does not consider processes like finding and picking up "clothes_pants".

After several steps, we can roughly solve the process of generating a skeleton plan from natural language instructions. Next, we need to flesh out this skeleton, that is, by generating dependencies between actions using ASP. In this step, we need to formalize the robot's action model into causal rules and transcribe them into ASP plans. These rules are fully reusable and are written once during the entire lifecycle of the robot. In the example, for the `switchon` action, the target item must be in a `found` state, which is caused by `find`. Therefore, the derived rules are:

```
:- occurs(C, switchon(O), t),  h(plugged_out(O), t).
:- occurs(C, switchon(O), t), not h(found(C, O), t).
h(found(C, O), t+1) :- occurs(C, find(O), t).
h(on(O), t+1) :- occurs(C, switchon(O), t).
```

For the skeleton plans given by the LLM, we use a Python-based module to automatically convert them into corresponding ASP rules. This process involves treating each subtask as a goal that needs to be time-sequenced, and finally integrating them together. They are linked to the action model by the following two ASP rules:

```
1{occurs(C, A, t): action_of(C, A), related_action(A)}1 :- is(C, character).
goal(C, A, t) :- occurs(C, A, t), action_of(C, A), is(C, character).
#program check(t).
:- query(t), not goal(Task, t), task(Task).
```

For the initial state used in ASP reasoning, we assume that the robot can obtain a global semantic map through observation. In the experiment, this semantic map is a directed acyclic graph, with each node representing an entity in the scene. Directed edges represent biased relationships, each entity has its own state and a unique ID. For example, `is(1, character)` indicates that the entity with ID=1 is a character, i.e., the robot. The state of entities is expressed by the predicate state, such as `state(7, dirty)` indicating that `clothes_pants` with ID=7 are dirty. Relationships between entities are expressed by the predicate relation, such as `relation(in, 5, 2)` indicating that the `washing_machine` with ID=5 is inside the `laundry_room` with ID=2.

After inputting into the ASP Solver for reasoning, the final output will be a sequence like:

```
occurs(1, walk(2), 1)
occurs(1, find(4), 2)
occurs(1, open(4), 3)
... (omitted) ...
occurs(1, putin(6,5), 15)
occurs(1, plugin(5), 16)
occurs(1, switchon(5), 17)
```

Here, all entities are represented by their unique ID numbers.

6 Experiments

This section outlines the experimental setup and results for evaluating our method.

6.1 Configuration

6.1.1 Experimental Platform and Dataset

We evaluate our method using the VirtualHome (VH), a virtual household simulation platform from [7]. VH consists of 50 custom-designed environments suitable for executing various activities. Each environment is represented through an Environment Graph, structured as dictionaries with nodes representing objects and edges depicting relationships

between them. This structure enables dynamic updates by modifying the Environment Graph, allowing real-time changes caused by actions in the scenario.

Additionally, the experiments are supported by a dataset [7], which includes 292 unique tasks, each described in natural language alongside action sequences refined through reinforcement learning and human intervention. For example, one task is "wash clothes", described as "load the dirty clothes into the washing machine" followed by a series of planned action sequences.

		Vanilla	**SR**	**RG**	**SR+RG**	**ASP**	**SR+ASP**	**RG+ASP**	**SR+RG+ASP**
GPT3.5	*Exec*	1.16%	2.07%	7.75%	10.47%	63.18%	64.73%	93.93%	**96.12±1.9%**
GPT4	*Exec*	1.94%	2.32%	8.90%	11.24%	66.28%	71.32%	89.15%	**94.57±2.30%**

Table 1: Comparison of the effects of different modules that SR represents self-refinement, RG denotes referring-grounding, ASP represents ASP planning. Vanilla indicating the standard LLM method without auxiliary techniques. It is evident that ASP contributes significantly to the enhancement of executability.

6.1.2 LLMs and ASP Solver

We use 2 language models, GPT-3.5-1106 (*GPT-3.5*) and GPT-4-0613 (*GPT-4*), representing advanced level of language models from OpenAI. These language models are sensitive to the sampling parameters; after extensive grid searching over the hyperparameters, we set *'temperature'*=0.9, *'frequency_penalty'*=0.9, *'presence_penalty'*=0.8 as the optimal settings for our experiments.

For logic program processing in CLMASP, we employ the ASP solver *clingo* [30] version 5.6.2 from the Potassco project [42] for logic program processing in CLMASP.

6.1.3 Evaluation Metrics

We use two metrics to evaluate the performance of CLMASP: Executability (*Exec*) and Goal Achievement Rate (*GAR*).

Exec assesses if action plans can be successfully executed within the simulator, measuring the operational feasibility of these plans.

GAR assesses if these plans effectively meet the task-specific goals by comparing the state changes in the environment before and after executing the plans against a ground truth state. This ground truth state, denoted as s_{gt}, represents an ideal state that reflects the necessary changes to achieve the task's goals starting from an initial state s_{initial}. GAR is calculated using the formula:

$$GAR = 1 - \frac{|(s_{gt} - s_{\text{initial}}) - (s' - s_{\text{initial}})|}{|s_{gt} - s_{\text{initial}}|}, \tag{2}$$

where $|\cdot|$ denotes the cardinality of the set, representing the number of state conditions changes. Essentially, *GAR* reflects how closely the actual outcome s' of the executed plan matches the ideal outcome s_{gt}.

6.2 Results and Analysis

6.2.1 Overview of Planning Performance

The results in Table 1 demonstrate the impact of integrating various modules within the CLMASP framework on the performance of LLMs, in task planning.

Both GPT-3.5 and GPT-4 struggle with generating executable plans due to incorrect object references, misuse of actions and the omission of necessary conditions. These issues stem from the LLMs' limited capabilities in handling the detailed constraints required for robust action planning. To address these issues, CLMASP employs the RG improving object referencing, and the ASP module correcting action misuse and aligning steps with constraints. This results in a boost in executability rate from as low as 1.42% in the vanilla configuration to over 95% in enhanced setups, highlighting ASP's effectiveness in overcoming the planning limitations of LLMs in constraint-heavy environments.

In terms of *GAR*, GPT-4 shows better performance across almost all configurations compared to GPT-3.5. The incorporation of the ASP module significantly enhances goal-oriented task performance, with the highest *GAR* observed in the SR+RG+ASP configuration for both models: 68.00% for GPT-3.5 and 71.51% for GPT-4. These findings suggest that the logical planning capabilities of ASP are crucial in achieving specific task goals effectively.

6.2.2 Impact of Individual Modules

SR shows moderate improvements when used alone, but is more effective when combined with other modules. RG improves both *Exec* and *GAR*, especially effective when used alongside ASP, indicating its utility in correct object referring-grounding. ASP, enhancing both *Exec* and *GAR* significantly, demonstrating its strength in logical planning and constraint adherence.

6.2.3 Model Comparison

The results reveal that GPT-4 shows higher planning capabilities than GPT-3.5, but still benefits greatly from module integrations. We notice the impact of RG+ASP is more pronounced on GPT-3.5, likely because GPT-4 already excels in aspects that RG aims to enhance, diminishing the relative improvement seen with this module integration.

7 Discussions & Future Work

In this work, we present CLMASP, a general robotic task planning framework that coupling LLMs with ASP to significantly enhance LLM Agents to generate executable plans in challenging test environments on complex household activities. Specifically, we have formally defined the task planning problem and provided a set of methods to convert it into corresponding ASP rules, while offering better maintainability and portability. Furthermore, we also explore how to generate ASP rules using LLMs with proper prompts, which allows non-KR experts to update the ASP program through the interaction with LLMs.

7.1 Environmental Flexibility Explanation

To adapt the CLMASP system to a new robot, a KR expert only needs to modify the ASP rules converted from the action model in the Figure 1. Theoretically, CLMASP is applicable to robots that use Directed Acyclic Graphs (DAGs) to represent global semantic maps and employ Verb-Noun commands. If a robot's semantic map is not represented as a DAG, slight modifications should be made to the State Translator to accommodate it. If the robot's commands do not follow a Verb-Noun format, adjustments should be made to the LLM Prompt, Syntax Checker, and Task Translator to suit this structure. Overall, the CLMASP system is modular; as long as the interfaces align, the aforementioned modifications can be independently implemented.

7.2 Closed-Source LLM Constraints

Due to the constraints of available computational resources, our experiments involving LLMs were conducted using the API provided by OpenAI [2]. Employing OpenAI's API for planning tasks presents two primary challenges: it requires a significant consumption of tokens, which can be costly, and it limits the ability to inject domain-specific knowledge through fine-tuning, a service that is also expensive when offered by OpenAI.

In our future research, we plan to explore the use of open-source LLMs as an alternative to the proprietary GPT series. The Llama series, represents a promising option. Our preliminary trials with Llama3 [43] and fine-tuned Llama2 [44] utilizing the CLMASP method have demonstrated encouraging outcomes.

7.3 Automating ASP Program Generation

The translation of causal rules into ASP programs still necessitates some level of expert knowledge, which currently constitutes approximately 30% of the content in ASP programs that articulate causal rules. For our future endeavors, we intend to adopt the KnowRob [45] methodology to develop an ontology-based knowledge graph [46]. This approach will allow us to further automate the generation of ASP programs in specific commonsense domains by integrating knowledge graph embeddings with inductive logic techniques [47].

References

[1] Dongcai Lu and Xiaoping Chen. Interpreting and extracting open knowledge for human-robot interaction. *IEEE/CAA Journal of Automatica Sinica*, 4(4):686–695, 2017.

[2] OpenAI. Gpt-4 technical report. Technical report, OpenAI, 2023.

[3] Zhenyu Wu, Ziwei Wang, Xiuwei Xu, Jiwen Lu, and Haibin Yan. Embodied task planning with large language models. *arXiv e-prints*, pages arXiv–2307, 2023.

[4] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. *Authorea Preprints*, 2023.

[5] Meiqi Chen, Yubo Ma, Kaitao Song, Yixin Cao, Yan Zhang, and Dongsheng Li. Learning to teach large language models logical reasoning. *arXiv preprint arXiv:2310.09158*, 2023.

[6] Vladimir Lifschitz. *Answer set programming*. Springer Heidelberg, 2019.

[7] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8494–8502, 2018.

[8] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[9] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[10] Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, Quan Vuong, Vincent Vanhoucke, Huong Tran, Radu Soricut, Anikait Singh, Jaspiar Singh, Pierre Sermanet, Pannag R. Sanketi, Grecia Salazar, Michael S. Ryoo, Krista Reymann, Kanishka Rao, Karl Pertsch, Igor Mordatch, Henryk Michalewski, Yao Lu, Sergey Levine, Lisa Lee, Tsang-Wei Edward Lee, Isabel Leal, Yuheng Kuang, Dmitry Kalashnikov, Ryan Julian, Nikhil J. Joshi, Alex Irpan, Brian Ichter, Jasmine Hsu, Alexander Herzog, Karol Hausman, Keerthana Gopalakrishnan, Chuyuan Fu, Pete Florence, Chelsea Finn, Kumar Avinava Dubey, Danny Driess, Tianli Ding, Krzysztof Marcin Choromanski, Xi Chen, Yevgen Chebotar, Justice Carbajal, Noah Brown, Anthony Brohan, Montserrat Gonzalez Arenas, and Kehang Han. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In Jie Tan, Marc Toussaint, and Kourosh Darvish, editors, *Proceedings of The 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pages 2165–2183. PMLR, 06–09 Nov 2023.

[11] Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, pages 287–318. PMLR, 2023.

[12] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[13] Rowan Zellers, Ari Holtzman, Matthew Peters, Roozbeh Mottaghi, Aniruddha Kembhavi, Ali Farhadi, and Yejin Choi. PIGLeT: Language grounding through neuro-symbolic interaction in a 3D world. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2040–2050, Online, August 2021. Association for Computational Linguistics.

[14] Kolby Nottingham, Prithviraj Ammanabrolu, Alane Suhr, Yejin Choi, Hannaneh Hajishirzi, Sameer Singh, and Roy Fox. Do embodied agents dream of pixelated sheep?: Embodied decision making using language guided world modelling. *arXiv preprint arXiv:2301.12050*, 2023.

[15] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[16] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE, 2023.

[17] Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023.

[18] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

[19] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

[20] Raymond Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT press, 2001.

[21] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 460–465, 1997.

[22] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.

[23] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of knowledge representation*. Elsevier, 2008.

[24] Xiaoping Chen, Jianmin Ji, Zhiqiang Sui, and Jiongkun Xie. Handling open knowledge for service robots. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, pages 2459–2465, 2013.

[25] Jianmin Ji and Fangzhen Lin. Turner's logic of universal causation, propositional logic, and logic programming. In *Proceedings of the 12th International Converence on Logic Programming and Nonmonotonic Reasoning (LPNMR-13)*, pages 401–413. Springer, 2013.

[26] Xiaoping Chen, Guoqiang Jin, Jianmin Ji, Feng Wang, and Jiongkun Xie. Kejia project: Towards integrated intelligence for service robots. *RoboCup@ Home League Team Descriptions, Istanbul*, 2011.

[27] Suraj Kothawade, Vinaya Khandelwal, Kinjal Basu, Huaduo Wang, and Gopal Gupta. Auto-discern: autonomous driving using common sense reasoning. *arXiv preprint arXiv:2110.13606*, 2021.

[28] Van Nguyen, Philipp Obermeier, Tran Son, Torsten Schaub, and William Yeoh. Generalized target assignment and path finding using answer set programming. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, pages 194–195, 2019.

[29] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1985–1993, 1995.

[30] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[31] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[32] Jianmin Ji and Xiaoping Chen. From structured task instructions to robot task plans. In *Proceedings of the 5th International Conference on Knowledge Engineering and Ontology Development (KEOD-13)*, pages 237–244, 2013.

[33] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080. Cambridge, MA, 1988.

[34] Tran Cao Son, Enrico Pontelli, Marcello Balduccini, and Torsten Schaub. Answer set planning: A survey. *Theory and Practice of Logic Programming*, 23(1):226–298, 2023.

[35] Kai Chen, Fangkai Yang, and Xiaoping Chen. Planning with task-oriented knowledge acquisition for a service robot. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, pages 812–818, 2016.

[36] Dongcai Lu, Yi Zhou, Feng Wu, Zhao Zhang, and Xiaoping Chen. Integrating answer set programming with semantic dictionaries for robot task planning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI-17)*, pages 4361–4367, 2017.

[37] Luca Iocchi, Dirk Holz, Javier Ruiz-del Solar, Komei Sugiura, and Tijn Van Der Zant. Robocup@ home: Analysis and results of evolving competitions for domestic and service robots. *Artificial Intelligence*, 229:258–281, 2015.

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[39] Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *arXiv preprint arXiv:2303.14100*, 2023.

[40] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[41] Michael Beetz, Daniel Beßler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoğlu, and Georg Bartels. Know rob 2.0—a 2nd generation knowledge processing framework for cognition-enabled robotic agents. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2018.

[42] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *Ai Communications*, 24(2):107–124, 2011.

[43] Meta AI. Introducing meta llama 3: The most capable openly available llm to date, April 2024. Accessed: 2024-05-01.

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[45] Moritz Tenorth and Michael Beetz. Knowrob—knowledge processing for autonomous personal robots. In *2009 IEEE/RSJ international conference on intelligent robots and systems*, pages 4261–4266. IEEE, 2009.

[46] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[47] Rudolf Carnap. On inductive logic. *Philosophy of science*, 12(2):72–97, 1945.