UNIVERSITY OF CALIFORNIA SAN DIEGO

Improvisational Robot Tool Use using Affordance based Planning

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science and Engineering

by

Shrutheesh Raman Iyer

Committee in charge:

Professor Henrik Christensen, Chair
Professor Sicun Gao
Professor Michael Yip

2023

The Thesis of Shrutheesh Raman Iyer is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my amazing advisor, Professor Henrik I. Christensen, for offering guidance throughout my two years here and always having some time to catch up, with both research and life. I'd also like to thank Andrea Frank, my mentor, who introduced me to this field and has always been supportive and has encouraged me to pursue my research interests. I will certainly miss whiteboarding our way through all problems.

I would like to extend my gratitude to the rest of the members of my committee, Professor Sicun Gao and Professor Michael Yip for their insights and comments. I would also like to acknowledge all the members at the CogRob lab, who have all provided me with guidance and support during various parts of my masters, and with whom I've developed a great bond. A special thank you to Jiaming Hu, without whom this work would not have been what it is.

I'm grateful to my parents, and my brothers for their constant love and motivation throughout this entire journey.

VITA

2020	Bachelor of Engineering, RV College of Engineering, India
2021–2023	Teaching Assistant, University of California San Diego
2023	Master of Science, University of California San Diego

ABSTRACT OF THE THESIS

Improvisational Robot Tool Use using Affordance based Planning

by

Shrutheesh Raman Iyer

Master of Science in Computer Science and Engineering

University of California San Diego, 2023

Professor Henrik Christensen, Chair

Tool use and improvisational tool use is a hallmark of human (and animal) intelligence. When an appropriate tool for a task is not available, we can innovate and design a new tool, or use an existing object in a non-canonical way to accomplish the task, by reasoning about the underlying nature of the task. Despite the impressive capabilities of robots to learn narrow yet complex tasks using tools, innovative tool use by robots still remains an open and a significant challenge. We seek to address this challenge in the context of causal affordance based planning for the robot to reason about the constraints of the task, and thereby select the appropriate tool and its usage. We do this by providing semantic annotations to task preconditions in a Planning Domain Definition Language (PDDL) framework, derived from the constraints of the

task. Subsequently, we extend a standard planning algorithm to exploit these semantics, and demonstrate its application in improvisational tool use.

Chapter 1

Introduction

Improvisational tool use is a defining feature of human species. It is the ability to fashion everyday objects as tools, and employ them in unconventional ways. It also encompasses creating new tools in the absence of appropriate ones. Tool use and improvisation allows them to adapt to new situations easily and has generally become an essential survival trait. A remarkable instance of such skills were demonstrated by the astronauts in the failed Apollo 13 moon mission, where they came up with an adapted CO_2 filter to save power in the Command Module, by using objects such as cardboard, socks, towels, plastic bags, astronaut suits and duct tape [6]. In simpler scenarios, humans rely on improvisation to achieve their goals, such as using a coat hanger to retrieve small objects that are out of reach or substituting a rock for a hammer. Even animals show traits of intelligent tool use. For instance, Caledonian crows have demonstrated capabilities of mental representation to solve tool problems by reasoning about tool dimensions in terms of goals and sub-goals to retrieve objects [23].

Similar capabilities to allow for tool use have been proposed for robots as well. Recent works in tool use have demonstrated the capability to perform impressive tasks with tools such as forceful manipulation with hammers and screwdrivers [27], unfastening screws [29] and gripping knives [47]. Yet most of these have been developed for highly specialized tasks where the task and the mode of tool use are well-defined. While these systems can work in ideal conditions, they still lack a solution for the challenges posed when a robot encounters unfamiliar situations

that require improvisation. In large and unstructured environments where a general purpose robot has a large factory of actions and tasks, it is not possible to pre-define every possible tool use mechanism for tasks. The brittleness of only having pre-defined actions and capabilities can be fatal, especially when a robot is faced with an unforeseen situation.

Consider an example of a general purpose home service robot that is capable of navigation, manipulation, and also high level reasoning. Since it is general purpose, the robot knows the functional usage of tools such as an umbrella, hammer, screwdriver etc. that it has gained. One of the modules of the robot involves tidying the house and restoring items to their designated places. However, if the robot encounters a situation like the one depicted in Figure 1.1, where a pill bottle is wedged behind a couch, it becomes "stuck" since it lacks knowledge on how to retrieve the object. In such instances, the incorporation of improvisational abilities can help the robot, enabling the robot to use an object in its vicinity as a tool to access and dislodge the item. Sometimes, the lack of improvisation can be fatal. Consider an emergency fire situation, where the robot has to break the glass containing the fire extinguisher to use the extinguisher. However, if it cannot find a hammer to break the glass, it must be capable of reasoning that a rock can also break the glass, and be able to look for rocks or heavy objects to accomplish the goal. Thus, there is a need for robot tool improvisation, for robots to be deployed in the real world, since every possible scenario cannot be predetermined and modelled.

For a robot to improvise actions (and tools), it requires two important concepts:

1. It must be able to repurpose existing knowledge, utilizing its knowledge base of actions to come up with new actions or new usages.

2. It must not only know what the task is, but also needs to understand *how* and *why* it performs a certain task.

The first question of leveraging existing knowledge to come up with a plan has been studied in AI planning for decades, in the field of task planning. Given a factory of available

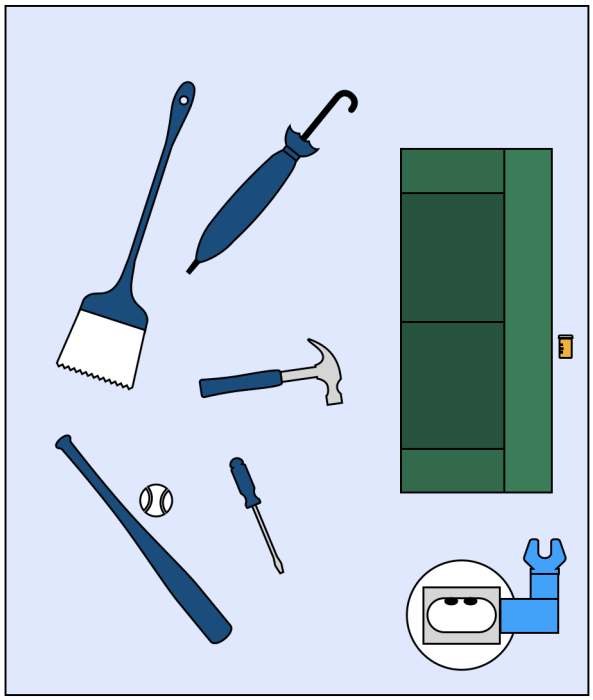


Figure 1.1. Problem Setup

discrete actions and states that the robot is capable of performing, generate a sequential plan of actions to achieve a goal state from the start state. Task planning has two components: the representation and plan computation.

1. **Representation** is concerned with describing an action, the necessary conditions for an action to be performed, and the effects of the action, along with the states of the environment, all described as sets of predicates. The most popular representation language used is the Planning Domain Definition Language (PDDL).

2. **Planning** is concerned with using the representation and start and goal states to compute a sequence of actions. It is modelled as a graph search problem, either across states or across plans.

The next step involves translating the high level symbolic action to low level motor controls for the robot to execute in the real world. This step is concerned with the field of task and motion planning (TAMP). However, this is not straightforward, since there is no guaranteed

one-to-one mapping between symbolic actions and motion controls in all states due to the constraints of the environment and the robot. Garrett et al. [19] provides a comprehensive survey on the different approaches towards TAMP problems.

The second question of understanding semantics of an action requires explicit modeling of the interaction between the agent (robot) and the environment. The semantics of the action must be able to describe relationships between properties and state changes from the knowledge base. For instance, it must be able to reason that shattering of glass is caused due to the fragility of glass, which is acted upon by impact of a heavy object. This enhances the generalization capabilities of the robot, since it can now infer that other objects that are fragile are also susceptible to shattering. This notion of building relationships between entities, actions, effects, and properties is closely linked with the field of affordances. Affordance is a concept from ecological psychology that contextualizes an agent's capabilities in terms of the *interaction* between properties of the agent and properties of the environment, and thus provide a natural framework to relate properties to potentials for actions.

However, the solutions to the two aforementioned questions are not directly compatible with each other, primarily due to the propositional nature of symbolic task planning. Task planning maintains a "black box" model of actions and predicates. By abstracting away the underlying mechanism, actions are represented as atomic units that cannot be inspected, limiting the complexity of planning. In this framework, a robot's behavior is limited to sequences of actions drawn from this finite set, and all achievable goals are limited to states described in those actions' effects. This means that the search space for planning is likewise limited, resulting in more efficient and performant task planning.

Unfortunately, this black-box like structure of task planning hampers semantic understanding of tasks, that is crucial for improvisation. To know how to use a tool, the planner requires access to underlying representations of state changes and effects in terms of real-world properties. This structure also makes TAMP problems harder to solve, since it assumes that all symbolic actions in a plan can be grounded and executed by the robot, as it has no knowledge

lower than the action, which is necessary for compliant manipulation [28].

Therefore, enhancing the representations of these task planning blocks is crucial for improvisation. To address this, we propose to enrich the predicate representation in symbolic planning in terms of affordances, to allow the planner to open the black box and perform introspective search. We use this parameterization to identify action relevance, and dependencies, which can help in improving TAMP problems.

To accomplish this, we identify four core questions that need to be investigated.

1. Is there a systematic method to describe manipulation actions for tool use to come up with preconditions for actions?

2. Can we encode semantics for predicates as affordances to provide more expressive capabilities for an AI planner to exploit?

3. How can this framework be used to perform improvisational tool use?

4. How to compute and execute these high level plans in dynamic settings?

Through this work, we seek to provide some insights on four questions. The rest of the text deals with answering the 4 questions. Chapter 2 provides a primer on task planning, and identifies the limitations of the existing system that motivated the work. Chapter 3 explores one approach to designing preconditions for actions. Chapter 4 is focused on answering question (2), by addressing the limitations discussed in Chapter 2. Chapter 5 discusses the application of the framework developed for improvisational tool use. Chapter 6 brings the ideas together and discusses how they can be used in a dynamic setting in deliberative fashion. Finally, chapter 7 discusses the system implementation details, experiments, and some results of the work.

1.1 Problem Setup and Scope

We implement our work on the Fetch [46] robot, which is a mobile manipulator with a 7DOF arm. To both respect the physical constraints of the robot, and ground the problem, we set

the scope of the problem in terms of the task and the type of tool use.

Tasks

We consider manipulation based tasks, in a table-top setting. Thus, the problem of retrieving a stuck object is equivalently converted to Figure 1.2. The goal is to retrieve the bean can object, that is both too far for the robot to reach, and is stuck behind in between the channel. The robot has to reason about both its length and width of its end effector to pick up the right tool to accomplish this. There are different objects present in the environment, placed to the side, that the robot can potentially use as tools. Some of these objects may also lie outside the reachable region of the robot.

In addition, purely kinematic constraints are considered for the robot. We do not consider forces and velocities for robot motion control, and hence perform path planning instead of motion planning. The robot has to reason only in terms of joint-angles kinematic configuration positions, and not other aspects of the motion planning problem. This is done to reduce the complexity of the problem.

Thus, within these constraints, we choose two problems, and implement the described algorithm for them. The first problem is the running example of extending the reach using a long object as a stick to either push or poke at an object to produce motion and the second problem is using a hook like tool to pull an object closer, while also reasoning about the dimensions of the object. These tasks are chosen, since tool use for pushing, pulling and poking have been studied in literature [2, 25, 17], to move an object around.

Tools

There are a large variety of tasks and tool use with distinct properties, and it is not easy to study all of them in one unifying framework. As with [34] and [42], we focus on tools that amplify the user's sensorimotor capabilities, since they are closely related with embodiment. Picking up a tool translates the problem from an agent-target interface to an agent-tool interface

Figure 1.2. Table-top setup

and a tool-target interface. We consider only Category-I tools that are tools which require the agent to be performing the same motion with or without the tool. In other words, the agent-target interaction in the absence of a tool is the same as the tool-target interface in the presence of a tool. Examples of Category-I tools are hammers and rakes. Tools such as levers and power drill are not considered since they are Category-II tools.

In specific, for this problem, we are interested in the shape and material of available objects to reason about tool use, as they have demonstrated the importance of reasoning about tasks [14, 13]. Despite this limited setting, we discuss how the framework can be potentially extended towards other categories of tools.

The properties of tools we are interested in are:

- Dimensions of tool (length and width).
- Shape of tool. (hook structure, flatness of edges).
- Material and rigidity of the tool.

Due to limited sensor capabilities, we assume that the material and rigidity of the tool are *provided* to us, and we demonstrate how we use them in the planning framework.

Chapter 2

Task Planning for Robotics

2.1 Task Planning

To enable robots to achieve goals and tasks, when the exact steps are not provided, it needs to be able to come up with a sequential plan of discrete actions to accomplish it. Task planning is the process of computing a sequence of high level discrete actions to transform a start state to a goal state. Task planning uses an internal model and reasons about the state of the world to achieve a goal.

In formal terms, let S denote the set of states, A denote the set of actions, γ denote state transitions, s_i the initial state, and s_g the goal state. A planning domain can be represented as $\Sigma = (S, A, \gamma)$ while a planning problem is represented by $P = (\Sigma, s_i, s_g)$. A plan, π is a solution to the planning problem, such that $s_g \in \gamma(s_i, \pi)$.

Research in Task Planning for Robotics has advanced significantly with research in automated planning. Also known as symbolic planning, it attempts to find the plan π, to reach the goal state s_g from the start state, s_i where each action a in the plan transforms the state. Since task planning seeks to reason about the state of the world to come up with discrete actions, it requires a way to represent the states and the actions, such that robots can take actions to transition between states. One of the earliest works in this domain was the STRIPS language for the Shakey Robot. The STRIPS language eventually gave way to PDDL, Planning Domain Definition Language. PDDL has been the community standard to represent and exchange

planning domain models since its introduction.

There are two components of PDDL:

1. State/task agnostic *domain* file, that contains information about actions and predicates.
2. Task specific *problem* file that contains the current state of the world and the required goal state.

PDDL uses *predicates* to represent boolean facts about an object or the environment, and *operators* to represent actions that can transition from one state to another. In the problem file, it uses *objects* to represent the different objects in the world and their types.

Predicates can be unary or binary predicates. They can also be negative predicates. An example of a predicate in this problem would be : $straightLineExists(?obj - phy\_obj, ?t - pose)$, where $?obj$ and $?t$ pose are the parameters that could be of types physical object and a Pose. This predicate checks if a straight line path exists between $?obj$ and the goal pose $?t$ and returns $True$ or $False$.

Operators contain two components:

1. Preconditions: A list of predicates that have to be true to be able to perform this action.
2. Effects: A list of predicates that will become true upon completion of the action.

For instance, for an action to pick up an object, $pickup(?o - phy\_obj ?r - robot)$.

The preconditions for this action are:

1. $robotHandEmpty(?r)$ that checks if the hand is empty
2. $isFree(?o)$ that checks if the object is free

The effects for this action are:

1. $isNotFree(?o)$. i.e., the object is no longer free.
2. $robotHandNotEmpty(?r)$, the robot hand is no longer empty.

Given the domain file and the problem file that contains the start state and goal predicates, the next step is to find a sequence of operators to satisfy the goal predicates, which is the planning component. Task planning essentially boils down to a graph search problem to find the path (or sequence of actions) from a start state node to one of the goal nodes. There have been several classes of planning algorithms developed over the years for efficient task planning. State space and plan space algorithms are the two popular approaches towards planning algorithms. State space algorithms search through the space of possible states, while plan space algorithms search through a space of possible partial plans. [21] provides a comprehensive description of different classes of algorithms used.

Since the goal of the work is not to design an AI planner, we use a simple task planner, the backward search planner (a state space planner), and discuss our work with respect to the backward search planner. The backward planner search starts with the goal state, and computes the inverse transitions (actions) that can help track back to the start state.

In our case, suppose the goal is to have an object o placed on a table t, then it spawns an action *place* to place the object on the table. But since $place()$ requires us to be holding the object and being near the table, o, the action of pickup the object is spawned to satisfy the object being held, and an action $moveToTable$ is spawned to be near the table. This way, until the start state is reached, the planner substitutes preconditions with actions in a backward fashion.

2.2 Task and Motion Planning

The task planner is a symbolic planner since it provides abstract high level actions given a description of states and actions. These actions further have to be refined using low level planners such as path planning and manipulation planning. For example, $grasp(obj)$ is an action that is composed of the following low level robot plans:

1. Open the gripper if not already opened.

2. Obtain the joint positions required for a particular grasp pose using inverse kinematics.

3. Find a collision free path in configuration space from current robot joint positions to the joint positions required to hold the object

4. Close the gripper

Translating the sequence of high level actions into a sequence of motion plans and executing them is however not straightforward and is a challenging problem. The task planner provides discrete high level actions. Robots and their actions reside in continuous space. For instance, to pick up and place the target object in a specified location, the robot has to reason about the different arm configurations to pick the object, the configurations to move the arm with the object and the configurations to place the object, and the choice of each configuration affects the possible configurations in the downstream tasks. Or in the case of using an object (as a tool) to push another object, the choice of the grasp of the tool informs the different ways of pushing the object. Task planning and motion planning thus reside in different spaces, each with a different purpose:

- **Task planning** is performed in discrete action and predicate space, to compute a long horizon sequence of actions.

- **Motion planning** is performed in continuous configuration space that can help the robot achieve a particular action.

This is a particularly challenging problem, since there is no guaranteed one-to-one mapping between symbolic plans and motion controls. The different configuration constraints imposed may lead to a sequence of actions (outputted by the task planner) infeasible. This requires Integrated Task and Motion planning (TAMP), which seeks to compute high level actions and ensure that they are feasibly executed by the robot.

There has been considerable research conducted in the field of Integrated Task and Motion planning. [19] provides a comprehensive survey of the different approaches towards TAMP. [16] introduced the concept of semantic attachments for planning, which offloads complex numerical

calls (to the motion planner for instance) as external modules that can return a boolean value, and demonstrated the use of the extension on standard PDDL planners (FF and TFD). [20] developed PDDLStream, a TAMP extension to PDDL that can sample continuous values from an external module and interfaces with PDDL via streams. These methods have demonstrated impressive performance and results. However, some of these methods require pre-computation of a few continuous variables, such as motion plans and Inverse Kinematics (IK) solutions. However, in the case of improvisational tool use where we do not know which tool and how it will be grasped, there is no way to pre-compute the motion planning primitives and optimize over them. In addition, these methods cannot detect dependent predicates and actions easily.

Most current implementations of high level AI planning treat actions and predicates as black boxes and do not describe how they are done, and encode effects as predicates. An action is considered to satisfy a predicate if the predicate is listed in its effects, and the planner performs pure pattern matching to link actions to predicates. However, high dimensional robotics actions are fairly complex, and it is not possible to list all possible predicates as effects. It would be better to encode the effects in a more compact yet descriptive format. In addition, treating actions as black boxes cannot capture dependencies between actions and predicates. There are a few reasons why capturing dependencies may be useful:

1. Choosing a particular action to satisfy a predicate p1, may invalidate another predicate p2, which may need to be satisfied. For instance, if an action has preconditions that the robot hand is both empty, and an object is not free, then grasping the object to make it not free invalidates the robot hand being empty, thus the object perhaps needs to be fastened or attached to another object in the environment, so that the robot hand is empty as well.

2. When a sequence of actions needs to be performed, the mechanism of some actions depends on other actions. For e.g., to sweep using the sweeper, the two actions required are: grab the sweeper, perform the sweep action. Traditional AI planners represent these actions independently, however the sweep action is parameterized by the choice of grasp

of the sweeper. [28]

Garagnani et al. [18] discuss similar and addition representation problems with PDDL for real-world rearrangement problems in more detail.

Chapter 3

Characterizing and Parameterizing Actions

This section attempts to answer the first question brought up; “is there a systematic way of finding preconditions for actions”. There are two key ingredients to this question:

1. Tool-agnostic action representation

2. Characterizing the phases of action, and identifying the physical constraints imposed.

3.1 Tool-Agnostic Action Encoding

Improvisational tool use is a defining feature of human and animal species, and understanding tool use is a fundamental open problem in cognition. There have been multiple hypothesis stated towards understanding tool use [45]. One of the key aspects of improvisational tool use is flexibility of usage. Humans and animals are able to improvise with tools by reasoning about the underlying physics of the task, which is agnostic of the tool, and using this consistent physical reasoning in flexible ways. By reasoning about the task in terms of contact points and forces, we can use tools in novel ways, to be able to satisfy the physical constraints of the task or action at hand [33].

One of the popular approaches towards understanding tool use is based on the embodiment theory. Studies have shown that humans and animals treat tools as attachments to the body to complete the goal [4], and that picking up a tool “induces the illusion of owning external

fake body parts". This is referred to as tool-as-embodiment [5]. According to this theory, grasping a tool is merely extending the kinematic chain and the grasped object now becomes the "end-effector" link. This approach has also been adopted in robotics work to enable robot tool use [42], [33]. To comply with this setting, we encode actions using an object-centric representation, as done in [31]. Actions are written in a tool-agnostic way, with an end-effector centered frame of reference.

3.2 Characterizing Actions

The process of determining preconditions for actions in high-dimensional tasks such as manipulation tasks can be challenging and, more importantly, subjective. One of the ways to derive these preconditions is by identifying the constraints that are inherent for performing an action, which can be done by studying the different phases of action in more detail. Following the terminology introduced by [27], all manipulation tasks considered in this scope consist of two stages: Engage (E) and Actuate (A). Although [27] represents in terms of the tool, we represent it in terms of the end effector, in line with the theory of "tool-as-embodiment". This eliminates the need to consider the **grasp** and the **release** phases discussed in their work. The engage stage specifies how the end effector must make contact with the target object, while the actuate stage specifies the direction of motion.

Preconditions for tasks arise out of constraints imposed during the various stages of the task. The important variables of interests upon which constraints are imposed are the kinematic and joint configuration of the robot, and the poses of the target object in question. Since we are primarily concerned with kinematic feasibility of the tasks, the constraints are described only at a kinematic level. More formally,

- During the **engage phase**, the key parameter is the pose of the end effector with respect to the object, which needs to be feasible and collision free and feasible in the joint space. The engage point is parameterized by the contact normal and the contact point between

the agent-target. For instance, the engage points in $push(obj)$ are all points on the object, obj that are on the near side, with their contact normals.

- During the **action phase**, both the set of poses of the object along the direction of motion, and the set of poses along the direction of motion are the relevant parameters. To make this more explicit, deviating from [27], we split this phase into two components, the direction of the target object, and that of the end effector.
  - **Actuate**$_o$: The set of poses that the target object must be in during the motion. Depending on the task, these may have to be collision free or not. For example: when pushing an object, every point that the object is in along the motion needs to be collision free, however if a nail is being screwed into a board, it can (or needs to be) in collision with the board.
  - **Actuate**$_e$: This set of end-effector poses must be feasible by the end effector and collision free. While for a *push* or a *pull* motion, this set of poses is the same as that of Actuate$_o$, for a *poke*, there need not be a collision free path for the end effector along the direction of motion.

Thus, using an object-centric approach and parameterizing an action in terms of engage poses and actuate poses, the base actions (that require no tools) are encoded as though the end effector of the robot is the tool. This is equivalent to the view that the tool becomes the last link (or the end effector) in the kinematic chain. These different constraints are encoded as (boolean) predicates (ideally in increasing complexities of computation). Through the actions of "push" and "pull," we highlight the different predicates that arise as a result of breaking down an action into its phases.

Thus, the three classes of predicates are:

- Engage phase : For push and poke, the engage points are such that there is a feasible collision free configuration to touch the near side of the object. Thus, the predicate is

$canReachObj(?obj, ?r)$ while for pull, it is for the far side of the object. Since the far side can be reached only with a "hook" like shape, we simplify the problem and encode it as two predicates: $canReachObj(?obj, ?r)$ and $hasHook(eef(?r))$. This deduction can be obtained by simple physics based reasoning, which is beyond the scope of this work. Similarly, the need to reach the base of the screw for the screwing action yields the predicate : $hasSharpEnd(eef(?r))$. We also have a $hasFlatEnd(eef(?r))$ as a predicate, since it is easier to push and pull with an object that has a flat edge, rather than a pointed edge. Figure 3.1 shows an example of the engage constraint that highlights the engage point in terms of the contact point and contact normal, as well as a shape property

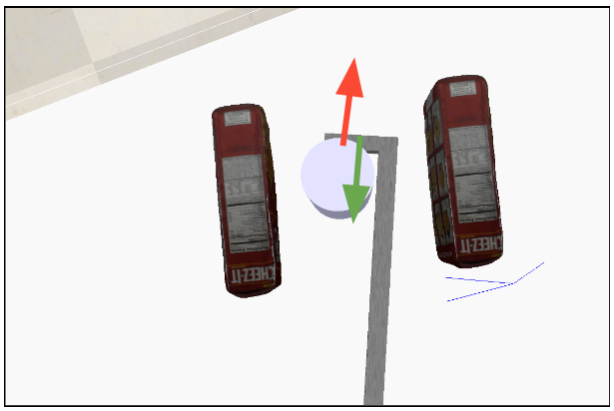


Figure 3.1. Constraints at engage point

- Actuate$_o$: For all three actions, a collision-free path must exist for the object along the direction of motion towards the goal pose p. For a linearly constrained motion such as the push, the predicate is : $straightlineExists(?obj, ?p)$. In addition, more knowledge specific information can also be encoded here for the object, such as $isSlidable(?obj)$ to check if the object can slide (for push and pull), and if the object is roll-able $isRollable(?obj)$ for poke.

- Actuate$_e$: Although for every point along motion, we must check a feasible collision free end effector configuration, we simplify the problem once again, and check if a feasible pose exists only for the final goal point. The predicate for this is: $canReachPose(?p, ?r)$.

While this may cause issues and give false positives (false satisfaction of predicates), we would like the predicate computations to be lightweight, and be admissible (i.e., should not eliminate any action falsely), and thus leave the heavy lifting to the motion planning component to reject infeasible actions at a later stage. (This could also be changed by selecting keypoints along the paths).

Thus, by characterizing actions through physical grounding, preconditions for actions are obtained, and encoded as boolean predicates. The predicates used for the tasks in our setting are :

1. **Engage Phase**
    - $robotHandEmpty(?r)$ for grasp
    - $canReachObj(?obj, ?r)$ for push, pull and grasp
    - $hasHook(eef(?r))$ for pull
    - $isObjectFree(?obj)$ for grasp
    - $hasFlatEnd(eef(?r))$ for push and pull
2. **Actuate Phase**
    - $straightLineExists(?p1, ?p2)$ for push, pull
    - $isSlidable(?obj)$ for push, pull
    - $isRollable(?obj)$ for poke
    - $canReachPose(?r, ?p)$ for all actions

There are two potential avenues to improve this step for the future.

- Instead of reasoning at a geometric level, predicates can be reasoned at a physical level, in terms of forces and impulses. For instance, in the case of hammering, the requirement is to impart a high impulse on the target object. This can either be done by increasing the mass

of the tool, or increasing the force at impact, or both. Using such a physics based reasoning provides a more consistent and compact grounding, which can improve the flexibility of actions.

- These predicates can be learned in an RL like setup. By splitting into 2 phases of engage and actuate, the system can observe and learn the constraints at each phase, in terms of the kinematics and dynamics.

Chapter 4

Planning with affordances

The black box structure of PDDL does not allow for detailed and flexible reasoning of *why* and *how* an action is performed. This highly propositional black box nature of PDDL-like planning also makes Integrated Task and Motion Planning a hard problem, due to the large gap between symbolic predicates and ground motion and path planning. Using it directly for real world problem is not straightforward, as one to one mapping between a symbolic plan and motion planning is not guaranteed to exist.

To address the insufficient representation of predicates and operators in PDDL, we propose an extension that incorporates semantics. Semantics should be able to model the interactions between the agent (the robot) and the environment. The semantics provide a connection between syntactic representation of identities of objects (i.e., the name "cup" for an object) to the real-world properties relevant for an action. (concave shape for holding/containment).

4.1 Background on Affordances

This notion of associating the properties of an object that can provide possibilities for action is strongly linked with the theory of affordances, which has its origins in psychology. The term affordance was first introduced by psychologist JJ Gibson [22] to describe the interactions between an agent and the environment. Affordances describe "opportunities for action", and can be thought of as possibilities provided by an object for action. For instance, the seat of a

chair affords sitting, and a mug affords holding and drinking. Affordances have been studied in great detail, primarily in the field of ecological psychology, with multiple approaches towards formalizing them. These works have led to different perspectives on affordances, spanning from perceptual approaches to cognitive and interactional approaches, with some of these views even contradictory to one another. According to Gibson, which has now been termed as the "Gibsonian approach", affordances are directly perceived by the agent, and that the agent does not build an internal model of the world, however some interactional approaches argue for the existence of an internal model that infers these affordances.

One of the more popular theories put forth was by Turvey et al.[44], who developed the concept of disposition. In this formalization, affordances are modelled as dispositional properties of the environment. Dispositions are properties that can offer hints on how to interact with the environment. More formally, "a disposition is a causal property that can be linked to a realization" [43]. For instance, a screw can be fastened onto another material because the helically threaded shape of a screw allows the screw being fastened. The disposition is the threaded shape of the screw, while fastening represents the realization. While the disposition exists even without it being triggered, triggering allows the disposition to be realized.

This connection between perception and action provides a natural framework for roboticists to develop cognition in robots to enable interaction with the environment. It can also provide a framework for automatic knowledge grounding, which has always been challenging for cognitive robotics. Affordances can determine what the robot must perceive and also inform what actions can be performed. Şahin. et al. [48] provided one of the earliest works in using the concept of affordances for autonomous systems. They argue that affordances are acquired relations, describing an agent-environment interaction. They model affordances as a tuple of $(effect, (entity, behavior))$ such that when a *behavior* is applied on an *entity*, an $effect$ is produced. Through this setup, they define equivalence classes, across different levels. Entity equivalences are classes of objects that produce the same effect on the same behavior; behavior equivalence are a class of behaviors that produce the same effect on the same entity, while

affordance equivalence are pairs of entity, behaviors that produce the same effect. Behavior equivalence and affordance equivalence are concepts that robots can exploit to achieve the same goal in multiple ways. They also introduce the concept of affordance from multiple perspectives, namely agent, environment and observer perspectives, where the agent perspective provides a natural means of encoding affordances for robotics.

This work has inspired multiple attempts at using affordances for robot planning and execution. Cruz et al. [12] extended this work by adding state to the relation, and argued that the current state of the environment and the agent are also crucial in describing an affordance. For instance, if a robot is holding a cup, then a die is not grasp-able, even though the die inherently affords grasping in other cases. Ardon et al. [1] provide a comprehensive survey on the different views on affordances towards robotic tasks. The works range from building a causal model for affordance recognition to directly learning affordances (such as [7]) implicitly for specific tasks.

However, most of the recent work on using affordances for robotics also treat affordances as another form of a label. Instead of identifying an object as a "knife", the system can now identify an object as something that can "cut". But the robot or the autonomous agent still does not have an understanding of what it means to cut, and it is just another label to the system. [32, 15]

4.2 Representing Predicates with Affordances

In this work, we borrow from the works of Turvey et al.[44] and Cruz et al. [12]. From Turvey's dispositional theory, affordances, or opportunities for action are discovered by the properties or dispositions of objects that can be used/altered. This view is similar to [30], in that affordances are modelled as "perceivable mappings of environmental features or characteristics to the abilities for interaction" (ibid.). Building on Cruz et al.'s[12] formalization, we view these dispositions to be parameterized by the current state as well. For instance, the heavy end of a hammer (the disposition) allows force amplification (realization), which depends on the weight

of the object being struck.

Contrary to Şahin et al.'s [48] and other similar works, instead of directly perceiving the mappings that associate objects to action, a state-varying dispositional approach can inform the type of mapping that is needed for interactions. The disposition provides hints for the type of state change that needs to occur for a predicate to be successful. For instance, the target object "pill_can" is *move-able* (or affords being moved) since the pose of the object can be changed, and it is *graspable* because the *boundedness* of the object can be changed (from false to true). The pose of an object is the disposition, that allows being modified (change of state) and moving the object. This naturally fits well with the task planning framework, where the dispositions can be thought of, as preconditions for a task, while the realizations can be equivalent to the effect. However, we take a slightly different approach, and propose to parameterize the predicates (both preconditions and postconditions) as explicit dispositions, such that the properties relevant to the predicate yield dispositions while the *consequence* of the predicate being satisfied is the realization.

For instance, for the predicate $canReachObject(?obj, ?r)$, the relevant properties are both the pose of the object, and the kinematic configuration of the robot, while the realization is that the object is now reachable by the robot.

This representation of semantics can be realized as:

1. Entities have properties that are observable by the agent. Some of these properties are dynamic, while others are static. The pose of an object is a dynamic property, while its mass is static. Dynamic properties can be altered through actions.

2. Predicates are described as properties of entities it depends on. E.g., : $objectAtPose(?x, ?t1)$ now has an entry: which is a tuple of $(?x, pose)$.

3. Similarly, the effects of an operator can also be encoded based on the property of the entity being changed. For instance, $push(?o, ?t1, ?r)$ has an effect of $(?o, pose)$ indicating that the pose of the object changes. This eliminates the need to list out the effects as predicates.

4. In addition to merely listing out the property of the entity in question, more relevant information can be given to the tuple. This can be used to describe the predicate or the effect in more detail, by specifying the relationship between the property, entity, and the effect.

We introduce the RelationTuple, also known as the semantics: $R(e,d,r,p)$ is a 4 element tuple of

1. e : the entity in question
2. d : the relevant disposition or the property of the entity
3. r : the realization of a disposition
4. p : parameters for the relation

This four-element tuple captures the specific semantics of a predicate with respect to an entity's properties. Relations can range from simple, such as "equality" and "existence," to complex functions that provide a more detailed description of the tuple. A RelationTuple describes one specific semantics of a predicate in terms of one of the properties of entities in question. A predicate can have multiple RelationTuple for each relevant property. This signature can be read as:

> "The property or disposition d of the entity e is related to the parameters p by the relation r."
>
> A few examples of signatures of predicates are provided here.

- $objectAtPose(?o,?p)$ has the RelationTuple : $(?o,\text{“}pose\text{”},\text{“}equality\text{”},?p)$.
- $straightLineExists(?o,?p)$ has the RelationTuple: $(?o,\text{“}pose\text{”},\text{“}f_{sl}\text{”},?p)$ where $pose(?o) = f_{sl}(,?p)$ is a function that can check if the pose of an object is in a straight line from $?p$.
- $push(?o,?p,?r)$ has the effect of : $(?o,\text{“}pose\text{”},\text{“}equality\text{”},?p)$.

4.3 Planning with Semantically Encoded Predicates

This explicit description of predicates in terms of its disposition can be directly exploited while planning. In standard PDDL based planning, predicates and actions are black box entities, and the planner lacks knowledge of the semantics. The predicate in the effect of an action is matched for equality with the predicate in question, i.e., $push()$ needs to have $objAtPose()$ as its predicate. The planner performs a naive pattern matching between effects of actions and preconditions to link actions to predicates. This requires that all possible effects must be written as predicates, and that while planning, the operators have to be grounded explicitly at this stage. However, through the description of predicates in terms of its semantics, to satisfy a predicate, the planner now looks at the dispositions of the entity that can be altered, and finds actions that have an effect of bringing about a change in that property. This can also allow the action to be grounded at a later stage, allowing for multi-level planning. By making use of RelationTuples, the planner now has access to richer information about relationships between entities, the properties, and the actions they afford as a result. Two examples of the reasoning steps are provided

- Sticking with the running example, to satisfy the predicate $objectAtPose(\text{“}pill\_can\text{”}, \text{“}t1\text{”})$, the following line of reasoning is performed.
  - First, the predicate has the RelationTuple $(?o, \text{“}pose\text{”}, \text{“}equality\text{”}, ?p)$.
  - Since the pose is a dynamic property, it can be changed, and thus the entity affords being *moved*.
  - Second, the action that can bring about a change in the pose is $push(?o, ?p, ?r)$ since it has in its effect: $(?o, \text{“}pose\text{”}, \text{“}equality\text{”}, ?p)$.
- To satisfy the predicate $straightLineExists(?o, ?p1)$, the following line of reasoning is performed
  - The predicate has a RelationTuple : $(?o, \text{“}pose\text{”}, \text{“}f_{sl}\text{”}, ?p$

- Since the pose is a dynamic property, it affords being changed (i.e., moving)
- Actions that can bring about a change in pose are : $push(?o, p)$ and $pull(?o, p)$
- The consequence of the predicate is checked if satisfied, i.e., the new pose the object is moved to, p is now on a collision free straight line from $p2$ to satisfy the relation "f_{sl}", as shown in Figure 4.1, where a position, which is not on a collision free path from the goal position is pruned out.

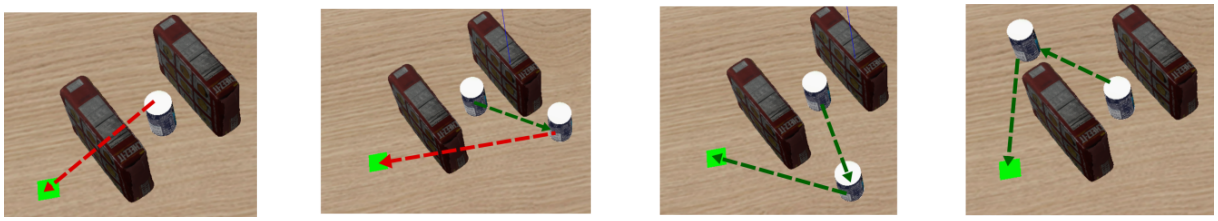


Figure 4.1. Planning to satisfy $straightLineExists()$ proposition.

This additional knowledge provided to the planner in the form of semantics allows the planner to reason about the effects in a contextually appropriate manner by leveraging affordances. The planner *discovers* affordances using the dispositions of the entities.

4.4 Benefits and contributions

Through this form of encoding, we demonstrate three contributions this can bring about. Most of the contributions are targeted towards accomplishing Task and Motion Planning, rather than specifically for tool use, but we show how these contributions for TAMP also contextualize tool use in later sections:

4.4.1 Inclusion of Property Subtypes

Some properties are subtypes of another property. For instance, the "hook"-like property is a subtype of "shape" property, which in itself is a subtype of the "configuration" property. With this framework, we can handle actions that can also change specific subtypes of properties

as well, and reason at multiple levels. For instance, to satisfy a predicate where a "hook" shape is needed, if an action changes the "shape" of an object, it is still a relevant action. We can prioritize actions in terms of how specific of a type it changes. This way, we can expand the types of interactions that can be captured by the planner.

4.4.2 Planning with functions

The relation entry in the semantics can aid in contextual planning. Although it seems trivial when the relation is just "equality", this is useful in situations where the exact relationship is either unknown or belongs to a set, and can be used for grounding the values. For instance, if the predicate $straightLineExists(?o, ?p)$ fails, then, by looking at its semantics, $(?o, \text{“}pose\text{”}, \text{“}f_{sl}\text{”}, \text{“}?p\text{”})$, the pose of the object can be altered to satisfy the predicate, and the *push* action is a relevant action. But the object has to be moved to a location $t2$, such that $t2$ belongs to the relation $?t2 = f_{sl}(?p)$. A *push* can be instantiated to any such $t2$ that belongs to the relation. This significantly reduces the search space when dealing with high dimensional continuous spaces.

Some relations can be hard or impossible to compute easily, such as the configuration of the end effect after it has grasped an object (due to multiple choices of grasp) and can be left as "unknown", which will be evaluated at a later stage with other predicates when computing the entire plan.

This brings down the search space significantly in the cases of large number of entities and/or continuous variables.

4.4.3 Identifying Dependencies

In addition to eliminating the need for specifying every effect in the predicate and making the problem formulation more readable, encoding predicates and effects in terms of its dispositions has the major advantage of being able to identify dependencies and the axis of dependency as well, both between predicates and between actions. As discussed earlier, it is not

possible to identify dependent preconditions of an action that may have to be satisfied together with a single action with the regular PDDL representation. Using the RelationTuple semantics, both inter-predicate and inter-operator dependencies can be identified in the following ways:

- all preconditions that share a common change in a specific disposition are all dependent on each other and have to be satisfied together. For instance, $straightLineExists(?o, ?t1)$ and $canReachObj(?o, ?t1)$ both share $(pose, ?o)$ in their RelationTuple and any action that alters the property of pose has to satisfy both the predicates together.

- If one action $a1$ has an "unknown" effect on an entity, $?e$ or an effect that belongs to a set instead of equality, and is followed by another action $a2$ such that it depends on the entity e (i.e., e belongs to one of its RelationTuple), then the two actions are dependent and may have to be computed together. For e.g., a *push* followed by another *push* are independent actions, while a *grasp* followed by a *push* are dependent, i.e., the choice of grasp determines how to perform the push. How this dependency is handled will be discussed in later sections.

4.4.4 Action Understanding through Affordance Representation

The proposed approach allows for robots to understand the actions they perform. By reasoning in terms of dispositions, the robot can understand how an affordance can be realized given the current (and future) states of the environment. This representation allows for the robot to reason in terms of properties, that are shared between different entities (instead of the identities), which is more generalizable and leads to more informed and sound decision-making. The self-understanding of actions for the robot can also enable the robot to both monitor and self-introspect its behaviors.

In terms of the affordance perspective, when all the preconditions are satisfied, the object affordances are now object-agent affordances, i.e., the object is not just *push-able*, but it is *push-able* by robot, r.

4.5 Summary

Thus, using the concept of affordances, we show how our parameterization of predicates can help in real-world planning and overcome some of the identified limitations of PDDL.

Chapter 5

Improvisational Tool Use

Given the framework for backward planning by parameterizing predicates in terms of semantics, and being able to discover affordances, we can now apply it to improvisational tool use.

In the disposition based planning framework, some of the predicates depend on the end effector configuration of the robot, such as $hasHook(?r)$, and $canReachObject(?obj, ?r)$. The end effector configuration is an inherent property of the robot. The only action that can alter the property of the robot is for the robot to augment itself with an object. When an object is picked up, it effectively becomes the new end effector, inheriting all the static properties of the object, such as shape and dimensions. This is also in line with the tool-as-embodiment theory. In the context of a robot arm, the end effector is added as a link to the kinematic chain of the robot. The relative pose of this link with respect to its parent is given by the choice of the grasp.

We introduce the special "Augment" operator for realizing tool use within the improvisational framework:

The semantics of the Augment function are given by :

- $R(?r, \text{“}eef_{config}\text{”}, \text{“}aff_{inh}\text{”}, ?t)$ which indicates that the end effector configuration is changed such that the last link now *becomes* the tool $?t$, where all the properties and affordances of the tool are inherited by the robot.

- $R(?t, \text{“}boundedness\text{”}, \text{“}equality\text{”}, True)$ which indicates that the tool is now bound to the

robot.

Using the planning method described in 4.3, when one of the precondition predicates that depends on $EEFConfig(?r)$ fails, then the corresponding action, Augment to satisfy all the sibling predicates (including the failed predicate) that depend on the end effector configuration is spawned. For instance, the preconditions for the $pull(?obj-object, ?p-pose, ?r-robot)$ action, as described by Chapter 3 are given in section . If the object is not in the collision free reachable region, i.e., if $canReachObject(?obj, ?r)$ fails, since it depends on the end effector configuration (from its semantics tuple), the augment action can be initiated. Since this action changes the end effector configuration, all the sibling preconditions that depend on end effector configuration also have to be satisfied by the Augment function, i.e., both $hasHook(?r)$ and $canReachPose(?p, ?r)$ also have to be satisfied by the same Augment function, as shown in Figure 5.1

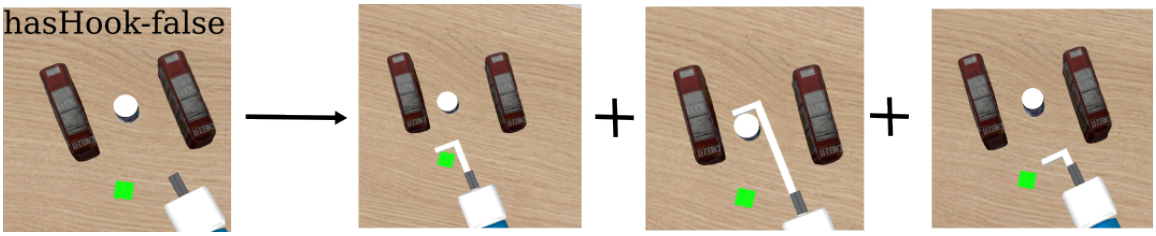


Figure 5.1. Augment operator needs to satisfy dependent predicates. If hasHook fails, the accompanying Augment operator must also satisfy *canReach* object and goal poses

The Augment function transforms the agent-target interface into an agent-tool and a tool-target interface. Since we are considering Category 1 tools here, the agent-target interaction is the same as the tool-target interaction. In other words, the engage and actuate phases (contact point and direction of motion) are the same whether or not the agent is holding a tool. The agent-tool interface is parameterized by the choice of grasp that is executed.

5.1 Details of Augment Operator

The augment operator is a complex operator that is composed of two components/phases:

1. Objects-agnostic Tool Prototype: A prototype of the ideal tool that satisfies all the relevant predicates (i.e., predicates that depend on the end effector configuration is computed. This does not depend on the available tools, but the ideal tool for the task is *imagined*, with the required dimensions, shape and other properties. In this step, the contact points of the tool are also identified, such that the predicates are satisfied.

2. Object dependent Evaluation and Grasp: For all objects available in the environment, this phase is performed. This has two components:

   (a) Compatibility Evaluation: The object is evaluated on how well it fits the prototype description, and is either accepted or rejected as a candidate tool. This could either be a series of boolean checks, such as the width, length, shape, rigidity. A more robust process is to obtain a similarity score with the prototype tool and then accept it based on a threshold. In this work, however, the simpler approach of boolean checks is done.

   (b) Object grasp: Grasping is the action of attaching the object to the robot's kinematic chain. Depending on the required contact points, and the tool dimensions, the grasp regions are identified appropriately.

In the task planning framework, for the augment operator, $Augment(?o-obj, ?r-robot)$, the RelationTuple or the semantics of the effect is encoded as: $(?r, \text{“}EEFConfig\text{”}, \text{“}affordance\_inheritance\text{”}, ?o)$ where the function indicates that all the static properties and affordances of the object, such as the dimensions, shape, and material properties, are inherited by the end effector of the robot. For instance, if the object has a hook-like shape, then after augment, the end effector will satisfy the predicate $hasHook(?eef_r) = True$. Another important point to note here is that the relation in the RelationTuple is not an equality relation, i.e., it is not know how the end effector will change exactly, since it depends on the choice of grasp, but it is known that the end effector changes. Thus, any subsequent action (pull) is

dependent on this action, according to the dependence identification discussed in the planning section.

5.1.1 Computing a Prototype Tool

In this step, an ideal tool that satisfies all the relevant predicates is imagined. This "imagination" is again a key concept in affordance planning that is done implicitly in most works, while we attempt to ground it based on physical and geometric constraints of the task. Alternatively, the step determines how to *deform* (by attachment) the end effector to achieve the poses. In more formal terms, the different properties of the tool described in 1.1 are identified for the task. This section describes the computation of these properties in detail:

Dimensions

The goal of this step is to obtain the dimensions of the prototype tool such that all the relevant predicates that require the end effector to reach a particular pose are satisfied. In the case of pull and push, $canReachObj(?obj, ?r)$ and $canReachPose(?p, ?r)$ need to be satisfied, i.e., there must exist a collision free joint configuration to the pose of the object, and the goal pose. Since the tool is considered as a link attachment to the robot kinematic chain (along the x-axis for simplicity), the objective of this step is to identify the minimum change required to change the dimensions of an imaginary object of zero length, and width and height equal to that of the end effector, to be able to achieve the poses, as given by [42]. In other words, the objective is to find a minimal augmentation to the end effector to achieve the poses.

Let $q = f_{IK}(X, l)$ be the collision free inverse kinematics (IK) function solver, $q = f'_{IK}(X, l)$ be the collision agnostic IK solver and $X = f_{FK}(q)$ be the forward kinematics function for the robot, where X is the target pose, q is the joint angles and l the kinematic chain of the robot.

Let $P = X$ be the set of all poses to reach. We first find the farthest pose for the robot to reach by trying to find a collision free Inverse Kinematics Solution for each pose, given by Q.

Then the forward kinematics is computed for these, given by : X. And the pose with the greatest $\|X' - X\|_2$ is selected as the farthest pose. The farthest pose denotes the hardest pose for the robot end effector to reach. The minimum augmentation function attempts to only reach this farthest point, (in the hope that it can also be able to achieve other *easier* poses).

From [42], mathematically, let X be the forward kinematics of the arm, given by:

$$X = f(q, l, \Delta)$$

where q denotes the feasible collision free joint angles, l denotes the dimensions of the links, and Δ denotes the minimum augmentation dimensions(Δl, Δw, Δh).

Let p be the goal pose required, then omitting the time and discomfort factors from [42], the objective function to minimize is:

$$J = \alpha \|f(q, l, \Delta) - p\|_2 + \beta \|\Delta\|$$

where α and β are constant weights for pose satisfaction and minimization of augmentation. The free variables are q and Δ.

The minimum augmentation is thus given by:

$$\Delta_{min}, q_{min} = \arg\min_{\Delta, q} J$$

This is however a highly complex and non-convex problem due to the nature of the parameter space (the joint configuration space in the presence of obstacles is non-convex). While [42] simplifies the problem and solves it by providing external information such as required width of channel and required length left to reach, it cannot work in generic environments without external information.

Thus, we try to solve the problem suboptimally, by relaxing the joint constraints and solving for the length separately from the width. (we assume that the height need not be altered,

but it can be easily integrated into this setup as well). We define the length to be the minimum additional link length needed to be able to have a feasible joint configuration without considering the obstacles (F'_{IK}), and the width to be the maximum width the end effector is allowed to have a collision free feasible joint configuration given the augment length. They're both converged using a simplified iterative optimization framework.

The augment length is computed iteratively by computing the *remaining* distance to a valid configuration. The algorithm is described in lines $12-18$ of the algorithm 28. To compute the width, the width of the end effector is constantly decreased with the updated length until a collision free configuration is detected. Using FCL collision checking [35], we can also get the depth of the collision, c_d between the augmented tool and the environment. The goal is thus to minimize this collision depth (to make it zero) and still be able to reach the goal pose.

Although this is a suboptimal estimate, it is a conservative estimate, and we believe that we would not eliminate any potential tools. The tip of the tool prototype is considered as the region for contact with the target object.

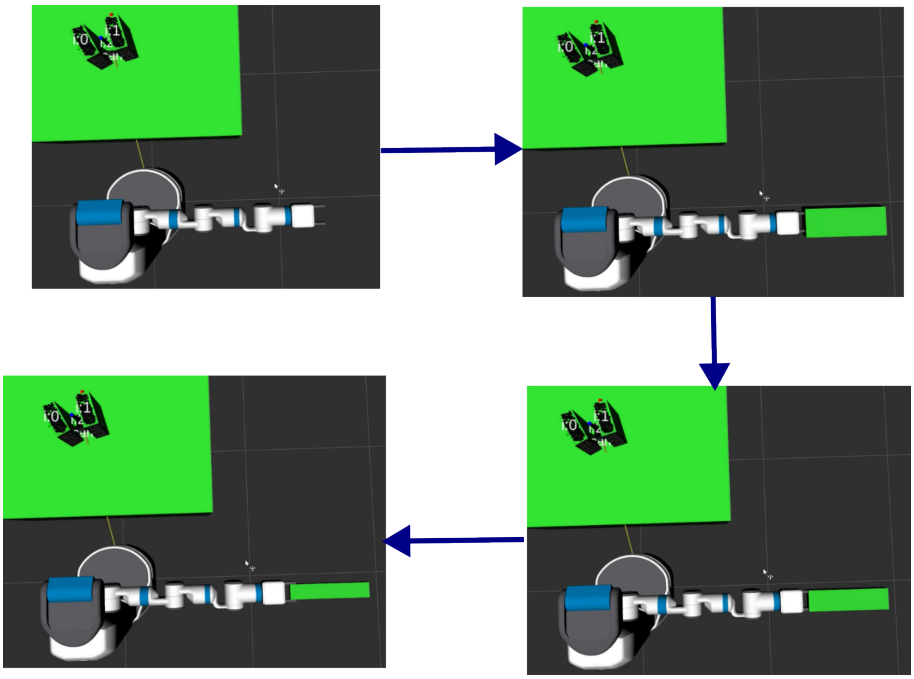


Figure 5.2. Evolution of the augment prototype action. First, an estimate of length is computed. Then iteratively the width is decreased until there is no collision

Algorithm 1: Dimension Augment Prototype

Data: List of poses to reach P

Result: Δ, d, the minimum augment dimensions and direction of augmentation

$\gamma \leftarrow 0.25$ $d_l \leftarrow 0$ $d_w \leftarrow 0$ $l_a \leftarrow 0$;

$w_a \leftarrow w$ (where w is width of eef);

for *Pose X in P* **do**

  $q = F_{IK}(X, l)$;

  $X' = F_{FK}(q)$;

  $err \leftarrow \|X' - X\|_2$;

  if $err > err_{max}$ **then**

    $err_{max} \leftarrow err$; $X_f \leftarrow X$;

  end

end

$err \leftarrow err_{max}$;

while $err > 0$ **do**

  $l \leftarrow l + l_a$ (Add length to link) ;

  $q = F'_{IK}(X_f, l)$;

  $X' = F_{FK}(q)$;

  $err \leftarrow \|X' - X\|_2$;

  $l_a \leftarrow l_a + \gamma * err$ $d_l \leftarrow +1$

end

$err \leftarrow err_{max}$;

while $err > 0$ **do**

  $l \leftarrow l - w_a$ (decrease link width) ;

  $q, c_d = F_{IK}(X_f, l)$;

  $X' = F_{FK}(q)$;

  $err \leftarrow \|X' - X\|_2 + c_d$;

  $w_a \leftarrow w_a - \gamma * err$;

  $d_w \leftarrow -1$

end

$\Delta \leftarrow (l_a, w_a, 0)$ $d \leftarrow (d_l, d_w, 0)$

Tool Shape

While a similar formulation such as the dimension computation can be obtained, the complexity of the problem increases and is beyond the scope of this work. Instead, the shape properties are taken as a boolean entity and passed over, i.e., the tool prototype is just annotated with the shape properties, such as : $hasHook()$ (in case of a pulling task) and $hasSharpEdge()$ (in case of a screwing task or a cutting task), or $hasFlatEdge()$ in case of pushing tasks.

The tool shape is associated with the tip of the tool prototype, since the tool-target contact happens at the tip of the tool, as described in the previous section.

Other Properties

The material and rigidity properties are also just annotated with the prototype tool, but they are not necessarily with the tip of the tool, but throughout the tool. This is highly task specific, and thus, they cannot be *imagined* easily.

Summary

The formulation of estimating the required minimum augmentation to achieve a set of poses is a complex problem with many open-ended challenges. While this work presents a very simple suboptimal solution, more sophisticated methods can be used to obtain these, that can also produce the required shape along with the dimensions. One potential exciting avenue can be the use of superquadrics to plan in narrow spaces, as performed by [40]. The augmentation can be treated as finding a path from the end effector to the goal pose by explicitly parameterizing the obstacle space, and thus analytically deriving the augment dimensions along with other shape features. The literature of deformable collision checking, can also provide some hints on finding good deformations for the end effector to reach goal poses.

5.1.2 Evaluation of Object against tool prototype

Once a tool prototype has been identified, an environment object, O has to be checked if it fits the description of the prototype. In this work, we assume that the material properties such as $isRigid(?obj)$ are known to us. Thus, we are primarily interested in the dimension and other shape related properties, by performing shape analysis on the tools. Shape analysis of objects have been studied in the field of topology, computer graphics and computer vision to model 3D objects for better understanding and representation.

Using shape primitives to represent shapes of objects have demonstrated impressive results for computer vision and geometry tasks [36]. Superquadrics are a parametric family of surfaces that can describe a variety of 3D convex shapes such as cubes, cuboid, spheres, and ellipsoids in a single continuous parameter space of 11 elements. Of the 11 elements, 6 are used for pose, 3 for dimensions and 2 for the shape or curvature of the object. 5.4 demonstrates the range of shapes that can be represented by the 2 parameters.

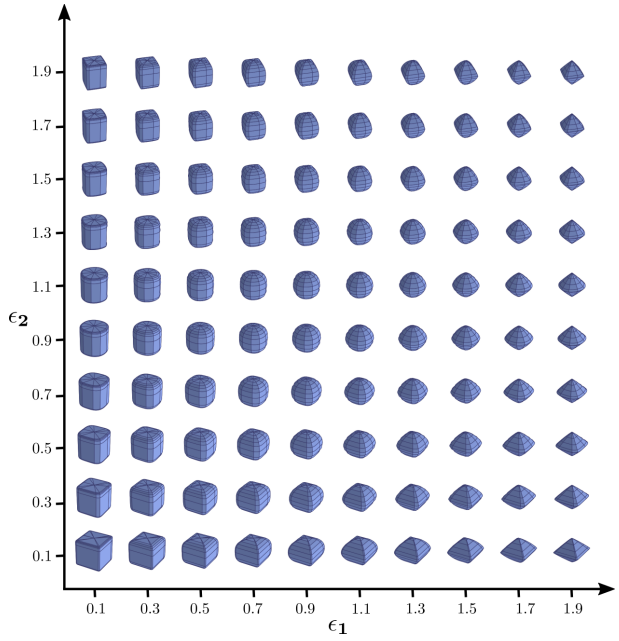


Figure 5.3. Superquadrics Shape Vocabulary, (from [36])

Superquadrics provide a unifying representation of various objects, and the 5 parameters

of dimension and shape obtained from the previous section can be used in a superquadrics setting.

First, the object of interest, O is decomposed into multiple convex shapes, using the popular DBScan algorithm [38]. This algorithm is provided both in the Open3D and Sklearn libraries. Then a superquadric is fit for each of the parts.

The dimension of the components is directly obtained from the superquadric parameters. The dimensions of the tool are just the concatenation of the dimensions of the components. Thus, given the prototype tool dimensions, and the augment directions (i.e., greater than, less than or not relevant), the tool is either accepted or rejected based on the dimensions.

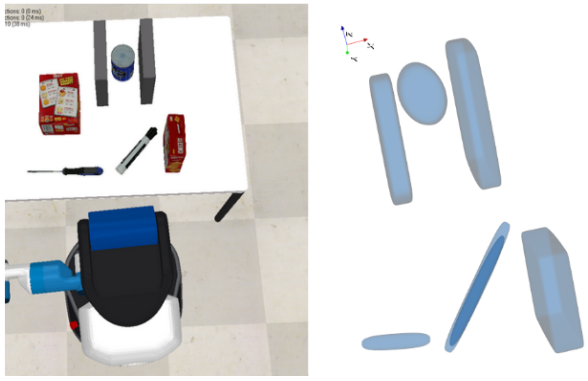


Figure 5.4. Superquadrics obtained from the given scene

Shape analysis can also be performed by using the superquadric parameters. This is highly property specific, and has to be specifically encoded for each property, but the superquadrics provide a good representation for encoding the specific shape properties. The relevant ones for this work are:

- $hasFlatEdge(?o)$: If there exists a surface such that either ε_1 or ε_2 is closer to 0.1, then that is a flat edge. The region of this region is also obtained.

- $hasHook(?o)$: This is for multipart superquadrics. If there exist two connected superquadrics that are aligned at 90 degrees to each other, then it is a hook. This region can also be identified.

- *hasSharpEdge*(?*o*): Similar to flat edges, sharp edges (for knife like objects) can be checked and obtained depending on the $epsilon_i$ value.

The contact regions on the tool are also identified, informed by shape properties.

5.1.3 Grasping

In this step, once an object has been identified as a potential tool, then it has to be grasped to complete the augment process. From the previous two steps, the contact regions and the dimensions of the tool have been identified. Grasp computation is performed in two steps:

1. Identify grasping regions on the tool such that the augment dimensions from the contact point are respected. i.e., along the direction of contact from the contact point, the tool must be grasped in a region such that the grasp region is augment-dimension distance away from the contact region. This is a simple geometric step of computing distances. For instance, for a pull, the *hook* region is identified when evaluating the tool, and the grasp region must be such that after grasping the object, there is at least a distance of Δl between the contact region and the grasp region, as shown in Figure 5.5

2. Computing a motion plan to grasp the object and hold it in the hand. This plan is however not executed in this step, since as previously discussed, the following action may depend on the choice of grasp, and they might have to be evaluated together. The task planner decides when to execute the plan, or if it needs to generate another grasp, which will be discussed in a later section.

5.2 Scope for Tool Construction

Although the primary focus of this work does not involve the creation of new tools, the proposed framework has the potential to facilitate the generation of novel tools. An action, *Attachment*, that can join two objects could be available to the robot. The *Attachment* action

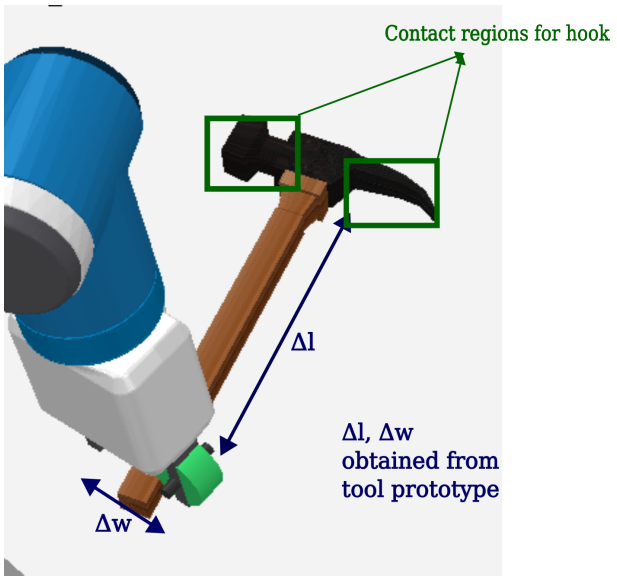


Figure 5.5. Grasping the tool

changes the "shape" of the end effector, with the RelationTuple being : $(?r, \text{“}shape\text{”}, \text{“}f_{shape}\text{”}, ?t)$. Consequently, any predicate that depends on the shape, such as $hasHook(?r)$ can now also be satisfied with attachment, where two objects can be put together in perpendicular directions to obtain a hook-like structure.

However, further exploration and detailed implementation of this concept on the robot have not yet been undertaken.

5.3 Summary

In this chapter, the special *Augment* function was introduced and discussed in detail to enable the robot to reason about the task in terms of the properties required to enrich its end-effector to be able to satisfy the predicates of the task that it is attempting to do. While category II tools have not been considered, we believe that this approach can also be extended for category II tools, where the tool target interface would be changed, in terms of the action direction and points.

Chapter 6

Task Execution using Behavior Trees

This section deals with the execution of the system, that includes both calling the planner and execution of the planned actions/motions.

One of the key components of affordance based planning for flexible tool use is the need for real time execution monitoring, since the predicates need to be dynamically evaluated. In addition, the nature of the backward chaining algorithm is similar to that of re-planning. In the core reasoning step of the algorithm, when a specific precondition is not satisfied, a new plan is generated to satisfy the precondition, and this process is repeated until all the preconditions can be satisfied from the start state. This reactive nature of the algorithm calls for a modular control architecture to orchestrate the system.

Behavior Trees(BTs) is a control architecture developed for controlling autonomous agents. BTs were initially developed by the gaming industry to control non-player-characters (NPC) in games. Recently, they've also found their way into robotics to control autonomous systems due to their modular and flexible nature, and have demonstrated their use in manipulation, task planning and multi robot system. [10]. A behavior tree consists of nodes where the leaf nodes are atomic actions or operations, also known as execution nodes, while the internal nodes are behavior compositions, or the control nodes. The atomic actions could both be sensing and actuation.

The different control nodes are Sequence, Fallback, Parallel, and Decorator nodes. The

sequence node is a node that succeeds if all of its children nodes succeed, a fallback node is one that succeeds if any of its children succeed. The parallel node executes all of its children together, while the decorator is a special node that can add some functionality to its children (such as repeat this five times). The two execution nodes are the action and condition nodes. More details regarding the purpose and implementation of these nodes can be found in [11].

Behavior trees have been used in robotics to monitor and guide executions of different behaviors of the robot in a coordinated manner. Figure 6.1 shows an example of a behavior tree for a fetch task. The robot first detects the object (if already not detected), then grasps the object and finally goes to the destination.

Figure 6.2 shows an example of a behavior tree for a service home robot that is performing a tidying task. It first identifies objects out of place, then picks it up, and then identifies the right placement locations for the object. It then navigates to the right room, looks for receptacles to place, and places it at the appropriate receptacle. Behavior trees are highly modular, and different behavior trees can be composed together to form a bigger tree. For instance, the fetch behavior from 6.1 can be placed as the "PickupObject" behavior in Figure 6.2. The Tidy behavior itself is one of the behaviors of the service robot, in addition to other behaviors that can be described similarly. This figure is provided primarily to highlight the color scheme that will be followed for the rest of the text.

- Orange represents a sequence node
- Gray represents both condition and sequence nodes
- Blue represents a fallback or a selector node
- White represents a decorator node to provide additional functionality, such as retrying the child node until it succeeds.

The structure and the different components of the behavior tree are highly compatible with the different components of the task planning. For instance, there is a one-to-one mapping

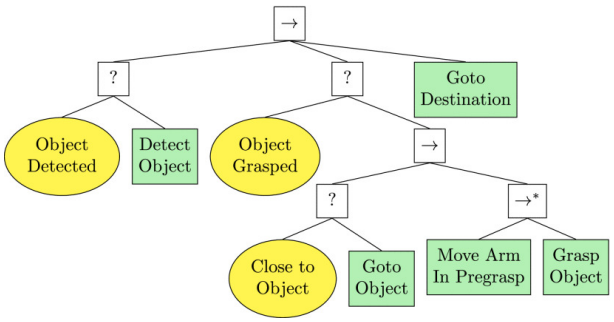


Figure 6.1. Example of a BT for a fetch task, (from [10])

between the following:

- predicates and condition node
- operators and action/execution node
- A plan as a sequence.
- choices of plans/sub-plans and fallback nodes.

The idea of backward chaining used through this text is similar to the concept of plan repair. Only when a predicate fails, a new action sequence is spawned to satisfy the failed predicate. This enables combining both planning and execution, allowing more reactive behaviors. Colledanchise et al. [9] showed how to merge planning and acting by creating an updating a behavior tree on the fly in the form of re-planning. Based on the idea of back chaining, it iteratively tries to achieve unmet preconditions/predicates starting from the goal condition, by inserting relevant actions. The system starts off with a trivial BT consisting of a single Sequence of all goal conditions. When any goal condition is not satisfied, small behavior trees in the form of Postcondition Precondition Action (PPA) [9] are created, where the actions are the execution nodes that can satisfy the failed condition. The PPA structure consists of a fallback node with the post-condition (that needs to be satisfied) and an action sequence, as shown in 6.3(b). The sequence node consists of the preconditions for the relevant action, followed by the

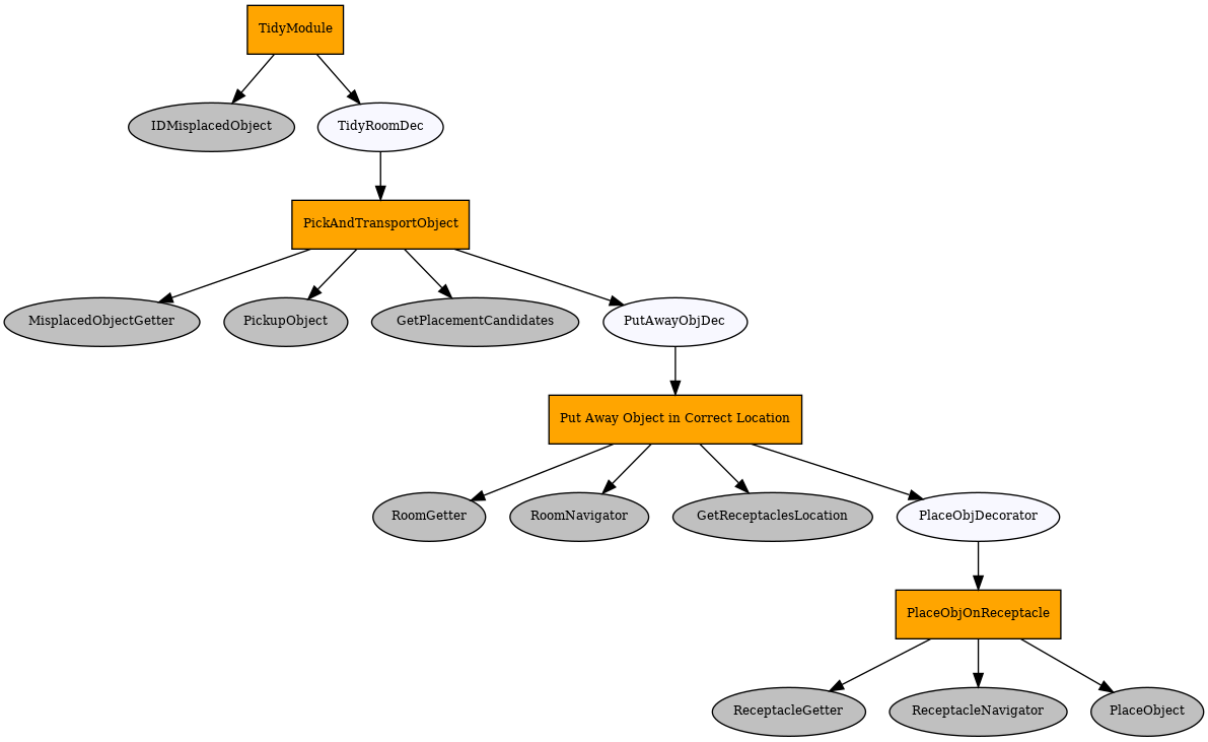


Figure 6.2. Behavior tree for a home-robot tidy module

action execution nodes. Recursively, whenever any of the preconditions for the actions fail, a new PPA is inserted to satisfy the failed predicate. The algorithm is explained in Figure 6.3.

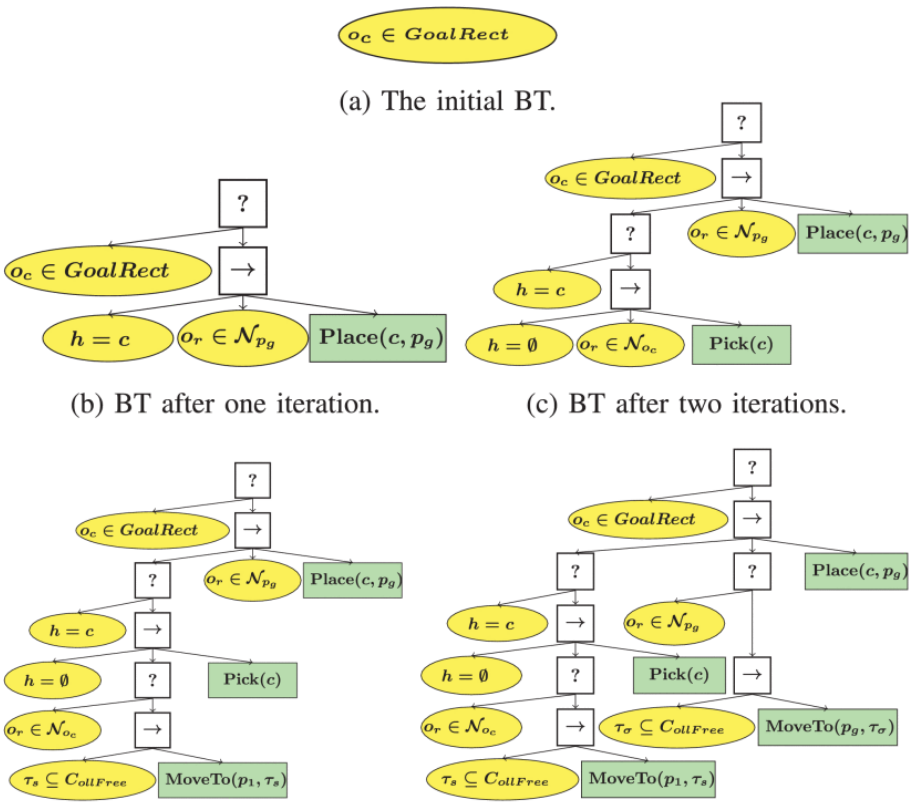
Since this algorithm is consistent with our backward chaining planner, and also allows the flexibility of dynamic planning, it has been adopted into our work, with small modifications. This original algorithm was developed to work in PDDL like settings, and thus suffers from some of the representational issues of PDDL based planning, which our planning representation and method seeks to overcome. It works on the assumption that failed preconditions can be solved independently by individual atomic actions, which may not be true in the case of dependent predicates. And it also works under the assumption that all actions are atomic and can be executed in isolation, which is also not guaranteed due to the lack of one-to-one mapping between actions and motion control (some of which is targeted in their follow-up work, PA-BT [11]).

Thus, in line with the modified backward search described in this work, when a pre-condition, P_c is failed, relevant actions are identified and the other sibling preconditions (that are

Algorithm 2:

Data: Goal Condition, C_{goal}

$T \leftarrow Sequence(\emptyset)$;
for c *in* C_{goal} **do**
  $T.add_child(c)$;
end
while *True* **do**
  $r \leftarrow SUCCESS$;
  while $r \neq FAILURE$ **do**
    $r \leftarrow Tick(\mathrm{T})$;
  end
  $c_f =$ GetConditionToExpand(T) ;
  $T \leftarrow ExpandTree(c_f, T)$;
end
Function `GetConditionToExpand(`T`):`
  for c *in* $T.get\_next\_condition()$ **do**
    if $c.status = FAILURE$ *and* $c \notin Expanded$ **then**
      $Expanded.insert(c)$;
      return c;
    end
  end
  return *None*;
Function `ExpandTree(`c_f, T`):`
  $A, param \leftarrow GetRelevantActions(c_f)$ (Get action and the altered property of c_f) ;
  $A \leftarrow RefineActions(A)$ (Grounds the action templates with values);
  $A_p \leftarrow GetParentActionForCond(c_f)$
  $T_{pred} \leftarrow Fallback(\emptyset)$
  for $a \in A$ **do**
    $c \leftarrow GetRelevantSiblingPredicates(c_f, a, param, T)$;
    $T_{post} \leftarrow Sequence(\emptyset)$;
    for c_s *in* c **do**
      $T_{post}.add\_child(c_s)$;
    end
    $T_{act} \leftarrow Sequence(\emptyset)$;
    for a_c *in* $a.precon$ **do**
      $T_{act}.add\_child(a_c)$;
    end
    $T_{act}.add\_child(act)$;
    if *dependent(*act, A_p*)* **then**
      $T_{act}.add\_child(A_p)$;
    end
    $T_{pred}.add\_child(T_{post})$;
    $T_{pred}.add\_child(T_{act})$;
  end
  $T \leftarrow Replace(T, c_f, T_{pred}$;
  return T;



(a) The initial BT.

(b) BT after one iteration.

(c) BT after two iterations.

(d) BT after three iterations.

(e) BT after four iterations. Final Tree

Figure 6.3. Deliberative backward chaining algorithm using BTs ([11])

siblings of the failed precondition node in the tree) are also identified based on the property that the action modifies. The algorithm is provided in Algorithm 2 A few important details relevant are :

- The PPA tree consists of a sequence of dependent postconditions instead of a single post condition, as shown in Algorithm 2. (lines 27 – 30). This is shown in the precondition sequence node in Figure 6.4.
- The postcondition in this step are symbolically linked to their upper level precondition counterparts, such that if this postcondition is satisfied, then the associated postcondition on the upper level is also satisfied.
- If there exists a dependency between the action, A, and the upper level action, A_p (i.e., A_p has the precondition P_c that failed to initialize the action A), then A_p is brought down to

the lower level and added to the sequence after action A, and this is again symbolically linked to A_p. (line 38). This can again be seen in Figure 6.4

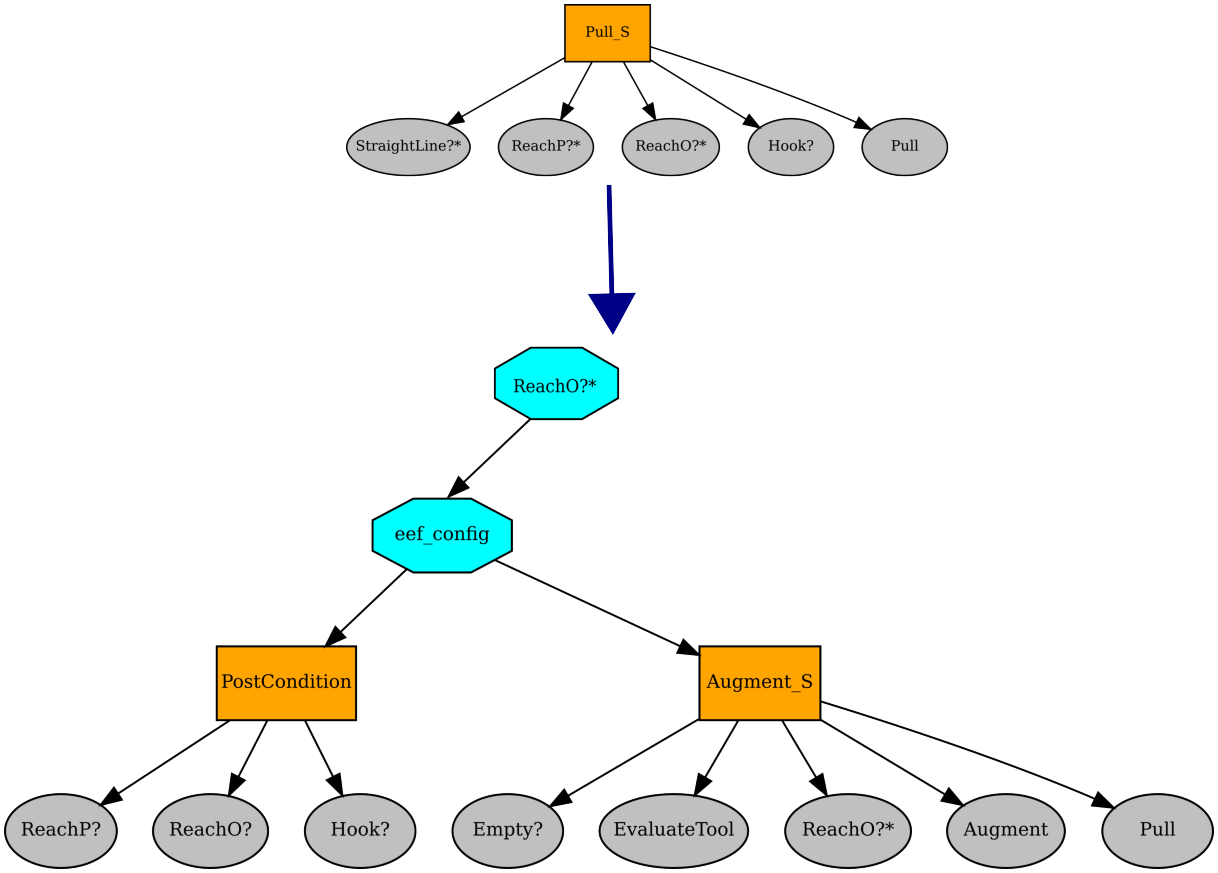
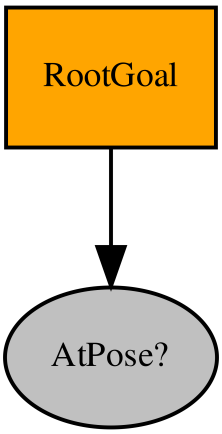


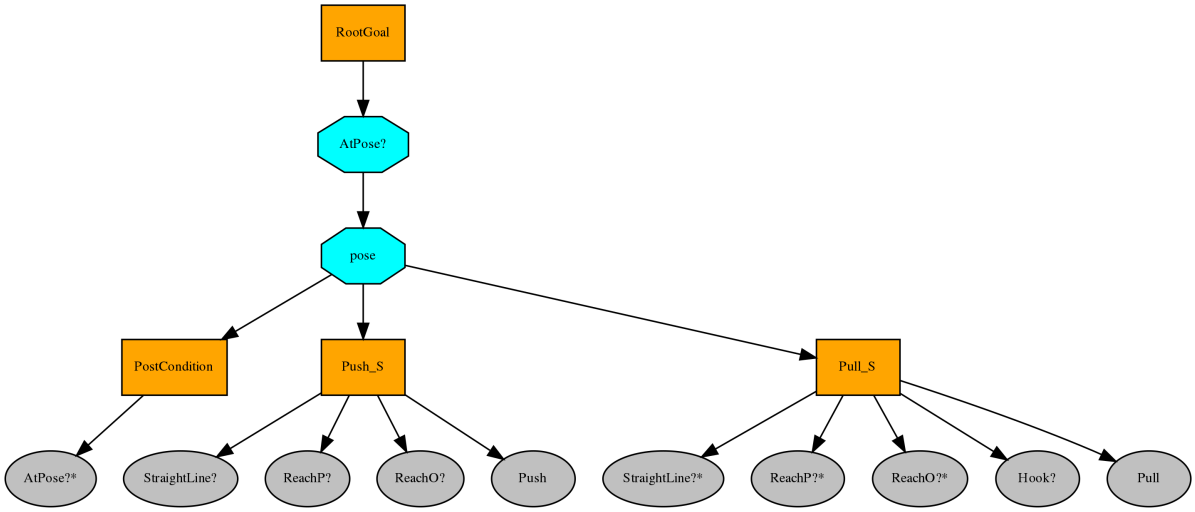
Figure 6.4. The failed predicate *canReachObj* is replaced with a PPA of Augment. Notice the sequence of dependent postconditions, and the pull action brought down due to action dependency.

The entire process of tree creation is shown

This process is shown in Figure 6.5. The goal is to have $objectAtPose(\text{“}can\text{”},\text{“}p1\text{”})$ (a). Since the predicate is not satisfied, relevant actions are initialized in the PPA structure, (pull and push) along with their preconditions (b). Since push is not a valid action (since the goal is farther away from the current position), the pull action is explored further. Now, as the precondition $canReachObject(\text{“}can\text{”},\text{“}r\text{”})$ for the robot fails, a new PPA is spawned to satisfy the predicate, using the $Augment(?r,?o)$ operator and another PPA is instantiated (d). In the PPA structure, the postcondition itself is a sequence, consisting of the relevant (dependent) preconditions of *pull*.



(a) Goal

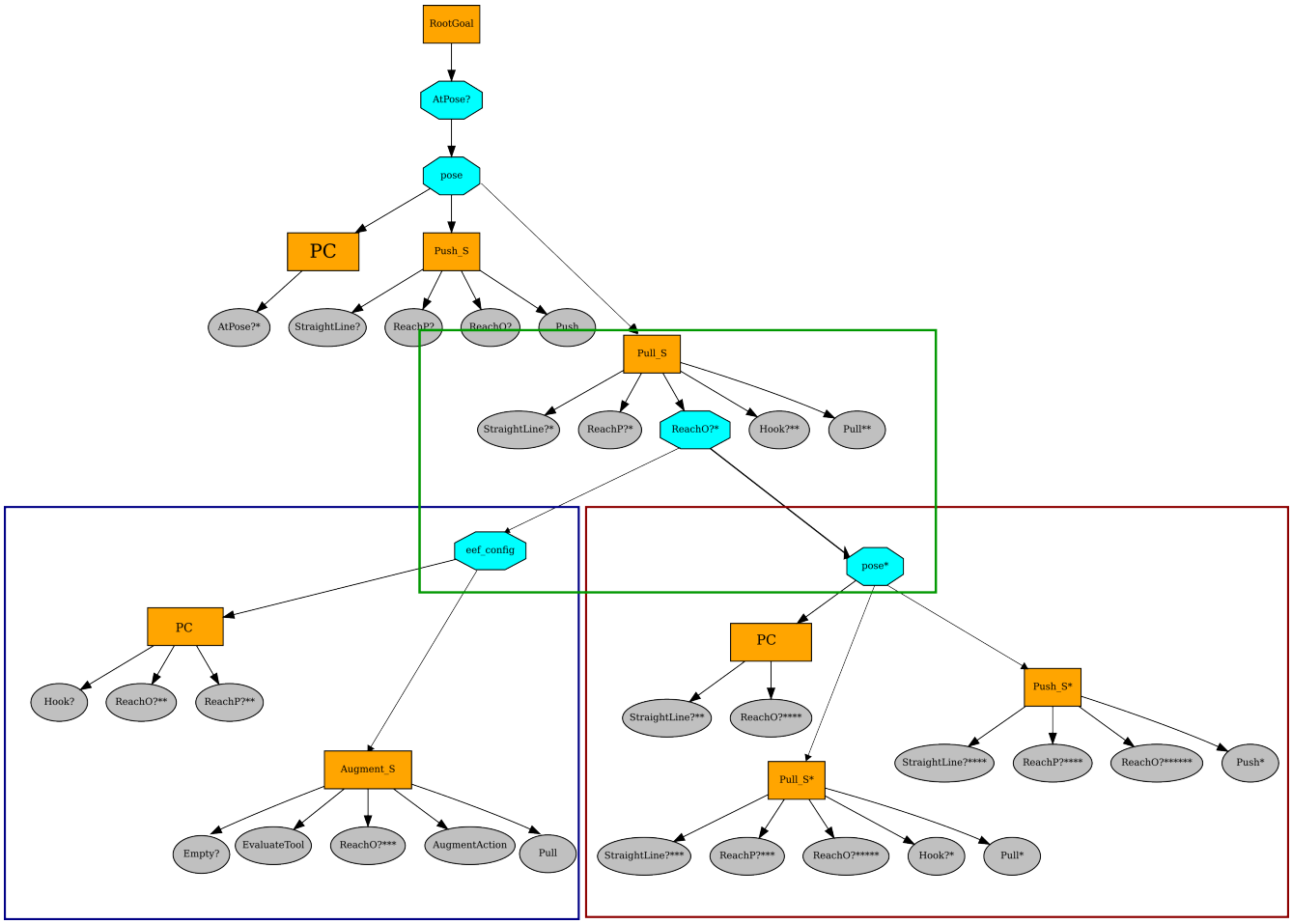


(b) Action to satisfy goal conditions

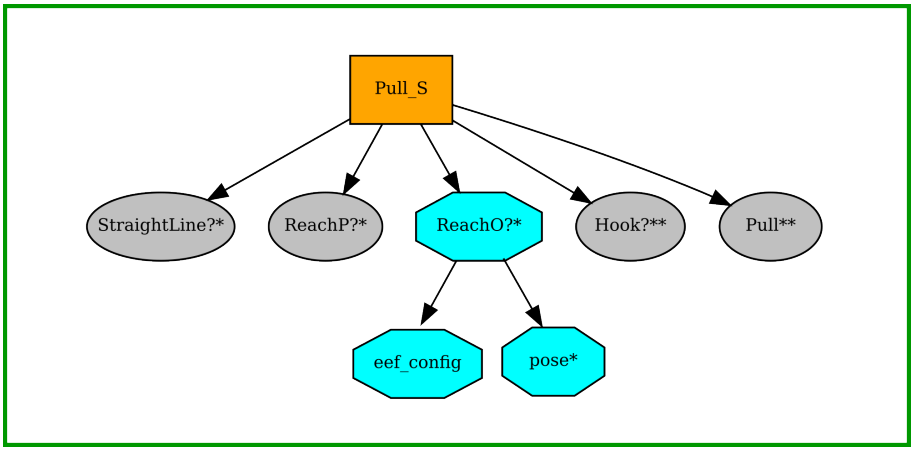
Figure 6.5. Evolution of Behavior tree

This here is indicated by the *PostConditionTree* node. The other important element is that the *pull* action from level 2, (b) has been brought down to the next level (e), after the *AugmentAction* operator node, in the sequence, and is symbolically linked to the upper one (indicated by the $*$). This means that, if this pull is accomplished, then the upper level pull will also be completed.

To accomplish this, a decorator node has been inserted, that attempts to retry all the actions of a sequence until the final action returns a success. In our problem, this translates to trying out multiple grasps, until the subsequent pull has succeeded.

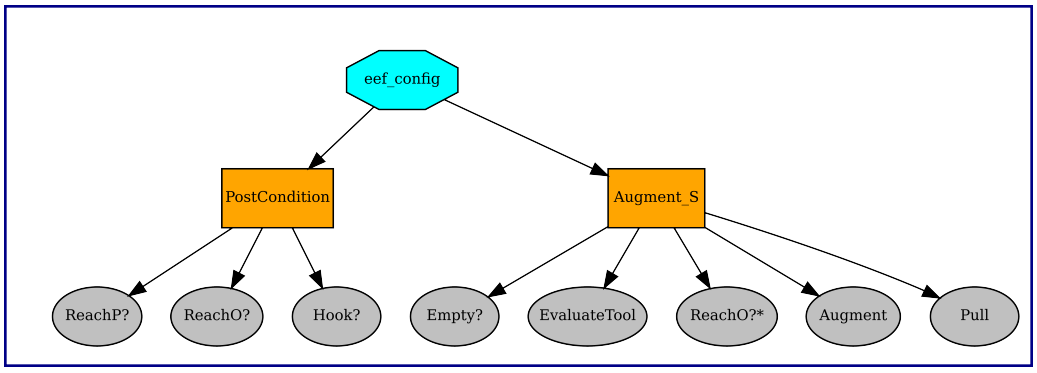


(c) Action to satisfy *canReachO*().

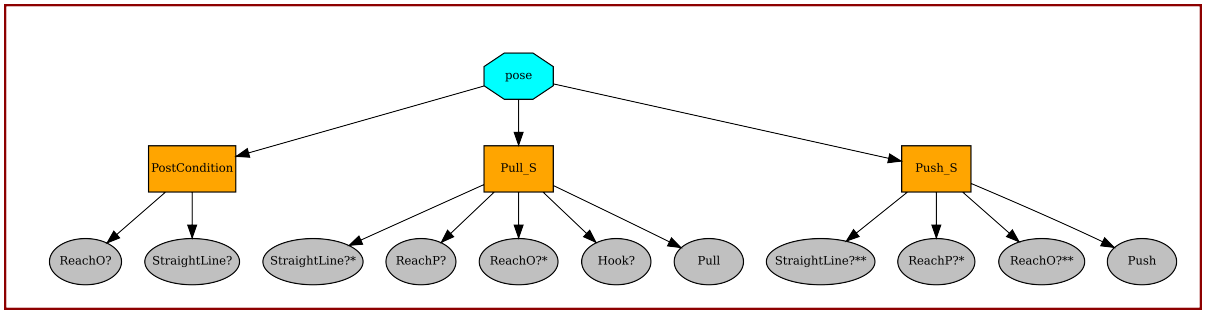


(d) Dispositions of *canReachOb ject*

Figure 6.5. Evolution of Behavior tree



(e) Details of the Augment behavior.



(f) Tree to change pose of object

Figure 6.5. Evolution of Behavior tree

The resolution of dependency of actions is also highlighted through the evolution of the Behavior Tree. The pull action generated in level 2 in Figure 6.5b is dependent on the Augment action, i.e., pulling with a tool depends on the choice of grasp. Thus, in the augment tree, in Figure 6.5d, the pull action has been chained after the Augment action.

However, looking at the other disposition of $canReachObject(?obj, ?r)$, i.e., to accomplish the predicate by changing the pose of the object, which can be done by either pushing or pulling to a new location such that it is reachable, as shown in Figure 6.5f. The parent pull action from level 2 is not brought down here, because the two *move* motions can be completed independently of each other. The second push/pull action does not depend on *how* the first push or pull was performed, and thus they are executed independently. Specifically, the relation in the semantics of effect of the intermediate pull is an equality, i.e., the position of the object at the end of the action is just a single entity instead of a set, thus, they are not dependent on each other.

For improved readability, for future images of the behavior tree generations, the Post-

ConditionTree has been omitted from the figures, but they are nonetheless generated during the runtime of the algorithm.

Thus, using behavior trees, we can design modular and reactive planners that aligns with the motivations and goals of improvisation.

Chapter 7

Setup, Implementation, and Experiments

7.1 System

The system has been built using the Robot Operating System (ROS) [37]. ROS provides a framework for developing robot software and provides tools for different programs to communicate with each other. In this work, all predicates nodes are written as ROS Services and all the execution nodes are written as ROS Actions. They work on a server-client based model. All the code has been written for ROS Melodic. Most of the code is written in Python, with very few nodes in C++. The system has been developed in both simulation and in the real world. For simulation, the environment has been developed in Coppeliasim (formerly VREP) [39], since it has excellent integration with ROS.

For building behavior trees, the "py_trees" [1] library has been used, along with its ROS wrapper "py_trees_ros" for communication with other ROS nodes. Since there could be multiple ways to perform an action (such as grasp), python generators are used, where each function call to the node yields a successful version of the action (i.e., one successful grasp). This way, actions that need to be chained together can be retried until the last action is successful. This is similar to the implementation by [20]. All motion planning has been performed using the MoveIt framework in Python [8].

[1] https://github.com/splintered-reality/py_trees

For perception, RAIL segmentation [2] is used to find the table surface and segment 3D point clouds of objects. Although most of the system works with unknown objects, since perception is not the main focus of the work, it is assumed that the 3D mesh of the object is known, purely for grasp computation. ICP algorithm [3] has been used for pose estimation. Grasps are pre-computed for the objects from the 3D mesh, although recent state-of-the-art works for grasp generation can be used for unknown objects, such as Contact-GraspNet [41].

Like in PDDL, a domain file and multiple problem files are maintained. The domain file is similar to a PDDL file, but written in Python, along with the additional RelationTuple semantics annotated. The problem file is also written in Python for simplicity. The domain and the problem files can also be written in PDDL, and conversion from PDDL is trivial.

7.2 Set-Up

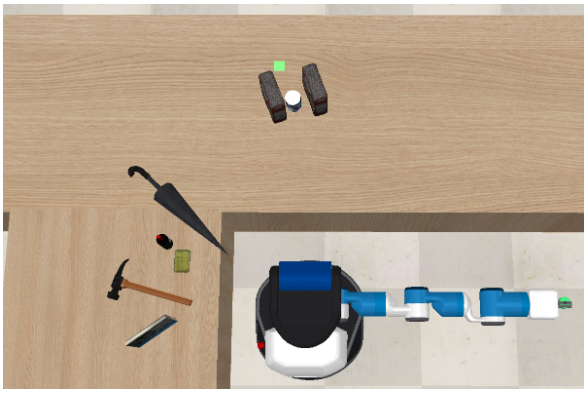
All development and experiments were done with the Fetch Robot [46], which has a mobile base, and a 7DOF arm with parallel grippers for the end effector. It has an RGBD camera that can provide dense colored point clouds that is useful for manipulation tasks. The robot has been deployed in simulation as well. The setup is shown in Figure 7.1. The goal is to get the can object in the front to the green area, using a set of objects as potential tools to the left. More formally, the goal is set as $objectAtPose(\text{“}can\text{”},\text{“}p1\text{”})$ where $p1$ is the green area. The green area is set at different places for different experiments.

A few objects have been chosen as potential interesting tools and placed in the scene. We believe that these objects have enough variety to demonstrate the different ways of using the objects as tools for different tasks. For instance the skillet and umbrella can be used as a pull tool, since they have hook like property, but cannot be used for pushing to the other end of the channel due to insufficient length in the case of the skillet and excessive cross-section in case of the umbrella. The book can however be used for the push task.

First, the robot scans to the left, scans the list of objects that can be used as tools. Then it

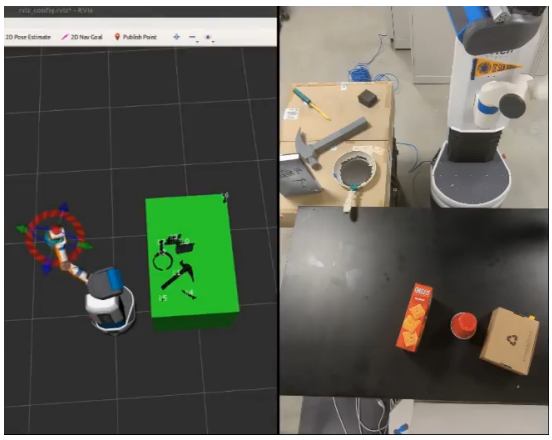
[2] https://github.com/GT-RAIL/rail_segmentation

(a) Setup (for a pull task)

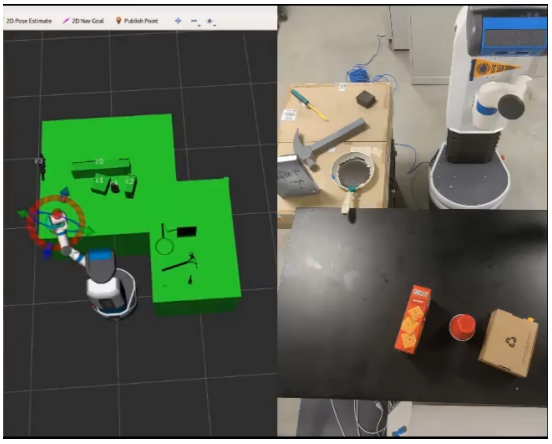


(b) Setup (for a push or a poke task)

Figure 7.1. Setup in Coppeliasim. The potential tools are to the left. The goal is to get the can in front of the goal region.



(a) Scan Tools



(b) Scan the scene

Figure 7.2. The scene and the objects provided are first identified

faces forward and scans the problem and obtains the pose of the can and the goal pose, as shown in Figure 7.2.

Then the behavior tree is spawned and the algorithm 2 is run.

7.3 Experiments

Since there exists no canonical tools for push, pull and poke, we seek to investigate the capacity of the robot to use everyday objects as tools for these actions. Different tasks are considered to test the different capabilities of the system. Empirical evaluations are not performed

since the developed system is proof-of-concept work. The situations and tasks presented to the robot are tailored for the tools provided in the scene. We consider two levels of tasks : (i) simple atomic tasks that require only one action, and (ii) complex longer horizon tasks that require reasoning multiple actions.

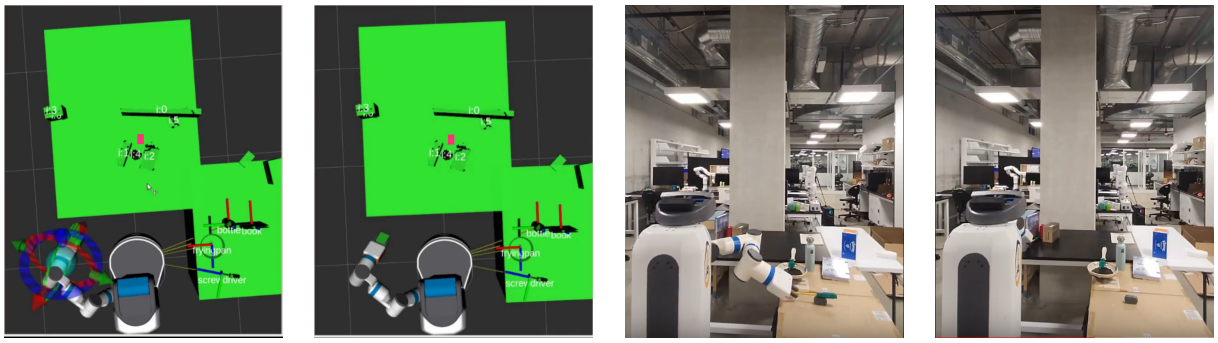
7.3.1 Simple Push and Pull Tasks

In this stage, we consider the task of pulling and pushing the "can" object to *easy* locations by using the objects around the robot.

In the first task, the goal is to have the object beyond the channel, as indicated by the pink box to the left in Figure 7.3a. The first two images indicate the visualization of different frames and geometric computations, while the last two frame demonstrates the real world execution. Key high-level decisions made for this task include:

- Spawn Push to move object to pink rectangle.
- Since the object cannot be reached, augment self with a tool
- Compute tool prototype, as shown in the visualization in Figure 7.3b.
- Evaluate the available tools.
- Identify that the screwdriver fits the prototype description.
- Grasp screwdriver, as shown in Figure 7.3c
- Perform the push. (Figure 7.3d)

An important observation in this context is the unconventional use of the screwdriver, wherein the handle serves as the contact point, while the shaft functions as the grasp region. This deviates from the conventional use of a screwdriver. This is because the handle region of the screwdriver has the flat end required to perform a better push, in comparison to the sharper end of the shaft. This demonstrates non-canonical tool use.



(a) Goal **(b)** Augment Prototype **(c)** Screwdriver Grasp **(d)** Screwdriver push

Figure 7.3. The decisions made and the action of pushing with the handle of the screwdriver are shown.

Similarly, a book is also used as an object to execute the push action, as shown in Figure 7.4, since the length and the *width* of the book respect the augment dimensions calculated for the tool prototype.

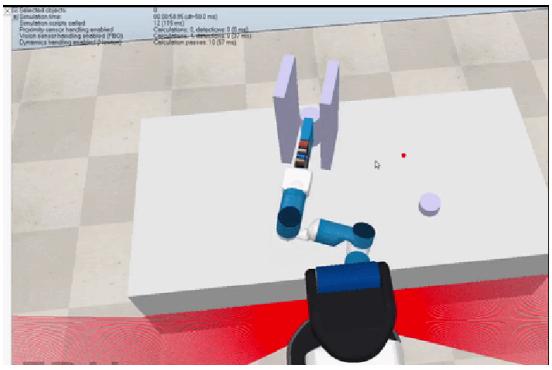
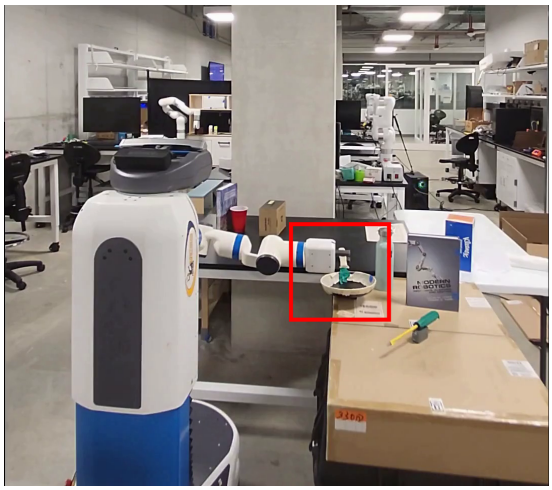


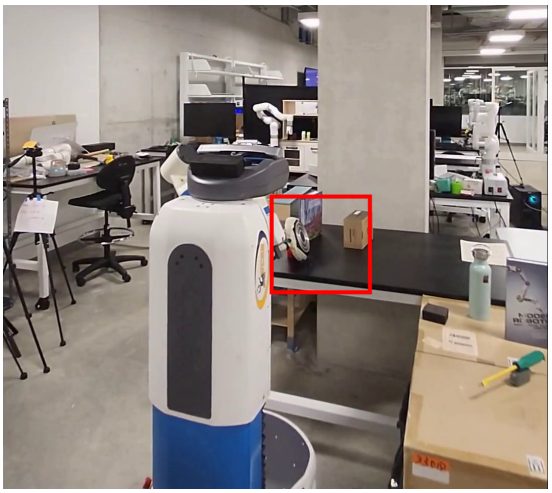
Figure 7.4. Pushing using a book is shown

In the second task, the objective is to pull the object closer to the robot. Using the line of reasoning presented in the behavior tree in Figure 6.5, the skillet object is picked up and used to pull the object closer, as shown in Figure 7.5, since the skillet has a hook-like structure. Likewise, the simulation shown in Figure 7.6 demonstrates the use of an umbrella to pull the object.

Thus, using this framework, we can show how the robot can use everyday objects as tools to perform push and pull tasks.



(a) Grasp skillet



(b) Pull with skillet

Figure 7.5. Using a skillet to pull the target object

7.3.2 Complex Tasks

In this setting, the goals provided to the robot are such that they require reasoning through multiple steps.

In the first task, the goal is to pick up the can object as shown in Figure 7.7. Again, the key reasoning steps done are:

- The goal of $isHolding(?obj, ?r)$ is spawned, which has preconditions of being able to reach the object, and the robot hand being empty.
- Since the robot cannot reach the object, it evaluates the dispositions of the predicate, which are the end effector configuration and the pose of the object.
- Although Augment can change the end effector configuration, it is an invalid action, since the robot hand being empty is a dependent predicate when changing the end effector configuration. Thus Augment is rejected.
- The other disposition is to change the pose, and a pull action is spawned.
- Similar to the steps discussed in the previous section, to pull the object, it augments itself

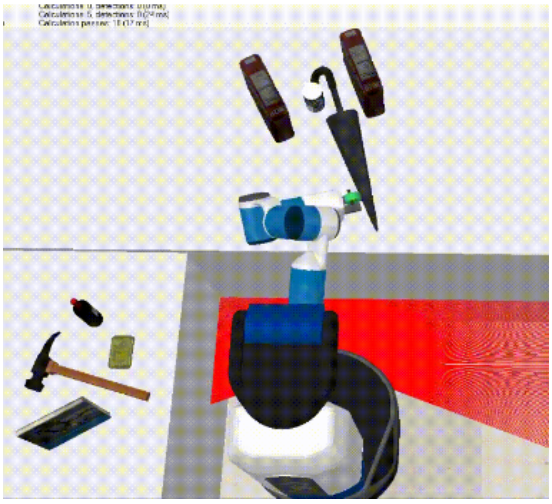


Figure 7.6. Pulling the object with an umbrella

with a tool and pulls it, by using a hammer.

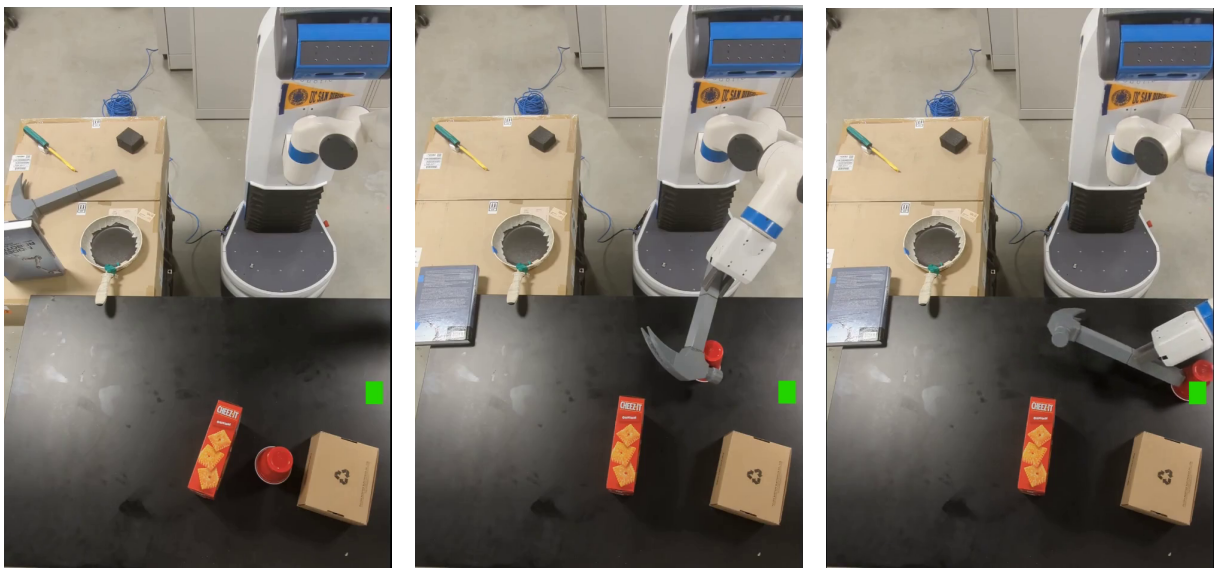
- However, to pick up the object, the robot hand needs to be empty.
- The robot hand is made empty by dropping the tool that was picked up.
- Finally, the target object is picked up.

Through this example, we demonstrate how multiple actions (augment, pull, drop, pickup) are reasoned in different time-steps to accomplish the final goal, and why identifying dependency between predicates is important to make logical and sound decisions.

In the second task, the goal is to move the object to the green region again, as shown in Figure 7.8. However, since a straight line does not exist between the object and the green region, it spawns a pull action to move it to a new location, and then attempt to push it from that new location to the green region. To pull the object, it uses the *hook* region of the hammer to make contact with the object. However, to push the object, it does not require the hook region, and the backside of the hammer makes contact with the target object and is pushed. This encapsulates the essence of the proposed work, where different constraints are achieved by different usage of the objects as tools.

(a) Pull Object **(b)** Drop Tool **(c)** Pickup Can

Figure 7.7. Sequence of actions to pull the object out using a hammer, drop the tool and then pick up the object



(a) Start state **(b)** Pull to intermediate location **(c)** Push to green location

Figure 7.8. Sequence of actions to pull to an intermediate location followed by push to goal location. Note the different contact points of the tool for the tasks

Chapter 8

Conclusion and Future Work

Through this work, we demonstrate how providing semantics to task planning leads to emergence of novel usage of objects as tools to accomplish goals. We realize this by identifying the different concepts required for an entire end-to-end system to work. In Chapter 3, the need for a tool-agnostic representation of actions is motivated, accompanied by a systematic approach to establishing preconditions for said actions, borrowing from multiple existing works in tool use and compliant manipulation. Chapter 4 discusses the role of affordances in providing semantics to predicates and introduces a mechanism through which affordances can be used for planning in a classical backward search planning. By encoding affordances, the planner can form meaningful relationships between properties of entities and actions. The benefits of using these affordances also help in addressing a few Task and Motion Planning problems, ultimately building a planner with enhanced reasoning capabilities. Chapter 5 is primarily concerned with the application of the affordance planning for tool use. It introduces the "Augment" operator that enables a robot to modify itself, thus augmenting its capabilities and altering its mode of interaction with the environment. A detailed examination of the functioning of the Augment operator is provided, that highlights both imagination of the right tool prototype, and selecting a tool that aligns with the prototype. Finally, all of these components are integrated into an execution framework using Behavior Trees, where planning and execution are interleaved.

While an end-to-end system has been developed to highlight the contributions of using

affordances for planning, due to the limits of time, a number of outstanding issues have been deferred, and left for future work. Some of these include.

- Empirical evaluation of different objects for different goals.
- A more optimal solution for computing tool prototype by a better optimization strategy (such as using differentiable collision checking)
- Extension to Category-II tools by explicit modelling of agent-tool interfaces and tool-target interfaces.
- Exploring the possibility of being able to learn and store some of the affordances discovered.
- Integration of more optimized planners such as Fast Downward [24] or Fast Forward [26].
- Increasing the set of actions available to the robot

Thus, reasoning in terms of affordances, and making connections between different properties of entities instead of identities holds promise in enhancing the operational scope of robots and enabling their effective deployment in the real world situations which are both unseen and noisy.

Bibliography

[1] Paola Ardón, Èric Pairet, Katrin S. Lohan, Subramanian Ramamoorthy, and Ronald P. A. Petrick. Affordances in Robotic Tasks – A Survey, April 2020. arXiv:2004.07400 [cs].

[2] Jennifer Barry, Kaijen Hsiao, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Manipulation with Multiple Action Types. In Jaydev P. Desai, Gregory Dudek, Oussama Khatib, and Vijay Kumar, editors, *Experimental Robotics: The 13th International Symposium on Experimental Robotics*, Springer Tracts in Advanced Robotics, pages 531–545. Springer International Publishing, Heidelberg, 2013.

[3] Paul J. Besl and Neil D. McKay. Method for registration of 3-D shapes. In *Sensor Fusion IV: Control Paradigms and Data Structures*, volume 1611, pages 586–606. SPIE, April 1992.

[4] Lucilla Cardinali, Francesca Frassinetti, Claudio Brozzoli, Christian Urquizar, Alice C. Roy, and Alessandro Farnè. Tool-use induces morphological updating of the body schema. *Current Biology*, 19(12):R478–R479, June 2009.

[5] Lucilla Cardinali, Alessandro Zanini, Russell Yanofsky, Alice C. Roy, Frédérique de Vignemont, Jody C. Culham, and Alessandro Farnè. The toolish hand illusion: embodiment of a tool based on similarity with the hand. *Scientific Reports*, 11(1):2024, January 2021. Number: 1 Publisher: Nature Publishing Group.

[6] S Cass. Apollo 13, We Have a Solution. *IEEE Spectrum On-line*.

[7] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. pages 2722–2730, 2015.

[8] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study, April 2014. arXiv:1404.3785 [cs].

[9] Michele Colledanchise, Diogo Almeida, and Petter Ögren. Towards Blended Reactive Planning and Acting using Behavior Trees. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8839–8845, May 2019. ISSN: 2577-087X.

[10] Michele Colledanchise and Lorenzo Natale. On the Implementation of Behavior Trees in Robotics. *IEEE Robotics and Automation Letters*, 6(3):5929–5936, July 2021. Conference Name: IEEE Robotics and Automation Letters.

[11] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI: An Introduction*. July 2018. arXiv:1709.00084 [cs].

[12] Francisco Cruz, Sven Magg, Cornelius Weber, and Stefan Wermter. Training Agents With Interactive Reinforcement Learning and Contextual Affordances. *IEEE Transactions on Cognitive and Developmental Systems*, 8(4):271–284, December 2016. Conference Name: IEEE Transactions on Cognitive and Developmental Systems.

[13] Nicola Cutting, Ian A. Apperly, and Sarah R. Beck. Why do children lack the flexibility to innovate tools? *Journal of Experimental Child Psychology*, 109(4):497–511, August 2011.

[14] Nicola Cutting, Ian A. Apperly, Jackie Chappell, and Sarah R. Beck. Is tool modification more difficult than innovation? *Cognitive Development*, 52:100811, October 2019.

[15] Thanh-Toan Do, Anh Nguyen, and Ian Reid. AffordanceNet: An End-to-End Deep Learning Approach for Object Affordance Detection. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5882–5889, May 2018. ISSN: 2577-087X.

[16] Christian Dornhege, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel. Semantic Attachments for Domain-Independent Planning Systems. In Erwin Prassler, Marius Zöllner, Rainer Bischoff, Wolfram Burgard, Robert Haschke, Martin Hägele, Gisbert Lawitzky, Bernhard Nebel, Paul Plöger, and Ulrich Reiser, editors, *Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*, Springer Tracts in Advanced Robotics, pages 99–115. Springer, Berlin, Heidelberg, 2012.

[17] Sarah Elliott, Michelle Valente, and Maya Cakmak. Making objects graspable in confined environments through push and pull manipulation with a tool. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4851–4858, May 2016.

[18] Max Garagnani and Yucheng Ding. Model-based planning for object-rearrangement problems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-03)-Workshop on PDDL*, pages 49–58, 2003.

[19] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated Task and Motion Planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1):265–293, May 2021.

[20] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDLStream: Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning, March 2020. arXiv:1802.08705 [cs].

[21] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 1 edition, July 2016.

[22] James J Gibson. The theory of affordances. *Lawrence Erlbaum Associates*, 1:67–82, 1977.

[23] Romana Gruber, Martina Schiestl, Markus Boeckle, Anna Frohnwieser, Rachael Miller, Russell D. Gray, Nicola S. Clayton, and Alex H. Taylor. New Caledonian Crows Use Mental Representations to Solve Metatool Problems. *Current Biology*, 29(4):686–692.e3, February 2019.

[24] M. Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, July 2006.

[25] Tucker Hermans, James M. Rehg, and Aaron Bobick. Guided pushing for object singulation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4783–4790, October 2012. ISSN: 2153-0866.

[26] Joerg Hoffmann. FF: The Fast-Forward Planning System. *AI Magazine*, 22(3):57–57, September 2001. Number: 3.

[27] Rachel Holladay, Tomas Lozano-Perez, and Alberto Rodriguez. Force-and-Motion Constrained Planning for Tool Use. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7409–7416, Macau, China, November 2019. IEEE.

[28] Daniel Sebastian Leidner. *Cognitive Reasoning for Compliant Robot Manipulation*, volume 127 of *Springer Tracts in Advanced Robotics*. Springer International Publishing, Cham, 2019.

[29] Ruiya Li, Duc Truong Pham, Jun Huang, Yuegang Tan, Mo Qu, Yongjing Wang, Mairi Kerin, Kaiwen Jiang, Shizhong Su, Chunqian Ji, Quan Liu, and Zude Zhou. Unfastening of Hexagonal Headed Screws by a Collaborative Robot. *IEEE Transactions on Automation Science and Engineering*, 17(3):1455–1468, July 2020. Conference Name: IEEE Transactions on Automation Science and Engineering.

[30] Christopher Lã. Grounding Planning Operators by Affordances.

[31] Toki Migimatsu and Jeannette Bohg. Object-Centric Task and Motion Planning in Dynamic Environments. *IEEE Robotics and Automation Letters*, 5(2):844–851, April 2020. arXiv:1911.04679 [cs].

[32] Austin Myers, Ching L. Teo, Cornelia Fermuller, and Yiannis Aloimonos. Affordance detection of tool parts from geometric features. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1374–1381, Seattle, WA, USA, May 2015. IEEE.

[33] Yuki Noguchi, Tatsuya Matsushima, Yutaka Matsuo, and Shixiang Shane Gu. Tool as Embodiment for Recursive Manipulation, December 2021. arXiv:2112.00359 [cs].

[34] François Osiurak and Arnaud Badets. Tool use and affordance: Manipulation-based versus reasoning-based approaches. *Psychological Review*, 123(5):534. Publisher: US: American Psychological Association.

[35] Jia Pan, Sachin Chitta, and Dinesh Manocha. FCL: A general purpose library for collision and proximity queries. In *2012 IEEE International Conference on Robotics and Automation*, pages 3859–3866, May 2012. ISSN: 1050-4729.

[36] Despoina Paschalidou, Ali Osman Ulusoy, and Andreas Geiger. Superquadrics Revisited: Learning 3D Shape Parsing beyond Cuboids, April 2019. arXiv:1904.09970 [cs].

[37] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System.

[38] Anant Ram, Sunita Jalal, Anand S. Jalal, and Manoj Kumar. A Density Based Algorithm for Discovering Density Varied Clusters in Large Spatial Databases. *International Journal of Computer Applications*, 3(6):1–4, June 2010.

[39] Eric Rohmer, Surya P. N. Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326, November 2013. ISSN: 2153-0866.

[40] Sipu Ruan, Karen L. Poblete, Hongtao Wu, Qianli Ma, and Gregory S. Chirikjian. Efficient path planning in narrow passages for robots with ellipsoidal components. *IEEE Transactions on Robotics*, 2022. Publisher: IEEE.

[41] Martin Sundermeyer, Arsalan Mousavian, Rudolph Triebel, and Dieter Fox. Contact-GraspNet: Efficient 6-DoF Grasp Generation in Cluttered Scenes. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 13438–13444, May 2021. ISSN: 2577-087X.

[42] Keng Peng Tee, Jun Li, Lawrence Tai Pang Chen, Kong Wah Wan, and Gowrishankar Ganesh. Towards Emergence of Tool Use in Robots: Automatic Tool Recognition and Use Without Prior Tool Learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6439–6446, May 2018. ISSN: 2577-087X.

[43] Fumiaki Toyoshima, Adrien Barton, and Jean-François Ethier. Affordances and their ontological core. *Applied Ontology*, 17(2):285–320, January 2022. Publisher: IOS Press.

[44] M.T. Turvey. Affordances and Prospective Control: An Outline of the Ontology. *Ecological Psychology*, 4(3):173–187, September 1992. Publisher: Routledge _eprint: https://doi.org/10.1207/s15326969eco0403_3.

[45] Elisabetta Visalberghi, Gloria Sabbatini, Alex H. Taylor, and Gavin R. Hunt. Cognitive insights from tool use in nonhuman animals. In *APA handbook of comparative psychology: Perception, learning, and cognition, Vol. 2*, APA handbooks in psychology®, pages 673–701. American Psychological Association, Washington, DC, US, 2017.

[46] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. Fetch & Freight: Standard Platforms for Service Robot Applications.

[47] Yuechuan Xue and Yan-Bin Jia. Gripping a Kitchen Knife on the Cutting Board. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9226–9231, October 2020. ISSN: 2153-0866.

[48] Erol Şahin, Maya Çakmak, Mehmet R. Doğar, Emre Uğur, and Göktürk Üçoluk. To Afford or Not to Afford: A New Formalization of Affordances Toward Affordance-Based Robot Control. *Adaptive Behavior*, 15(4):447–472, December 2007.