2024 IEEE International Conference on Robotics and Automation (ICRA 2024)
May 13-17, 2024. Yokohama, Japan

When Prolog Meets Generative Models: a New Approach for Managing Knowledge and Planning in Robotic Applications

Enrico Saccon[1], Ahmet Tikna[1], Davide De Martini[1], Edoardo Lamon[1,2], Luigi Palopoli[1], Marco Roveri[1]

Abstract— **In this paper, we propose a robot oriented knowledge representation system based on the use of the Prolog language. Our framework hinges on a special organisation of Knowledge Base (KB) that enables: 1) its efficient population from natural language texts using semi-automated procedures based on Large Language Models (LLMs); 2) the seamless generation of temporal parallel plans for multi-robot systems through a sequence of transformations; 3) the automated translation of the plan into an executable formalism. The framework is supported by a set of open source tools and its functionality is shown with a realistic application.**

I. Introduction

In the last few years, the rising tide of AI has brought the promise of a radical change, in which robots will be able to autonomously move in unstructured environment, react to unanticipated events, and cooperate with humans and also other robots. The bedrock of this revolution rely on an effective and efficient way to manage knowledge. While the way humans produce and apply knowledge is only superficially understood, some defining and commonly recognized features of human knowledge representation are:

1) *understandability* and *explainability*: we use different types of languages to accumulate information that can be shared with other humans;
2) *scalability*: we use hierarchies and conceptual links to organise massive amounts of knowledge;
3) *usability*: conceptual entities are combined with a vocabulary of actions that humans can generate or receive.

In the past, different researchers have sought ways to express knowledge that could meet at least part of these requirements and be usable in computer programmes (and hence in a robot). Logic languages, like Prolog, or functional languages, like Lisp, hold the promise to be a *lingua franca* between humans and robots [1], [2].

Prolog is a logic programming language used for the representation of knowledge and symbolic reasoning. In Prolog, one can define a KB, i.e., a set of facts and rules describing the problem, that can be queried to obtain information regarding the satisfiability of more complex conditions. In robotics, it is a useful tool to represent knowledge about robots, their actions, and the environment. These interesting features of Prolog, and in general of logical languages, have led to their application to construct knowledge representations for robotics, such as KnowRob [3], to solve planning problems [4] and to foster human-robot interaction when paired with natural language processing [5], [6].

In this paper, we focus on two fundamental questions. First, while it is true that a Prolog KB is understandable and interpretable by a human reader, it is also not easy to write for a non-specialist. On the other hand, the automatic extraction of a strongly structured computer artefact, like a Prolog KB, from natural language transcript of conversations with humans is a very challenging and expensive task for traditional natural language processors. In this context, the emergence of large language models (LLMs) [7]–[9] could come to the rescue. They are a class of AI models aimed at natural language processing. They are often built upon transformer networks [10], which utilise self-attention mechanisms to gain a better understanding of the context of the words in a sentence. They are typically trained with enormous amounts of data and have hundreds of billions of parameters, which can also be fine-tuned for the task in which they have to be employed [11], [12]. LLMs have been applied to a growing number of different fields, from healthcare [13] to planning [14], demonstrating also their limitations [15]. LLMs abilities have recently been the focus of many studies on planning [16], [17], which have tested LLMs either to provide a PDDL problem to a certain domain [16], or to directly create Pythonic code [17]. In particular, given the outcome of these works, it appears that LLMs cannot be directly used to plan [15], whereas they may be more than capable to integrate common-sense knowledge that is otherwise elusive to capture.

This leads us to our first research question: is it possible to use LLM to populate a robot-oriented Prolog KB, at least as part of a semi-automated procedure? Given that Prolog is a widespread choice for expressing KBs in terms of logic clauses (predicates), symbolic reasoning and natural language processing, the second research question is: could this programming language be leveraged to create a framework that, exploiting an LLM to process the input, is able to smoothly provide a resilient plan and learn from possible unforeseen events?

We address these two questions by proposing a novel Prolog-based knowledge representation system, which 1) simplifies the population of the KB with a semi-automatic procedure relying on LLMs. As a first step towards the

Co-funded by the European Union under NextGenerationEU (FAIR - Future AI Research - PE00000013), under project INVERSE (Grant Agreement No. 101136067), under project MAGICIAN (Grant Agreement n. 101120731) and by the project MUR PRIN 2020 (RIPER - Resilient AI-Based Self-Programming and Strategic Reasoning - CUP E63C22000400001).

[1]Dipartimento di Ingegneria e Scienza dell'Informazione, Università di Trento, Trento, Italy. {enrico.saccon, ahmet.tikna, edoardo.lamon, luigi.palopoli, marco.roveri}@unitn.it, davide.demartini@studenti.unitn.it

[2]Human-Robot Interfaces and Interaction, Istituto Italiano di Tecnologia, Genoa, Italy.

generation of the whole KB, in this work we present the generation of the initial and final states of the KB; 2) enables the seamless generation of temporal parallel plans that support parallel actions of multiple agents, and 3) automatically translates the plan into a formalism (the Behaviour Trees (BT) [18]). We evaluated the proposed method by verifying: the correctness of the initial and final states returned by the LLM, comparing the performances of 3 golden standards in terms of LLMs, such as GPT 3.5, GPT 4.0, and BARD; and the consistency of the schedule generated by the planner. The results demonstrated the capability of LLMs in interpreting the natural language inputs in terms of plan requests and the potential suitability for further advanced requests towards the generation of the whole knowledge base.

II. Problem description and conceptual scheme

In this section, we define the addressed problem and we describe the workflow we adopted to solve it. We focus on the challenge of orchestrating a sequence of actions for multiple agents using the Prolog programming language. Our rationale for choosing Prolog is based on the fact that it is suitable for constructing knowledge bases well perceived in robotics, and it allows for symbolic reasoning.

In order to produce a feasible and robust plan, we rely on (state-space) temporal task planning, here briefly described. In *temporal task planning*, one does not only reason about the ordering of actions, but also about their metric duration. A *temporal task planning problem* is a tuple $TP = (F, DA, I, G)$, with F, I and G defined as the sets of fluents, initial and final states, respectively (*STRIPS* classical planning problem [19]), and with DA being a set of *durative actions*. A literal is either a fluent or its negation. Every action $a \in A$ is defined by two sets of literals: the preconditions, written $pre(a)$, and the effects $eff(a)$. Following [20]–[22], a durative action $a \in DA$ is given by i) two classical planning actions $a_{\vdash}$ and $a_{\dashv}$, with preconditions and effects at start and at end; ii) and a minimum $l_a \in \mathbb{R}^+$ and maximum $u_a \in \mathbb{R}^+$ duration. The ordering between the two snap actions is enforced by an overall condition. A *temporal plan* $\pi = \{tta_1, \cdots, tta_n\}$ is a set of time-triggered temporal actions, and it is a *valid temporal plan* if and only if it can be simulated, i.e., starting from the initial state we apply each timed triggered action $tta_i = \langle t_i, a_i, d_i \rangle \in \pi$ at time t_i with duration d_i, and at the end of the simulation, we obtain a state fulfilling the goal condition. We refer the reader to [23] for a thorough discussion on the semantics.*State-space temporal planning* is a specific approach to temporal planning, combining i) a classical forward state-space search to generate a candidate plan outline; and ii) a temporal reasoner to check its temporal feasibility [20]–[22]. The search extracts a classical planning and then checks if the associated temporal network is consistent. If it is, then a time-triggered plan can be computed and the search stops, otherwise the process starts again.

In this work, we focus on solving the problem of how to stack blocks in order to produce pillars. The blocks are scattered, and some agents must cooperate in order to stack them correctly. The goal of our framework is to take the set of actions that the agents can do (in Prolog) with a natural language description of the initial and final states of the blocks, and exploit the LLMs to produce the remaining KB needed in order to correctly generate a task plan.

The framework we propose is depicted in Figure 1. It starts by creating a knowledge base defined in Prolog. While one could directly implement the KB in Prolog, the goal of the framework is to automatise the KB generation process through an LLM, such as GPT or LaMBDA, which is able to parse natural language descriptions (e.g., manuals) and provide correct actions and predicates to model the problem. Having such a feature would greatly increases the usability of the framework since it would be possible to specify in natural language the environment and the actions that the agents can perform. In this work, we have focused and tested the usage of LLMs to generate the initial and final states, whereas the actions where provided. More information on the knowledge base will be provided in Section III. Subsequentially, we use Prolog to compute a total-order plan based on snap actions. The plan is obtained by leveraging a forward search approach that progresses the initial state, taken from the KB, till the goal has been reached. The progess is carried out by applying the effects of actions and exploiting the symbolic reasoning provided by Prolog. Differently from what is commonly done in temporal planning, we do not use any heuristic to guide the search, which will be investigated in the future. As a result, the computed total order plan with the snap actions may not be optimal. While constructing the total-order plan, we also save all the states in which the chosen action had its preconditions satisfied and could hence be executed. We leverage this information to build a partial order of the actions for the computed total order plan, which in turn is encoded as a Disjunctive Temporal Network (DTN), which we then strengthen to an Simple Temporal Network (STN) by considering only the last achiever [22] of the action preconditions. To complete the STN, we also include constraints on the duration of each action (between the start snap action and its corresponding end [22]). We remark that the strengthening of the DTN to the STN is not achieved using Prolog (specifically, we implement it in Python). For more information about the planning steps, see Section IV. Then, the consistency of the computed STN is checked by examining whether there are negative cycles in the graph [24]. If there are, then we can ask Prolog to generate a new total-order plan and subsequently a new partial-order plan and STN. If this operation fails multiple times, then there might be an error inside our knowledge base, which should be verified and corrected accordingly, hence also improving the domain. Once the STN is constructed and verified, we convert it into a Behavior Tree (BT), a formalism that can then be directly executed by the robots. While executing the constructed BT, we are aware that unexpected exceptions or problems may arise since the KB used to compute the plan may be not aligned with the real world. In such a case, there are two possible solutions to the problem, either we refine the KB or we add constraints

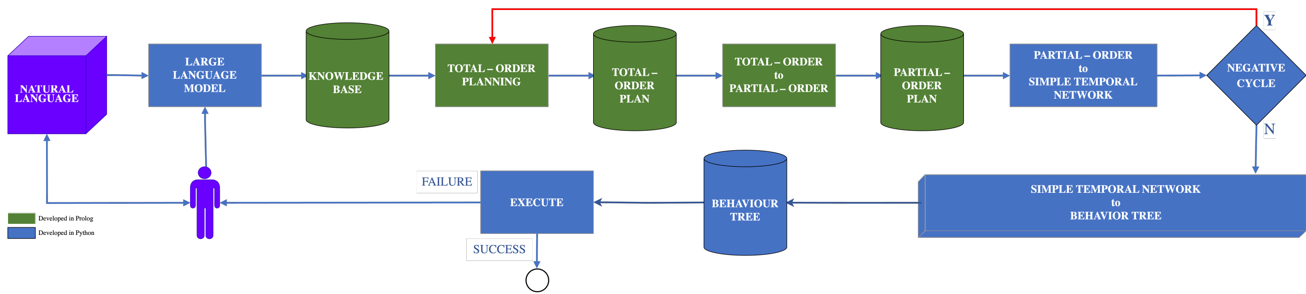

Fig. 1: The general diagram of the framework. The main idea is to create a feedback loop in different parts of the system to correctly modify the domain where needed and to learn from the environment.

on the agents so that the same situation does not arise again.

III. The Knowledge Base

In this section, we describe the general structure of the knowledge base and how LLMs are used to extract initial and final goals to expand the KB.

A. KB Structure

We employ SWI-Prolog [25] to easily create, modify, and query the knowledge base. We describe each state with a list of predicates, which define the position of the blocks and the status of the agents. For example, `ont(B, X, Y)` describes the fact that block `B` is on the table at coordinates `(X, Y)`, and `av(A)` states that agent `A` is available for use. Our KB is composed of predicates describing the states, and Prolog rules describing the actions, which have the following structure:

```
action(Name, ValidConditions, InvalidConditions,
       InvalidConditionsAtEnd, ConditionsOnKB, Effects).
```

Of which an example is here shown:

```
action(grip_ontable_start(Agent, Block),
       [ont(Block, X, Y), av(Agent), clr(Block)],
       [gripped(_, Block), gripping(_, Block)],
       [ont(Block, X, Y)],
       [],
       [del(av(Agent)), add(gripping(Agent, Block))]).
```

The variable `Name` defines the name of the action and its argument, e.g., for the action corresponding to the gripping of an agent `A` on a block `B`, we use `Name=grip(A,B)`. The four variables that follow are lists of conditions that must be checked before deciding whether to add the action or not:

- `ValidConditions` contains conditions that should be verified in the current state;
- `InvalidConditions` contains conditions that must not be verified in the current state;
- `ValidConditionsAtEnd` contains conditions that must not be verified in both the current state and the goal;
- `ConditionOnKB` contains conditions that must be verified on the knowledge base before deciding on the action.

The list `ValidConditionsAtEnd` checks if a condition from the goal has already been achieved and avoids actions which may make one of the contained conditions not to hold. `ConditionOnKB` is a list used to force the Prolog interpreter to ground the variables of the action to some values. The grounding predicates inside the KB are:

- `pos(X,Y)`, which indicates positions that may be used by the agents to temporarily store blocks;
- The predicates inside the goal state, which are added to avoid trivial and useless actions.

Indeed, the search in Prolog is not guided and if we were not to match the goal when choosing an action, the program may add useless actions such as the movement of a block to the same position it already is in.

Finally, `Effects` contains a list of predicates on how to modify the current state in a new state. Each predicate is in the form of either `add(P(...))`, which adds `P(...)` to the state, or `del(P(...))`, which looks for `P(...)` in the current state and removes it.

To query for a solution, we provide the initial and final states as input parameters to the `go` function. A detailed discussion of this function can be found in Section IV. An example query is the following one:

```
test1 :- go([av(ag1), ont(b1, 1, 1), clr(b1)],
            [av(ag1), ont(b1, 2, 2), clr(b1)]).
```

B. Large Language Models

While LLMs excel at learning complex patterns and information from vast training data, they primarily rely on statistical associations. They do not possess genuine inferential reasoning capabilities, and consequently, LLMs struggle when confronted with tasks different from the data they were trained on [15], [17]. Despite this, they can provide acceptable starting points for further refinements.

In our approach, LLMs are employed in order to generate the initial and goal states in Prolog for problems specified as natural language queries. LLMs are provided with queries involving multiple test cases. In the query configuration, there are multiple examples of test cases with explanatory comments and a problem description, as shown in Figure 2. These queries specify the desired initial and final states of the environment for the different examples. By setting the temperature parameter to zero, the stochasticity of the LLM's responses is minimized, providing more consistent outputs. Finally, we used GPT APIs to make tests on ChatGPT and GPT4, whereas we simply used the prompt-chat for Bard. The APIs were automatically called by running Python scripts and the results from all the LLMs were manually checked. Once the initial and goal states are generated, they are included into the KB.

```
Consider the following test cases.
Each of them moves a set of boxes (b1, b2, b3, ...) from an initial state to a final state using agents(ag1, ag2,..).
```
% from b1 at the point (2,2), b2 on the table at point (1,1) to b2,b1 stacked at point (3,3).
test1 :- go([av(ag1), av(ag2), av(ag3), ont(b1, 2, 2), ont(b2, 1, 1), clr(b1), clr(b2)],
            [av(ag1), av(ag2), av(ag3), ont(b2,3,3), on(b1, b2, 3, 3), clr(b1)]).
% from b2,b1 stacked to b1, b2 on the table.
test2 :- go([av(ag1), av(ag2), av(ag3), ont(b2,1,1), on(b1, b2, 1, 1), clr(b1)],
            [av(ag1), av(ag2), av(ag3), ont(b1,2,2), ont(b2, 3, 3), clr(b1), clr(b2)]).
% from b2,b1 stacked and b3 on the table to b1,b2,b3 stacked.
test3 :- go([av(ag1), av(ag2), av(ag3), ont(b2,1,1), on(b1, b2, 1, 1), clr(b1), ont(b3, 2, 2), clr(b3)],
            [av(ag1), av(ag2), av(ag3), ont(b1,3,3), on(b2, b1, 3, 3), on(b3, b2, 3, 3), clr(b3)]).
```
Can you generate a prolog code containing a new test case, namely test_case, in which we use 2 agents to move the boxes
b1, b2, on the table, which are at (1,1) and (2,2), respectively, to a final stack [b1,b2] at point (3,3), which is
ordered from top to bottom?
```

Fig. 2: Example of message used to query the LLM.

IV. Task Planning

In this section, we first describe how the total order plan is computed to solve the task planning problem. We then discuss how to extract a partial-order plan from the total-order one, followed by its transformation into an STN. Finally we outline how to obtain a BT from the STN.

A. Total-Order Plan

To compute a total-order plan we have developed the `plan` function (see Algorithm 1). This function recursively checks whether an action from the available action list can be scheduled for execution, by: i) verifying that the action's preconditions hold in the current state; and ii) assessing that none of the undesired predicates holds in the current state or in the final state. If both statements are met, `plan` applies the effects specified by the action, resulting in a new state. Subsequently, the function is invoked recursively to determine whether the current state matches the goal state. If the two states match, the process terminates and the computed plan is returned, otherwise, it continues to search for a viable plan. The function also checks that the depth of the recursion (length of the total order plan) does not exceed a given threshold by forcing a fail, which in turns triggers a backtrack to search for other solutions. Indeed, the search in Prolog is depth first and uninformed search, meaning that it may get stuck in a cycle of actions without reaching an optimal result. Indeed, this search may lead to very deep and time-consuming, albeit valid, plans, and to compensate for this, we have limited the depth of the recursion.

B. Partial-Order Plan and STN

To obtain the partial order plan we compare the preconditions of each newly chosen action with the effects of the previous actions. Indeed, while the chosen action was correctly added at a given moment since its preconditions were verified, this does not capture all the causality relationships between the different actions. What we want to capture are all the *achievers*, that is, actions whose effects allow the last added action to be executed. The goal of this step is to obtain a graph of temporal-causal relationships between the different actions so that an action can be executed only when, and as soon as, its preconditions are satisfied. Creating such a graph allows for concurrent actions to be carried out.

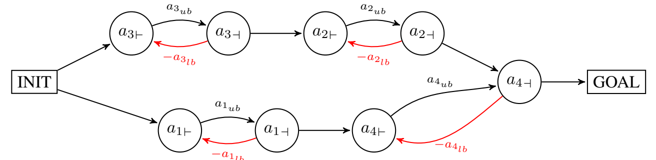

Fig. 3: In black, the graph representing the partial-order plan obtained from the total-order plan described in Table I. In red, the constraints on the durations to obtain the STN.

To correctly bind actions between each other, we split the actions into two snap actions, a starting and a terminating one, e.g., `move_block` becomes `move_block`$_\vdash$ (start) and `move_block`$_\dashv$ (end). In this way, we can safely state that another action can start only when the previous one is finished, e.g., a grip on a block can only start when the move on the same block has ended. Moreover, constraints on the duration of the actions have been put in place, that is, a terminating action cannot happen after a certain amount of time d_a from the starting of the action: $\mathtt{a}_\dashv - \mathtt{a}_\vdash \leq d_a$.

We create the above-mentioned graph by calling the `partial_order` function each time an action is added to the total order plan. Such a function takes the considered action and the list of previous actions and recursively checks if any of the preconditions of the chosen action is satisfied by the effects of another previous action. If it is, then there is a causal link between the two actions. Actions that do not have a causality relationship can be executed in parallel.

Consider the total-order plan shown in Table I. By applying the above function we obtain, for each action, a list of achievers that are needed for the pre-conditions of the action (Table II). From this list, we can construct a graph similar to the one shown in black in Figure 3, from which we can see that the two series of action can be run in parallel.

At this point, we want to obtain an STN from the partial-order. To do this, for each action, we keep only the earliest timestamp at which the action was executable. Moreover, we enforce the constraints on the duration of the action by inserting backward links of negative weight between the nodes. Once the STN has been built, we check that it is consistent, i.e., that there are no negative cycles. The

State	Action
`[av(ag1), av(ag2), ont(b1, 1, 1), ont(b2, 2, 2), clr(b1), clr(b2)]`	$a_{1\vdash}$: `grip(ag1, b2)`$_\vdash$
`[gripping(ag1, b2), av(ag2), ont(b1, 1, 1), ont(b2, 2, 2), clr(b1), clr(b2)]`	$a_{1\dashv}$: `grip(ag1, b2)`$_\dashv$
`[gripped(ag1, b2), av(ag2), ont(b1, 1, 1), ont(b2, 2, 2), clr(b1)]`	$a_{2\vdash}$: `move_block(ag1, b2, 2, 2, 3, 3)`$_\vdash$
`[moving(ag1, b2, 2, 2, 3, 3), av(ag2), ont(b1, 1, 1), clr(b1)]`	$a_{3\vdash}$: `grip(ag2, b1)`$_\vdash$
`[gripping(ag2, b1), moving(ag1, b2, 2, 2, 3, 3), clr(b1)]`	$a_{3\dashv}$: `grip(ag2, b1)`$_\dashv$
`[gripped(ag2, b1), moving(ag1, b2, 2, 2, 3, 3)]`	$a_{2\dashv}$: `move_block(ag1, b2, 2, 2, 3, 3)`$_\dashv$
`[gripped(ag2, b1), ont(b2, 3, 3), clr(b2), av(ag1)]`	$a_{4\vdash}$: `move_block(ag2, b1, 1, 1, 3, 3)`$_\vdash$
`[moving(ag2, b1, 1, 1, 3, 3), ont(b2, 3, 3), clr(b2), av(ag1)]`	$a_{4\dashv}$: `move_block(ag2, b1, 1, 1, 3, 3)`$_\dashv$
`[on(b1, b2, 3, 3), clr(b1), av(ag2), ont(b2, 3, 3), clr(b2), av(ag1)]`	

TABLE I: Table describing how the actions change the prior state. `av(ag`$_i$`)` states that agent ag$_i$ is available, `ont(b`$_i$`, X, Y)` states that block b$_i$ is in position `(X,Y)`, `on(b`$_i$`, b`$_j$`, X, Y)` states that block b$_i$ is on top of block b$_j$ in position `(X,Y)`. On the right, the total order plan to move from the initial state to the final one.

Action	$a_{1\vdash}$	$a_{1\dashv}$	$a_{2\vdash}$	$a_{2\dashv}$	$a_{3\vdash}$	$a_{3\dashv}$	$a_{4\vdash}$	$a_{4\dashv}$
Achievers	[]	$a_{1\vdash}$	$a_{1\dashv}$	$a_{2\vdash}$	[]	$a_{3\vdash}$	$a_{3\dashv}$	$a_{2\dashv}$, $a_{4\dashv}$

TABLE II: Achievers for the total order plan in Table I.

possibility of finding negative cycles comes from the fact that we consider a lower and an upper bound for the action duration, which increases the resilience of the final plan to possible delays in the real-world execution of the actions. The constraint on the duration $\mathrm{a}_\dashv - \mathrm{a}_\vdash \leq d_a$ becomes $l_a \leq \mathrm{a}_\dashv - \mathrm{a}_\vdash \leq u_a$, where l_a and u_a are the lower and upper bounds on the duration of action a, respectively. The final phase, involving the construction and validation of the STN, was done exploiting Networkx Python framework [26].

Algorithm 1 Pseudo-code for total and partial-order plan.

function PLAN($State$, $Goal$, $Been_list$, $Actions$, $Times$, $MaxDepth$)
  if $State = Goal$ **then**
    $print(Actions)$
  else
    $length(Actions) < MaxDepth$
    $choose_actions(Name, Preconditions, Effects)$
    $check_conditions(Preconditions, State)$
    $Child_state \leftarrow change_state(State, Effects)$
    if $Child_state \notin Been_list$ **then**
      $Stack(Child_state, Been_list)$
      $Stack(Name, Actions)$
      $partial_order(Preconditions, Been_list, Time)$
      recursively call $plan$
function PARTIAL_ORDER($Conditions$, $Action_list$, $Achievers$)
  for $Action_i \in Action_list$ **do**
    if $achiever(Conditions, Action_i)$ **then**
      $Achievers \cup \{i\}$

C. Behaviour Trees

The final step of the diagram shown in Figure 1 is the modelling of a BT from the STN. We will not delve into deep in the description of this step as this is not the main focus of the work and a complete explanation is available [28]. Starting from the root of the STN, we perform a deep-first search (DFS) of the network adding a sequential behaviour sub-tree each time that an action has only one exiting link, or a parallel behaviour sub-tree when the action has multiple exiting links. A parse action is executed before calling the function to create the BT in order to remove the backward links, allowing avoiding the insertion of waiting action and hence the creation of narrower BTs. Every time a start action is encountered during the DFS, a sub-tree in charge of starting the action is created, which checks the preconditions and applies the effects at the start. Similarly, every time an end action is met, the algorithm inserts a sub-tree, whose role is to apply the correct effects at the end.

V. Evaluation with an Example Application

Here we first show the application of the framework using an example application, which consists in creating a pillar by stacking blocks scattered on a surface (the same example has been used throughout this manuscript). Then, we evaluate the results obtained by using different LLMs to update the KB with new tests, and finally we describe how to link the outcome of the framework in a practical setup with 2 robotic arms in a co-manipulation task in simulation.

A. Multi-Robot Manipulation Setup in Simulation

We provide a KB containing the number of agents and the actions that they can do, as described in Section III. In order to obtain a valid plan, i.e., a series of actions that let us move from the initial state to the goal state, we need to specify the initial and final states. These states were generated by an LLM in response to a query in the format represented by Figure 2. Moreover, we provide a natural language description of the current state of the the environment and of what we want the state to be at the end. An example of the query is show in Figure 2, and a response obtained from GPT3.5 is shown in Figure 5.

Once we have the complete KB, we can use the SWI-Prolog interpreter to solve the problem and find a plan. First we compute a total-order plan (right of Table I) in which all the actions are executed sequentially. Given the correctness of the KB, the plan is valid and consistent since it was obtained through inference steps using Prolog. Then, we extract the causal relationships between the different actions by checking the achievers and produce a partial order plan (the black graph in Figure 3). At this point, we can obtain the STN by considering the constraints on the actions duration as negative weighted backwards edges of the graph (in red in Figure 3) and finally, we can extract a BT, as shown in Figure 6, which can then be executed.

The BT was then used to coordinate the movements of two simulated robotic arms by requesting the robot's motion according to the resulting schedule. Two different robots, an UR5e and an UR3e from Universal Robots, are included in the Gazebo-based physics simulator setup. The UR5e is adjusted upside down and attached to a workbench, while the UR3e is mounted on its custom base and it faces the

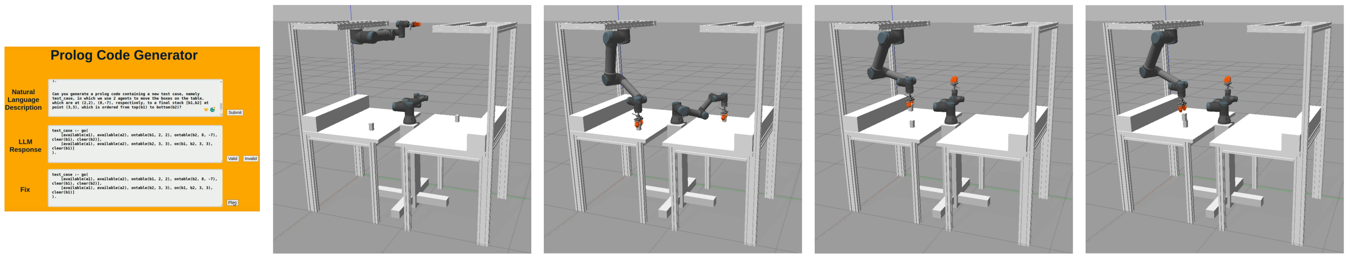

Fig. 4: (Left) The UI in which we specify the problem we want to solve. (Right) The Gazebo simulation environment. The arms work together to stack two blocks one on top of the other. A video of the experiment is available at [27].

```
test_case :- go( [av(a1),av(a2),av(a3),ont(b1,1,1),ont(b2,2,2),ont(b3,3,3),ont(b4,4,4),clr(b1),clr(b2),clr(b3),clr(b4)],
                 [av(a1),av(a2),av(a3),on(b1,b2,5,5),on(b2,b3,5,5),on(b3,b4,5,5),ont(b4,5,5),clr(b1)]).
```

Fig. 5: An example response from GPT3.5 given the query in Figure 2.

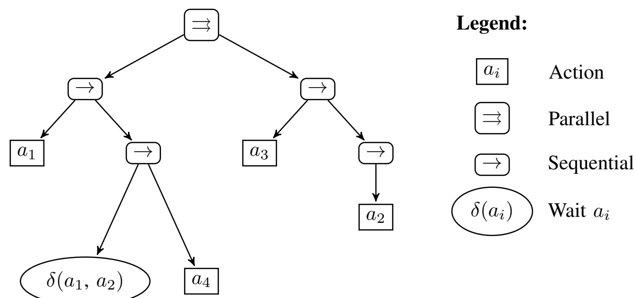

Fig. 6: The BT for the STN obtained in Figure 3.

workbench, as shown in Figure 4. In order to pick and place objects, both robots mount a SoftRobotics two-finger gripper. During the task execution (video available in [27]), the manipulators perform a pick-and-place task to stack elementary megablocks in defined positions, one on top of the other, to obtain the required pillar. While the two megablocks are gripped simultaneously by the two arms, since they are independent actions, the UR5e waits for the completion of the UR3e block placement before completing its own, as the placement of the megablock by the UR3e is a precondition for the placement of the megablock by the UR5e. In this way, we demonstrate the capability of the architecture to produce temporally consistent plans from a KB, whose initial and final states are generated by a LLM, and then orchestrate them by means of a BT.

B. LLMs Evaluation

We evaluate the performance of 3 LLMs, namely GPT-3.5, GPT-4, and BARD. We used GPT APIs to make tests on ChatGPT and GPT4, whereas we simply used the prompt-chat for Bard. The APIs were automatically called by running Python scripts. We designed 10 different scenarios, where the framework was assigned the task of picking and placing a number of boxes ranging from 3 to 5, with a number of manipulators varying from 2 to 4. In the experiments, the LLMs operate under identical configurations. As depicted in Table III, GPT-4 commits the least number of errors. The most frequent mistake made by LLMs is stacking boxes in the wrong order at the correct coordinates. For instance, the query shown in Figure 2 asks to stack block `b1` above `b2`, but this may be misinterpreted by the LLM, which instead produces the opposite predicates: {`ont(b1, 3, 3), on(b2, b1, 3, 3)`} The majority of the mistakes occur in the final states.

	# of predicates (avg)	# of literals (avg)	# of error in predicates (avg)	# of error in literals (avg)	success rate
BARD	16.5	33.4	2.4	5.7	0.4
GPT-3.5	16.5	33.4	1.3	3	0.4
GPT-4	16.5	33.4	0.6	1.6	0.8

TABLE III: Large Language Model Evaluation.

VI. Conclusions and Future Work

In this paper, we have shown a robot-oriented knowledge representation and planning system based on Prolog. A key feature of the approach is its effective integration with LLMs, which simplifies the generation of the initial and final states for the KB from an informal textual description, and a bumpless procedure that produces an executable plan to orchestrate the parallel operations of a set of robotic agents.

Many issues remain open and will require future investigations. The most important research directions that we intend to follow are: 1) the expansion of the use of LLMs to generate the whole KB, including also the list of actions, and to automate the detection and correction procedure of errors and logical inconsistencies; 2) the integration of probabilistic clauses that can be associated with uncertain events or perceptions (e.g., "This could be a hammer with 0.65 probability"), 3) the dynamic generation of new clauses and facts when the system comes across an unmodelled aspect of its operation domain (e.g., object too heavy to be lifted by one arm), 4) the optimisation of the STN or of the BT focusing on the temporal span, flexibility and resilience, and 5) the improvement of the search for the total plan guiding it with an heuristic.

References

[1] D. Paulius and Y. Sun, "A survey of knowledge representation in service robotics," *Robotics and Autonomous Systems*, vol. 118,

pp. 13–30, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889018303506

[2] J. Crespo, J. C. Castillo, O. M. Mozos, and R. Barber, “Semantic information for robot navigation: A survey,” *Applied Sciences*, vol. 10, no. 2, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/2/497

[3] M. Tenorth and M. Beetz, “Knowrob: A knowledge processing infrastructure for cognition-enabled robots,” *The International Journal of Robotics Research*, vol. 32, no. 5, pp. 566–590, 2013. [Online]. Available: https://doi.org/10.1177/0278364913481635

[4] T. C. Son, E. Pontelli, and N.-H. Nguyen, “Planning for multiagent using asp-prolog,” in *International Workshop on Computational Logic in Multi-Agent Systems*. Springer, 2009, pp. 1–21.

[5] C. Bitter, D. A. Elizondo, and Y. Yang, “Natural language processing: a prolog perspective,” *Artificial Intelligence Review*, vol. 33, pp. 151–173, 2010.

[6] A. Lally and P. Fodor, “Natural language processing with prolog in the ibm watson system,” *The Association for Logic Programming (ALP) Newsletter*, vol. 9, p. 2011, 2011.

[7] OpenAI et al., “Gpt-4 technical report,” 2023.

[8] OpenAI, “Introducing chatgpt,” http://web.archive.org/web/20231031083802/https://openai.com/blog/chatgpt/, accessed: 2023-10-31.

[9] R. Anil *et al.*, “Palm 2 technical report,” 2023.

[10] A. Vaswani, N. M. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *NIPS*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:13756489

[11] M. Shanahan, “Talking about large language models,” 2023.

[12] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, “A survey of large language models,” 2023.

[13] B. Meskó and E. J. Topol, “The imperative for regulatory oversight of large language models (or generative ai) in healthcare,” *npj Digital Medicine*, vol. 6, no. 1, p. 120, 2023.

[14] A. Levy and E. Karpas, “Understanding natural language in context,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 33, no. 1, pp. 659–667, Jul. 2023. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/27248

[15] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati, “Large language models still can’t plan (a benchmark for llms on planning and reasoning about change),” 2023.

[16] T. Silver, V. Hariprasad, R. S. Shuttleworth, N. Kumar, T. Lozano-Pérez, and L. P. Kaelbling, “PDDL planning with pretrained large language models,” in *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022. [Online]. Available: https://openreview.net/forum?id=1QMMUB4zfl

[17] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, “Code as policies: Language model programs for embodied control,” 2023.

[18] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, “A survey of behavior trees in robotics and ai,” *Robotics and Autonomous Systems*, vol. 154, p. 104096, 2022.

[19] M. Ghallab, D. S. Nau, and P. Traverso, *Automated planning - theory and practice*. Elsevier, 2004.

[20] W. Cushing, S. Kambhampati, Mausam, and D. S. Weld, “When is temporal planning really temporal?” in *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, M. M. Veloso, Ed., 2007, pp. 1852–1859. [Online]. Available: http://ijcai.org/Proceedings/07/Papers/299.pdf

[21] A. Coles, M. Fox, K. Halsey, D. Long, and A. Smith, “Managing concurrency in temporal planning using planner-scheduler interaction,” *Artif. Intell.*, vol. 173, no. 1, pp. 1–44, 2009.

[22] A. J. Coles, A. Coles, M. Fox, and D. Long, “COLIN: planning with continuous linear numeric change,” *J. Artif. Intell. Res.*, vol. 44, pp. 1–96, 2012. [Online]. Available: https://doi.org/10.1613/jair.3608

[23] M. Fox and D. Long, “PDDL2.1: an extension to PDDL for expressing temporal planning domains,” *J. Artif. Intell. Res.*, vol. 20, pp. 61–124, 2003. [Online]. Available: https://doi.org/10.1613/jair.1129

[24] L. Hunsberger and R. Posenato, “Simple Temporal Networks: A Practical Foundation for Temporal Representation and Reasoning,” in *28th International Symposium on Temporal Representation and Reasoning (TIME 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), C. Combi, J. Eder, and M. Reynolds, Eds., vol. 206. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 1:1–1:5. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14777

[25] SWI-Prolog. (2023) SWI-Prolog. Accessed on 13/09/2023. [Online]. Available: https://www.swi-prolog.org/

[26] NetworkX. (2024) NetworkX. Accessed on 13/02/2024. [Online]. Available: https://networkx.org/

[27] Interdep. Institute of Robotics (IDRA) - UNITN. (2023) A new approach for managing knowledge and planning in robotic applications . Accessed on 13/02/2024. [Online]. Available: https://youtu.be/zNF1T8efGW0

[28] J. Zapf, M. Roveri, F. Martín, and J. C. Manzanares, “Constructing behaviour trees from pddl temporal plans,” Tech. Rep., 2023.