ABSTRACT

Title of Dissertation:	TRANSLATING NATURAL LANGUAGE TO VISUALLY GROUNDED VERIFIABLE PLANS
	Angelos Mavrogiannis Doctor of Philosophy, 2025
Dissertation directed by:	Prof. Yiannis Aloimonos Department of Computer Science

To be useful in household environments, robots may need to understand natural language in order to parse and execute verbal commands from novice users. This is a challenging problem that requires mapping linguistic constituents to physical entities and at the same time orchestrating an action plan that utilizes these entities to complete a task. Planning problems that previously relied on querying manually crafted knowledge bases can now leverage Large Language Models (LLMs) as a source of commonsense reasoning to map high-level instructions to action plans. However, the produced plans often suffer from model hallucinations, ignore action preconditions, or omit essential intermediate actions under the assumption that users can infer them from context and prior experience. In this thesis, we present our work on translating natural language instructions to visually grounded verifiable plans.

First, we motivate the use of classical concepts such as Linear Temporal Logic (LTL) to verify LLM-generated action plans. Our key insight is that combining a source of cooking domain knowledge with a formalism that captures the temporal richness of cooking recipes could enable the extraction of unambiguous, robot-executable plans. Building on this insight, we present Cook2LTL, a system that receives a cooking recipe in natural language form, reduces high-level cooking actions to robot-executable primitive actions through the use of LLMs, and produces unambiguous task specifications written in the form of Linear Temporal Logic (LTL) formulae. By expressing action plans in a formal language notation that adheres to a set of rules and specifications, we can generate discrete robot controllers with provable performance guarantees.

Second, we focus on grounding linguistic instructions to visual sensory information and we find that Vision Language Models (VLMs) often struggle with identifying non-visual attributes. Our key insight is that non-visual attribute detection can be effectively achieved by active perception guided by visual reasoning. To this end, we present a Perception-Action API that consists of perceptual and motoric functions. When prompted with this API and a natural language query, an LLM generates a program to actively identify attributes given an input image.

Third, we present NL2PDDL2Prog, a system that incorporates the Planning Domain Definition Language (PDDL) as an action representation as a means to combine the ability of LLMs to decompose a high-level task to a set of actions with the correctness of symbolic planning. Prior work has often relied on manually crafting PDDL domains, which can be a difficult and tedious process, especially for non-experts. To circumvent that, we obtain visual observations before and after the execution of an admissible action in our environment. We pass them to a VLM to derive the action semantics which are then sent to an LLM to infer the entire domain. Given the generated domain and an initial visual observation of the scene, the LLM can produce a PDDL problem description that is then solved by a symbolic planner and parsed into an executable python program. By binding the perceptual functions to action preconditions and effects explicitly modeled in the PDDL domain, we visually validate successful action execution at runtime, producing visually grounded verifiable action plans.

To demonstrate the applicability of our work in the real world, we design a ROS-powered robotic system capable of receiving natural language instructions and implementing simple cooking recipes on a kitchen counter. We begin by bootstrapping a proof-of-concept system where each object has an ArUco marker on it to facilitate tracking. At runtime, our system receives a natural language instruction, calls Cook2LTL or NL2PDDL2Prog and passes the produced action plan to a python Pick-and-Place API that we developed for recipe execution on a Sawyer robot. We include demonstrations of experiments we conducted on the simple recipe of making a burger using artificial food items in the laboratory.

To conclude, we discuss ongoing and future work on improving our existing systems. We plan to incorporate object affordances in the safeguarding formalisms we have used to verify LLM plans. This can be achieved by introducing a more fine-grained action representation to support lower-level primitive actions and produce affordance-aware policies. We also focus on supporting contact-rich manipulation tasks such as grasping delicate and deformable items that are not only ubiquitous in the kitchen but in other domains, too. By leveraging visual context, textual descriptions, and feedback from tactile sensors, we could learn a mapping from the visual and textual space to the amount of current required for compliantly grasping delicate objects. Finally, we are working on extending the tracking functionality of our robotic system by incorporating Deep Object Pose Estimation (DOPE) to track objects of known 3D models from the YCB dataset, without the need of markers.

TRANSLATING NATURAL LANGUAGE TO VISUALLY GROUNDED VERIFIABLE PLANS

by

Angelos Mavrogiannis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2025

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACM	Allowed Collision Matrix
AI	Artificial Intelligence
API	Application Programming Interface
ArUco	Augmented Reality University of Cordoba
BERT	Bidirectional Encoder Representations from Transformers
BLIP	Bootstrapping Language-Image Pre-training
CLIP	Contrastive Language-Image Pre-training
CNN	Convolutional Neural Network
DH	Denavit-Hartenberg
DL	Deep Learning
DoF	Degrees of Freedom
DOPE	Deep Object Pose Estimation
GLIP	Grounded Language-Image Pre-training
GPT	Generative Pre-Trained Transformer
GPU	Graphics Processing Unit
HRI	Human-Robot Interaction
KLT	Kanade-Lucas-Tomasi
LiDAR	Light Detection and Ranging
LLM	Large Language Model
LSTM	Long Short-Term Memory
LTL	Linear Temporal Logic
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multilinear Perceptron
NER	Named Entity Recognition
NLP	Natural Language Processing
NMN	Neural Module Network
OVD	Open-Vocabulary Object Detection

PDDL	Planning Domain Definition Language
PID	Proportional Integral Derivative
POS	Part of Speech
RAG	Retrieval-Augmented Generation
RGB	Red Green Blue
RGBD	Red Green Blue Depth
RL	Reinforcement Learning
RNN	Recurrent Neural Network
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
SSH	Secure Shell
STL	Signal Temporal Logic
TEC	Theory of Event Coding
TF	Transformation
VHB	Very High Bond
VLM	Vision Language Model
VQA	Visual Question Answering
VR	Virtual Reality
YAML	Yet Another Markup Language
YCB	Yale-CMU-Berkeley

Chapter 1: Introduction

Robots have been traditionally and predominantly inhabitants of research laboratories with the exception of some well-monitored environments where they are usually tasked with completing a set of very specific and well-defined tasks. Examples of these tasks include manipulating and transporting goods in warehouses, executing specific pre-defined tasks as part of production lines in factories, patrolling or monitoring areas of interest, and performing intricate tasks to assist humans, often relying on teleoperation, such as robot (-assisted) surgery. These tasks are often tackled by planning, perception, and control systems that are designed to operate under precise conditions and yield relatively predictable outcomes, they usually lack support for any natural language interface between the user and the system, and may require trained engineers using sophisticated software [5] to operate, debug, and repair.

The advent of deep learning - and more specifically Large Language Models (LLMs) - has revolutionized the field of robotics. Robots that once relied on the tireless efforts of skilled programmers, typically graduate students running on little sleep, are gradually transitioning to helpful assistants capable of communicating with non-expert users and aspiring to become valuable members of our households. To be useful in household environments, robots may need to understand and execute instructions from novice users who are agnostic about their underlying technology and have no prior programming experience or ROS (Robot Operating System) communication skills. Although we are still far from an incarnation of a functional anthropomorphic robot butler who is capable of handling open-ended natural language queries, voice assistants are widely being used to answer everyday queries, such as describing the local weather, by querying APIs and accessing sources on the web at real time.

Such voice assistants can potentially connect to household appliances, creating the illusion of artificially intelligent home assistants by being capable of parsing verbal commands and completing simple tasks. Examples of these tasks include turning the lights on, setting a specific temperature, dimming the lights in a room, or deploying a robotic vacuum to wipe or mop the floor. Reliable systems that support such functionalities can often look impressive, although the underlying implementation reduces to a speech-to-text module mapped to a simple control command.

LLMs have demonstrated remarkable results on handling free-form natural language queries without the need of any task-specific training but rather by exclusively relying on the general pre-training process on vast internet corpora and on the autoregressive token generation at runtime. Consider the simple case where a user instructs a robot to make a sandwich in the kitchen. This would require parsing and adapting the input utterance to specified templates. For example, a hypothetical system could only support queries of the form "*make a* _____ *sandwich*" where the type of sandwich would probably have to be picked from a fixed set e.g. $\{$*chicken, BLT, grilled cheese*$\}$. An utterance deviating from the expected query format would not work based on this template. Furthermore, plan generation would have to rely on a manually-crafted knowledge base or recipe book, mapping the supported sandwich types to sequences of actions the robot would need to take. The lack of a key ingredient could mean that the recipe is impossible to execute. Such systems are rigid both when it comes to the constrained supported format for the user's command, and in terms of their limited ability to adapt the steps of the plan to the characteristics of a specific scene. Even if a system afforded querying the web for a recipe, some of the actions recovered from a recipe could be exclusively intended for a human to execute and therefore mapping them to robot actions without explicit pre-programming could be difficult or not possible at all. LLMs can operate as a real-time mechanism of commonsense reasoning to bridge these gaps by interpreting any reasonable input utterance, adapting the generated plan to the specifics of the scene, and providing a means to translate free-form actions to a limited vocabulary of admissible actions that a robot can actually support. Naturally, this implies that the input LLM prompt includes an accurate textual description of the scene, resulting from entities perceived by robot sensors, and a set of admissible robot actions.

LLMs possess the expressiveness and commonsense reasoning to decompose high-level task descriptions expressed in natural language to lower-level action plans. One of the key mechanisms that has found success in breaking down tasks to admissible robot actions has been few-shot prompting. In few-shot prompting the LLM is prompted with a few examples of input-output pairs to bias the answer structure of the model so that the output for a new query imitates the pattern of the example outputs. However, even in that case, LLM output comes from autoregressive token generation, predicting the next word that has the highest probability out of an available vocabulary, as described in Section 3.2. Consequently, as they have been exclusively exposed to linguistic signals, these models lack connection to the physical world and therefore the generated action plans often omit essential intermediate steps, assuming prerequisite knowledge and experience that robotic systems do not possess. Additionally, these plans might include redundant actions or exhibit the so-called hallucinations, where the model introduces out-of-context objects or unsupported actions. To limit the space of possible next-word predictions to an admissible set and constrain LLM output to executable plans, prior work has integrated LLMs with formalisms that are more well structured and rigorous than free-form natural language [3, 6, 7]. This thesis builds upon these formalisms, expressing actions and plans as computer programs, and using LLMs as an interface to translate unstructured natural language to formal logic and classical AI planning representations that provide provable guarantees on the feasibility of the derived plans. More specifically, Chapter 3 introduces Cook2LTL, a system that parses natural language instructions into Linear Temporal Logic (LTL) task specifications, and Chapter 5 presents a system that infers a PDDL domain through visual demonstrations before and after action execution, visually grounding action preconditions and effects.

At the same time, having only been exposed to linguistic signals, LLMs are "*blind*", and therefore need to ground linguistic constituents to physical entities perceived by robots. Vision Language Models (VLMs) incorporate the additional modality of vision, but they often struggle in spatial and physical reasoning tasks, or when dealing with identifying non-visual object attributes. Modeling actions as programs benefits from the code synthesis and commonsense reasoning abilities of LLMs and can guide attribute detection by invoking active perception behaviors through few-shot prompting-inspired schemes. In Chapter 4 we implement a system that follows this paradigm both in simulation and in the real world. The following section analyzes the benefits of modeling actions as programs, aligned with our philosophy to bridge the gap between perception and planning, and highlights several unexplored directions that arise from this type of modeling.

1.1 Actions as Programs

Perception and Action: The problems of planning and perception were traditionally studied separately by different communities. The modeling of the planning problem occurred on a symbolic level where many mathematical frameworks such as behavior trees, parse trees, graphs, and other mechanisms have been employed to represent the plan of an action [8]. Integrating visual feedback from the environment into a planning problem is challenging, and in practice often involves the underlying assumption that perception is represented as a black box that continuously delivers 3D descriptions of the world with a number of labels attached to them. However, this distinction between perception and planning as two separate processes has not been helpful for the evolution of the field of robotics. As the plan of the action is unraveling, a number of perceptual tests is required at various stages of the computation. Some of these tests ensure that action pre-conditions are satisfied, recognize the current stage of the action so that appropriate motions are generated by the control system, and track the progress of the robot towards achieving a goal. This highlights the fact that perception and action are intertwined, and that one cannot exist without the other in robotics. Based on these observations and inspired by the Theory of Event Coding (TEC) [9], we believe that the functional architecture supporting perception and action planning should be formed by a common representational domain for perceived events (perception) and intended or to-be generated events (action). Motivated by this idea, our insight is that linking actions and percepts by projecting them to a common programmatic representation space leverages the expressiveness of programming constructs and the commonsense reasoning capabilites of Large Language Models (LLMs), and enables us to reason towards executing complex tasks and generate perceptually-grounded action plans. Equipped with a set of logical reasoning constructs such as loops and conditionals, a program serves as an ideal representational medium [10], enabling function calls to Vision-Language Models (VLMs) [11, 12] to segment and classify objects of interest [13], and saving their output to intermediate variables that are consumable downstream. These intermediate variables illustrate a step-by-step explainable reasoning process towards the task at hand, often expressed in simple python pseudocode-resembling grammar [14]. The produced plans are interpretable and flexible in terms of incorporating any vision or language model API as a function call. At the same time, the generated programs benefit from the compositional power of LLMs to decompose high-level tasks into sub-tasks on a lower level of abstraction without the need of any additional training.

Commonsense Reasoning: The idea of modeling actions as programs has been explored in prior literature [15]. However, composing these programs previously required skilled programmers with domain knowledge, and the produced software was often inadvertently tied to a specific robot or application. The advent of LLMs and their ability to synthesize programs have revolutionized this idea. Leveraging a notion of artificial commonsense reasoning acquired from pre-training on vast internet corpora, these models have enabled new capabilities especially in the domain of robot task planning. Applications that previously relied on existing or manually-crafted knowledge bases to reason about task execution can now obtain such information by querying a language model on demand in real time. Experimental results have shown [3] that the power of LLMs can be leveraged more efficiently in robot task planning when actions and tasks are modeled as programs, compared to natural language representations. This occurs due to the vast presence of code that is publicly available on the web. Considering a causal language modeling objective where an LLM predicts the next word autoregressively, it is harder to reason in the practically infinite space of free-form natural language. On the other hand, programming logic can be found in abundance on the internet and is bound by stricter grammatical and syntactic rules. This makes programs a convenient medium for modeling problems and solving them through modular decision making and counterfactual reasoning through control flow tools and other expressive programming constructs.

Expressiveness: Control flow tools are the structural components of a programming language. Modeling actions as programs inherently facilitates counterfactual reasoning through the imaginary "*if...then...else*" statement. This statement enables decision making and planning that can be triggered by potential events, subsuming the functionality of previously used representations such as binary decision trees. Furthermore, a lot of tasks in robotics are inherently repetitive on a certain level of granularity. For example, a cooking robot instructed to stir a pot might perform the repetitive motion of rotating a spatula multiple times, which can be modeled with the construct of a loop. The action repetition can occur dynamically through a `while` loop with an appropriate stopping condition, or statically for a fixed number of times with a `for` loop. Additional programming concepts such as polymorphism are also applicable to robot planning applications and are currently underexplored. Overloading a function can map to different program implementations for the same task depending on the context and the robot perception of the environment. For example, imagine that we have access to an LLM planner that receives a function name describing a high-level task along with a set of appropriate parameters, and returns a program that implements this task using a set of actions that are admissible in the environment. Then, the program for implementing the function `cook(obj: pasta)` might map to a different implementation than the program for the function `cook(obj: pasta, loc: oven)` as we demonstrate in Chapter 3 [16], where `obj` and `loc` correspond to the direct object and location of the action, respectively. The former program will most likely assume that a pot is available and will generate a sequence of instructions that boil the pasta using a pot, following the most likely pattern seen in the data during the pre-training of the LLM. However, in the case of adding the oven as an extra parameter, the latter program will require cooking the pasta in the oven and will hence map to a different low-level implementation.

Distributability: There is a well-established and widely used functional infrastructure for software version control [17], enabling programmers to share, interactively modify, test, and deploy programs to build software applications. While there have been attempts on gathering crowdsourced programs for planning in robotics, they are focusing on a high-level prompt engineering layer of abstraction [18] which requires additional intermediate modules for mapping high-level plans to lower-level realizable robot controllers. Therefore, there are still exciting opportunities for future research on leveraging this existing infrastructure and adapting it to the programmatic action representation in downstream layers, mapping high-level programs to low-level robot control programs. We believe that this could be a promising direction towards building a large dynamic open-source repository of action programs that are hardware-agnostic and applicable among different platforms and applications.

Theoretical Benefits: One of the key ideas of the minimalist program in linguistics [19] is the optimality of the human language ability with its underlying components reducing to a very simple computation. From an evolutionary perspective, we view the development of action through the prism of language acquisition. Under this assumption, having access to a program that describes an action one can ask further questions about quantifying the optimality of the action as it exhibits itself through the complexity of the program, using a computational complexity metric such as the Kolmogorov complexity. Clearly, when thinking about the cognitive system of an agent, it is often required that this agent behaves in an optimal way and hence that the actions performed by the agent are orchestrated in such a way that the minimum amount of effort or energy is spent. The amount of energy spent by a program can theoretically be computed under the assumption that the cost of all operations is known. We can extend these thoughts about a single action to relationships between multiple actions. For example, a robot programmer can compose three different programs that all implement the same task with different sequences of actions. But which one is the best? What is "*best*" and what are the criteria? Some criteria could be the minimization or maximization of the contact of the robot with the world, or various other constraints and desiderata posed by the problem at hand. It is one thing to say that you can program the robot to perform an action and another for the robot to learn how to reason about which action is better, faster, slower or whatever the attribute is for the comparison. We believe that these ideas, driven by thinking of actions as programs, can give rise to new research directions in the field of robotics, both in theory, and in practice.

Chapter 2: Related Work

We begin our literature review by surveying robotic cooking (Sec. 2.1), which is our key domain for studying the translation of natural language to verifiable robot-executable plans. We continue by reviewing recent work in LLM planning (Sec. 2.2) and ways to validate LLM-generated plans using classical concepts like LTL or PDDL (Sec. 2.3). Finally, we look into attribute detection (Sec. 2.4) and visual reasoning with programs (Sec. 2.5) as they contribute towards our effort in invoking active perception behaviors through LLM-generated programs.

2.1 Robotic Cooking

Cooking has been an important means of studying action understanding [20–23]. The EU project POETICON [20] viewed cognitive systems as a set of languages {natural, visual, motoric} and integrated these languages towards understanding cooking actions. Along these lines, Yang et al. [21] processed YouTube videos using Convolutional Neural Networks (CNNs) and a grammatical approach [22] to produce parse trees that could be used for generating cooking actions. Several works have focused on modeling and learning the lower-level mechanics of manipulation in cooking actions with the end-goal of building intelligent aspiring robot chefs [24–26]. Bollini et al. [27], Beetz et al. [23], and Liu et al. [24] developed autonomous end-to-end robotic cooking systems tailored to the specific tasks of baking, stir-frying, and making pancakes, respectively. RoboCook [28] and Mobile ALOHA [29] recently introduced versatile learning-based systems that intricately handle cooking-specific complex soft body manipulation tasks and mobile manipulation tasks in the kitchen, respectively. Most of these works are constrained to preparing a unique dish [23, 24, 27], require significant adaptations and additional training [28] to support novel recipes, or cannot handle verbal instructions as input [26, 29]. Translating recipes to actionable robot plans is a challenging problem due to the linguistic richness and semantic ambiguity of recipes [30]. Recent approaches have built LLM-powered human-robot collaborative cooking interfaces equipped with a human intention prediction module [31, 32]. We leverage LLMs to decompose abstract free-form recipe text into executable robot plans [16]. Our AI2-THOR [2] simulation in Sec. 3.4.2 demonstrates the transferability of our work to a real robot while allowing the system to adapt to new recipes. In our ongoing work, proposed in Chapter 5, we incorporate perceptual evidence to visually ground action preconditions and postconditions at runtime.

2.2 LLM Planning

The field of robotics has been broadly and profoundly impacted by LLMs. Planning problems that previously relied on querying manually-crafted knowledge bases [33] can now leverage LLMs as a source of commonsense reasoning to map natural language commands to sequences of lower-level actions that can be easier to parse by robot controllers [34–36]. This has been achieved predominantly by few-shot prompting, a method that allows pre-trained LLMs to perform in-context learning, leveraging their pre-training on vast internet corpora and avoiding costly and time-consuming fine-tuning. In few-shot prompting the LLM receives a set of examples of sample tasks and action plans as input, and generates an action plan for an unseen task at inference time. These LLM-based works have shown great performance in grounding high-level actions to a well-defined set of actions for task planning but come with certain limitations. For instance, the framework of Ichter et al. [34] cannot handle open-vocabulary or combinatorial tasks, the one by Huang et al. [36] might produce action plans including items that are not present in the current environment, and the model of Huang et al. [35] does not guarantee that the returned actions are admissible in the current context.

To constrain the output to a given environment, subsequent works have leveraged the code synthesis ability of LLMs to project natural language to an intermediate representation of commands or function calls in the form of a computer program, bound by the more rigorous syntactic rules of programming languages [3, 10, 37]. More specifically, these works include action precondition checking through conditional and assertion statements [3], reasoning about task execution using control flow tools [37] and recursively defining undefined functions [10], or simply invoking VLMs in task execution functions [13]. In the context of cooking, Wang et al. [38] have used LLMs to break down high-level cooking actions into actionable plans. However, their approach requires access to demonstrations of the intermediate steps of the cooking task at hand. In our work [16], we adapt the methodology proposed by Singh et al. [3], where the task planning problem is formulated as a pythonic few-shot prompting scheme. The prompt consists of a pythonic import of a set of primitive actions, a definition of a list of available objects, and a few example task plans in the form of pythonic functions. Their experiments showed that prompting an LLM for task planning in a programmatic fashion outperforms verbose descriptive prompts by restricting the output plan to the constrained set of primitive actions and objects available in the current environment.

Furthermore, many of these approaches primarily focus on action planning and ground linguistic constituents to physical entities by using a single call to an Open-Vocabulary object Detection (OVD) model [11, 12, 39, 40]. Due to the known limitations of these models [41], they might not be able to handle attribute detection in challenging scenarios. Similar to [42], we combine the expressiveness of an intermediate programmatic representation and the complementary reasoning capabilites of LLMs and VLMs [43] to reason about attribute detection under an LLM-prompting scheme [44] invoking active perception robot policies.

2.3 Validating LLM Output with Safeguarding Mechanisms

While enforcing more rigor to the structure of the output by prompting the model to output a computer program, there are no guarantees on the validity of the program [3], which might require ad-hoc code reviewing to ensure compliance with the specifications of the environment. In long-horizon planning, such methods still suffer from model hallucinations, often ignore action pre-conditions, or omit essential intermediate actions under the assumption that users can infer them from context and prior experience [45]. To bridge this gap, a recent line of research has introduced formal logic and standardized formulations from classical AI planning as additional mechanisms to ensure plan validity before robot deployment [6, 7, 16, 46–56]. One of these mechanisms is Linear Temporal Logic (LTL) [57], which was initially used in formal verification for computer programs. Since then, it has been extensively used in robotics [58–60] as a formalism that enables the extraction of guarantees on robot performance given a robot model, a high-level description of its actions, and a class of admissible environments.

There has been considerable work on translating natural language instructions to task specifications in the form of LTL [46, 61–65] and its variants [48, 66] with the end-goal of using the output specifications to generate an automaton that validates LLM-generated plans [50, 51]. Most approaches try to address the main bottleneck which is the high cost of obtaining annotations of natural language with their equivalent LTL logical forms. Gopalan et al. [61] orchestrate a data collection and augmentation pipeline to build a synthetic domain and translate natural language to LTL formulae using Seq2Seq models [67]. Alternatively, Patel et al. [62], Wang et al. [63] learn from trajectories paired with natural language to reduce the need for human annotation, however a lot of trajectories are required to implicitly supervise the translator. Berg et al. [64], Liu et al. [65] ground referring expressions to a known set of atomic propositions and translate to LTL formulae using Seq2Seq models [68] and LLMs [69], respectively. Similarly, Pan et al. [46], Chen et al. [49] use the paraphrasing abilities of LLMs to generate synthetic datasets tackling the scarcity of labeled LTL data.

Our work [16] is more similar to the work of Chen et al. [49] and Hsiung et al. [70], abstracting natural language to an intermediate representation layer before grounding to the final atomic propositions. An important limitation of these methods is that they are based on thoroughly curated datasets or well-structured synthetic data generation pipelines. On the contrary, we deal with unstructured free-form recipe text scraped from the internet. Moreover, most of such works in embodied settings have mainly been applied to navigation and simple pick-and-place tasks or combinations of these. Our web-scraped cooking recipe corpus offers a richer and more diverse action space.

LTL can succinctly and elegantly express verbose recipes in the form of unambiguous task specifications, enabling us to effectively model temporally interconnected tasks. By incorporating LTL in our modeling, we also inherit the benefit of acquiring provable performance guarantees on the produced action plans when generating discrete robot controllers and using them in downstream motion planning systems. However, discrete synthesis requires generating an automaton. The computational complexity of generating such an automaton can be doubly exponential in the size of the formula, or potentially reduced to polynomial time when considering the special class of Generalized Reactivity(1), or GR(1), LTL formulae [59]. This can still be inefficient in long-horizon planning tasks. As a more human-readable specification language that has been widely used to enforce a standardized structure in long-horizon planning problems [71], PDDL [72] has been an alternative mechanism to validate LLM-generated plans. Recent work has been translating natural language to PDDL specifications [6, 52–56] via LLM few-shot prompting. However, many of these approaches often rely on feedback from interactions with simulators to evaluate predicates [73], which can be domain-specific or unrealistic, incorporating deterministic dynamics that do not accurately reproduce real-world scenarios. In contrast, in our work in Chapter 5 we automatically infer PDDL domains from real-world robot and human demonstrations. Moreover, we attach perceptual callback functions to action preconditions and postconditions and visually ground the corresponding predicates, inspired by the work of Migimatsu and Bohg [74].

PDDL offers a well-structured task representation that is convenient for modeling cooking activities by explicitly encoding action preconditions and effects in the domain. This is particularly useful for tracking the state change of an ingredient or food item. State changes can be phase transitions, for example butter melting on a pan, or changes in quantity in terms of the visible ingredients in the kitchen, such as the tomato slices resulting from slicing a tomato. Using this representation also allows us to leverage already existing PDDL infrastructure, such as VAL [75], a tool for syntactically validating generated PDDL files. Our approach is similar to the work of Shirai et al. [76] but they only leverage visual input at the initial stage to acquire a list of available objects or in the case of plan failure. By incorporating perceptual feedback to ground action preconditions and effects between every action, our proposed framework is applicable to a wider variety of kitchen tasks of higher complexity.

A significant amount of work has focused on using LLMs to generate the intermediate problem specifications which they then solve with symbolic planners. Some of these works assume that the domain file is known a priori or manually defined [76, 77]. However, the difficulty of manual domain crafting scales with the complexity of the domain and could be tedious, time-consuming, and erroneous, especially for non-experts. Zhu et al. [73] present an approach for domain induction that requires an iterative process of interacting with the environment and obtaining feedback, as well as a lot of information in a lengthy input prompt, similar to the approach presented by Guan et al. [6]. Liu et al. [78] proposed BLADE, a framework for long-horizon robotic manipulation that learns action semantics along the way. However, our approach infers PDDL domains directly from real-world image pairs using VLMs, requiring no language annotations, symbolic state labels, or simulation. In contrast, BLADE uses language-annotated demonstrations and LLMs to generate symbolic behavior descriptions, relying on contact-based segmentation, predefined predicate names, and classifier training for visual grounding.

2.4 Attribute Detection

Attribute detection has been a fundamental problem in the computer vision community with early work [79–82] on learning visual attribute classifiers to describe unseen objects. There has been work on identifying relative attributes [83, 84] but these approaches require prior training and can be limited to a certain domain ([84] demonstrate relative attributes of shoes and facial characteristics), while in Chapter 4 we show a general method [44] that works in a zero-shot fashion. Recent approaches have applied Open-Vocabulary object Detection (OVD) task [41, 85] to attribute detection. The goal in OVD is to detect unseen classes of objects defined at inference time in the form of textual queries. Bravo et al. [41] showed that the performance of various VLMs in zero-shot attribute detection is still low compared to OVD. However, most of these approaches focus on visually-perceivable attributes in disembodied settings. On the other hand, we focus on embodiment-crucial attributes such as the weight of an object, leveraging the physical reasoning capabilities of VLMs [86]. Our simulations in AI2-THOR and our robot demonstration (See Fig. 4.1) shows that our end-to-end framework can invoke active perception behaviors to reason about object attributes, inspired by prior work [87, 88].

2.5 Visual Reasoning with Programs

Generating and executing programs for vision applications originated from Neural Module Networks (NMNs) [89–91], on the basis of the idea that complex vision tasks are fundamentally compositional. Motivated by this idea, NMNs decompose a task into trainable modules that learn specific perceptual functions. However, these models produce domain-limited programs, rely on hand-tuned parsers [89] or are difficult to optimize [90, 91]. To overcome these shortcomings, a recent line of work has proposed a formulation of generating visual programs to deal with image-based natural language queries through in-context learning with an LLM. The programs consist of pseudocode instructions [92] or executable python code [14, 93, 94] and intermediate variables that map to computer vision models, image processing subroutines, or LLMs. These intermediate variables are consumable downstream and illustrate a step-by-step reasoning process towards the task at hand, which is primarily related to language grounding or Visual Question Answering (VQA). Our work (Chapter 4 [44]) invokes active perception robot behaviors guided by visual programming towards attribute detection.

Chapter 3: Translating Natural Language to Verifiable Plans with Large Language Models

3.1 Introduction

To be useful in household environments, robots may need to understand and execute instructions from novice users. Natural language is possibly the easiest way for users to provide instructions to robots but it is often too vague. This motivates the need for mapping natural language to actionable, robot-executable commands. This is a challenging problem, especially for complex activities that include temporally correlated subtasks, such as following instructions in a manual, or performing a delicate assembly task.

In this paper, we focus on translating cooking recipes into executable robot plans. Cooking is one of the most common household activities and poses a unique set of challenges to robots [27]. It usually requires following a recipe, written assuming that the reader has some background experience in cooking and commonsense reasoning to understand and complete the instruction steps. Recipes often feature ambiguous language [30], such as omitting arguments that are easily inferred from context (the known "Zero Anaphora" problem [95]; see Fig. 3.3b where the direct object of the verb "cook" is missing), or, more crucially, underspecified tasks under the assumption that the reader possesses the necessary knowledge to fill in the missing steps. For example, recipes with eggs do not explicitly state the prerequisite steps of cracking them and extracting their contents. Additionally, although inherently sequential, recipes often include additional explicit sequencing language (e.g. until, before, once) that clearly defines the temporal action boundaries.

Motivated by these observations, our key insight is that combining a source of cooking domain knowledge with a formalism that captures the temporal richness of cooking recipes could enable the extraction of unambiguous, robot-executable plans. Our **main contribution** is Cook2LTL, a system that receives a cooking recipe in natural language form, reduces high-level cooking actions to robot-executable primitive actions through the use of LLMs, and produces unambiguous task specifications written in the form of LTL formulae (See Fig. 3.1). These plans are then suitable for use in downstream robotic tasks. We build and evaluate our method based on a subset of recipes from the Recipe1M+ corpus [1]. We run Cook2LTL on these recipes and show that by caching the action reduction policy, we incrementally build a queryable action library and limit proprietary LLM API calls with significant benefits in cost (-42%) and computation time (-59%) compared to a baseline that queries the LLM for every unseen action at runtime. We demonstrate the transferability of Cook2LTL to a robotic platform through experiments in a simulated kitchen in AI2-THOR [2].

3.2 Preliminaries

This section provides a short background on LTL and LLMs, which are the tools we are using in our pipeline.

3.2.1 Linear Temporal Logic

LTL is a temporal logic that was developed for formal verification of computer programs through model checking [57]. It is suitable for expressing task specifications and verifying system performance in safety-critical applications. These task specifications are expressed through the use of this grammar:

$$\phi ::= p \,|\, \neg p \,|\, \phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2 \,|\, \mathcal{G}\, \phi \,|\, \mathcal{F}\, \phi \,|\, \phi_1 \mathcal{U}\, \phi_2 \tag{3.1}$$

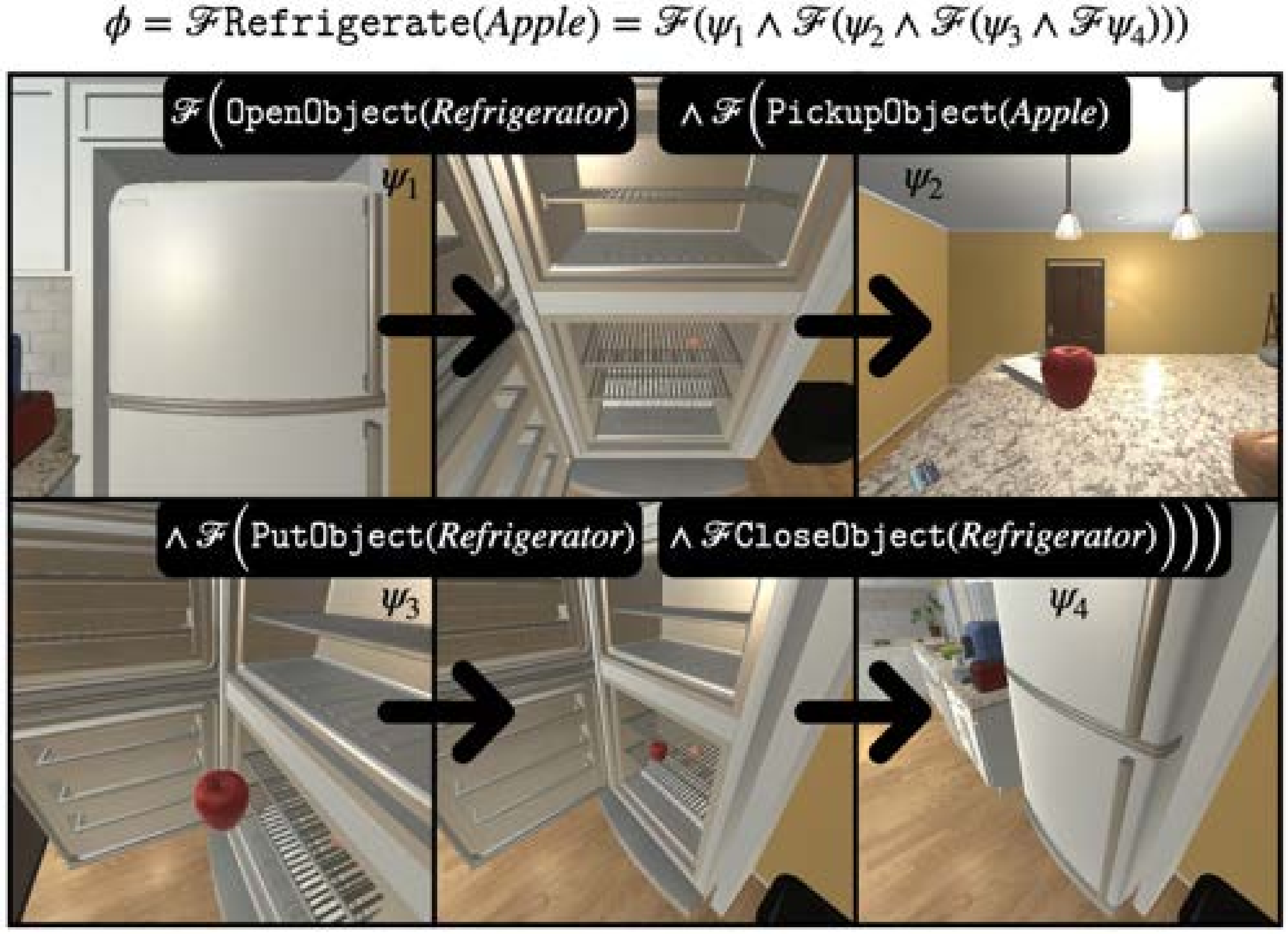

$\phi = \mathscr{F}\texttt{Refrigerate}(Apple) = \mathscr{F}(\psi_1 \wedge \mathscr{F}(\psi_2 \wedge \mathscr{F}(\psi_3 \wedge \mathscr{F}\psi_4)))$
$\mathscr{F}\Big(\texttt{OpenObject}(Refrigerator)$
$\wedge \mathscr{F}\Big(\texttt{PickupObject}(Apple)$
ψ_1
ψ_2
$\wedge \mathscr{F}\Big(\texttt{PutObject}(Refrigerator)$
$\wedge \mathscr{F}\texttt{CloseObject}(Refrigerator)\Big)\Big)\Big)$
ψ_3
ψ_4

Figure 3.1: **Cook2LTL in AI2-THOR** [2]: The robot is given the instruction *Refrigerate the apple*. Cook2LTL produces an initial LTL formula ϕ (*top left*); then it queries an LLM to retrieve the low-level admissible primitives for executing the action; finally it generates a formula consisting of 4 atomic propositions $(\psi_1, \psi_2, \psi_3, \psi_4)$ that provide the required task specification and yield these consecutive scenes.

where ϕ is a task specification, ϕ_1 and ϕ_2 are LTL formulae, and $p \in$ is an atomic proposition drawn from a set $\mathcal{P}$ of atomic propositions (APs). $\neg, \wedge, \vee$ are the known symbols from standard propositional logic denoting negation, conjunction, and disjunction, respectively. As an extension, LTL supports additional temporal operators. More specifically, $\mathcal{G}\,\phi$ denotes that ϕ holds globally, $\mathcal{F}\,\phi$ denotes that ϕ must eventually hold, and $\phi_1\,\mathcal{U}\,\phi_2$ indicates that ϕ_1 must hold for all time steps until ϕ_2 becomes true for the first time. In this work, we utilize LTL as a formal language to express temporally-extended cooking tasks.

3.2.2 Large Language Models

Given a piece of text $W = \{w_1, w_2, \ldots, w_n\}$ consisting of n words $w_i, i = 1, \ldots, n$, a language model estimates the probability $p(W)$. This is done in an auto-regressive manner, leveraging the chain rule to factorize the probability [96]:

$$p(W) = p(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} p(w_i | w_1, \ldots, w_{i-1}) \tag{3.2}$$

Generating text can then be achieved recursively. Given a set of preceding words $\{w_1, w_2, \ldots, w_{i-1}\}$, the model estimates the probability distribution for the next word $p(w_i|w_1, \ldots, w_{i-1})$. LLMs, such as BERT [97] and GPT-3 [69] are pre-trained on large-scale internet corpora and have dominated across a series of downstream natural language processing (NLP) tasks [98]. In this work, we leverage the domain knowledge encoded into such models in order to reduce high-level tasks to actions on a lower level of abstraction.

3.3 Translating Cooking Recipes to LTL Formulae

3.3.1 Problem Statement

Consider a robot in a kitchen, equipped with a limited set of primitive actions $\mathcal{A}$. We assume that a primitive action in a cooking environment can be described by a set of salient categories $\mathcal{C}$ ={`Verb`, `What?`, `Where?`, `How?`, `Time`, `Temperature`}. We define an action description `a` as a function consisting of a main `Verb` as the function name, with a set of one or more of the other categories as its parameters:

$$\mathtt{a} = \mathtt{Verb}(\mathtt{What?}, \mathtt{Where?}, \mathtt{How?}, \mathtt{Time}, \mathtt{Temperature})$$

The robot is tasked with executing a cooking recipe R that consists of a list of k instruction steps $\{r_1, r_2, \ldots, r_k\}$, where each instruction step r_i is an imperative sentence in natural language describing a robot command. Each instruction step r_i may include one or more cooking actions. Our goal is to generate a set of task specifications written in the form of a set of LTL formulae $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ that implement the recipe under the constraint of **only** including actions that belong to the set of primitive actions $\mathcal{A}$ that the robot is capable of executing.

3.3.2 System Architecture

To solve this problem, we propose Cook2LTL, the system architecture summarized in Fig. 3.2. Given an instruction r_i and a set of actions $\mathcal{A}$, Cook2LTL:

1. Semantically parses r_i into a function representation `a` for every detected high-level action.
2. Reduces each high-level action $\mathtt{a} \notin \mathcal{A}$ to a combination of primitive actions from $\mathcal{A}$.
3. Caches the action reduction policy for future use, thereby gradually building an action library that

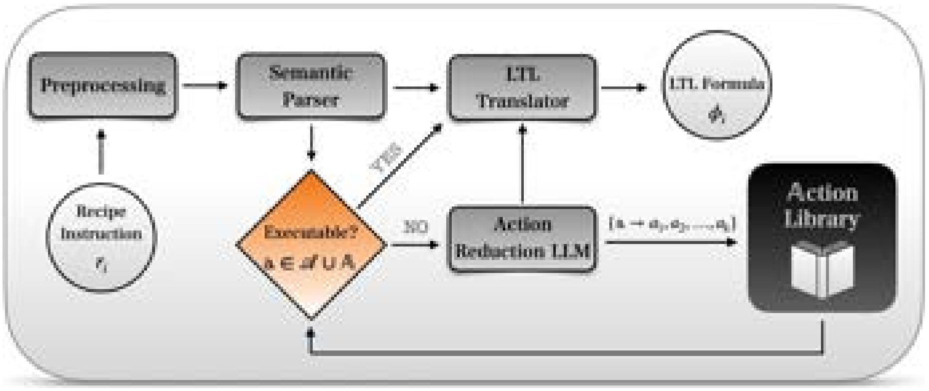


Figure 3.2: **Cook2LTL System:** The input instruction r_i is first preprocessed and then passed to the semantic parser, which extracts meaningful chunks corresponding to the categories $\mathcal{C}$ and constructs a function representation a for each detected action. If a is part of the action library $\mathbb{A}$, then the LTL translator infers the final LTL formula ϕ. Otherwise, the action is reduced to a sequence of lower-level admissible actions $\{a_1, a_2, \ldots a_k\}$ from $\mathcal{A}$, and the reduction policy is cached to $\mathbb{A}$ for future use. The LTL translator then yields the final LTL formulae based on the derived actions.

consists of parametric functions that express high-level cooking actions in the form of primitive actions.

4. Translates r_i into an LTL formula ϕ_i with function representations as atomic propositions.

Algorithmically, these steps are summarized in Alg. 1. In the following subsections, we expand on the components of Cook2LTL in more detail.

3.3.3 Semantic Parsing and Data Annotation

Our translation system requires a semantic parsing module capable of extracting meaningful chunks corresponding to the parametric function representation components of a cooking action. To this end, we fine-tune a named entity recognizer with the addition of salient categories $\mathcal{C}$ as labels. We choose a neural approach over a syntactic parse because the latter would require arduous manual rule crafting for every different mapping of part-of-speech (POS) tags to these categories. Additionally, explicit POS-tagging-

Algorithm 1 Cook2LTL

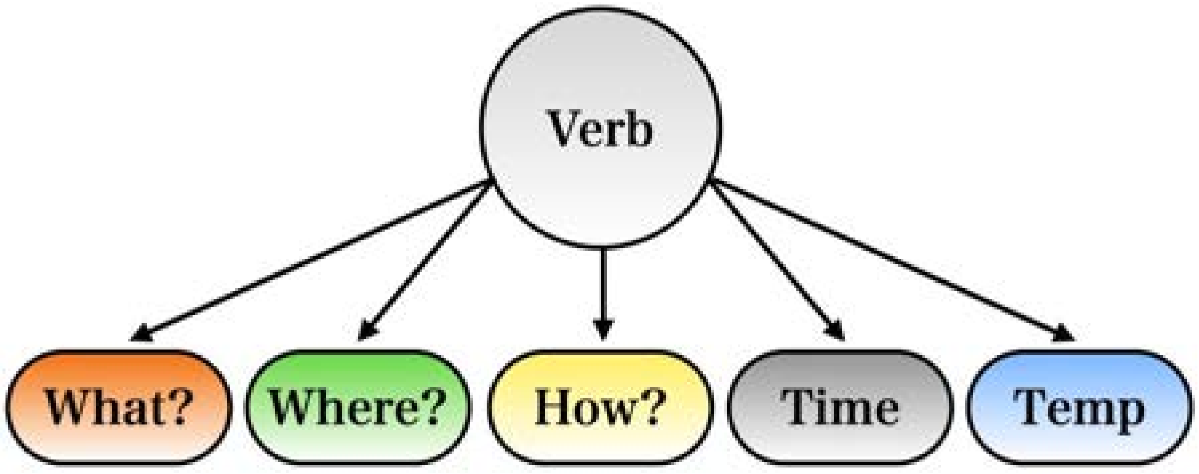
Input: A high-level instruction step r, a set of primitive actions $\mathcal{A}$, and an action library $\mathbb{A}$
Output: An LTL action formula ϕ

```
 1: $\mathbb{A} \leftarrow \mathbb{A} \cup \mathcal{A}$
 2: $r \leftarrow f_{PRE}(r)$                                          ▷ Preprocessing
 3: $\{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_\mathtt{n}\} \leftarrow f_{SP}(r)$                      ▷ Semantic Parsing
 4: $A \leftarrow \{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_\mathtt{n}\}$
 5: $\phi \leftarrow f_{LTL}(\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_\mathtt{n})$                       ▷ Initial LTL Translation
 6: for $\mathtt{a}_i \in A$ do
 7:     if $\mathtt{a}_\mathtt{i} \notin \mathbb{A}$ then
 8:         $\{a_1, a_2, \ldots, a_k\} \leftarrow f_{AR}(\mathtt{a}_i)$                    ▷ Action Reduction
 9:         $\mathtt{a}_\mathtt{i} \leftarrow \{a_1, a_2, \ldots, a_k\}$
10:         $\mathbb{A} \leftarrow \mathbb{A} \cup \{\mathtt{a} \rightarrow a_1, a_2, \ldots, a_k\}$                 ▷ Caching
11:     end if
12: end for
13: $\phi \leftarrow f_{LTL}(A)$                                      ▷ Final LTL Translation
14: return $\phi$
```

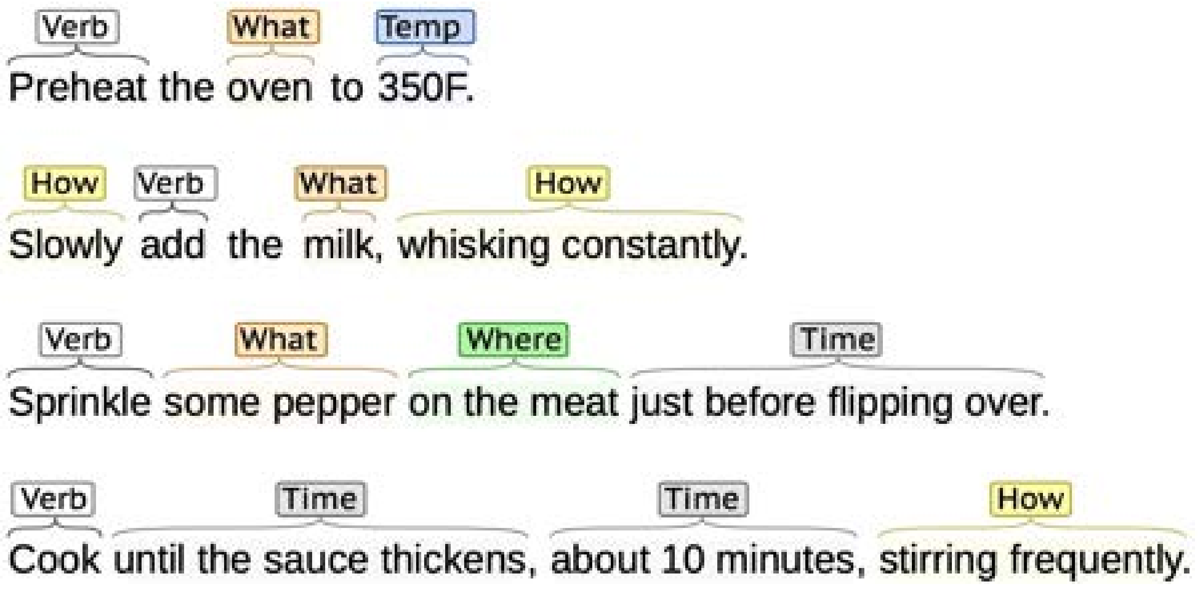
based approaches often struggle with handling the intricacies of cooking discourse, such as imperative form sentences omitting context-implicit parts of speech.

In the absence of a labeled dataset with a schema matching $\mathcal{C}$, we create our own data building upon the large cooking recipe dataset Recipe1M+ [1]. Specifically, we consider a subset of 100 recipes from Recipe1M+, leading to 1000 recipe instruction steps. We use brat [99] to manually annotate chunks in each step corresponding to the following salient categories: $\mathcal{C}$ ={`Verb, What?, Where?, How?, Temperature, Time`}, which is a similar annotation scheme as the one seen in recent work [100].

Fig. 3.3 shows these categories and a set of example recipe steps taken from Recipe1M+ [1]. `Verb` is the main action verb in a recipe step. `What?` represents the direct object of the `Verb` and is often an ingredient, but can correspond to other entities such as a kitchen utensil or an appliance. `Where?` is usually a prepositional phrase, it implies a physical location (e.g. table, bowl) but can often be an ingredient to which the `Verb` applies. `How?` is usually either a gerund form of a verb, expressing concurrency and hence giving rise to a secondary cooking action, or complements the main cooking

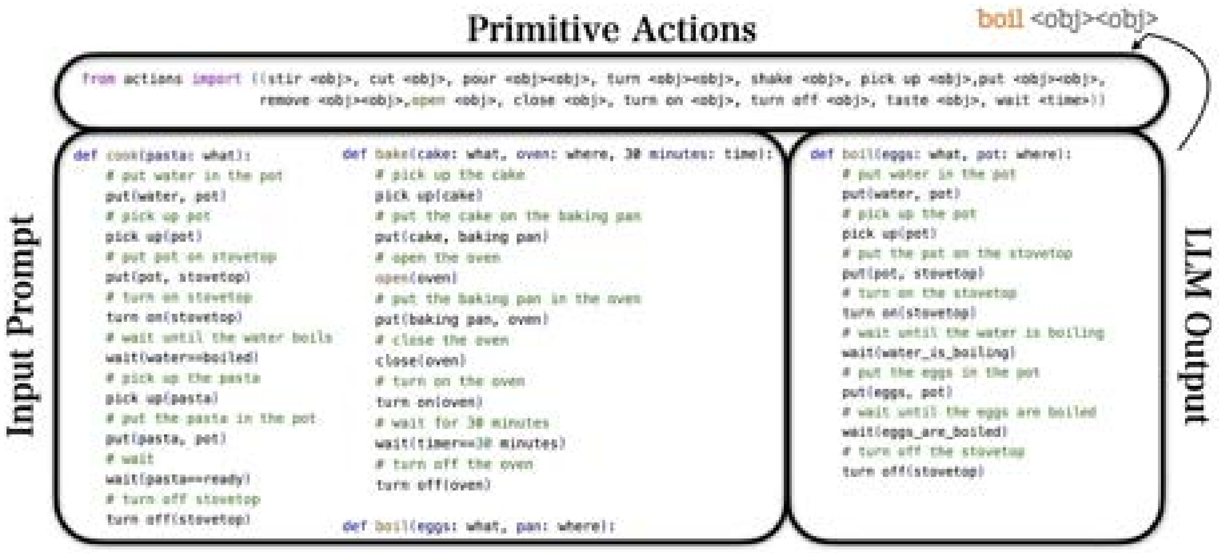


(a) Salient categories $\mathcal{C}$ considered for semantic parsing.


Verb
What
Temp
Preheat the oven to 350F.
How
Verb
What
How
Slowly add the milk, whisking constantly.
Verb
What
Where
Time
Sprinkle some pepper on the meat just before flipping over.
Verb
Time
Time
How
Cook until the sauce thickens, about 10 minutes, stirring frequently.

(b) Recipe steps annotated with the salient categories $\mathcal{C}$

Figure 3.3: We annotate Recipe1M+ [1] instruction steps with the salient categories $\mathcal{C}$ ={`Verb`, `What?`, `Where?`, `How?`, `Temperature`, `Time`} and fine-tune a named entity recognizer to segment chunks corresponding to $\mathcal{C}$.



```
from actions import ((stir <obj>, cut <obj>, pour <obj><obj>, turn <obj><obj>, shake <obj>, pick up <obj>,put <obj><obj>,
                      remove <obj><obj>,open <obj>, close <obj>, turn on <obj>, turn off <obj>, taste <obj>, wait <time>))
```

```
def cook(pasta: what):
    # put water in the pot
    put(water, pot)
    # pick up pot
    pick up(pot)
    # put pot on stovetop
    put(pot, stovetop)
    # turn on stovetop
    turn on(stovetop)
    # wait until the water boils
    wait(water==boiled)
    # pick up the pasta
    pick up(pasta)
    # put the pasta in the pot
    put(pasta, pot)
    # wait
    wait(pasta==ready)
    # turn off stovetop
    turn off(stovetop)
```

```
def bake(cake: what, oven: where, 30 minutes: time):
    # pick up the cake
    pick up(cake)
    # put the cake on the baking pan
    put(cake, baking pan)
    # open the oven
    open(oven)
    # put the baking pan in the oven
    put(baking pan, oven)
    # close the oven
    close(oven)
    # turn on the oven
    turn on(oven)
    # wait for 30 minutes
    wait(timer==30 minutes)
    # turn off the oven
    turn off(oven)

def boil(eggs: what, pan: where):
```

```
def boil(eggs: what, pot: where):
    # put water in the pot
    put(water, pot)
    # pick up the pot
    pick up(pot)
    # put the pot on the stovetop
    put(pot, stovetop)
    # turn on the stovetop
    turn on(stovetop)
    # wait until the water is boiling
    wait(water_is_boiling)
    # put the eggs in the pot
    put(eggs, pot)
    # wait until the eggs are boiled
    wait(eggs_are_boiled)
    # turn off the stovetop
    turn off(stovetop)
```

Figure 3.4: Inspired by ProgPrompt [3], Cook2LTL uses an LLM prompting scheme to reduce a high-level cooking action (e.g. `boil eggs`) to a series of primitive manipulation actions. The prompt consists of an import statement of the primitive action set and example function definitions of similar cooking tasks. The key benefit of using this paradigm is that it constrains the output action plan of the LLM to only include subsets of the available primitive actions. We extend this prompting scheme by reusing derived LLM policies. In this case, the action `boil` is added to future import statements in the input prompt, enabling the model to invoke the derived `boil` function which is now considered given to the system.

action (e.g. "Drizzle *with olive oil*"). The `Time` category consists of temporal expressions composed of keywords that are important for the translation of the commands to LTL formulae (*until*, *before* etc.). Finally, `Temperature` can explicitly list the degrees (e.g. 350F) to which food should be cooked or refer to a temperature-related state of some ingredient (e.g. *medium heat*). These salient categories form the function representation of an action found in r_i.

3.3.4 Reduction to Primitive Actions

Some of the function representations captured in the previous step contain high-level actions that might not be directly executable by the robot, which can only execute actions that belong to the primitive set $\mathcal{A}$. Therefore, our system requires a module capable of mapping an action $\texttt{a} \notin \mathcal{A}$ to an action $a \in \mathcal{A}$, if possible, or reducing $\texttt{a}$ to a sequence of actions $a_1, a_2, \ldots, a_k$ where $a_i \in \mathcal{A}, i = 1, 2, \ldots, k$. Our system initially checks whether $\texttt{a} \in \mathcal{A}$ to validate a formula for execution, and if $\texttt{a} \in \mathcal{A}$, $\texttt{a}$ is forwarded to the LTL translator.

LLM Action Reduction: If $\texttt{a} \notin \mathcal{A}$ we employ an LLM-based methodology inspired by the work in [3] to extract a lower-level plan exclusively consisting of primitive actions from $\mathcal{A}$. Specifically, we design an input prompt consisting of: i) a pythonic import of the available actions in the environment, ii) two example function definitions decomposing high-level cooking actions into primitive sets of actions from $\mathcal{A}$, iii) the function representation $\texttt{a}$ extracted by the semantic parsing module in the form of a pythonic function name with its parameters. As shown in [3] and Fig. 3.4, the LLM follows the style and pattern of the input function and only includes available actions in the output. The key advantage of this method is the flexibility in changing the admissible primitive actions depending on the robot capabilities and the environment. This change can simply be achieved by modifying the primitive actions in the pythonic import.

Action Library: Extending ProgPrompt [3], every time we query the LLM for action reduction, we cache $\texttt{a}$ and its action decomposition for future use through a dictionary lookup manner. This gradually builds a dynamic knowledge base in the form of an executable action library $\mathbb{A}$ consisting of various high-level actions along with their function bodies made out of primitive actions from $\mathcal{A}$. At runtime, instead of only checking whether a detected action $\texttt{a}$ matches an action $a \in \mathcal{A}$, we additionally check if $\texttt{a} \in \mathbb{A}$. In case there is a match, we replace $\texttt{a}$ with the action in $\mathbb{A}$. Additionally, we add $\texttt{a}$ to the pythonic import part of the prompt, allowing the model to invoke it when generating future policies (e.g. the action `boil` in Fig. 3.4). The key benefit comes from avoiding to continuously query an LLM for action reduction, thus replacing potential latency resulting from an LLM API call with a fixed $\mathcal{O}(1)$ dictionary lookup time. It also reduces the cost associated with querying a proprietary LLM API.

3.3.5 LTL Translation

The final step in our pipeline translates the intermediate function representations acquired from semantic parsing and action reduction into an LTL formula. The implicit sequencing of recipes is elegantly captured by the sequenced visit specification pattern [101]:

$$F(l_1 \wedge \mathcal{F}(l_2 \wedge \ldots \mathcal{F} l_n))) \tag{3.3}$$

This pattern has been used [62, 65, 102] to model a visit of a set of locations $L = \{l_1, l_2, \ldots, l_n\}$ in sequence one after the other in a navigational setting, adapted to the execution of consecutive cooking actions $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n$ in our case. Building on this pattern, we acquire conjunction, disjunction, and negation constituents for each segmented chunk corresponding to the categories $\mathcal{C}$ through a dependency parse. Then, we write down a formula ϕ which includes high-level actions $\mathtt{a}$ with a combination of the following LTL operators $\{(\mathcal{F} : \mathtt{Finally}), (\wedge : \mathtt{and}), (\vee : \mathtt{or}), (\neg : \mathtt{not})\}$. Every action $\mathtt{a}_i$ is translated to one or more primitive actions from $\mathcal{A}$. In the latter case, the generated low-level plan for $\mathtt{a}_i$ is parsed into a subformula ψ_i based on Equation 3.3. The `Time` parameter passed to the action reduction LLM often includes explicit sequencing language (such as *until*, *before*, or *once*). The LLM has been prompted to return a `Wait` function in these cases (see example in Fig. 3.4), which is then parsed into the $(\mathcal{U} : \mathtt{until})$ operator and substituted in ψ. The final formula ϕ consists of subformulae $\psi_1, \psi_2, \ldots, \psi_n$ comprised by primitive actions in $\mathcal{A}$:

$$\phi = F(\mathtt{a}_1 \wedge \mathcal{F}(\mathtt{a}_2 \wedge \ldots \mathcal{F}\mathtt{a}_n))) = F(\psi_1 \wedge \mathcal{F}(\psi_2 \wedge \ldots \mathcal{F}\psi_n))) \tag{3.4}$$

where:

$$\begin{cases} \psi_i = a_i & , a_i \in \mathcal{A} \text{ ,or} \\ \psi_i = f(a_1, a_2, \ldots, a_k, \mathbb{O}) & , \mathbb{O} = \{\mathcal{F}, \wedge, \vee, \neg, \mathcal{U}\} \end{cases} \tag{3.5}$$

3.4 Evaluation

3.4.1 Ablation Study

To investigate the performance of Cook2LTL, we conduct an ablation study against two variants. For each run, the input is a recipe from a held-out subset of Recipe1M+ and the output is a series of task specifications in the form of LTL formulae Φ towards executing the recipe under the constraints of admissible actions $\mathcal{A}$. In all the experiments we use the OpenAI API and the *gpt-3.5-turbo* model. The initial preprocessing step consists of filling in the implicit objects (zero anaphora resolution) in the recipes and segmenting each recipe into sentences. We begin by deploying a partial version of our system (AR*) as a baseline, consisting of the preprocessing, semantic parsing, and action reduction modules. We expect that our action reduction policy adheres to the admissible actions of the environment by a significant amount. We incrementally add the functionality of invoking cached policies, first when encountering a primitive action (AR), and then when an action is found in the action library (AR+$\mathbb{A}$), starting from an empty library and gradually building it with the LLM-generated policies along the way. We anticipate a significant benefit in terms of computational load and cost efficiency resulting from capitalizing on reusable policies, compared to querying the action reduction LLM for every unseen action encountered at runtime. We formalized these insights into the following hypotheses:

H1: Our action reduction policy generation constrains the LLM output to the admissible actions $\mathcal{A}$ in our environment.

H2: Our enhanced Cook2LTL system that includes the action library component is more time- and cost-efficient than the baseline action reduction-comprised partial system.

To evaluate these hypotheses, our metrics are: 1. *Executability (%)*, which is the fraction of actions in the generated plan that are admissible in the environment; 2. *Time (min* or *sec)* which measures the runtime influenced by the LLM API calls; 3. *Cost ($)* which is the overall cost for a batch of experiments and depends on the number of input and output tokens; 4. the number of the LLM *API calls*.

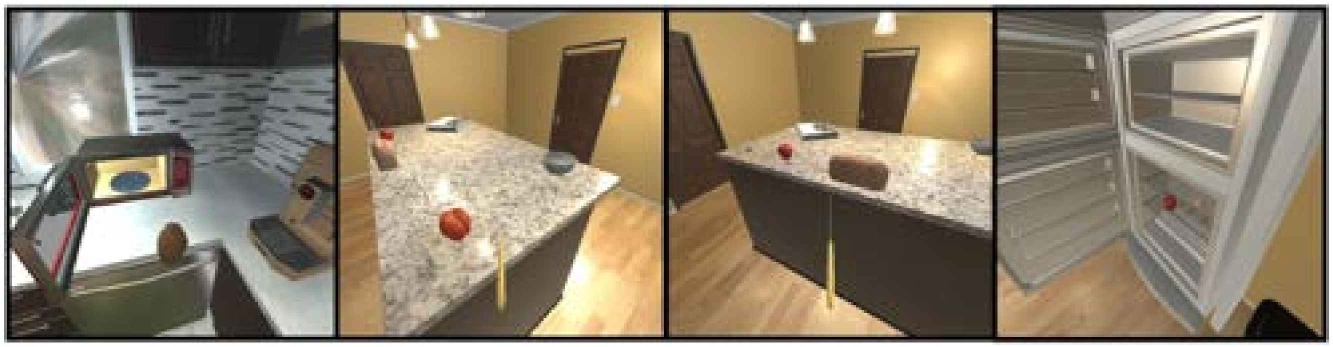


Figure 3.5: Tasks we tested Cook2LTL in AI2-THOR (left to right): `microwave the potato`; `chop the tomato`; `cut the bread`; `refrigerate the apple`.

	Active Modules		
Metric	AR*	AR	Cook2LTL (AR+$\mathbb{A}$)
Executability (%)	0.91 ± 0.01	0.92 ± 0.01	$\mathbf{0.94 \pm 0.01}$
Time (min)	14.85 ± 1.05	9.89 ± 0.46	$\mathbf{6.05 \pm 0.12}$
Cost ($)	0.19 ± 0.01	0.16 ± 0.00	$\mathbf{0.11 \pm 0.00}$
API calls (#)	275 ± 0.00	231 ± 0.00	$\mathbf{134 \pm 0.00}$

Table 3.1: Performance of Cook2LTL against baselines across 50 Recipe1M+ [1] recipes (10 runs per recipe).

3.4.2 Results & Discussion

Based on the quantitative results in Table 3.1 we make the following observations regarding our hypotheses.

H1: Our first hypothesis is confirmed. In every part of the ablation study the system has a high executability with a maximum value of 94% when using the action library. This is a natural consequence of incorporating a new action in the prompt every time it is decomposed to sub-actions by the LLM. The policies for the cached actions are now part of the system, and hence they are considered admissible in

	AR		Cook2LTL (AR+$\mathbb{A}$)	
Task	SR (%)	Time (sec)	SR (%)	Time (sec)
Microwave the potato	5.4 ± 1.95	27.29 ± 3.66	$\mathbf{8 \pm 4.47}$	$\mathbf{3.26 \pm 1.30}$
Chop the tomato	2.4 ± 1.52	16 ± 0.96	$\mathbf{4 \pm 5.47}$	$\mathbf{1.61 \pm 0.76}$
Cut the bread	$\mathbf{9 \pm 0.71}$	12.85 ± 0.84	8 ± 4.47	$\mathbf{1.12 \pm 0.16}$
Refrigerate the apple	7.6 ± 0.55	14.6 ± 0.38	$\mathbf{8 \pm 4.47}$	$\mathbf{1.56 \pm 0.44}$

Table 3.2: We demonstrate the performance of Cook2LTL on 4 simple cooking tasks in AI2-THOR. We observe that Cook2LTL (AR+$\mathbb{A}$) is time efficient but propagates initial incorrect LLM-generated sets of actions to subsequent runs.

the environment, leading to an increased executability value.

H2: The enhanced action library-based Cook2LTL system (AR+$\mathbb{A}$) outperforms the baseline (AR*) and primitive action-focused variant (AR) in all 4 metrics. We have discovered that learning new action policies through prompting an LLM and reusing them in a dictionary lookup manner in subsequent recipes decreases the number of API calls by 51% and 50% compared to the AR* and AR versions of the system. Consequently, a lower number of API calls leads to a significantly reduced runtime and cost. More specifically, the integration of the action library into our system decreases runtime by 59% and 42% compared to the AR* and AR versions, and cost by 42% and 31%, respectively.

3.4.3 Demonstration in AI2-THOR

We demonstrate the performance of Cook2LTL in a simulated AI2-THOR [2] kitchen environment (See Fig. 3.1). AI2-THOR has a small set of ingredients and objects and hence cannot support the full execution of recipes found on the web; however the limited action space aligns with the notion of primitive actions and offers room for highlighting the key ideas of our system. To showcase the potential of our approach, we constructed a set of 4 kitchen tasks that are admissible in AI2-THOR and executed them by invoking Cook2LTL. We assume that the kitchen is *mise en place* so the locations of the objects are known to the agent. In AI2-THOR, we design a minimal parser that receives an LTL formula and converts it to a series of actions. We adapt the imported primitive actions and example functions in the prompt to the ones that are supported in the simulation. Fig. 3.5 contains screenshots from our experiments. We run 5 sets of experiments where we execute each task 10 consecutive times. We measure the success rate SR and execution time due to the LLM API calls and compare the performance of the AR and Cook2LTL (AR+$\mathbb{A}$) variants. The success rate is the fraction of executions that achieved the task-dependent goal conditions (e.g. *tomato=sliced*) that we defined a priori. During our simulations we observe that Cook2LTL is still significantly more time efficient compared to baselines, however its SR is entirely dependent on the first LLM-generated plan, and fails when this plan is not executable (See Table 3.2).

3.5 Limitations & Future Work

System: We annotated a small part of the Recipe1M+ dataset [1] with our salient categories but we would need more data to improve the entity recognizer for reliably transferring the system to a real-world robot. Finally, some actions being substituted by action library policies lead to non-executable plans. Our system would benefit from an additional mechanism that robustly ensures the correctness of the LLM-generated plans based on environment feedback.

Sim2real: AI2-THOR is not tailored towards simulating cooking tasks but rather supports the general area of task planning. Thus, we would need a cooking-specific simulator to support a more diverse set of recipes that correspond to the rich web-scraped recipes that we built our system on. In terms of transferring simulation to a real robot, we plan to use the Yale-CMU-Berkeley (YCB) Object and Model set [103] towards supporting a basic set of simple cooking tasks for benchmarking preliminary experiments.

Task representation: The final layer of our system uses LTL as an expressible notation tool capturing temporal task interdependence, but our system is compatible with other task representations, such as PDDL [104], which incorporates action preconditions and postconditions in the problem setting and has recently been explored with LLMs [52, 54].

Chapter 4: Grounding Object Attributes through LLM-Generated Perception-Action Programs

4.1 Introduction

Connecting natural language instructions to the physical world often requires robots to detect object attributes in order to discriminate between candidate objects. This is a challenging problem as instructions might be linguistically ambiguous, for example “Can you please get me the second mug from the right on that shelf?”. Identifying attributes can also be required implicitly to determine the state or affordance [105] of an object in order to verify the feasibility of an action. These attributes might not be directly perceivable through vision sensors, for example “Is this lightweight enough to pick up?”.

To ground linguistic instructions in embodied settings, we focus on actively identifying object attributes in a programmatic fashion, combining perceptual and motoric functions. Attributes usually appear in the form of descriptive adjectives, some of which might not be adequately represented in the training sets of data-driven perceptual models. Furthermore, while attributes might not necessarily characterize an object in an absolute scale, they are often applicable based on context [83]. We argue that attribute detection is highly contextual - an analogy of J. R. Firth’s famous quotation “You shall know a word by the company it keeps” [106] applies to attributes. More specifically, characterizing an object as big, tall, or heavy can sometimes depend on the other objects and their respective attribute values in the current environmental context. Ambiguity can also arise due to occlusion or partial observability [87]. These problems might not occur when studying attribute detection in static images, but can be common in a household environment where a robot is tasked with executing user instructions. In these cases, erroneous attribute detection can be detrimental, producing and executing an action plan involving an incorrect object, or misinterpreting the affordance of an object and failing to even execute the action.

Existing attribute detectors [41, 85, 87, 88, 107–109] are mainly obtained by either supervised training [82] or contrastive pre-training [12]. While attribute detection is an active area of research, it is often studied separately from embodied reasoning. To bridge this gap, we model attribute detection as visual reasoning with programs. This provides us with a powerful representation for reasoning in the presence of embodied agents and allows us to utilize the space of plans and movements via robot actions as programs [3].

Summarizing these ideas, our main observation is that modern real-world robotic systems relying on visually-driven attribute detection using VLMs in isolation can be myopic in language grounding. Our key insight is that combining different VLMs as visual reasoning functions with a robot control API can benefit from the code synthesis and commonsense reasoning capabilities of LLMs to actively reason about attribute detection in the form of computer programs. We prompt an LLM with an attribute detection API on a dataset that we curate, consisting of embodiment-crucial **location**-, **size**-, and **weight**-related attributes and construct a perception-action API for active attribute detection [44]. Our key contributions can be summarized as follows:

- We highlight some of the drawbacks of using VLMs for attribute detection in isolation and the complementary reasoning capabilities that emerge by reasoning in the form of LLM-generated visual programs.
- We construct a perception-action API by integrating visual reasoning with robot control functions and demonstrate its benefits by invoking active perception behaviors towards solving attribute detection queries.
- We release an end-to-end framework that integrates this perception-action API on a real robotic platform using visual servoing-based control.

4.2 Method

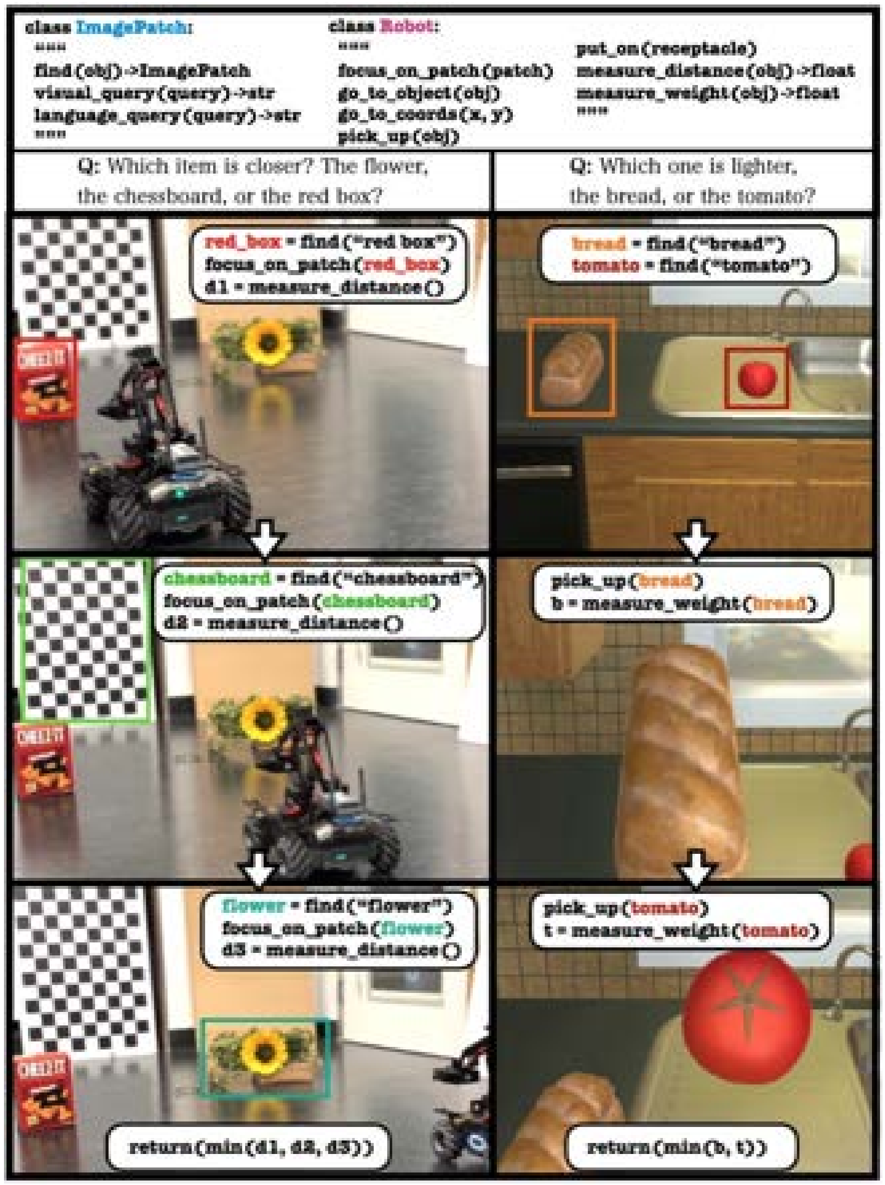
4.2.1 Problem Statement

Consider a robot equipped with a set of sensors $\mathcal{S}$ in a scene with a set of objects $\mathcal{O}$. The robot is tasked with executing a natural language instruction $inst = f(a, g, o, img)$, where a is a high-level action, g is an object attribute, o is an object, and img is an input image. Our goal is to determine whether an object exists with this attribute, expressed by the predicate $g(o)$, and localize it in img by obtaining its bounding box coordinates $\mathcal{X} = \{x_{min}, y_{min}, x_{max}, y_{max}\}$. If $\exists o$, such that $g(o)$ holds, a visual navigation policy $\pi(a(\mathcal{X}))$ is deployed, which allows the robot to leverage its sensors S, navigate and manipulate object o given its 2D bounding box coordinates $\mathcal{X}$ towards task completion.

We adopt a generalized definition of an attribute, viewing it as an abstract property of an object that does not necessarily map to a visual representation, including primarily descriptive adjectives related to the size (*big*) or weight (*heavy*) of an object, but also prepositional phrases indicating spatial relationships (*the second object from the left*).

4.2.2 Prompt-based Attribute Detection

We adopt the methodology of Surís et al. [14] into constructing a Python API for attribute detection. The API consists of a main `ImagePatch` class that is instantiated by an input image img. `find` is the fundamental function of the API that uses an OVD model (MM-Grounding-DINO [110] or GLIP [111]) to locate an object o and return the detection-resulting cropped patch $\mathcal{X}$ from the image given a chunk of natural language $inst$. While we do not explicitly define a function for spatial reasoning, we provide in-context examples encoded in the docstring of the function. The examples perform pixelwise math given bounding box coordinates $\mathcal{X}$ of detected objects $\mathcal{O}$ that are returned from calls to `find` in order to reason about relative object locations on the image frame. `visual_query` calls a pre-trained VLM (BLIP-2 [112]) to provide a textual answer to a visual query given an image. `language_query` recursively



```
class ImagePatch:
  """
  find(obj)->ImagePatch
  visual_query(query)->str
  language_query(query)->str
  """

class Robot:
  """
  focus_on_patch(patch)
  go_to_object(obj)
  go_to_coords(x, y)
  pick_up(obj)
  put_on(receptacle)
  measure_distance(obj)->float
  measure_weight(obj)->float
  """
```

Q: Which item is closer? The flower, the chessboard, or the red box?

```
red_box = find("red box")
focus_on_patch(red_box)
d1 = measure_distance()
```

```
chessboard = find("chessboard")
focus_on_patch(chessboard)
d2 = measure_distance()
```

```
flower = find("flower")
focus_on_patch(flower)
d3 = measure_distance()
```

```
return(min(d1, d2, d3))
```

Q: Which one is lighter, the bread, or the tomato?

```
bread = find("bread")
tomato = find("tomato")
```

```
pick_up(bread)
b = measure_weight(bread)
```

```
pick_up(tomato)
t = measure_weight(tomato)
```

```
return(min(b, t))
```

Figure 4.1: Demonstration of our perception-action API solving a minimum distance query on a real robot (*left*) and a minimum weight query in simulation (*right*). The LLM receives a perception-action API and a natural language query as input (*top*). It then generates code that invokes API functions leveraging on-board sensors (camera, distance sensor, force/torque sensor) to actively identify these attributes.

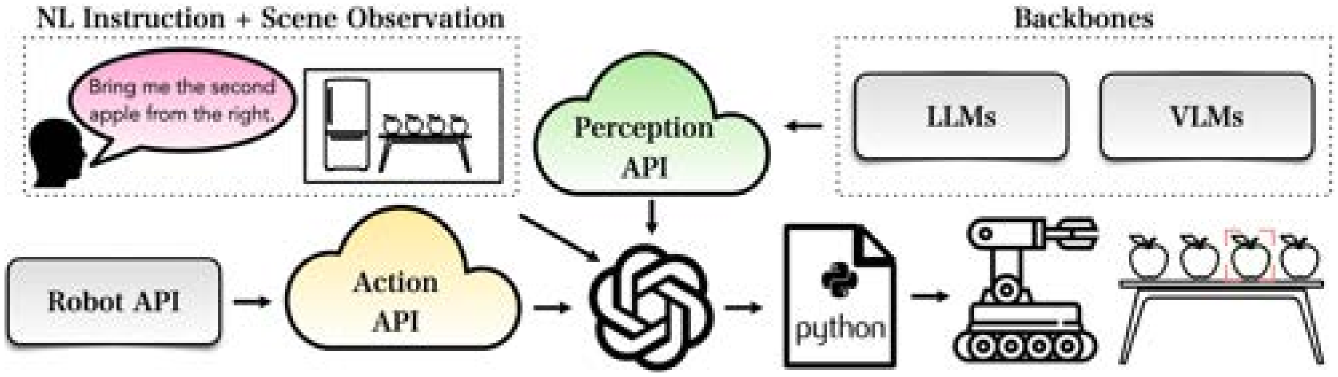

NL Instruction + Scene Observation
Bring me the second apple from the right.
Backbones
LLMs
VLMs
Perception API
Robot API
Action API
python

Figure 4.2: We describe our end-to-end framework for embodied attribute detection. The LLM receives as input a perception API with LLMs and VLMs as backbones, an action API based on a Robot Control API, a natural language (NL) instruction from a user, and a visual scene observation. It then produces a python program that combines LLM and VLM function calls with robot actions to actively reason about attribute detection.

calls the LLM with a textual query such as the visually-extracted return value from `visual_query`. This API can be viewed as an internal dialogue between VLMs and LLMs, similar to the idea of Socratic Models [43], but enhanced by structural programming tools hosted on a pythonic platform. In the following subsections we describe the complementary reasoning capabilities that emerge from calling these functions.

4.2.3 Programmatic Reasoning

Reasoning in the form of programs inherits the expressiveness of programming languages through control flow tools, data structures, and built-in methods. LLM-generated programs invoke loops to iterate over detected object patches and conditional statements to determine whether an object exists ($\exists o$) in the input image img and whether it possesses an attribute g, grounding the predicate $g(o)$. Python lists are used to store instances of image patches dynamically with the `append` function. Other built-in functions such as `sort` and the `lambda` function utilize horizontal and vertical coordinates of the detected bounding boxes and their centroids within simple mathematical operations and leverage basic geometrical notions (e.g. computing the area of an image patch) to reason about the size or relative position of detected entities. The generated code is interpretable and mainly consists of elementary arithmetic in the image frame. The commonsense reasoning component of the LLM proves to be essential in mapping complex language queries to these computations, as well as adapting attribute interpretation to image-specific contexts, as shown in Sec. 4.3.5.

4.2.4 Vision-Informed Language Reasoning

Combining LLMs and VLMs in the input API prompt unlocks complementary reasoning capabilities through information passing between different model calls in the the context of an LLM-generated program. This model interplay can be particularly efficient when dealing with non-visually perceivable attributes, such as estimating the weight of an object. In this case, `visual_query` can serve as a zero-shot object recognition function, subsequently passing information to the input of the `language_query`, which deduces factual knowledge on the weight of the recognized object, as explained in Sec 4.3.2. Similarly, when a task instruction involves an attribute that does not typically describe an object in an absolute scale, recognizing adjacent objects in addition to the object at hand establishes context. Then, posing a textual query with this visually-acquired information might reduce ambiguity and provide the correct grounding thanks to the domain knowledge of LLMs. We demonstrate an empirical evaluation of such use cases in Sec. 4.3.

4.2.5 Embodied Attribute Detection

Attribute detection in embodied settings often requires active perception. To this end, we formulate an action-perception API by integrating the attribute detection API with a high-level robot control API (See Fig. 4.2). The robot control API is implemented as a `Robot` Python class that consists of sensors as member variables and methods that map to simple navigation and pick-and-place actions. The robot can navigate to an object with the `go_to_object` function which implements a visual navigation policy by calling `go_to_coords` with an image patch $\mathcal{X}$ as a parameter. `pick_up` and `put_on` implement picking and placing actions. Assuming the lack of an on-board RGB-D camera or a depth estimation model, a robot could employ additional sensors to measure the distance to an object in order to reason about scene geometry or depth. This can be achieved by the `measure_distance` function which calls `focus_on_patch`, a function that aligns the geometric center of the image frame to an object patch and can then retrieve the distance sensor measurement to compute the distance from the camera to that object. A demonstration on a real robot is shown in Fig. 4.1 (*left*). Similarly, `measure_weight` can measure the weight of an object grasped by the robot, under the precondition that the robot first navigates and picks it up. These preconditions are encoded in the example use of the function in a docstring. We integrate this perception-action API into an AI2-THOR simulated environment and a real robot and demonstrate its benefits in Sec. 4.3.5.

4.3 Evaluation

We design a set of experiments to showcase some of the drawbacks of using OVD or VQA models for attribute grounding in isolation and we highlight the complementary commonsense reasoning that emerges by visual reasoning with LLM-generated programs including actions.

4.3.1 Spatial Reasoning

We evaluate the spatial reasoning capabilities of the attribute detection API by comparing its ability to ground linguistically complex spatial queries with an open-vocabulary object detector [111]. We manually craft a dataset that consists of 200 challenging spatial queries based on the Odd-One-Out (O^3) Dataset [113]. Every image in this dataset includes multiple instances of an object or similar objects with an instance being slightly different to stand out. We leverage the multiple instances of an object to invoke reasoning that requires differentiating between objects based on their relative attributes rather than obvious qualitative differences between entirely different objects. Therefore, instead of focusing on relative attributes that localize the object with respect to another object of different type in the image [14, 114] (e.g. "*the car to the left of the tree*"), our 100 **location** queries require commonsense reasoning in the form of counting and establishing the relative order of an arranged set of objects, such as "*second umbrella from the left at the second to last row*" or "*the window in the middle at the bottom*". Our 100 **size** queries utilize descriptive size-related adjectives (long, wide, short, large etc.) in their superlative and absolute form, such as "*the tallest item*" or "*the wide line*", respectively. We test the same queries on both forms and expect that the superlative form will outperform the absolute, forcing a specific object to stand out by emphasizing its attribute. We anticipate that the attribute detection API will outperform OVD by incorporating pixelwise mathematical operations and expressive python utility functions.

4.3.2 Non-visually Perceivable Attributes

To evaluate more profound reasoning capabilities, we focus on the **weight** of an object as a representative sample of non-visually perceivable attributes, as it is crucial for executing essential manipulation tasks. In the absence of a dataset with a suitable schema for our use case, we prompt GPT-4 to generate sets of objects of different weight, along with the ground truth label of the heaviest object. To simplify the task, we design the prompt so that the object weight distribution is monotonic and clearly distinguishable by a human observer: *Generate* 100 *triplets of objects where each object is significantly heavier than the other (for example: feather, dog, car)*. After acquiring the generated textual data, we utilize it to extract relevant images from the web and arrange them to form an image dataset where each data sample is an image that includes three objects of monotonically decreasing weight. Assuming some rudimentary commonsense reasoning functionality in VLMs [115], we expect that they are capable of identifying obvious differences in weight and hence dealing with examples that are intuitive to humans, such as selecting the heaviest object between a handbag, a kangaroo, and a bus (See Fig. 4.5). We compare the performance of OVD: `find(`*"a heavy object"*`)`, VQA: `visual_query(`*"Out of these items, which one is the heaviest?"*`)`, and vision-informed language

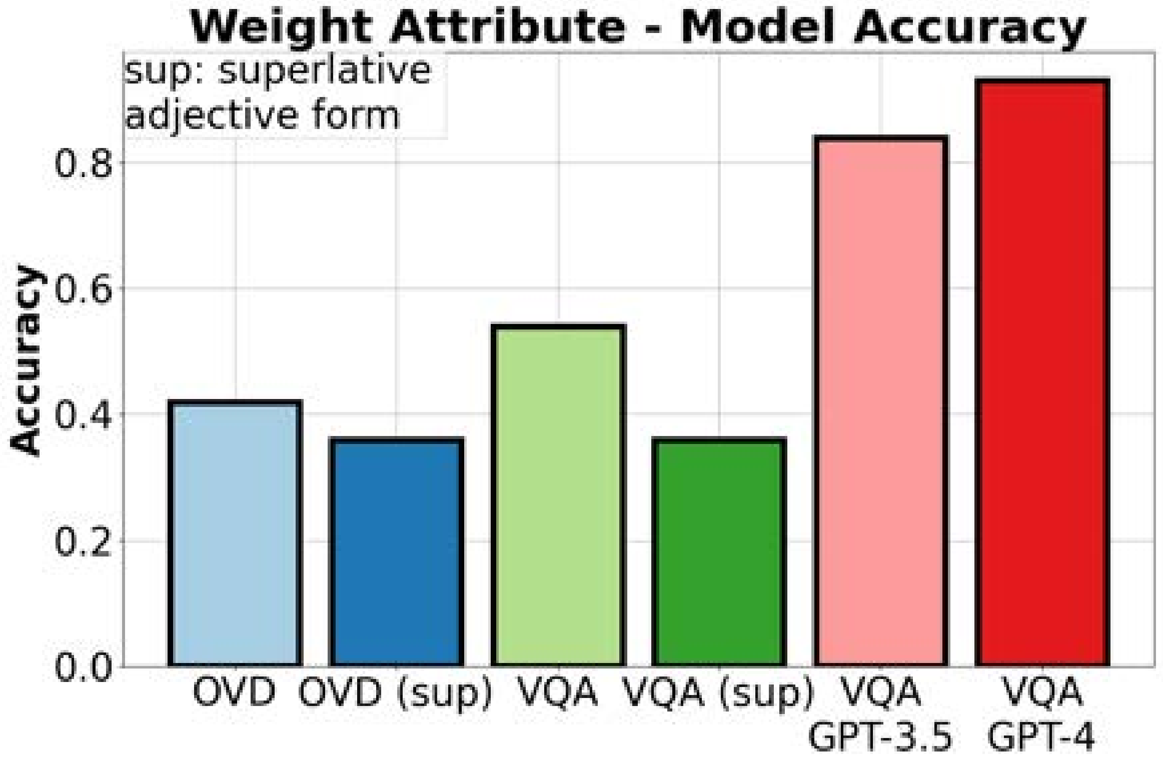


Figure 4.3: The accuracy of OVD (GLIP), VQA (BLIP-2), and VQA+GPT in determining the heaviest object in an image.

reasoning (VQA+GPT) that is invoked by our prompt API: `visual_query(`*"What are the items in this image?"*`)` $\rightarrow$ `language_query(`*"Out of these items, which one is more likely to be the heaviest one?"*`)`.

4.3.3 Evaluation in Embodied Settings

To evaluate our perception-action API in embodied settings, we adapt it to a simulated AI2-THOR [2] household environment. We assume that the robot comes with a proximity sensor and a force/torque sensor mounted on the wrist of the gripper, capable of measuring the weight of an object. To replicate the behavior of these sensors, we query the simulator for the distance between an object centered on the frame captured by the on-board robot camera, and build a queryable dictionary that maps an object to an approximate weight when the robot is holding that object. We measure the accuracy (%) of our perception-action API in estimating the relative distances of objects from the robot camera, and identifying the most lightweight object. Our baselines are OVD, VQA, GPT-4o, and the attribute detection API (VQA/OVD+GPT-4). We use the following prompt templates: "*Out of the {objects}, which one is closer to me?*", "*Out of the {objects}, which one is the most lightweight?*". We anticipate that the LLM-generated programs from the perception-action API are capable of actively interacting with the environment to identify object attributes leveraging sensor-powered visual reasoning functions and robot actions.

4.3.4 Hypotheses

We formalize these insights into the following hypotheses:

H1: OVD+GPT outperforms OVD- and VQA-only baselines in **location**- and **size**-related queries.

H2: VLMs possess the rudimentary reasoning capability to tackle evident **weight** estimation queries.

H3: The superlative form of a descriptive adjective yields a better grounding performance than the

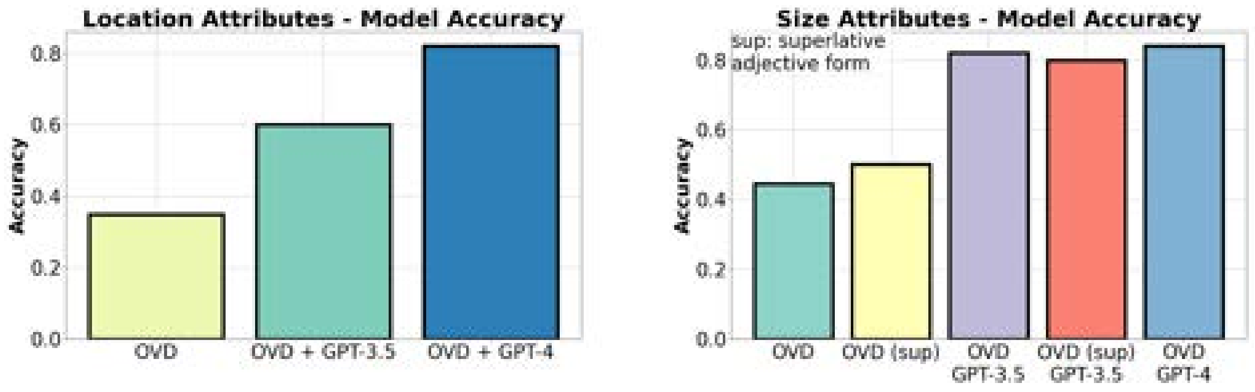


Figure 4.4: We compare the accuracy of OVD-only (GLIP) with (OVD+GPT) on our **location** (*left*) and **size** (*right*) datasets.

absolute form.

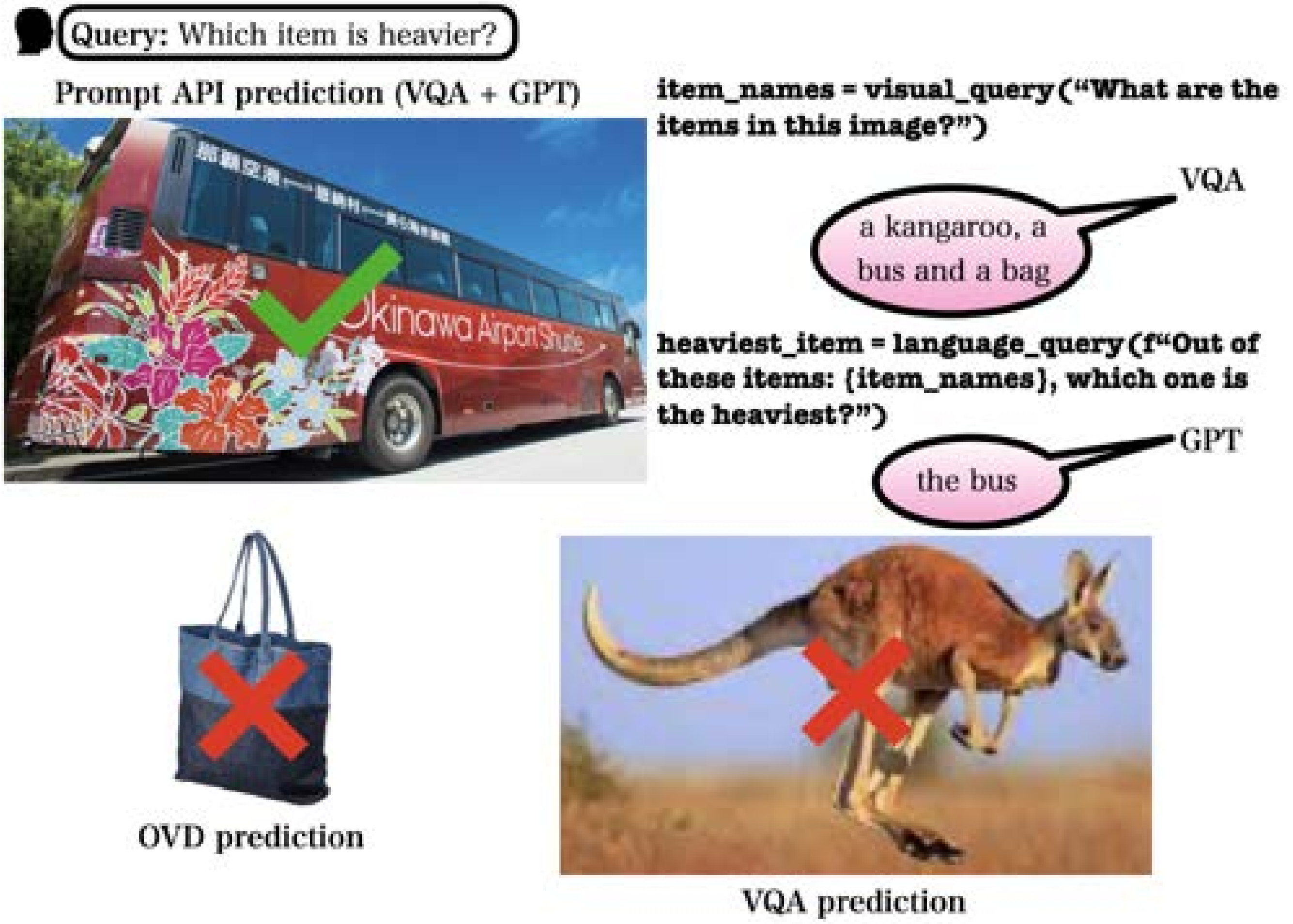
H4: Our perception-action API solves attribute detection queries by actively interacting with the environment.

To evaluate these hypotheses, we measure the grounding accuracy by comparing the bounding boxes returned by OVD and OVD+GPT. For OVD, we report results from GLIP [111] since it outperforms MM-Grounding-DINO [110] on our data. In the case of the weight attribute, we additionally consider the textual output of VQA+GPT. We demonstrate results from GPT-3.5, GPT-4, and GPT-4o in the embodied settings.

4.3.5 Results & Discussion

Based on the results in Fig. 4.3, Fig. 4.4, and Table 4.1 we make the following observations regarding our hypotheses.

H1: Our first hypothesis is confirmed. We find that OVD+GPT significantly outperforms OVD (See Fig. 4.4) in **location** queries by 134% and in **size** queries by 67%. Qualitative examples are shown in Fig. 4.6. In example *a*, OVD and OVD+GPT correctly identify the apple as the small fruit. Example *b* shows a common mutual failure case where the bounding box for one paper clip mistakenly includes all


Query: Which item is heavier?
Prompt API prediction (VQA + GPT)
Okinawa Airport Shuttle
OVD prediction
VQA prediction
VQA
a kangaroo, a bus and a bag
GPT
the bus

```
item_names = visual_query("What are the items in this image?")
heaviest_item = language_query(f"Out of these items: {item_names}, which one is the heaviest?")
```

Figure 4.5: An example where OVD and VQA fail to identify the heaviest object in the image (), but the API prompt-generated code (VQA+GPT) returns the correct answer ().

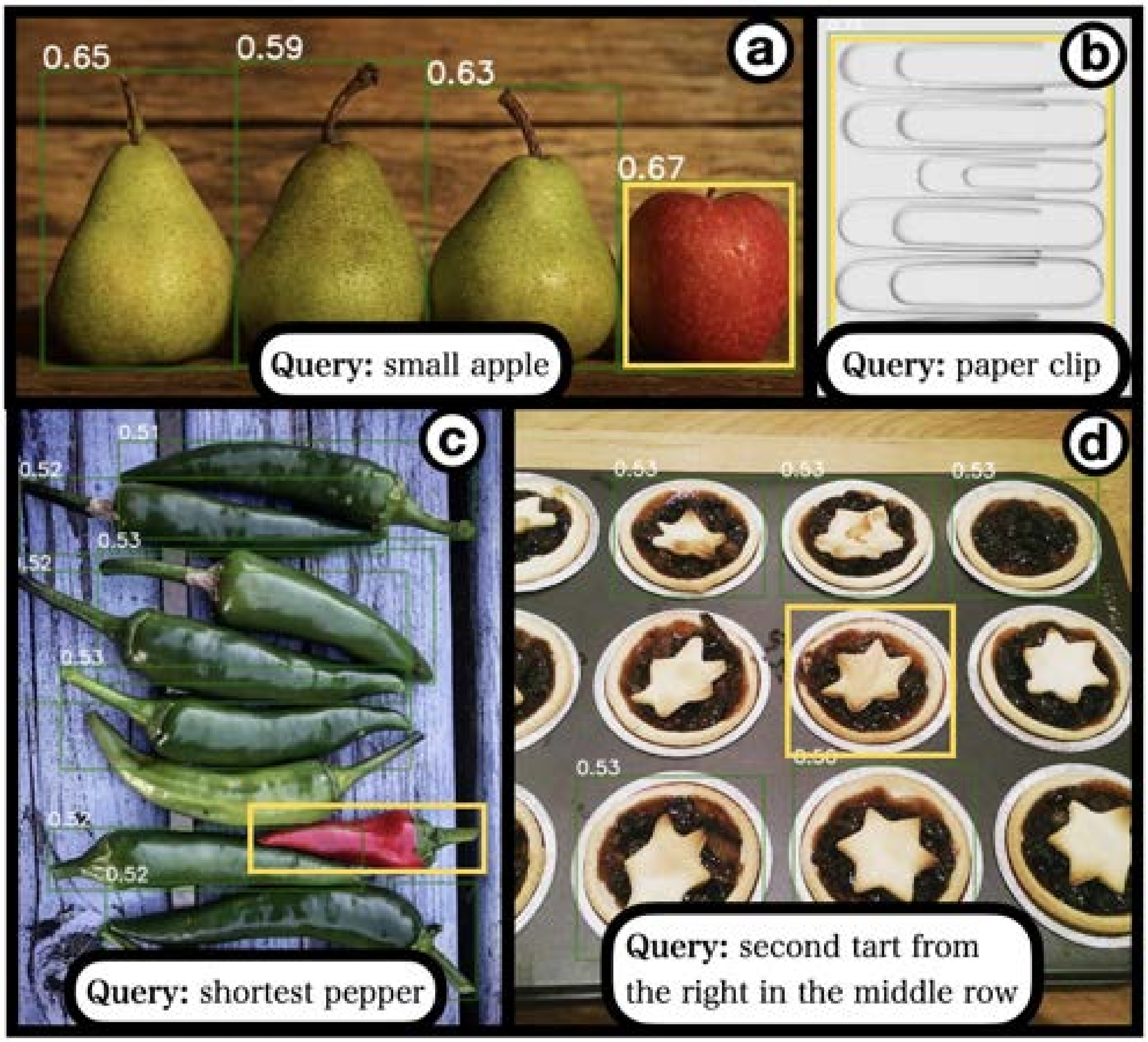


Figure 4.6: OVD (numbers in white denote the OVD confidence score) and OVD+GPT predictions are shown with green and yellow bounding boxes, respectively. *a* shows an example of agreement between OVD and OVD+GPT, *b* a mutual failure case, and *c*, *d* show cases where OVD+GPT exhibits superior performance compared to OVD.

.

of them. We also observe failure cases due to occlusion. Examples *c*, *d* show cases where OVD+GPT outperforms OVD. In these examples, all instances of the object in the query are localized with the `find` function, and then the coordinates of the bounding boxes and their centroids are used to compute relative distances and areas and sort them, if needed. In terms of the **size**-related queries, GPT-4 can adapt the generated code to the context of an image and interpret what dimension *long* or *short* corresponds to based on the orientation of an object in an image. On the other hand, GPT-3.5 is more rigid and tends to tie certain adjectives to hardcoded dimensions following common norms. For example, in Fig. 4.6 (*c*), it cannot connect the adjective *short* to the red pepper, since it is horizontally aligned. Such minor details explain the slight discrepancy in performance, which is still superior than vanilla OVD when identifying attributes with an attribute detection API. We would need to add targeted examples in the prompt API covering all failure cases to induce equal accuracy from both models. Finally, in example *d*, OVD+GPT-4 understands that the arrangement of the tarts is forming a row and column pattern. On the contrary, OVD fails to recognize this pattern, and OVD+GPT-3.5 exhibits a context-agnostic interpretation of rows, dividing the image into parts based on the image height, which yields incorrect results.

H2: Our second hypothesis is only partially confirmed. Based on Fig. 4.3, the combination of VQA, followed by a call to an LLM, significantly outperforms OVD-only (+121%) and VQA-only (+72%) solutions although we expected that all the models would be able to handle simple comparative attribute detection tasks. We believe that the step-by-step reasoning process followed by the prompt API leverages the strength of each model separately. On the contrary, burdening a model with additional reasoning tasks upon the ones that it was naturally tasked with in the first place (zero-shot language-conditioned object detection for GLIP, and zero-shot object recognition for BLIP-2) might be the reason we are missing out on its full task-specific potential. Another potential reason for this discrepancy in performance is that the pre-training objective of these models might not be aligned to our specific use case, which is identifying non-visually perceivable attributes.

H3: Our third hypothesis is rejected, mainly because the absolute form of an adjective yields better performance in the weight estimation task (See Fig. 4.3). In Fig. 4.3, Fig. 4.4, *sup* stands for superlative

Method	**Task**	
	Weight	Distance
OVD	0.14	0.64
VQA	0.64	0.56
Attribute Detection API	0.90	0.22
GPT-4o	0.88	0.70
Perception-Action API	**0.96**	**0.94**

Table 4.1: Performance of our perception-action API in 50 weight and 50 distance estimation queries in simulated AI2-THOR [2] household environments against baselines.

adjective form in the prompt. GPT-4 demonstrates the same performance for both forms, therefore we report a joint measurement for this model in Fig. 4.4.

H4: Confirming our hypothesis, our perception-action API solves both tasks and outperforms all baselines based on the results shown in Table 4.1. In distance estimation, the robot first identifies an object patch with `find`, and then leverages its distance sensor by focusing on the detected image patch with `focus_on_patch` and calling `measure_distance` to get the measurement. In weight estimation, after locating the patch with `find`, the robot navigates (`go_to_object`) and proceeds to `pick_up` every object and measure its weight using the force/torque sensor by calling `measure_weight`. At every measurement, the generated programs compare the currently measured value with a previously stored minimum and update it if the current value is lower, finally yielding the minimum distance or weight. In some cases, `find` cannot locate the object patches leading to failures. We believe that this occurs because of the image quality of the simulated objects, which are not as realistic as the ones in real-world images and the OVD model that `find` uses struggles to locate them. In distance estimation, the generated code by the attribute detection API (OVD+GPT-4) incorrectly hardcodes the distance from the object to the robot camera to the distance of the object from the geometric center of the image frame, leading to a very low accuracy.

4.3.6 End-to-End Framework - Robot Demonstration

We integrate our perception-action API into a real robot by implementing a wrapper over a robot-specific API. We deploy the combined end-to-end framework on DJI® RoboMaster™ EP [116], an affordable ground robot with holonomic movement and pick-and-place capabilities. A demonstration of our framework in action is shown in Fig. 4.1. Picking-and-placing an object first requires navigating in front of it with the appropriate orientation (`go_to_object`). To this end, we leverage sensory information to design a control policy for implementing the `go_to_object` function to navigate to a detected object. The control policy is further divided into two sub-policies: i) a visual servoing-based control policy for the lateral movement that aligns the center of the patch of the detected object to the center of the image frame captured by the on-board robot camera (`focus_on_patch`), ii) a control policy for the longitudinal movement that steers the robot towards a proximal position to the object at hand based on an infrared distance sensor. The target distance from the gripper to an object is a pre-computed functional gap, or in other words an experimentally determined *sweet spot* for picking and placing. Each of these policies is separately handled by a hand-tuned Proportional-Integral-Derivative (PID) controller. The robot is connected to a (*local*) computer via wifi connection and communicates with a (*remote*) computing cluster through a client-server architecture running on an SSH tunnel. To reduce latency due to the computational load of deploying a VLM on the cluster, we only run OVD on the first frame captured by the robot camera, and then track the corresponding position(s) with the Kanade–Lucas–Tomasi (KLT) feature tracker[1] [117].

[1]We follow the implementation in https://github.com/ZheyuanXie/KLT-Feature-Tracking.git.

4.4 Limitations & Future Work

Sensor Integration: Our experiments provide some insights on failure cases and emerging reasoning capabilities of VLMs in attribute detection. We demonstrate the applicability of our action-perception API on a robot in simulation and in the real world. In the future we plan to leverage the compositionality of our API and extend its sensing capabilities by incorporating more sensors (e.g. IMU, temperature sensor) via wrapper functions, supporting the discovery of additional attributes through active perception.

Error Propagation across Model Calls: In Sec. 4.3.5 we showed how the attribute API (VQA+GPT) outperforms calling an OVD or VQA model in isolation. However, if the first call yields an incorrect result, any downstream calls consume erroneous parameters and hence lead to an incorrect final result. In the future we plan to develop mechanisms that leverage additional feedback from the environment to catch such exceptions before errors propagate downstream.

Chapter 5: Planning Domain Induction from Demonstrations

5.1 Introduction

Everyday tasks that often seem trivial to humans can be challenging for robots. Consider the task of boiling pasta, which is ubiquitous in college student kitchens as a means to a quick and easy meal. Besides the dexterous manipulation required for handling ingredients and utensils, robots need to keep track of temporal task interdependencies, such as understanding when the water is boiling and only then adding the pasta (*action precondition*). Additionally, they have to monitor the kitchen to track the progress towards completing a recipe, such as understanding when the pasta is cooked to the desired (e.g. al dente) state (*action postcondition* or *effect*).

While LLMs are capable of translating natural language to actionable plans, they often fall short in long-horizon planning. For example, when executing cooking recipes in the kitchen they may omit crucial action preconditions or hallucinate, introducing out-of-context objects or unsupported actions to the plan. To mitigate these shortcomings, a recent line of research has instead been using LLMs to translate natural language to intermediate structured formats, such as PDDL specifications that afford precise solutions via symbolic planners. PDDL models an action as a set of preconditions and effects that must be satisfied to begin and deem action execution successful, respectively. However, devising PDDL specifications often requires lengthy prompts assuming domain knowledge or exhaustively hand-crafting the PDDL domain, which is a non-trivial task for non-experts. What happens when the designer has no prerequisite domain knowledge or the ability to manually craft the PDDL domain?

Furthermore, data collection towards imitation learning can be expensive, especially when dealing with intricate actions such as the ones observed in the cooking domain [28, 29]. However, when it comes to the task of defining a PDDL domain, the temporal sequence of actions, as well as their STRIPS action representation [118], consisting of preconditions and effects, can all be inferred through static images, without the need of observing lengthy robot trajectories. To this end, we can leverage robot or human demonstrations [119] as a convenient method to collect data about recipe execution in the kitchen, which can then be translated into PDDL format.

Based on these observations, our key insight is that acquiring a visual observation before and after action execution could provide adequate information towards inferring the PDDL domain. Our **main contribution** is NL2PDDL2Prog, a system that receives a visual observation before and after action execution, uses a VLM to generate their captions and passes them to an LLM to infer the PDDL action semantics for every admissible action. We gather visual data by recording robot demonstrations in the lab and human demonstrations in an actual kitchen. We demonstrate how we can combine LLM-produced action semantics from multiple demonstrations to obtain accurate generalized action semantics. The action semantics are fed back to the LLM to infer the entire PDDL domain. The LLM can then use the derived domain along with a natural language instruction and an initial visual observation of the scene to generate the problem specification, which is passed to a symbolic solver that produces a plan. We parse the resulting plan into a python executable program that visually grounds successful action execution at runtime by binding perceptual functions to action preconditions and effects.

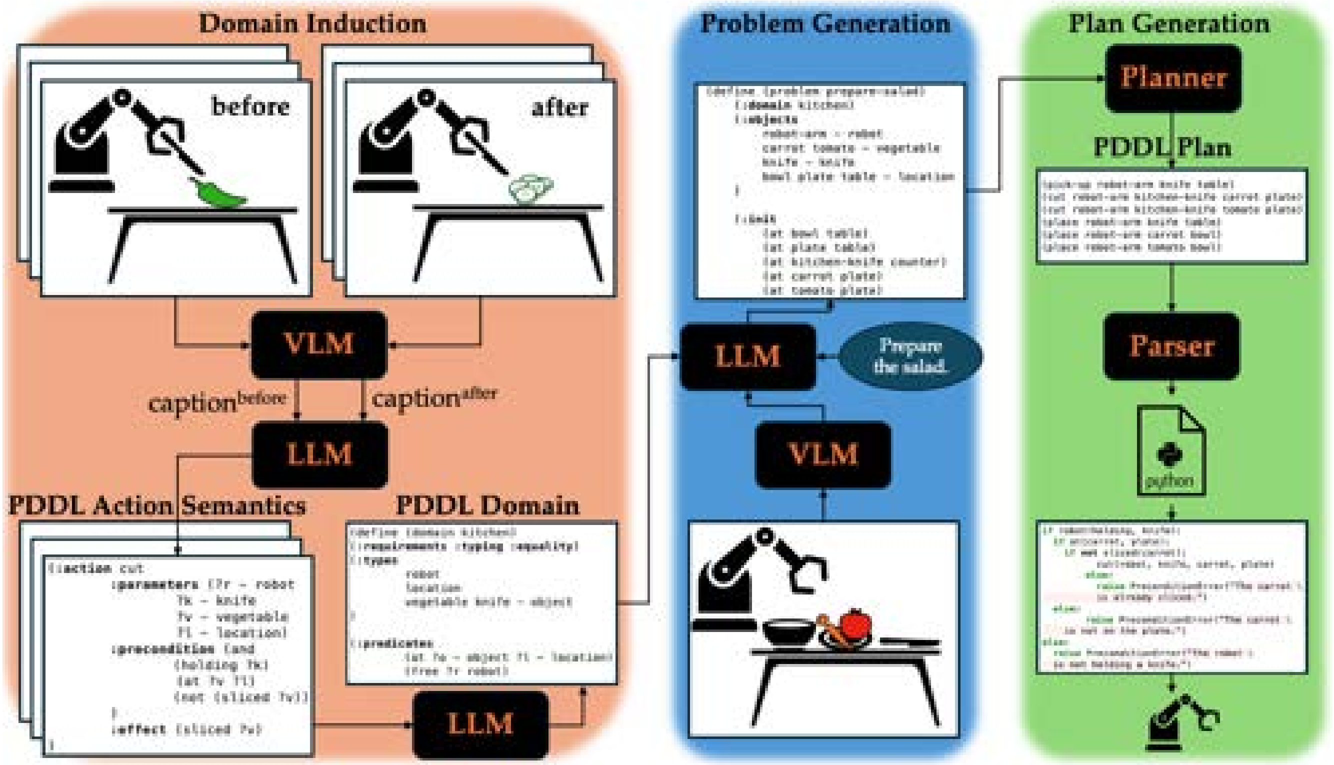
5.2 Background & Problem Statement

5.2.1 Planning Domain Definition Language (PDDL)

PDDL is a standardized means of encoding classical AI planning problems, developed by McDermott [72]. It clearly separates the representation of the planning problem into a domain file and a problem file. The domain file includes a lifted representation of the state space and action space. The state space defines the types of objects that are admissible in the environment and the allowable states in the form of boolean predicates that consume objects of the given types. PDDL supports hierarchical object relationships and property inheritance. This translates to hypernymy or IS-A [120] hierarchical relationships. For example, a kitchen knife IS-A utensil and a utensil IS-An object, hence the knife inherits all properties of its parent object types. The action space consists of STRIPS-style [118] actions with a set of objects as parameters, a set of preconditions that are prerequisite predicate values for action execution, and a set of effects that correspond to the resulting predicate values from successful action execution. The problem file enumerates the specific instances of the objects in the environment, the initial states, and the goal of the plan, or in other words, what state we want the world to be in at plan completion. The initial and goal states are both expressed in the form of predicates.

5.2.2 Problem Statement

Consider a mobile manipulation robot with an action space $\mathcal{A}$. The robot is tasked with executing a natural language instruction $\mathcal{L}$ that involves completing a task in the kitchen. We adopt a STRIPS [118] action representation that associates each action $a_i \in \mathcal{A}$ with its action semantics $\Phi_{a_i} = \{Pre(a_i), Eff(a_i)\}$, where $Pre(a_i)$ and $Eff(a_i)$ are sets of preconditions and effects of a_i, respectively. We assume access to a a camera that streams discrete visual observations of the kitchen scene: $\mathcal{O} = \{\mathcal{O}^{before}, \mathcal{O}^{after}, o^0\}$, where $\mathcal{O}^{before}$ and $\mathcal{O}^{after}$ are sets of observations before and after every admissible action $a_i \in \mathcal{A}$, respectively, and o^0 is an initial scene observation at the time of the instruction $\mathcal{L}$. Our goal is to infer the PDDL action semantics Φ for every action $a_i \in \mathcal{A}$ towards automatically generating the domain file f^D from the available observations. We can then use f^D to generate a plan in the form of an executable program f^{python} that visually grounds preconditions and effects. To execute such a plan, we assume access to a robot control API that maps high-level actions $\mathcal{A}$ to low-level control policies.



```
(:action cut
        :parameters (?r - robot
                ?k - knife
                ?v - vegetable
                ?l - location)
        :precondition (and
                (holding ?k)
                (at ?v ?l)
                (not (sliced ?v)))
        )
        :effect (sliced ?v)
)
```

```
(define (domain kitchen)
(:requirements :typing :equality)
(:types
        robot
        location
        vegetable knife - object
)

(:predicates
        (at ?o - object ?l - location)
        (free ?r robot))
```

```
(define (problem prepare-salad)
    (:domain kitchen)
    (:objects
        robot-arm - robot
        carrot tomato - vegetable
        knife - knife
        bowl plate table - location
    )

    (:init
        (at bowl table)
        (at plate table)
        (at kitchen-knife counter)
        (at carrot plate)
        (at tomato plate)
```

```
(pick-up robot-arm knife table)
(cut robot-arm kitchen-knife carrot plate)
(cut robot-arm kitchen-knife tomato plate)
(place robot-arm knife table)
(place robot-arm carrot bowl)
(place robot-arm tomato bowl)
```

```
if robot(holding, knife):
  if at(carrot, plate):
    if not sliced(carrot):
        cut(robot, knife, carrot, plate)
      else:
        raise PreconditionError("The carrot \
        is already sliced.")
  else:
      raise PreconditionError("The carrot \
  is not on the plate.")
else:
  raise PreconditionError("The robot \
  is not holding a knife.")
```

Figure 5.1: Schematic overview of our NL2PDDL2Prog system. **Domain Induction:** We use a VLM to obtain captions $c_{a_i}^{before}, c_{a_i}^{after}$ of an observation of the scene before ($o_{a_i}^{before}$) and after ($o_{a_i}^{after}$) every admissible action a_i. These captions are passed to an LLM to infer the action semantics Φ_{a_i}. The action semantics are fed back to the LLM to produce the entire PDDL domain f^D. **Problem Generation:** Given the derived domain, a natural language instruction $\mathcal{L}$, and a caption c^0 of an initial observation of the scene obtained by a VLM, the LLM generates a problem specification f^{prob}. **Plan Generation:** A classical planner receives the domain f^D and problem f^{prob} as input and produces a plan f^{plan}. This plan is parsed into a python executable program f^{python} that visually grounds action preconditions and effects at runtime through a VLM.

5.3 NL2PDDL2Prog

5.3.1 System Architecture

To solve this problem, we present a system with an architecture summarized in Fig. 5.1. Specifically, our system:

1. Infers the PDDL action semantics and domain given a set of scene observations $\{\mathcal{O}^{before}, \mathcal{O}^{after}\}$ before and after action execution.

2. Generates a PDDL problem file given a natural language instruction $\mathcal{L}$, a domain file f^D, and an initial scene observation o^0.

3. Uses a classical planner [121] to acquire a feasible plan f^{plan} given the generated domain f^D and problem file f^{prob}.

4. Parses the generated plan into a robot-executable python program f^{python} that visually grounds action preconditions and effects at runtime.

Our system is algorithmically summarized in Alg. 2. The following sections expand on its individual components.

5.3.2 PDDL Domain Induction from Visual Observations

To infer the action semantics Φ, we rely exclusively on scene observations assuming no prior domain knowledge. We obtain captions $c_{a_i}^{before}$ and $c_{a_i}^{after}$ from an observation before $o_{a_i}^{before}$ and after $o_{a_i}^{after}$ every admissible action $a_i \in \mathcal{A}$, respectively, by querying a VLM with the prompt: *"Provide a short caption of this image."*. We then feed $c_{a_i}^{before}$ and $c_{a_i}^{after}$ to an LLM as context and ask for the *"PDDL action semantics for the executed action"*. Note that we do not specify the action but rather let the model predict it from the context of the captions, assuming access to a set of visual observations of the scene

Algorithm 2

Input: A set of images before $\mathcal{O}^{before}$ and after $\mathcal{O}^{after}$ every action $a \in \mathcal{A}$, an image of the current state of the kitchen o^0, and a natural language instruction $\mathcal{L}$

Output: A robot-executable action plan in the form of a python program f^{python}

```
 1: $\mathcal{C} \leftarrow VLM(\mathcal{O}^{before}, \mathcal{O}^{after})$                ▷ Domain Induction
 2: $\Phi \leftarrow LLM(\mathcal{C})$
 3: $f^D \leftarrow LLM(\Phi)$
 4: $c^0 \leftarrow VLM(o^0)$                                          ▷ Problem Generation
 5: $f^{prob} \leftarrow LLM(f^D, c^0, \mathcal{L})$
 6: $f^{plan} \leftarrow FastDownward(f^D, f^P)$                        ▷ Plan Generation
 7: $f^{python} \leftarrow Parser(f^{plan})$
 8: **return** $f^{python}$
    ***function*** $Parser\big(\mathcal{A}^*, Pre(\mathcal{A}^*), Post(\mathcal{A}^*)\big)$
 9: **for** $a_i \in f^{plan}$ **do**
10:     **for** $pre_i \in Pre(a_i^*)$ **do**
11:         **if** $!VQA(pre_i)$ **then**
12:             raisePreconditionError($pre_i$)
13:         **end if**
14:     **end for**
15:     execute($a_i$)
16:     **for** $post_i \in Eff(a_i^*)$ **do**
17:         **if** $!VQA(eff_i)$ **then**
18:             raisePostconditionError($eff_i$)
19:         **end if**
20:     **end for**
21: **end for**
```

but without further domain-specific information. We find that using a single caption before and after an action produces action semantics that are biased towards specific action instances, for example slicing bread on the counter yields a `slice_bread` action that must necessarily take place on a `counter`. To generalize the inferred action semantics we obtain observations and captions from multiple different instances of the same action, such as slicing bread on the counter, slicing a tomato on the table etc. This yields a `slice` action that is executable on any valid location or surface. As the model might infer a nominally different but semantically very similar action name for different pairs of images (e.g. *cut* and *slice*), we keep the most common inter-sample occurrence as the main action verb. At the same time, we retain the derived action semantics if an action verb is semantically similar to the most common one by comparing their cosine similarity. Using a sentence transformer [122], we embed every new action verb and take its cosine similarity to the existing action verb. If the similarity is below an empirically tuned threshold, we discard the output for that pair of images, otherwise, we append it to the existing samples. After acquiring the LLM predictions for all samples, we concatenate the predictions and feed them to an LLM asking for the "*generalized PDDL action semantics*". Finally, we pass the generalized action semantics Φ to the LLM which infers the entire PDDL domain, combining the different object types and predicates from the different samples and filling in any missing ones.

5.3.3 Problem File Generation

To generate the PDDL problem file f^{prob}, we prompt an LLM with a natural language instruction $\mathcal{L}$, the domain file f^D generated in the previous step, and a caption c^0 of the initial scene observation o^0 that we acquire from a VLM. The main functionality of the LLM is to derive the predicates corresponding to the initial state and the goal conditions required to complete $\mathcal{L}$. This is achieved by matching $\mathcal{L}$ to the world representation encoded in f^D and the natural language description c^0 of the initial scene observation o^0.

5.3.4 Plan Generation & Parsing

To generate the plan f^{plan} given the domain f^D and problem specifications f^{prob}, we use Fast Downward[1] [121], a heuristic search-based classical planning system widely used in the literature [6, 52–55]. The planner generates a finite sequence of admissible actions $\mathcal{A}^*$, if there is one, and produces an output file f^{plan} listing these actions and the specific parameter values used for solving the input planning problem. Based on the structure of this file, we synthesize a parser that matches the parameter values to lifted action definitions in the domain file and translates the generated plan into a python program that visually grounds preconditions $Pre(\mathcal{A}^*)$ and effects $Eff(\mathcal{A}^*)$. An action is only executed if all preconditions are satisfied. This is achieved by nested conditional statements that query a VLM with question templates corresponding to each predicate. If any of these preconditions does not hold, an exception is raised informing the user about the specific source of failure. Action effects are similarly handled with conditional statements after action execution, halting program execution and displaying the error source if any of them is not satisfied (an example is shown in Fig. 5.6). The parser is algorithmically summarized in Alg. 2.

5.4 Experiments

5.4.1 Data Collection

Our domain induction module requires a set of visual observations before $\mathcal{O}^{before}$ and after $\mathcal{O}^{after}$ the execution of every admissible action. We consider the following action space that includes some of the most frequent action verbs in the cooking domain [123] $\mathcal{A}$ ={`pick`, `place`, `pour`, `cut`, `open`, `close`}. We obtain a dataset consisting of robot demonstrations and human demonstrations. The setup for the robot demonstrations (shown in Fig. 5.2) consists of a Sawyer robot used in Zero-G mode allowing us to freely move the arm to various poses, and a PS4 controller. The controller is configured

[1]We use the implementation in https://github.com/aibasel/downward.

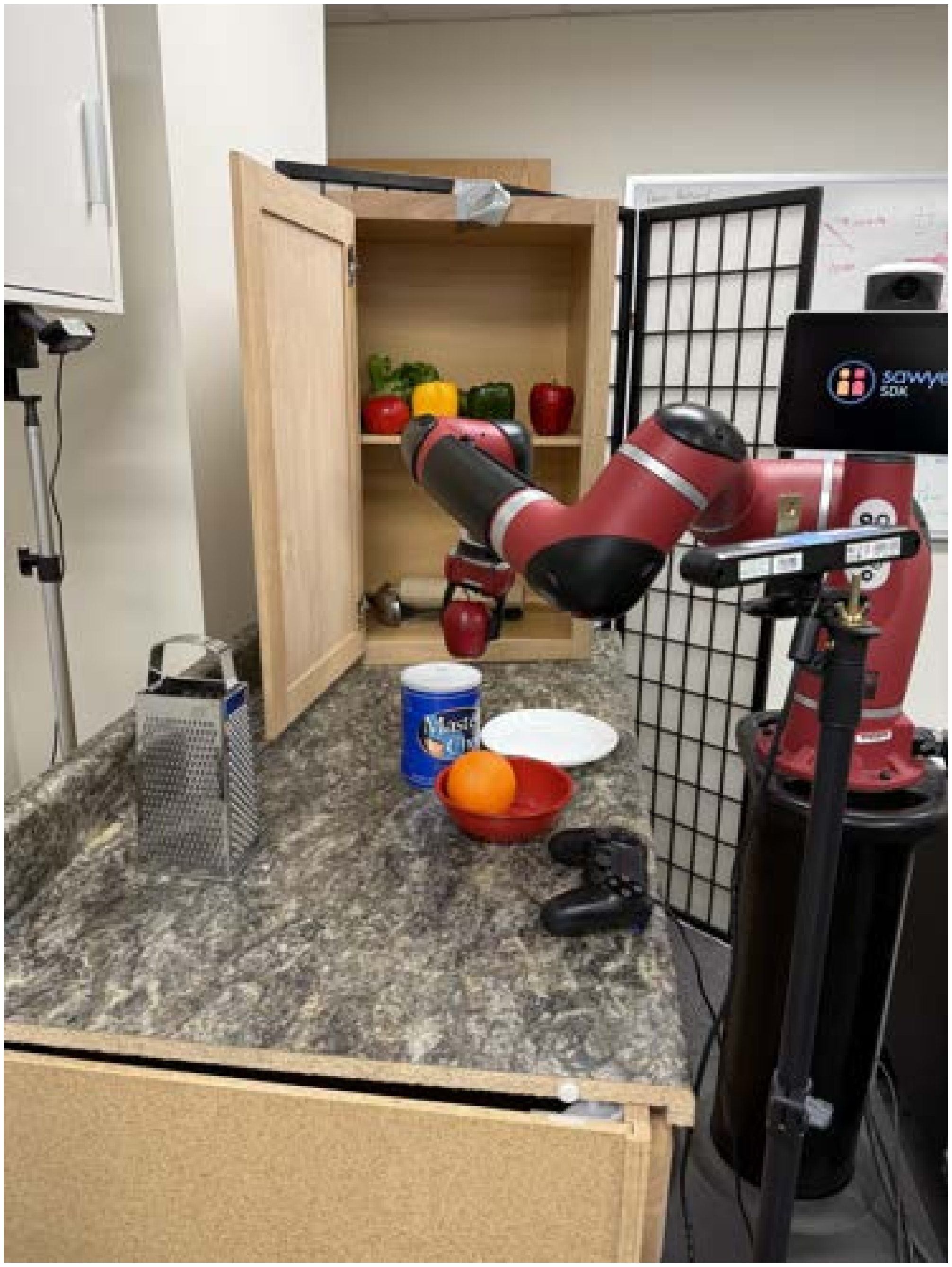


Figure 5.2: Our setup for recording robot demonstrations in the lab consists of a Sawyer robot in Zero-G mode, a PS4 controller to take pictures using a camera mounted on a tripod and open or close the gripper.

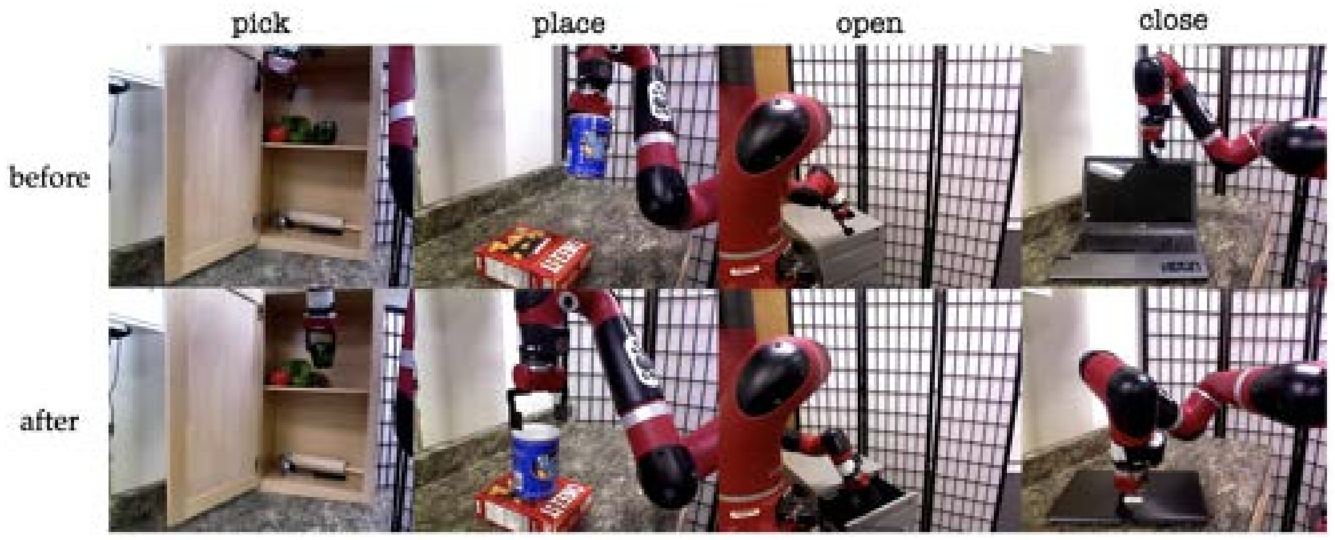


Figure 5.3: Robot demonstrations recorded in the lab illustrating snapshots before and after a robot executing the `pick`,`place`,`open`,`close` actions.

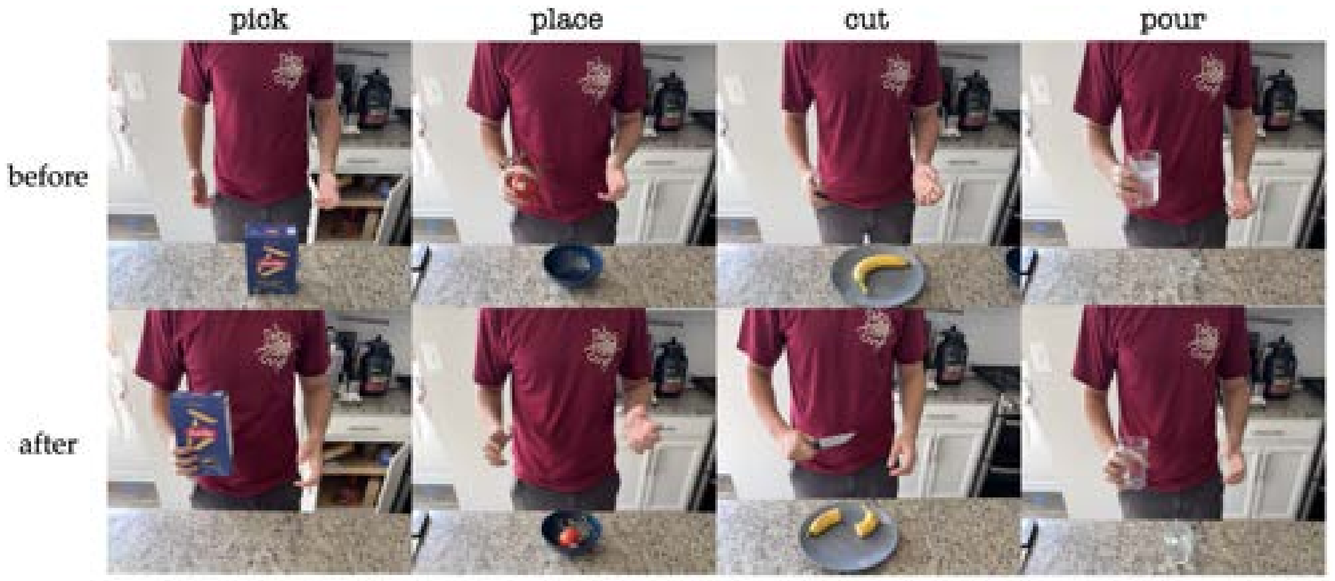


Figure 5.4: Human demonstrations recorded in a real kitchen illustrating snapshots before and after a human executing the `pick`,`place`,`cut`,`pour` actions.

to trigger a camera to take pictures (shown in Fig. 5.3) and open or close the gripper. We obtain human demonstrations by taking pictures of a human performing actions in a real kitchen. These pictures are snapshots of different kitchen configurations corresponding to preconditions and effects of every action $a_i \in \mathcal{A}$ as a result of a human executing these actions, as shown in Fig. 5.4.

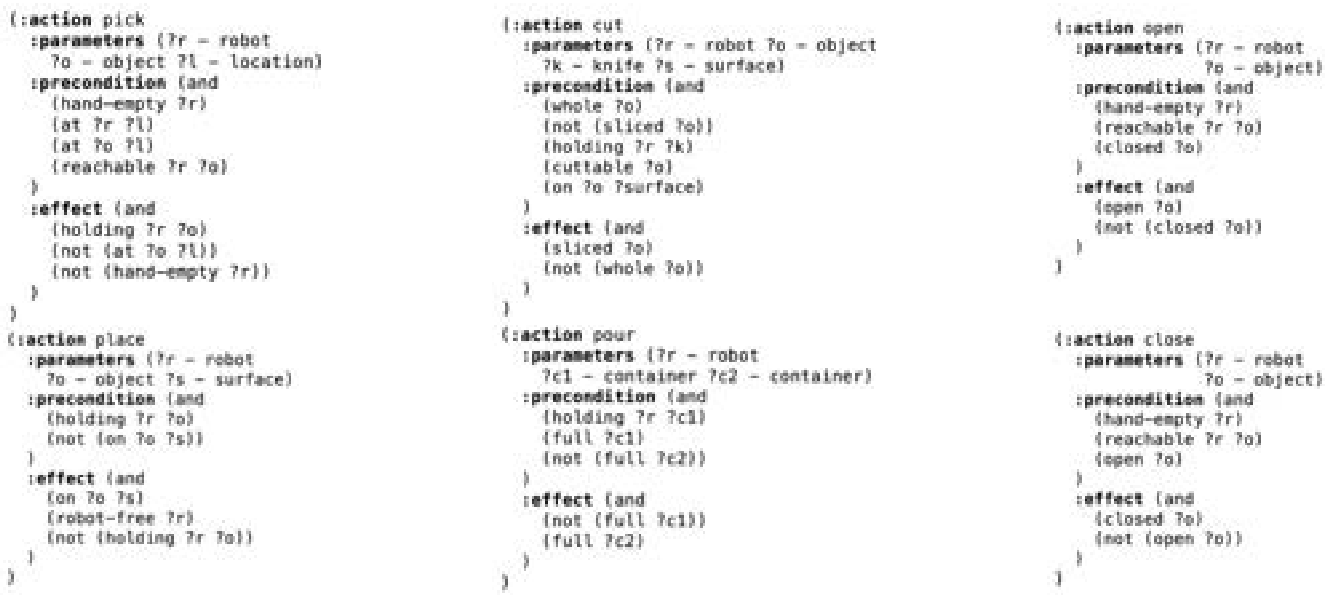
5.4.2 Domain Induction Analysis

We analyze the ability of our system to infer the correct action semantics Φ from the generated captions $\mathcal{C}$. For evaluation, we compare the accuracy of the generated $\tilde{\Phi}_{a_i}$ for every action $a_i \in \mathcal{A}$ with manually-crafted ground truth action semantics Φ_{a_i}. If any of the preconditions or postconditions does not match the ground truth, we consider the prediction incorrect. The results in Table 5.1 show that per-sample precondition (Pre) and effect or postcondition (Post) predictions are partially correct. However, Action Recognition (AR) is 100% across all actions both in the robot and in the human demonstration dataset. This means that we **always** correctly recognize the executed action without explicitly including the action name or any other hints in the input prompt to the LLM, but rather by using the generated captions as intermediate annotations.

By concatenating all generated action semantics for every action at a time and prompting the LLM for the "*generalized action semantics*" we obtain all action semantics and therefore the entire domain shown in Fig. 5.5 that matches the ground truth domain used for comparison. The model corrects the previously observed per-sample overfitting to specific action instances (e.g. `slice_bread`) by combining the predictions for all samples of an action and producing the action semantics of the general action (e.g. `slice`) that was executed.

5.4.3 Visually Grounding Action Preconditions and Effects

To highlight the practical implications of using a PDDL action representation we demonstrate the usage of the python executable PDDL plan generated using the inferred PDDL domain and our parser



```
(:action pick
  :parameters (?r - robot
    ?o - object ?l - location)
  :precondition (and
    (hand-empty ?r)
    (at ?r ?l)
    (at ?o ?l)
    (reachable ?r ?o)
  )
  :effect (and
    (holding ?r ?o)
    (not (at ?o ?l))
    (not (hand-empty ?r))
  )
)
(:action place
  :parameters (?r - robot
    ?o - object ?s - surface)
  :precondition (and
    (holding ?r ?o)
    (not (on ?o ?s))
  )
  :effect (and
    (on ?o ?s)
    (robot-free ?r)
    (not (holding ?r ?o))
  )
)
```

```
(:action cut
  :parameters (?r - robot ?o - object
    ?k - knife ?s - surface)
  :precondition (and
    (whole ?o)
    (not (sliced ?o))
    (holding ?r ?k)
    (cuttable ?o)
    (on ?o ?surface)
  )
  :effect (and
    (sliced ?o)
    (not (whole ?o))
  )
)
(:action pour
  :parameters (?r - robot
    ?c1 - container ?c2 - container)
  :precondition (and
    (holding ?r ?c1)
    (full ?c1)
    (not (full ?c2))
  )
  :effect (and
    (not (full ?c1))
    (full ?c2)
  )
)
```

```
(:action open
  :parameters (?r - robot
               ?o - object)
  :precondition (and
    (hand-empty ?r)
    (reachable ?r ?o)
    (closed ?o)
  )
  :effect (and
    (open ?o)
    (not (closed ?o))
  )
)

(:action close
  :parameters (?r - robot
               ?o - object)
  :precondition (and
    (hand-empty ?r)
    (reachable ?r ?o)
    (open ?o)
  )
  :effect (and
    (closed ?o)
    (not (open ?o))
  )
)
```

Figure 5.5: The produced PDDL action semantics Φ synthesizing the entire PDDL domain.

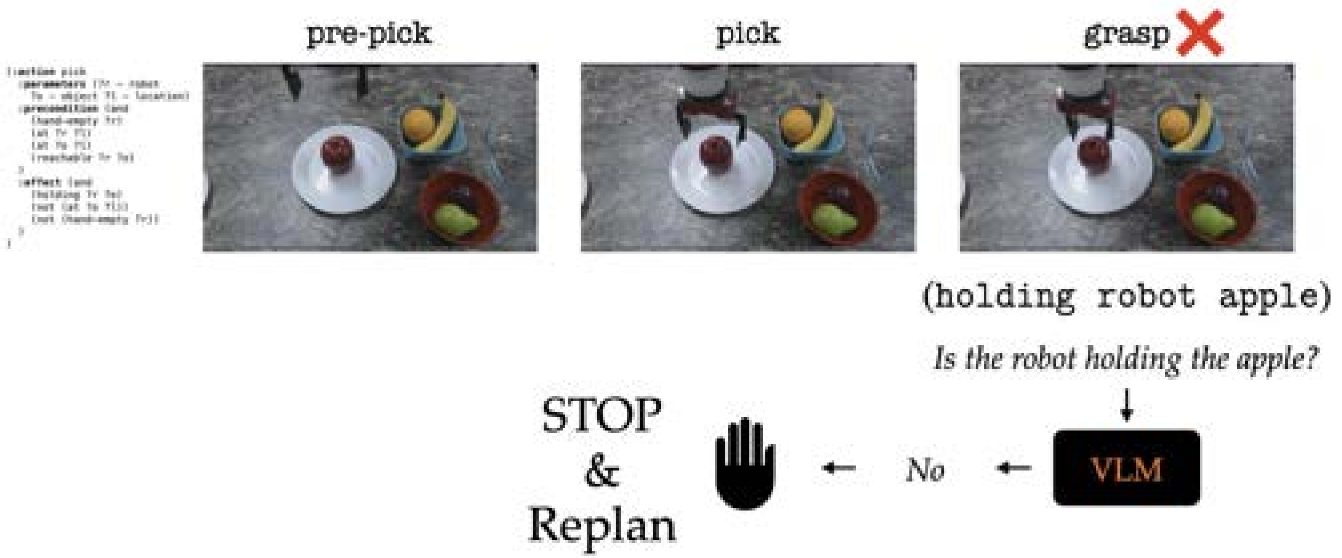


Figure 5.6: Detecting failed action execution by visually grounding PDDL action postconditions allows us to stop plan execution and repair the plan.

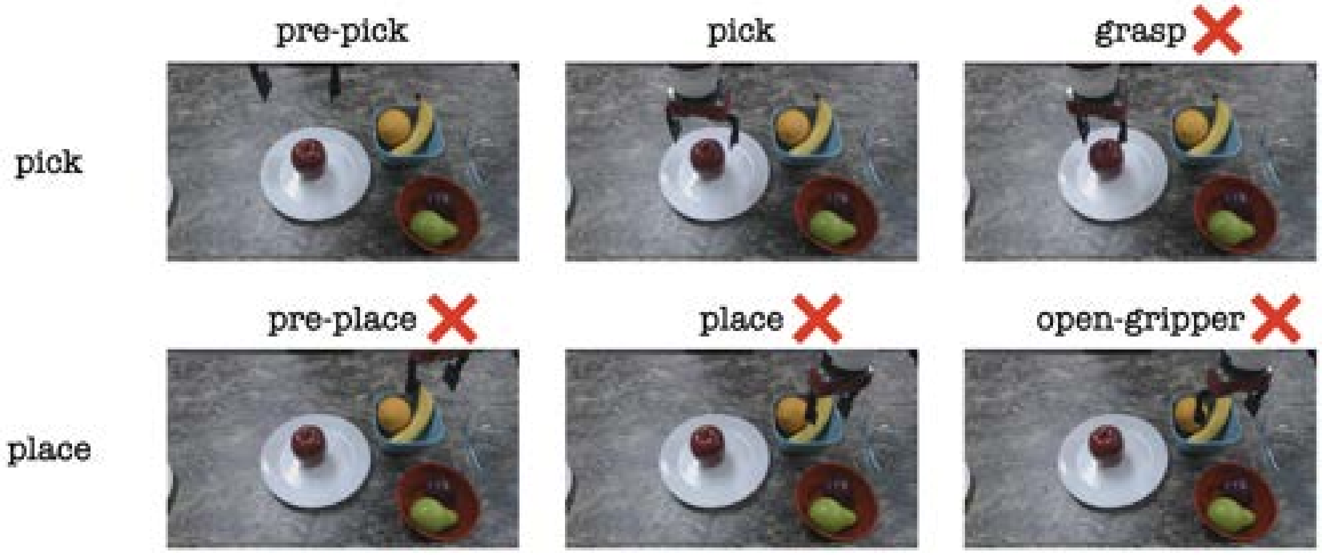
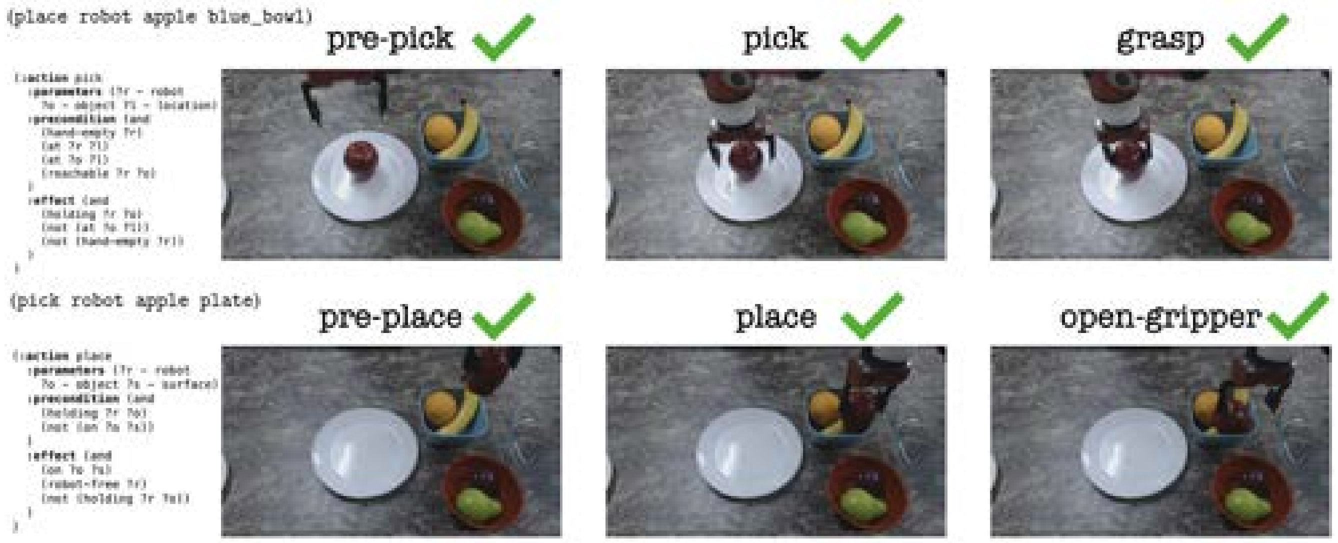


Figure 5.7: The lack of a mechanism to validate action postconditions can lead to redundant action execution, resulting in wasted resources and time.



```
(place robot apple blue_bowl)

(:action pick
  :parameters (?r - robot
    ?o - object ?l - location)
  :precondition (and
    (hand-empty ?r)
    (at ?r ?l)
    (at ?o ?l)
    (reachable ?r ?o)
  )
  :effect (and
    (holding ?r ?o)
    (not (at ?o ?l))
    (not (hand-empty ?r))
  )
)

(pick robot apple plate)

(:action place
  :parameters (?r - robot
    ?o - object ?s - surface)
  :precondition (and
    (holding ?r ?o)
    (not (on ?o ?s))
  )
  :effect (and
    (on ?o ?s)
    (robot-free ?r)
    (not (holding ?r ?o))
  )
)
```

Figure 5.8: Successful action execution by visually grounding PDDL action preconditions and effects.

Robot Demonstrations

Task	Action			
	`pick`	`place`	`open`	`close`
AR (%)	**100**	**100**	**100**	**100**
Pre (%)	60	100	35	40
Post (%)	100	100	80	65

Human Demonstrations

Task	Action			
	`pick`	`place`	`cut`	`pour`
AR (%)	**100**	**100**	**100**	**100**
Pre (%)	70	100	90	85
Post (%)	95	100	100	90

Table 5.1: Comparison of per-sample generated PDDL action semantics $\tilde{\Phi}_{a_i}$ for every action $a_i \in \mathcal{A}$ to ground truth action semantics Φ_{a_i}. AR stands for Action Recognition, Pre stands for Precondition prediction, and Post corresponds to Postcondition prediction.

for the task of picking up an apple located on a plate and placing it in an adjacent blue bowl. In Fig. 5.6 we show how visually grounding a failed action execution resulting from a pose estimation error and a failed grasp allows us to detect the error at runtime, stop plan execution, and take action to repair the plan. On the contrary, Fig. 5.7 shows how the lack of such a mechanism fails to detect runtime errors and can lead to the execution of redundant actions, resulting in wasted resources and time. Finally, Fig. 5.8 demonstrates a successful action execution instance that stems from correct pose estimation and visual grounding of the preconditions and effects of the `(place robot apple blue_bowl)` and `(pick robot apple plate)` generated PDDL plan steps.

5.5 Limitations & Future Work

Error Propagation: Our system architecture consists of multiple layers that rely on a call to a generative model. Besides the potential latency and cost resulting from these calls, this also means that an error at an initial stage will propagate to the subsequent layers and will lead to an inflated error on the generated problem and domain file. As a remedy, we need to incorporate mechanisms that verify the correctness of the output of each layer, such as cross-validating the caption generation process through different models, or ensuring that problem files and plans are verified by the VAL validation software [75].

Different Domains for Different Environments: The idea of grounding preconditions and effects to visual observations relies on images taken before and after executing an action. The mechanics of these actions might differ depending on the environments where these images have been captured. For example, our system might ground `open_door` to a top-down movement of the gripper if dealing with a lever door handle, but in a different domain the same action might be grounded to a grasp-and-rotate motion if dealing with a round door knob. This highlights the significance of the notion of affordances [105], as highlighted in our future work ideas in Chapter 7. Knowing the affordances of an object in a new environment can assist in detecting potential mismatches between the learned action semantics and the action semantics corresponding to that environment, enabling early error detection before action execution.

Task Complexity: We collected demonstrations for a set of essential but simple actions that are commonly observed in a kitchen and demonstrated the usage of our NL2PDDL2Prog system in a simple pick-and-place task. However, executing more complex recipes requires supporting more intricate actions that might not be easily or accurately recognized by snapshots before and after action execution. This means that we might require additional observations during the execution of an action to fill in some missing information.

Chapter 6: Robotic System

6.1 Overview

Our work has focused on translating natural language instructions to action (or task) plans. These action plans consist of high-level actions that are often modeled under the assumption that all high-level actions are executable by internal simulator controllers. However, mapping these actions to low-level robot control commands in a real environment is not a trivial problem and requires a combination of perception and planning mechanisms, as well as a framework for them to communicate. To bridge this gap, we set up a real-world robotic system capable of receiving natural language instructions and executing minimal manipulation tasks. Although our system is applicable to various domains, we focus on executing high-level manipulation tasks in the kitchen, and specifically tasks that can be implemented by sequences of actions that reduce to picking and placing or simply grasping an object. Examples include making a burger by stacking ingredients on a bun, or making a salad by picking and placing every ingredient from the kitchen counter surface into a bowl.

In this section we describe the architecture of our robotic system. A user provides a natural language instruction based on a set of ingredients on a kitchen counter. The instruction is converted to text using speech-to-text. An image displaying the initial state of the counter is fed from the camera to an object recognition module that returns the labels of the objects. To kickstart object tracking, we follow the simple approach of printing ArUco markers [124], gluing them on cardboard, and then mounting them on top of objects, but our system is modular and can be combined with any approach capable of

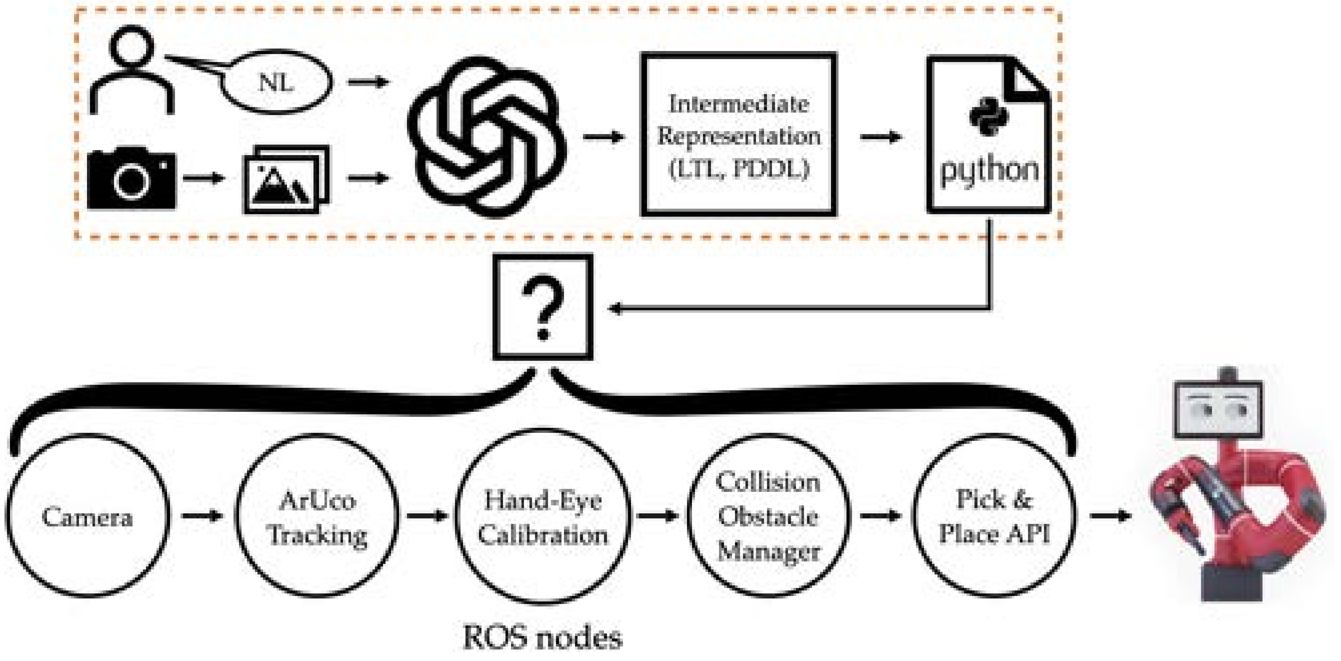


Figure 6.1: The ROS architecture of our robotic system, bridging the gap between higher-level action plans produced by Cook2LTL (Chapter 3) and NL2PDDL2Prog (Chapter 5), and low-level motion control handled by the Pick-and-Place python API that we developed.

publishing pose estimation information from RGB images. Based on the labels of the detected objects, we create an internal dictionary that maps every labeled object to a marker id. The labeled objects, formulated as a textual scene description, along with the textual instruction are then passed to Cook2LTL, NL2PDDL2Prog, or a vanilla LLM invoked through an API call, treating it as an end-to-end task planner. Once we acquire the action plan expressed in the action vocabulary of a Pick-and-place python API that we developed, we kickstart our robotic system for recipe execution through Pick-and-Place API calls with marker ids as parameters.

We use the Sawyer robot (by Rethink Robotics) with python 2.7 and ROS Kinetic, a single-arm collaborative robot with 7 degrees of freedom, and a Logitech C920 HD webcam to capture our workspace, which is a $123 \times 63 \times 91cm$ kitchen counter. The overall ROS architecture of our system is shown in Fig. 6.1.

6.2 Camera Calibration

We begin by calibrating the camera's intrinsics parameters under a 1920×1080 resolution using the ROS `camera_calibration` package. We print a checkerboard pattern, glue it on cardboard and move it around so that it is captured at various tilts, rotations, and distances within the camera's field of view. The ROS package returns a YAML file with the focal lengths f_x, f_y, the principal point (c_x, c_y), and the radial distortion k_1, k_2, k_3 and tangential distortion p_1, p_2 coefficients (under the Plumb Bob model). These parameters are then passed to a downstream node using the `image_proc` ROS package to undistort and rectify the camera input. The computed camera matrix and distortion coefficients can be seen in the following equations:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1443.895 & 0 & 933.989 \\ 0 & 1446.086 & 534.447 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.1}$$

$$D = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} = \begin{bmatrix} 0.054 & -0.161 & 0.003 & -0.001 & 0 \end{bmatrix} \tag{6.2}$$

The rectified camera input is passed to an `aruco_detect` node that tracks the poses of the markers with respect to the camera coordinate system and a subscriber node `fiducial_to_tf` publishes these poses for downstream usage.

6.3 Hand-Eye Calibration

To perform manipulation actions in our workspace given visual observations from the camera, our robot needs to be able to generate motion plans and hence compute the inverse kinematics to reach any 3D point in the workspace with its end-effector. When using ArUco markers, we can easily obtain the pose of a marker with respect to the camera coordinate system, however the robot inverse kinematics and motion planning operate with respect to the robot coordinate system. Accordingly, we need to establish a mapping of an arbitrary 3D point in the workspace to the (base) coordinate system of the robot. To this end, we perform hand-eye calibration, a procedure that is conceptually aligned with the notion of hand-eye coordination. In a cognitive perspective, hand-eye coordination is a form of multisensory integration that amounts to synchronized motor control of eye movement with hand movement, relying on the processing of visual input and proprioception to guide manipulation actions such as reaching or grasping.

While the core idea remains the same, there are two different variants of the hand-eye calibration process. In the first "eye-to-hand" scenario, the camera is placed at a fixed location in the scene and a calibration object of known geometry is mounted on the end-effector. The specific mounting location should match the location associated with a known transform with respect to the robot base. In the second "eye-in-hand" scenario, the camera is mounted at a point with a known robot transform and the calibration object is placed at a fixed location. Following the former "eye-to-hand" method, we attach the camera to a tripod so that it captures a panoramic view of the kitchen counter and a printed ArUco marker glued on a piece of cardboard at the edge of the right gripper finger tip of the Sawyer (corresponding to the known `right_gripper_r_finger_tip` transform). The transformation of the right gripper tip with respect to the base of the robot is known from the robot geometry and from the DH (Denavit-Hartenberg) parameters [125] so when we mount the marker exactly on that point, we consider the pose of the marker with respect to the robot base coordinate system to be identical with the known pose of the gripper tip (see Fig. 6.3) Fig. 6.2 shows the robot with the attached marker on the gripper. Leveraging the `aruco_detect` and `fiducial_to_tf` nodes that we set up for object tracking, we visualize the detected marker pose in rviz along with the right finger tip robot transform and properly align the orientation of the marker so that it matches the orientation of the gripper tip, as shown in Fig. 6.3. The `aruco_tf` node prompts the user to record a number of samples that consist of different poses of the marker at different robot configurations in the workspace, making sure that when a sample is recorded, the marker is detectable. Upon gathering 50 samples, we apply the Umeyama

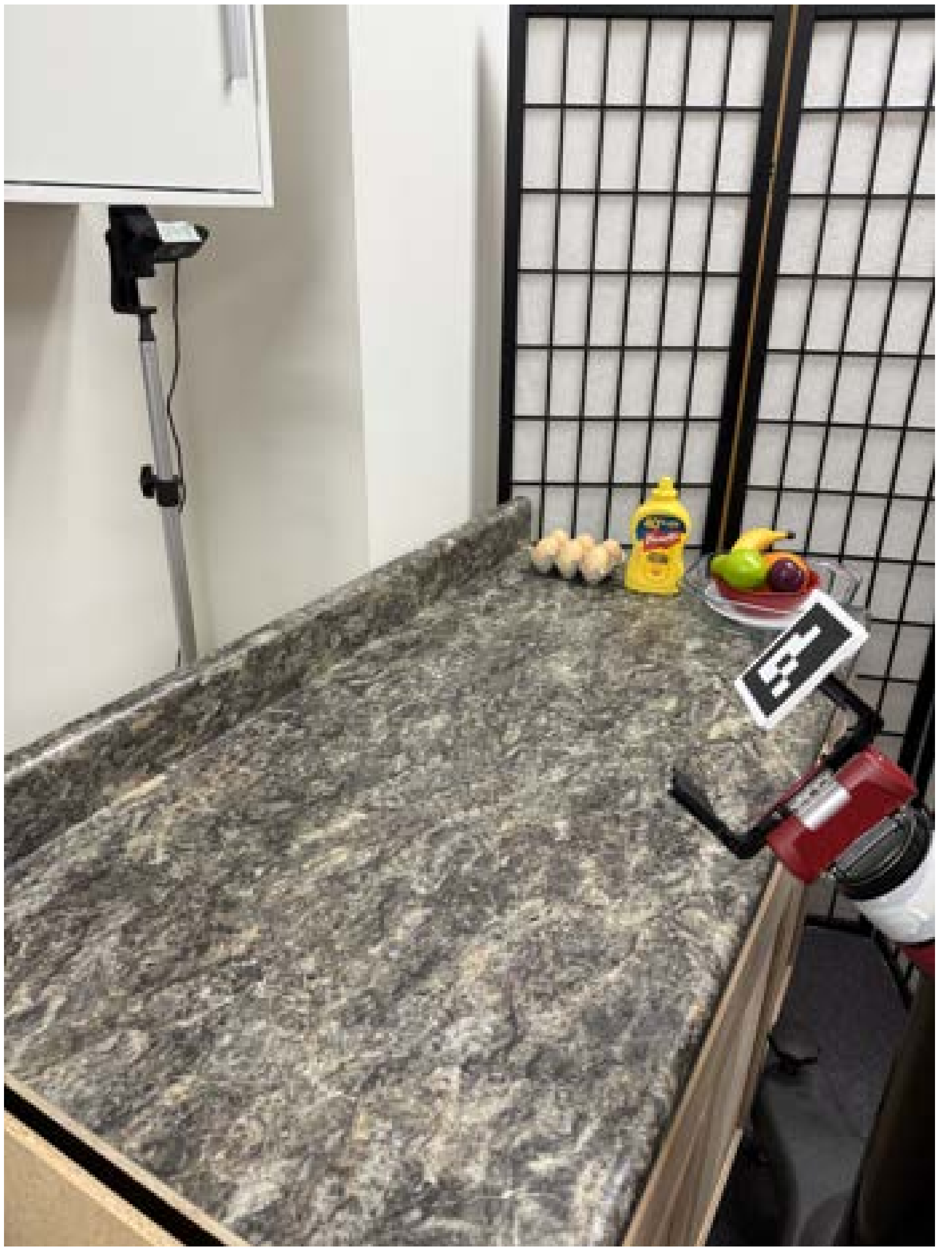


Figure 6.2: The hand-eye calibration system consisting of an HD camera fixed on a tripod and the Sawyer robot with an attached ArUco marker on its `right_gripper_r_finger_tip` link.

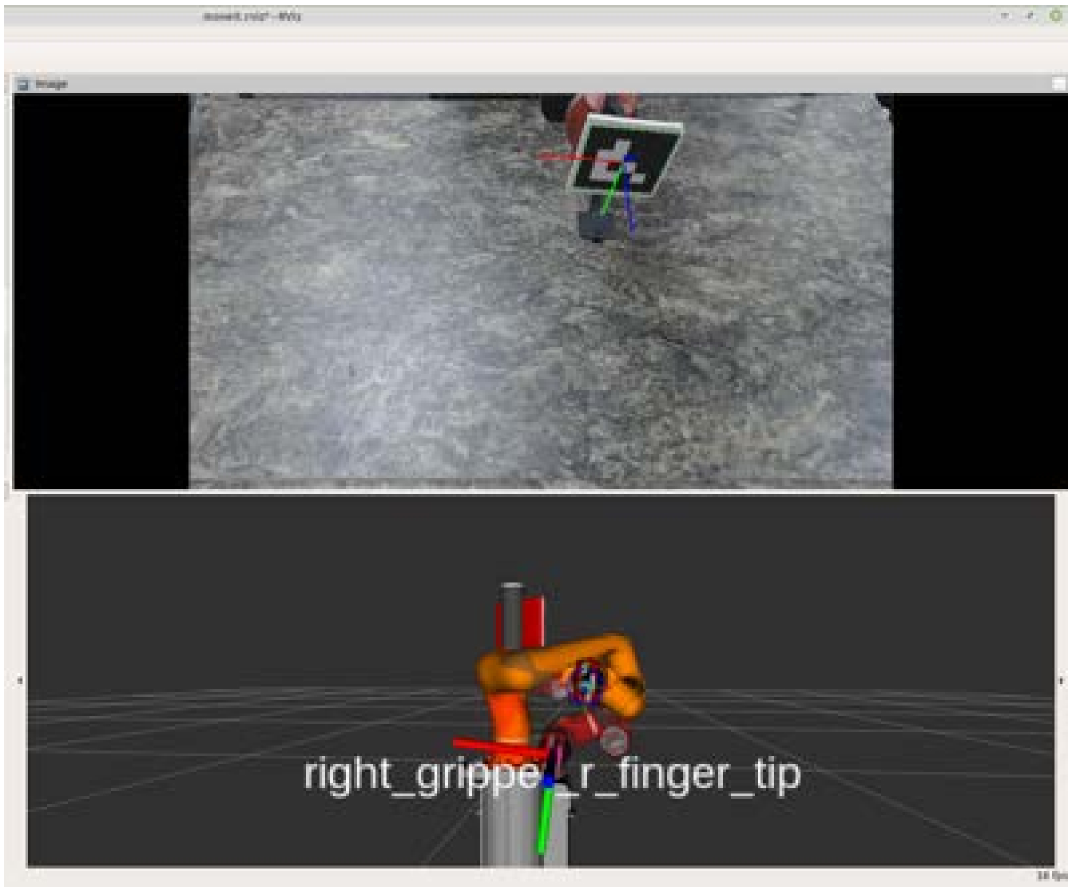


Figure 6.3: After mounting the marker on the right gripper tip of the robot we visualize the tracked pose in rviz and ensure it is properly aligned with the `right_gripper_r_finger_tip` transform.

algorithm [126] that provides a closed-form solution in the form of a rotation and a translation matrix that minimize the mean squared error between the two sets of points. Finally, we obtain the extrinsic camera parameters (a rotation and a translation matrix) with respect to the robot base coordinate system, save them to a YAML file and let the `aruco_tf` node continuously advertise the camera-to-robot-base transform. When launched with a `_load_calibration:=true` argument, the node skips the sample recording phase and directly broadcasts the transform, loading it from the YAML file.

6.4 Planning

For planning we launch the pre-configured Sawyer MoveIt ROS node that imports the robot in a MoveIt planning scene. We build a `collision_object_manager` node that adds static and dynamic objects in the scene through an `ApplyPlanningScene` ROS service. We start by importing the kitchen counter to the scene as a static obstacle. After manually counting the height of the counter, we enter "Zero-G" mode on Sawyer, allowing us to freely move the arm around, and record the 3D positions of the gripper tip after maneuvering it to touch the four corners of the counter. We then add it to the scene as a `CollisionObject` of type `BOX` with the recorded coordinates of the four corners and the measured height as parameters. In the final version of our system, we attach a marker to every corner of the counter, extract its 3D position with respect to the robot base from the hand-eye calibrated system, and import the counter based on these positions, without having to move the arm or manually measure the height of the counter.

To dynamically add the objects that can be manipulated to the scene, we make two simple assumptions: i) all objects can be approximated as boxes, and ii) the dimensions of the objects are known to us. Based on these assumptions, we create a dictionary that maps every object name to the marker id attached to it, as well as its dimensions, and continuously add the objects as `CollisionObjects` of type `BOX` to the scene with a refresh rate of 10Hz if they are being detected by the `aruco_detect` node. Fig. 6.4 illustrates the tracked markers by the `aruco_detect` node and their transforms

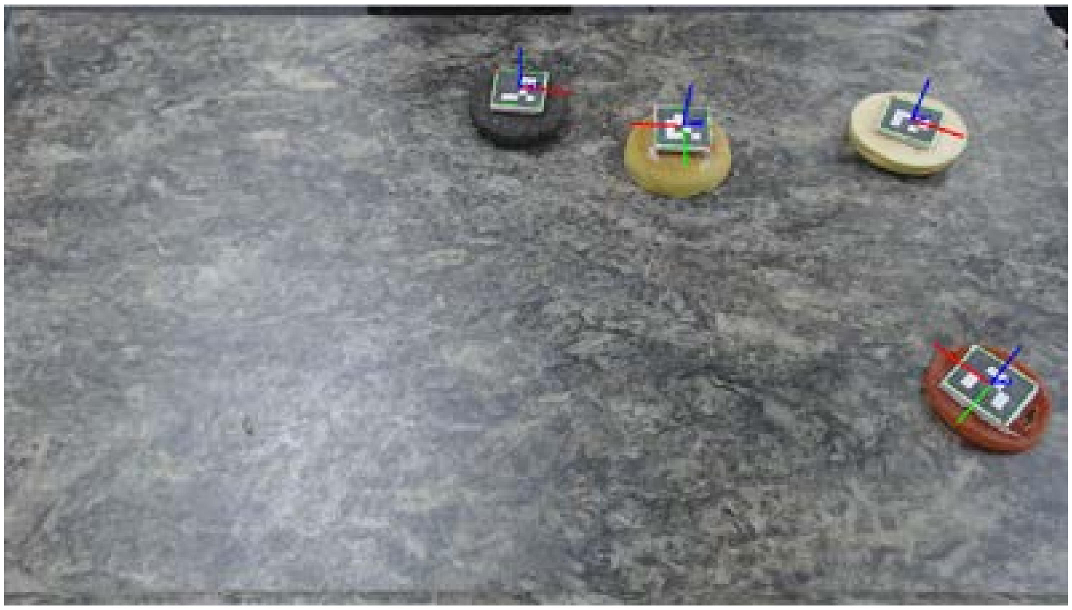


Figure 6.4: Transforms of the markers tracked through the `aruco_detect` node and published through the `fiducial_to_tf` node in rviz.

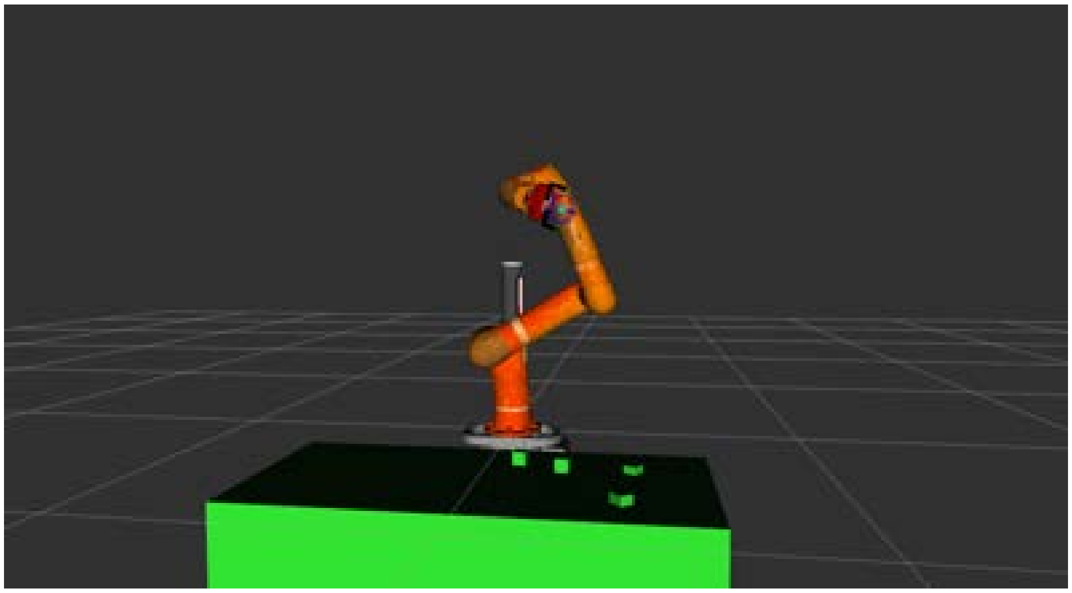


Figure 6.5: The planning scene in MoveIt after importing the kitchen counter as a static obstacle and the dynamic objects as continuously updated collision objects. This scene corresponds to the configuration shown above in Fig. 6.4.

published by the `fiducial_to_tf` node. Fig. 6.5 shows the corresponding planning scene in MoveIt after importing the kitchen counter as a static obstacle and the objects as continuously updated collision objects.

Picking and placing is handled by a python API-like script that communicates with the `collision_object_manager` through a `manage_tracked_object` ROS service used to handle MoveIt collisions during picking and placing. More specifically, commanding the gripper to move to a pose where an object lies between the gripper fingers leads to a collision in MoveIt. To remedy this, we temporarily modify the `AllowedCollisionMatrix (ACM)` during manipulation to allow collisions between the robot gripper links and the object to be manipulated. Furthermore, as every object has a marker attached on top of it, marker detection is expected to cease when the gripper attempts to grasp it due to occlusion. As a workaround, once the grasp is complete, we approximate continuously tracking the object by attaching it to a known robot transform on the gripper (`right_hand`) and computing its pose as relative to that link. Once the object is released and the gripper moves away, the object is detached and the `aruco_detect` node resumes tracking its pose. We limit manipulation actions to top-down grasping and placing for simplicity and separate grasping to: i) move to pre-grasp pose, ii) move to grasp pose, iii) close gripper, iv) attach object to `right_hand` transform, v) retreat (move to pre-grasp pose), and similarly, placing to i) move to pre-place pose, ii) move to place pose, iii) open gripper, iv) detach object, and then v) retreat (move to pre-place pose). This simplified sequential pick-and-place policy is illustrated in Fig. 6.6.

Our pipeline consists of multiple connected components, each contributing to an accumulated error that leads to a misalignment when we deploy a control command to send the gripper to a 3D point above the center of a marker. First, the hand-eye calibration yields some error in pixels. We can estimate this error by tracking a marker in our workspace and obtaining one of its corner coordinates. Multiplying with the hand-eye transformation matrix yields the pose of that point with respect to the robot base. Then, manually guiding the gripper to that point and comparing the resulting gripper pose to that returns the desired error. The placement of the markers on the objects is another important source of error, both due to

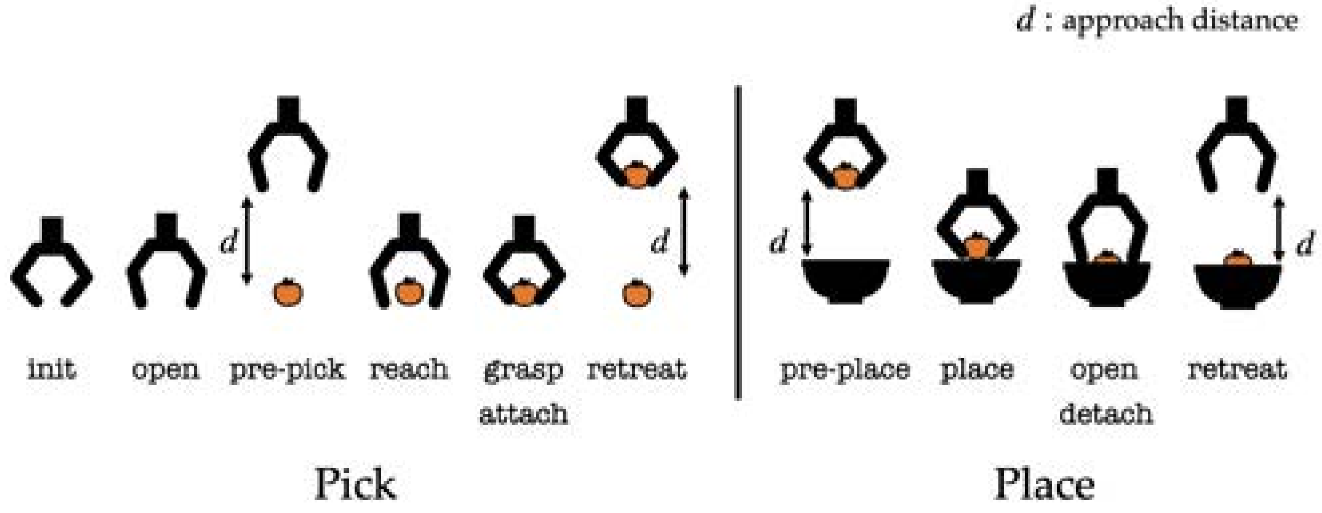


Figure 6.6: Picking and placing is performed by following this simplified top-down grasping policy. We attach and detach the manipulated object to and from a known robot transform after picking and after placing, respectively, to approximate tracking during the grasp.

the approximate manual placement in the center of the objects with cardboard and double-sided adhesive VHB (Very High Bond) tape, but also because of the uneven surface of some of the objects. Additionally, detecting a marker of low printing quality or a marker that is far from the camera can be intermittent, causing flickering in the MoveIt planning scene. Finally, Sawyer is a legacy robot and therefore not as precise as more contemporary robots. Hence, there is no guarantee that a control command to reach a 3D point accurately reaches that point at every attempt.

To mitigate the accumulated error from the components of our pipeline, we implement a PID control loop to align the X and Y coordinates of the `right_gripper_tip` to the center of a marker after the pre-grasp or pre-place motion. We tune one controller for the X and one for the Y component and set an error threshold of $3mm$. The error signal is computed by calculating the difference between the X and Y coordinates of the gripper and the marker. However, the coordinates of the marker with respect to the robot base are obtained through the hand-eye calibration so any error resulting from it will be propagated regardless of the PID control correction. Therefore, this control loop is introduced as an additional verification mechanism, or in other words, as an effort to mitigate the error from the rest of the downstream components. To further improve the accuracy of our system we would need to mount an additional camera on the arm right above the gripper and align the gripper with the center of the marker. We could achieve that through visual servoing, a PID control loop to align the center of the marker to the center of the camera feed, or to an offset from the center, accounting for the exact pose that the additional camera would be mounted at.

6.5 Demonstration - Recipe Implementation

We integrate our robotic system with Cook2LTL and NL2PDDL2Prog to demonstrate its functionality on the simple recipe of making a burger using artificial food items. The scene can be seen in Fig. 6.7. The implementation of the recipe reduces to stacking the burger ingredients on top of each other by executing a series of pick and place actions. Every ingredient has an ArUco marker on its top surface, used for tracking its pose both when it's being picked up or for placement, serving as a target receptacle. We use a simple object detector to acquire a list of objects that are visible in our scene {`top_bun`, `bottom_bun`, `beef_patty`, `tomato`}, which gets passed to Cook2LTL along with a natural language instruction from the user asking the robot to "*make a burger*". Cook2LTL returns the following task specification:

$$\begin{aligned}\phi =&F(\texttt{make}(\texttt{WHAT}:\texttt{burger})) = \\ &F(\texttt{pick\_up}(\texttt{WHAT}:\texttt{beef\_patty})\wedge \mathcal{F}(\texttt{place}(\texttt{WHAT}:\texttt{beef\_patty},\texttt{WHERE}:\texttt{bottom\_bun})\wedge \\ &F(\texttt{pick\_up}(\texttt{WHAT}:\texttt{tomato})\wedge \mathcal{F}(\texttt{place}(\texttt{WHAT}:\texttt{tomato},\texttt{WHERE}:\texttt{beef\_patty})\wedge \\ &F(\texttt{pick\_up}(\texttt{WHAT}:\texttt{top\_bun})\wedge \mathcal{F}(\texttt{place}(\texttt{WHAT}:\texttt{top\_bun},\texttt{WHERE}:\texttt{tomato})))))))\end{aligned} \tag{6.3}$$

Our internal dictionary maps every detected object to a marker ID and the task specification ϕ

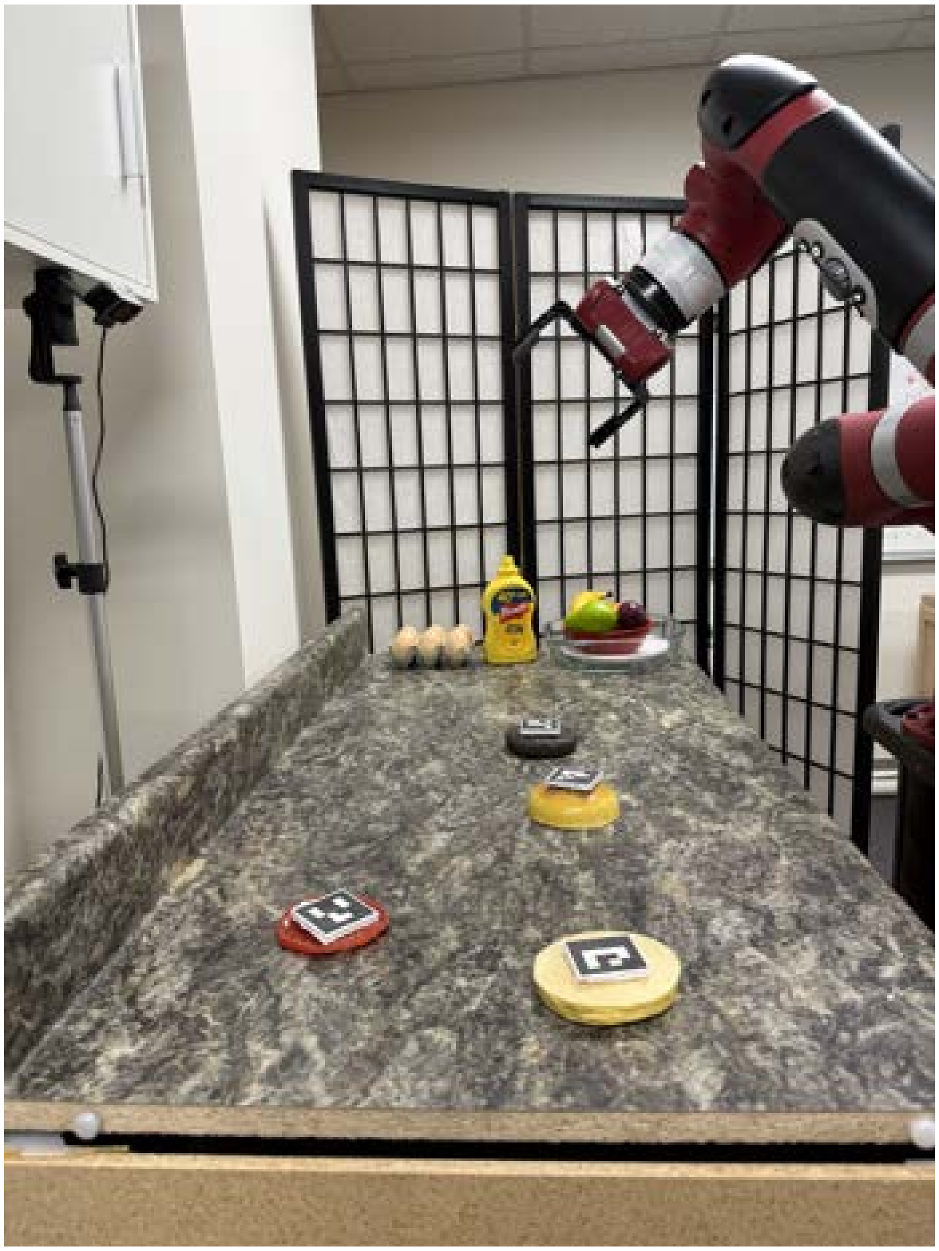
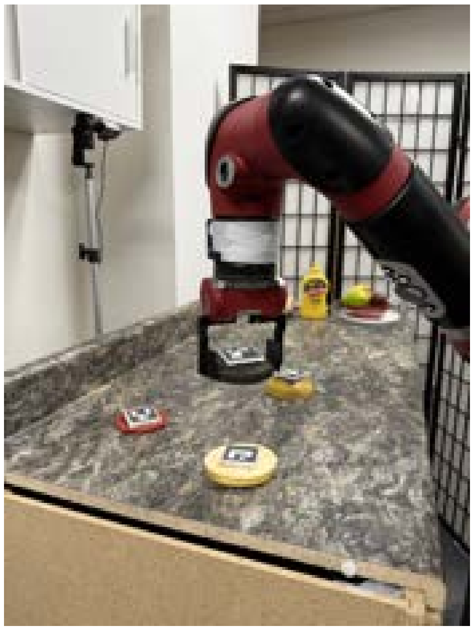


Figure 6.7: The initial state of the scene featuring the Sawyer robot-HD camera hand-eye calibrated system and the ingredients, randomly placed on the kitchen counter.

is converted to a sequence of calls to the `pick` and `place` functions in our python API. Fig. 6.7 shows a side view of the initial state of the scene, and Figures 6.8, 6.9, 6.10 illustrate the step-by-step implementation of the recipe by the Sawyer robot using the pick-and-place API that we developed.

(a) Pre-grasp pose

(b) Post-grasp pose


(c) Pre-place pose

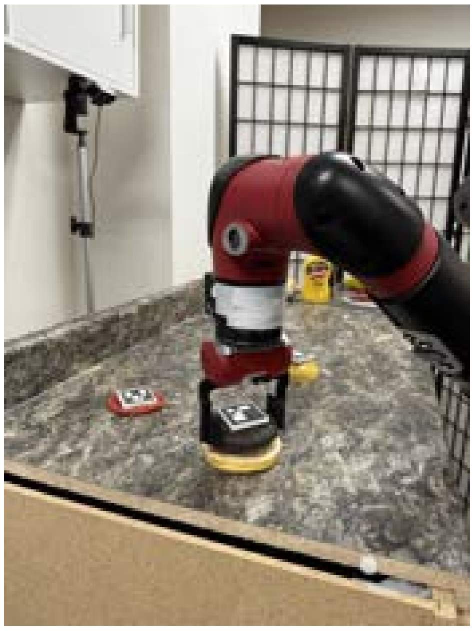
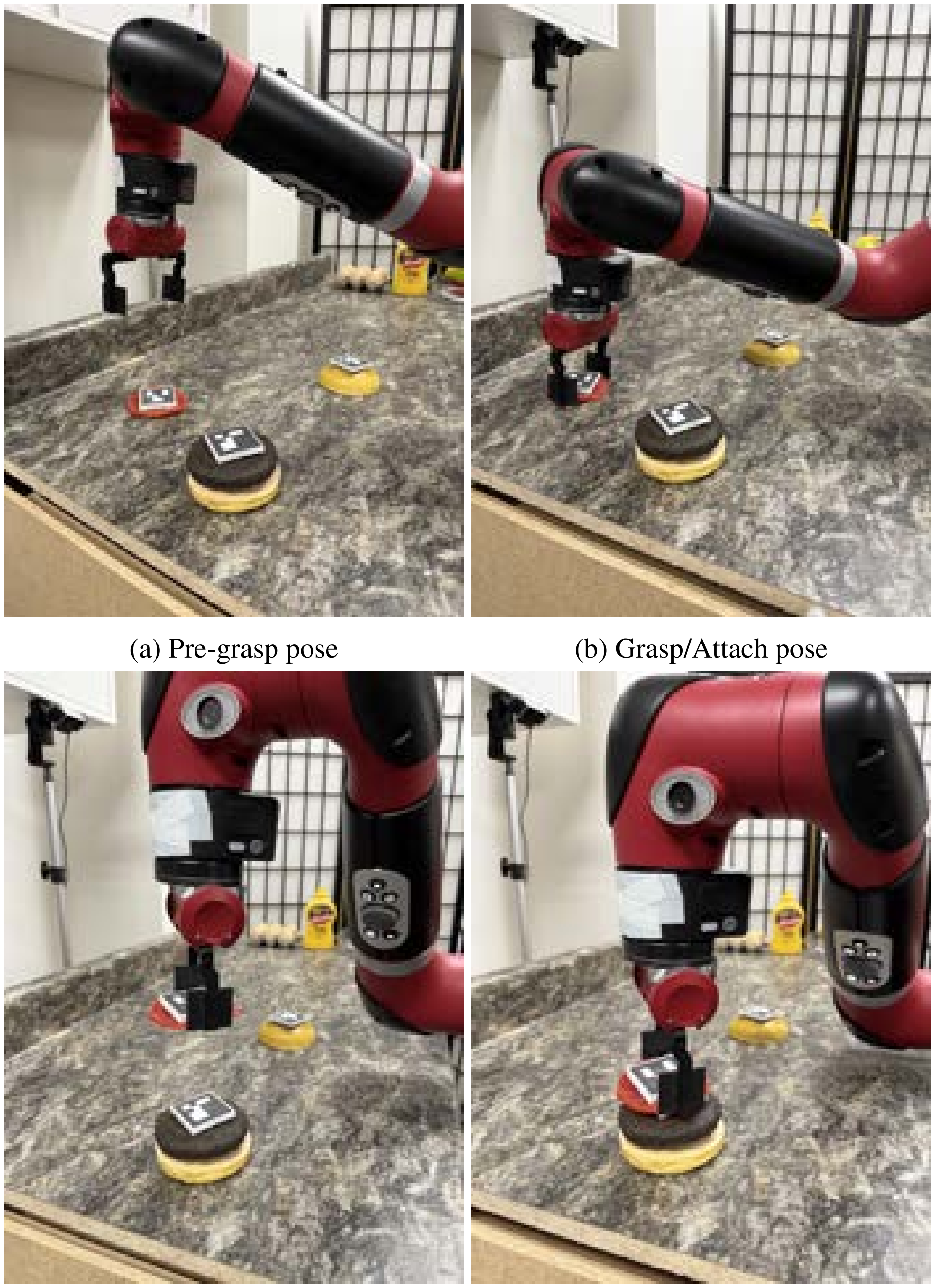

(d) Place/Detach pose

Figure 6.8: Picking the `beef_patty` and placing it on top of the `bottom_bun`.



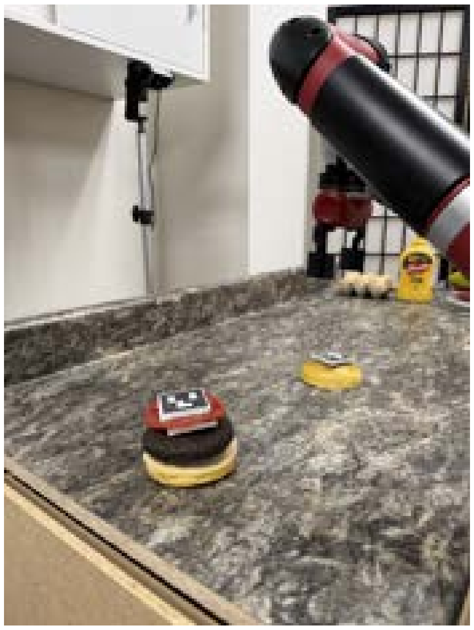
(a) Pre-grasp pose

(b) Grasp/Attach pose

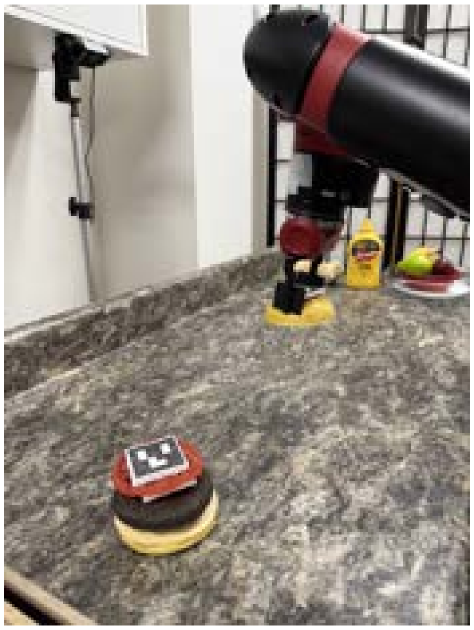
(c) Pre-place pose

(d) Place/Detach pose

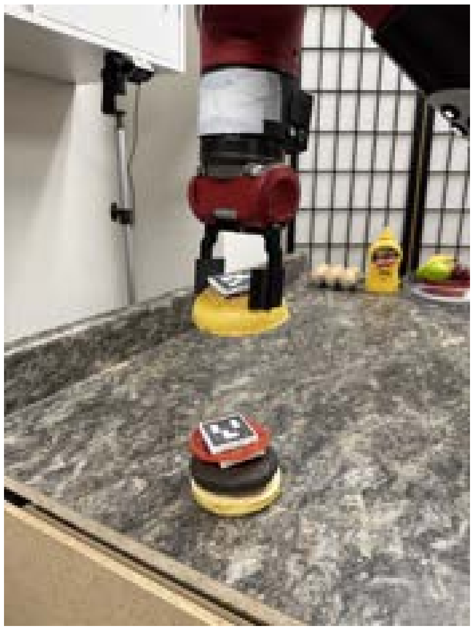
Figure 6.9: Picking the `tomato` and placing it on top of the `beef_patty`.



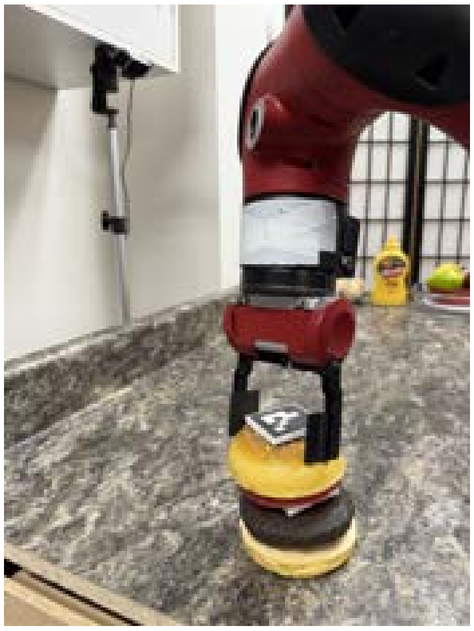
(a) Pre-grasp pose



(b) Post-grasp pose



(c) Pre-place pose



(d) Place/Detach pose

Figure 6.10: Picking the `top_bun` and placing it on top of the `tomato`.

Chapter 7: Conclusion and Future Work

Our work has focused on safeguarding LLM output through formal logic (Cook2LTL, Chapter 3) and classical AI planning formalisms (NL2PDDL2Prog, Chapter 5), as well as grounding physical properties of objects through active perception behaviors (Perception-Action API, Chapter 4). In Chapter 6, we presented the design and implementation of a real-world robotic system built with a Sawyer robot communicating through ROS. This system receives a natural language instruction, generates a plan by incorporating one of our previous systems (Cook2LTL, NL2PDDL2Prog), and implements the low-level motion plans through a python Pick-and-Place API that we built using the MoveIt ROS interface. In this section, we summarize our key findings.

In Chapter 3 we showed that caching LLM-derived action plans resulting from few-shot prompting significantly decreases the latency and cost associated with calling proprietary LLM APIs. We achieved that by dynamically building an action library that maps high-level tasks and actions to a set of actions that are admissible in our environment. The plans are parsed in the form of LTL task specifications and can generate discrete robot controllers that are provably correct [59].

In Chapter 5 we transitioned to PDDL as an action representation, allowing us to explicitly model action preconditions and effects. Our literature review highlighted that one of the key limitations of current methods combining LLMs and symbolic planning lies in the manual definition of the PDDL domain, assuming pre-existing domain knowledge that often does not exist, especially for non-experts. We demonstrated that having access to pairs of visual observations before and after executing every admissible action in our environment can assist a VLM to infer the action semantics, which are then passed to an LLM to automatically generate the entire PDDL domain file. By incorporating PDDL as an action representation, we inherit the additional benefit of visually grounding successful action execution by binding PDDL predicates (action preconditions and effects) to perceptual functions in the context of a python executable perception-action program. This is a particularly valuable insight on whether an action was successfully executed, as it is often erroneously assumed that the satisfaction of action preconditions implies successful action execution, neglecting potential issues (e.g. dynamic obstacles, robot malfunctions, perception errors) that may hinder plan execution at runtime.

One of our key insights in this thesis is modeling actions as computer programs (Section 1.1). Combining their well-established structure, flexibility, and expressiveness, and leveraging the code synthesis aptitude of LLMs, we can produce powerful perception-action programs that invoke active perception behaviors to ground object attributes at runtime (Fig. 4.1). We demonstrated using this paradigm both in simulation and in the real world on estimating the relative distance of an object to the robot and estimating the weight of an object. Our Perception-Action API can be updated with new functions given additional sensors or system functionalities, such as a function computing adaptive grasp policies described in the following section.

7.1 Future Work

7.1.1 Object Pose Estimation

So far we have assumed that every object in our workspace comes with an ArUco marker on top of it to facilitate tracking. To make our robotic system more realistic, we are incorporating Deep Object Pose Estimation (DOPE) [127], a deep learning method that provides us with 6-DoF object pose estimation on objects of known 3D models based on a single RGB image, without the need of markers. As we already have a set of objects from the YCB dataset [103] in our lab, we are currently experimenting with integrating DOPE into our robotic system to pick and place YCB objects with our python API. More specifically, we are building a `dope` node in ROS that subscribes to the plain `usb_cam` camera node, runs inference on our local GPU and publishes the 6-DoF pose of the detected object(s). To facilitate implementation, we approximate pose tracking of a grasped object by attaching it to a known link on the gripper, as described in Chapter 6. Additionally, we assume that the objects in the scene are static during the execution of the motion plans. Dynamic object tracking would require running DOPE or a state-of-the-art pose estimation approach like FoundationPose [128] in real time which is currently not supported by our local computer workspace.

7.1.2 Adaptive Grasp Policies

Our experiments on the robotic system use artificial food items as ingredients, which is a simplification that allows us to use the Sawyer non-current-controllable gripper that has a binary gripper control command (`gripper_open` or `gripper_close`). However, when deploying this system in a real kitchen, many of the real ingredients required for cooking recipes are delicate and deformable, which means that such grasp policies could damage the ingredients or lead to unsuccessful grasps. Although we could train a naïve policy that adapts the grasp based on a visual observation of an ingredient, we could be missing salient information encoded in different modalities that might be available to us. Consider the case of a user asking a robot to pick up the ripest avocado out of a set of avocados that are lined up on the kitchen counter. Our system needs to be able to tell the difference between a ripe and an unripe avocado, grounding the notion of ripeness to its perception, and second, compute a grasp policy with the appropriate motor current that does not damage the avocado (see Fig. 7.1 for a visualization of this idea).

DeliGrasp [129] is the most relevant work that aligns with our direction. DeliGrasp prompts an LLM with the description of an object and then infers certain object attributes (mass m, friction coefficient μ, and spring constant k) to parameterize an adaptive grasp controller with slip detection that returns the amount of current required for the desired grasp. It can also act as an estimator of an object's attribute since it actively computes the force feedback and the spring constant for the desired grasp and

Figure 7.1: Learning adaptive grasp policies based on visual observations, textual descriptions, and tactile feedback obtained by grasping different delicate and deformable objects in the lab.

can therefore derive a descriptive adjective for the grasped object based on these values. As an extension to this idea, we plan to associate the force feedback received during grasping objects of different texture or delicacy (e.g. a set of avocados of varying ripeness) with visual observations and textual descriptions (e.g. very ripe, hard etc.) of these objects. Our goal is to obtain tuples of (`force,image,description`) and train a model that predicts a grasp policy (amount of current required for compliant grasps) based on a natural language instruction and a visual observation as input. We can then incorporate this functionality in our Perception-Action API described in Chapter 4. Based on this modification, grasping the tomato in Fig. 4.1 would first call a function like `compute_adaptive_grasp`, that runs inference on our trained model to acquire an adaptive grasp policy. Consequently, the examples encoded in the docstrings of the API would also add a call to that function before every call to the `pick_up` function.

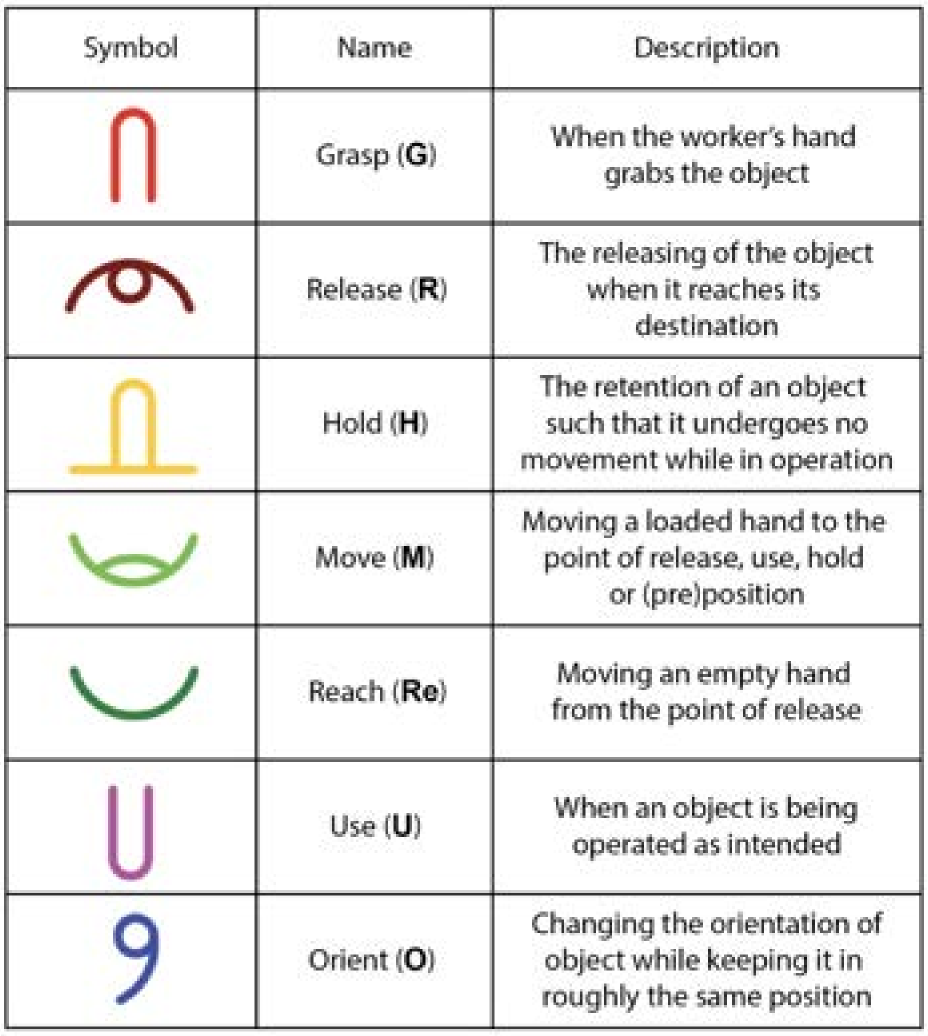
7.1.3 Object Affordances in Safeguarding Formalisms

The robotic system we implemented in Chapter 6 currently supports picking and placing, enabling the execution of simple manipulation tasks in the kitchen. To carry out more complex tasks and recipes, we need to adapt our action plans, safeguarding mechanisms, and controllers based on the geometry, properties, use cases, or more generally the affordances [105] of the ingredients and objects we manipulate. For example, opening a cabinet with a lever should reduce to a different low-level motion planning policy compared to opening a cabinet with a knob. The temporal sequence of such policies can still be expressed through the safeguarding formalisms we presented in Chapters 3 and 5 but requires a more fine-grained representation of lower-level motion primitives.

To this end, we plan to incorporate the Therbligs [4] as a lower-level grammar of action. Therbligs were developed in the context of workplace motion economy and establish a system that allows identifying and classifying the smallest indivisible actions involved in a task, such as reaching, orienting, moving, or grasping, as shown in Fig. 7.2. Transitioning to such an action representation characterized by a finer level of granularity allows us to model and generate affordance-aware policies. For example, a policy for opening a cabinet with a knob handle can consist of moving (`move(handle)`) towards the lever handle, reaching (`reach(handle)`) for it, grasping (`grasp(handle)`) it, rotating or orienting (`orient(handle)`) for an appropriate amount of degrees, moving (`move(forward)`) forward, and finally releasing (`release(handle)`). In LTL notation, a formula $\phi = F(\texttt{open}(\texttt{cabinet}))$ representing a simple sequential execution of these movements could be written as follows:

$$\begin{aligned}\phi =& F(\texttt{open}(\texttt{cabinet})) = \\ & F(\texttt{move}(\texttt{handle}) \wedge \mathcal{F}(\texttt{reach}(\texttt{handle})\wedge \\ & F(\texttt{grasp}(\texttt{handle}) \wedge \mathcal{F}(\texttt{orient}(\texttt{handle})\wedge \\ & F(\texttt{move}(\texttt{forward}) \wedge \mathcal{F}(\texttt{release}(\texttt{handle})))))))\end{aligned} \tag{7.1}$$

Our safeguarding formalisms can now operate over two layers, a higher-level layer including general action descriptions such as `pick` or `open`, and a Therblig-based representation that maps such actions to affordance-aware lower-level policies.



Symbol	Name	Description
	Grasp (**G**)	When the worker's hand grabs the object
	Release (**R**)	The releasing of the object when it reaches its destination
	Hold (**H**)	The retention of an object such that it undergoes no movement while in operation
	Move (**M**)	Moving a loaded hand to the point of release, use, hold or (pre)position
	Reach (**Re**)	Moving an empty hand from the point of release
	Use (**U**)	When an object is being operated as intended
	Orient (**O**)	Changing the orientation of object while keeping it in roughly the same position

Figure 7.2: Some of the Therbligs used in prior work [4], their symbolic illustrations as introduced by the Gilbreths, and brief descriptions of their usage.

Bibliography

[1] Javier Marin, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. Recipe1m+: A dataset for learning cross-modal embeddings for cooking recipes and food images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2019.

[2] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, et al. Ai2-THOR: An interactive 3d environment for visual AI. *arXiv preprint arXiv:1712.05474*, 2017.

[3] 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.

[4] Eadom Dessalene, Michael Maynord, Cornelia Fermüller, and Yiannis Aloimonos. Therbligs in action: Video understanding through motion primitives. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10618–10626, 2023.

[5] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[6] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094, 2023.

[7] Jason Xinyu Liu, Ziyi Yang, Ifrah Idrees, Sam Liang, Benjamin Schornstein, Stefanie Tellex, and Ankit Shah. Grounding complex natural language commands for temporal tasks in unseen environments. 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 1084–1110. PMLR, 06–09 Nov 2023. URL https://proceedings.mlr.press/v229/liu23d.html.

[8] BK Patle, Anish Pandey, DRK Parhi, AJDT Jagadeesh, et al. A review: On path planning strategies for navigation of mobile robot. *Defence Technology*, 15(4):582–606, 2019.

[9] Bernhard Hommel, Jochen Müsseler, Gisa Aschersleben, and Wolfgang Prinz. The theory of event coding (tec): A framework for perception and action planning. *Behavioral and brain sciences*, 24 (5):849–878, 2001.

[10] 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.

[11] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. *arXiv preprint arXiv:2304.02643*, 2023.

[12] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[13] Siyuan Huang, Zhengkai Jiang, Hao Dong, Yu Qiao, Peng Gao, and Hongsheng Li. Instruct2act: Mapping multi-modality instructions to robotic actions with large language model. *arXiv preprint arXiv:2305.11176*, 2023.

[14] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 11888–11898, 2023.

[15] Gayane Kazhoyan and Michael Beetz. Programming robotic agents with action descriptions. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 103–108, 2017. doi: 10.1109/IROS.2017.8202144.

[16] Angelos Mavrogiannis, Christoforos Mavrogiannis, and Yiannis Aloimonos. Cook2ltl: Translating cooking recipes to ltl formulae using large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 17679–17686, 2024. doi: 10.1109/ICRA57147.2024.10611086.

[17] GitHub. GitHub, 2024. URL `https://github.com`.

[18] Microsoft. PromptCraft-Robotics, 2023. URL `https://github.com/microsoft/PromptCraft-Robotics`.

[19] Noam Chomsky. *The minimalist program*. MIT press, 2014.

[20] POETICON Project. The "Poetics" of Everyday Life: Grounding Resources and Mechanisms for Artificial Agents. https://cordis.europa.eu/project/id/215843. Accessed: 2023-09-28.

[21] Yezhou Yang, Yi Li, Cornelia Fermuller, and Yiannis Aloimonos. Robot learning manipulation action plans by" watching" unconstrained videos from the world wide web. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

[22] Katerina Pastra and Yiannis Aloimonos. The minimalist grammar of action. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1585):103–117, 2012.

[23] Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. Robotic roommates making pancakes. In *2011 11th IEEE-RAS International Conference on Humanoid Robots*, pages 529–536. IEEE, 2011.

[24] Junjia Liu, Yiting Chen, Zhipeng Dong, Shixiong Wang, Sylvain Calinon, Miao Li, and Fei Chen. Robot cooking with stir-fry: Bimanual non-prehensile manipulation of semi-fluid objects. *IEEE Robotics and Automation Letters*, 7(2):5159–5166, 2022.

[25] Donghun Noh, Hyunwoo Nam, Kyle Gillespie, Yeting Liu, and Dennis Hong. Yori: Autonomous cooking system utilizing a modular robotic kitchen and a dual-arm proprioceptive manipulator. *arXiv preprint arXiv:2405.11094*, 2024.

[26] Moley Robotics. Moley kitchen. URL https://www.moley.com. Accessed: 2023-05-29.

[27] Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. Interpreting and executing recipes with a cooking robot. In *Experimental Robotics: The 13th International Symposium on Experimental Robotics*, pages 481–495. Springer, 2013.

[28] Haochen Shi, Huazhe Xu, Samuel Clarke, Yunzhu Li, and Jiajun Wu. Robocook: Long-horizon elasto-plastic object manipulation with diverse tools. In *7th Annual Conference on Robot Learning*, 2023. URL https://openreview.net/forum?id=69y5fzvaAT.

[29] Zipeng Fu, Tony Z. Zhao, and Chelsea Finn. Mobile aloha: Learning bimanual mobile manipulation with low-cost whole-body teleoperation. In *arXiv*, 2024.

[30] Jonathan Malmaud, Earl Wagner, Nancy Chang, and Kevin Murphy. Cooking with semantics. In *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pages 33–38, 2014.

[31] Huaxiaoyue Wang, Kushal Kedia, Juntao Ren, Rahma Abdullah, Atiksh Bhardwaj, Angela Chao, Kelly Y Chen, Nathaniel Chin, Prithwish Dan, Xinyi Fan, et al. Mosaic: A modular system for assistive and interactive cooking. *arXiv preprint arXiv:2402.18796*, 2024.

[32] Zhe Huang, John Pohovey, Ananya Yammanuru, and Katherine Driggs-Campbell. Lit: Large language model driven intention tracking for proactive human-robot collaboration–a robot sous-chef application. *arXiv preprint arXiv:2406.13787*, 2024.

[33] David Paulius, Yongqiang Huang, Roger Milton, William D Buchanan, Jeanine Sam, and Yu Sun. Functional object-oriented network for manipulation learning. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2655–2662. IEEE, 2016.

[34] Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of the Conference on Robot Learning (CoRL)*, volume 205, pages 287–318, 2023.

[35] 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.

[36] 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.

[37] Sai H Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. *IEEE Access*, 2024.

[38] Huaxiaoyue Wang, Gonzalo Gonzalez-Pumariega, Yash Sharma, and Sanjiban Choudhury. Demo2code: From summarizing demonstrations to synthesizing code via extended chain-of-thought, 2023.

[39] Xiuye Gu, Tsung-Yi Lin, Weicheng Kuo, and Yin Cui. Open-vocabulary object detection via vision and language knowledge distillation. *International Conference on Learning Representations*, 2022.

[40] Aishwarya Kamath, Mannat Singh, Yann LeCun, Gabriel Synnaeve, Ishan Misra, and Nicolas Carion. Mdetr-modulated detection for end-to-end multi-modal understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1780–1790, 2021.

[41] Maria A Bravo, Sudhanshu Mittal, Simon Ging, and Thomas Brox. Open-vocabulary attribute detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7041–7050, 2023.

[42] Harel Biggie, Ajay Narasimha Mopidevi, Dusty Woods, and Chris Heckman. Tell me where to go: A composable framework for context-aware embodied robot navigation. In *7th Annual Conference on Robot Learning*, 2023.

[43] Andy Zeng, Adrian Wong, Stefan Welker, Krzysztof Choromanski, Federico Tombari, Aveek Purohit, Michael Ryoo, Vikas Sindhwani, Johnny Lee, Vincent Vanhoucke, et al. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv preprint arXiv:2204.00598*, 2022.

[44] Angelos Mavrogiannis, Dehao Yuan, and Yiannis Aloimonos. Discovering object attributes by prompting large language models with perception-action apis, 2024. URL `https://arxiv.org/abs/2409.15505`.

[45] Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

[46] Jiayi Pan, Glen Chou, and Dmitry Berenson. Data-efficient learning of natural language to linear temporal logic translators for robot task specification. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11554–11561. IEEE, 2023.

[47] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pages 383–396. Springer, 2023.

[48] Sara Mohammadinejad, Jesse Thomason, and Jyotirmoy V Deshmukh. Interactive learning from natural language and demonstrations using signal temporal logic. *arXiv preprint arXiv:2207.00627*, 2022.

[49] Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. Nl2tl: Transforming natural languages to temporal logics using large language models. *arXiv preprint arXiv:2305.07766*, 2023.

[50] Ziyi Yang, Shreyas S Raman, Ankit Shah, and Stefanie Tellex. Plug in the safety chip: Enforcing constraints for llm-driven robot agents. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14435–14442. IEEE, 2024.

[51] Benedict Quartey, Eric Rosen, Stefanie Tellex, and George Konidaris. Verifiably following complex robot instructions with foundation models, 2024. URL https://arxiv.org/abs/2402.11498.

[52] 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, 2023.

[53] Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128*, 2023.

[54] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 20256–20264, 2024.

[55] Ishika Singh, David Traum, and Jesse Thomason. Twostep: Multi-agent task planning using classical planners and large language models. *arXiv preprint arXiv:2403.17246*, 2024.

[56] Zhehua Zhou, Jiayang Song, Kunpeng Yao, Zhan Shu, and Lei Ma. Isr-llm: Iterative self-refined large language model for long-horizon sequential task planning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2081–2088. IEEE, 2024.

[57] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. ieee, 1977.

[58] Georgios Fainekos, Hadas Kress-Gazit, and George Pappas. Temporal logic motion planning for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2020–2025, 2005.

[59] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 25(6):1370–1381, 2009.

[60] Stephen L Smith, Jana Tumova, Calin Belta, and Daniela Rus. Optimal path planning for surveillance with temporal-logic constraints. *The International Journal of Robotics Research*, 30(14):1695–1708, 2011.

[61] Nakul Gopalan, Dilip Arumugam, Lawson LS Wong, and Stefanie Tellex. Sequence-to-sequence language grounding of non-markovian task specifications. In *Robotics: Science and Systems*, volume 2018, 2018.

[62] Roma Patel, Ellie Pavlick, and Stefanie Tellex. Grounding language to non-markovian tasks with no supervision of task specifications. In *Robotics: Science and Systems*, volume 2020, 2020.

[63] Christopher Wang, Candace Ross, Yen-Ling Kuo, Boris Katz, and Andrei Barbu. Learning a natural-language to ltl executable semantic parser for grounded robotics. In *Conference on Robot Learning*, pages 1706–1718, 2021.

[64] Matthew Berg, Deniz Bayazit, Rebecca Mathew, Ariel Rotter-Aboyoun, Ellie Pavlick, and Stefanie Tellex. Grounding language to landmarks in arbitrary outdoor environments. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 208–215. IEEE, 2020.

[65] Jason Xinyu Liu, Ziyi Yang, Ifrah Idrees, Sam Liang, Benjamin Schornstein, Stefanie Tellex, and Ankit Shah. Lang2ltl: Translating natural language commands to temporal robot task specification. *arXiv preprint arXiv:2302.11649*, 2023.

[66] Yongchao Chen, Jacob Arkin, Charles Dawson, Yang Zhang, Nicholas Roy, and Chuchu Fan. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6695–6702. IEEE, 2024.

[67] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[68] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*, 2016.

[69] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[70] Eric Hsiung, Hiloni Mehta, Junchi Chu, Xinyu Liu, Roma Patel, Stefanie Tellex, and George Konidaris. Generalizing to new domains by mapping natural language to lifted ltl. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 3624–3630. IEEE, 2022.

[71] Yu-qian Jiang, Shi-qi Zhang, Piyush Khandelwal, and Peter Stone. Task planning in robotics: an empirical comparison of pddl-and asp-based systems. *Frontiers of Information Technology & Electronic Engineering*, 20:363–373, 2019.

[72] Drew M McDermott. The 1998 ai planning systems competition. *AI magazine*, 21(2):35–35, 2000.

[73] Wang Bill Zhu, Ishika Singh, Robin Jia, and Jesse Thomason. Language models can infer action semantics for symbolic planners from environment feedback. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume*

1: Long Papers), pages 8751–8773, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. URL https://aclanthology.org/2025.naacl-long.440/.

[74] Toki Migimatsu and Jeannette Bohg. Grounding predicates through actions. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 3498–3504. IEEE, 2022.

[75] Richard Howey, Derek Long, and Maria Fox. Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301. IEEE, 2004.

[76] Keisuke Shirai, Cristian C Beltran-Hernandez, Masashi Hamaya, Atsushi Hashimoto, Shohei Tanaka, Kento Kawaharazuka, Kazutoshi Tanaka, Yoshitaka Ushiku, and Shinsuke Mori. Vision-language interpreter for robot task planning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2051–2058. IEEE, 2024.

[77] Naoaki Kanazawa, Kento Kawaharazuka, Yoshiki Obinata, Kei Okada, and Masayuki Inaba. Real-world cooking robot system from recipes based on food state recognition using foundation models and pddl. *Advanced Robotics*, 38(18):1318–1334, 2024.

[78] Weiyu Liu, Neil Nie, Ruohan Zhang, Jiayuan Mao, and Jiajun Wu. Learning compositional behaviors from demonstration and language. In Pulkit Agrawal, Oliver Kroemer, and Wolfram Burgard, editors, *Proceedings of The 8th Conference on Robot Learning*, volume 270 of *Proceedings of Machine Learning Research*, pages 1992–2028. PMLR, 06–09 Nov 2025. URL https://proceedings.mlr.press/v270/liu25d.html.

[79] Vittorio Ferrari and Andrew Zisserman. Learning visual attributes. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[80] Ali Farhadi, Ian Endres, Derek Hoiem, and David Forsyth. Describing objects by their attributes. In *2009 IEEE conference on computer vision and pattern recognition*, pages 1778–1785. IEEE, 2009.

[81] Christoph H Lampert, Hannes Nickisch, and Stefan Harmeling. Learning to detect unseen object classes by between-class attribute transfer. In *2009 IEEE conference on computer vision and pattern recognition*, pages 951–958. IEEE, 2009.

[82] Olga Russakovsky and Li Fei-Fei. Attribute learning in large-scale datasets. In *Trends and Topics in Computer Vision: ECCV 2010 Workshops, Heraklion, Crete, Greece, September 10-11, 2010, Revised Selected Papers, Part I 11*, pages 1–14. Springer, 2012.

[83] Devi Parikh and Kristen Grauman. Relative attributes. In *2011 International Conference on Computer Vision*, pages 503–510. IEEE, 2011.

[84] Steven Chen and Kristen Grauman. Compare and contrast: Learning prominent visual differences. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1267–1276, 2018.

[85] Keyan Chen, Xiaolong Jiang, Yao Hu, Xu Tang, Yan Gao, Jianqi Chen, and Weidi Xie. Ovarnet: Towards open-vocabulary object attribute recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 23518–23527, 2023.

[86] Yi Wang, Jiafei Duan, Dieter Fox, and Siddhartha Srinivasa. NEWTON: Are large language models capable of physical reasoning? In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9743–9758, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.652. URL `https://aclanthology.org/2023.findings-emnlp.652`.

[87] Xiaohan Zhang, Saeid Amiri, Jivko Sinapov, Jesse Thomason, Peter Stone, and Shiqi Zhang. Multimodal embodied attribute learning by robots for object-centric action policies. *Autonomous Robots*, pages 1–24, 2023.

[88] Gyan Tatiya, Jonathan Francis, Ho-Hsiang Wu, Yonatan Bisk, and Jivko Sinapov. Mosaic: Learning unified multi-sensory object property representations for robot perception. *arXiv preprint arXiv:2309.08508*, 2023.

[89] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48, 2016.

[90] Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 804–813, 2017.

[91] Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE international conference on computer vision*, pages 2989–2998, 2017.

[92] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962, 2023.

[93] Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. Modular visual question answering via code generation. *arXiv preprint arXiv:2306.05392*, 2023.

[94] Aleksandar Stanić, Sergi Caelles, and Michael Tschannen. Towards truly zero-shot compositional visual reasoning with llms as programmers. *arXiv preprint arXiv:2401.01974*, 2024.

[95] Yiwei Jiang, Klim Zaporojets, Johannes Deleu, Thomas Demeester, and Chris Develder. Recipe instruction semantics corpus (RISeC): Resolving semantic structure and zero anaphora in recipes. In *Proceedings of the Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*, pages 821–826, 2020.

[96] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

[97] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[98] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

[99] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. brat: a web-based tool for NLP-assisted text annotation. In *Proceedings of the Demonstrations Session at EACL 2012*, Avignon, France, April 2012. Association for Computational Linguistics.

[100] Dim P Papadopoulos, Enrique Mora, Nadiia Chepurko, Kuan Wei Huang, Ferda Ofli, and Antonio Torralba. Learning program representations for food images and cooking recipes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16559–16569, 2022.

[101] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 47(10): 2208–2224, 2019.

[102] Roma Patel, Roma Pavlick, and Stefanie Tellex. Learning to ground language to temporal logical form. In *NAACL*, 2019.

[103] Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M. Dollar. The ycb object and model set: Towards common benchmarks for manipulation research. In *2015 International Conference on Advanced Robotics (ICAR)*, pages 510–517, 2015.

[104] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[105] James J Gibson. *The ecological approach to visual perception: classic edition*. Psychology press, 2014.

[106] John Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, pages 10–32, 1957.

[107] Khoi Pham, Kushal Kafle, Zhe Lin, Zhihong Ding, Scott Cohen, Quan Tran, and Abhinav Shrivastava. Learning to predict visual attributes in the wild. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 13018–13028, 2021.

[108] Xiaoyuan Guo, Kezhen Chen, Jinmeng Rao, Yawen Zhang, Baochen Sun, and Jie Yang. Lowa: Localize objects in the wild with attributes. In *R0-FoMo: Robustness of Few-shot and Zero-shot Learning in Large Foundation Models*, 2023.

[109] Lorenzo Bianchi, Fabio Carrara, Nicola Messina, Claudio Gennaro, and Fabrizio Falchi. The devil is in the fine-grained details: Evaluating open-vocabulary object detectors for fine-grained understanding. *arXiv preprint arXiv:2311.17518*, 2023.

[110] Xiangyu Zhao, Yicheng Chen, Shilin Xu, Xiangtai Li, Xinjiang Wang, Yining Li, and Haian Huang. An open and comprehensive pipeline for unified object grounding and detection. *arXiv preprint arXiv:2401.02361*, 2024.

[111] Liunian Harold Li, Pengchuan Zhang, Haotian Zhang, Jianwei Yang, Chunyuan Li, Yiwu Zhong, Lijuan Wang, Lu Yuan, Lei Zhang, Jenq-Neng Hwang, et al. Grounded language-image pre-training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10965–10975, 2022.

[112] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023.

[113] Iuliia Kotseruba, Calden Wloka, Amir Rasouli, and John K. Tsotsos. Do Saliency Models Detect Odd-One-Out Targets? New Datasets and Evaluations. In *British Machine Vision Conference (BMVC)*, 2019.

[114] Sahar Kazemzadeh, Vicente Ordonez, Mark Matten, and Tamara Berg. Referitgame: Referring to objects in photographs of natural scenes. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 787–798, 2014.

[115] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

[116] DJI. DJI RoboMaster EP. https://www.dji.com/robomaster-ep. Accessed: 2024-02-01.

[117] Carlo Tomasi and Takeo Kanade. Detection and tracking of point. *Int J Comput Vis*, 9(137-154): 3, 1991.

[118] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[119] Dima Damen, Hazel Doughty, Giovanni Maria Farinella, Sanja Fidler, Antonino Furnari, Evangelos Kazakos, Davide Moltisanti, Jonathan Munro, Toby Perrett, Will Price, et al. Scaling egocentric vision: The epic-kitchens dataset. In *Proceedings of the European conference on computer vision (ECCV)*, pages 720–736, 2018.

[120] Dan Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

[121] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[122] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410/.

[123] Daniel Nyga and Michael Beetz. Everything robots always wanted to know about housework (but were afraid to ask). In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 243–250. IEEE, 2012.

[124] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, and Manuel Jesús Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[125] Jacques Denavit and Richard S Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, 1955.

[126] Shinji Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 13(04):376–380, 1991.

[127] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. In *Conference on Robot Learning*, pages 306–316. PMLR, 2018.

[128] Bowen Wen, Wei Yang, Jan Kautz, and Stan Birchfield. Foundationpose: Unified 6d pose estimation and tracking of novel objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17868–17879, 2024.

[129] William Xie, Maria Valentini, Jensen Lavering, and Nikolaus Correll. Deligrasp: Inferring object properties with llms for adaptive grasp policies. In *Proceedings of The 8th Conference on Robot Learning*, pages 1290–1309. PMLR, 2024. URL `https://proceedings.mlr.press/v270/xie25a.html`.