Chair of Computer Science 6 (Machine Learning and Reasoning)

RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair of Machine Learning and Reasoning.
Diese Arbeit wurde vorgelegt am Lehrstuhl für Maschinelles Lernen und Inferenz.

Learning Critical Regions via Feasibility Constraints in Task and Motion Planning

Bachelor Thesis
Bachelorarbeit

Presented by / **Vorgelegt von**

Baran Dello
422616

Supervised by / **Betreut von**	Dr. Zlatan Ajanović
1st Examiner / **1. Prüfer**	Prof. Hector Geffner, Ph.D.
2nd Examiner / **2. Prüfer**	Prof. Dr. rer. nat. Christopher Morris

Aachen, May 19, 2025

Abstract

In the field of task and motion planning, identifying the feasibility of a task (PDDL action) is challenging, and it is usually validated on a sample-to-sample basis. This can be critical, as many samples fail, leading to wasted computation, because they fall outside the true feasibility region, where an action can succeed without kinematic or collision violations. Dealing with regions of feasibility allows us to reason about sets of states that are feasible for a given action. However, learning such regions can be challenging, especially in high-dimensional spaces. This thesis proposes a method for learning action feasibility regions based on version space learning that leverages high-level symbolic models with a rich feature grammar to generate feature constrains, construct a hypothesis space and then learn generalizable action feasibility regions, represented by a set of consistent hypotheses that form the feasibility bounds of the action by bounding the continuous parameters of the action to guarantee valid action simulation. This is then integrated into a Task and Motion Planning framework to improve the planning process. Finally, we demonstrate the effectiveness of our approach in a simulated environment and present significant results in certain cases.

Contents

1 Introduction

1.1 Motivation

In robot planning, a fundamental challenge in Task and Motion Planning (TAMP) domains lies in effectively reasoning about high-level symbolic plans as well as low-level continuous actions required to successfully perform actions for a given task. Task and Motion Planning is essential in robot planning for specifying the discrete tasks and the corresponding motions required to perform certain goals. Traditional TAMP methods, where a task planner first generates a candidate task plan as a sequence of actions and then a motion planner attempts to bring these actions into motions, can effectively work well, especially the motion planner to generate feasible motions in the robot's *configuration space* to perform high-level tasks in low-dimensional and structured environments. However, as robotic tasks become more complex, these traditional methods face difficulties in navigating through high-dimensional configuration spaces.

Consider a simple pick-and-place task, where a robot is required to grasp a block from a tabletop and then place it inside a shelf. A typical TAMP planner might choose an initial grasp pose, for example, a top-down grasp, only to discover upon the full motion simulation that this pose is infeasible due to some collision. The planner then selects another sample, runs the simulator again, and repeats this process until a feasible grasp is found. Each of these simulation attempts can be quite costly, and the planner may waste a lot of time and resources on infeasible actions.

This is where *critical regions* come into play. They are specific areas important for the successful execution of tasks and can represent areas with significant action occurrence, such as reachable areas on a tabletop or grasping points. By focusing on these critical regions, motion planners can focus on the most important parts of the configuration space, thereby reducing computational overhead and improving performance. One way to implement these critical regions into planning is to learn the critical region for each action independently, where a critical region for an action represents the set of continuous parameters under which the action can be successfully executed. For example, in the pick-and-place task, the critical region for the pick action would be the set of all possible grasp poses that do not result in collisions with the environment or other objects.

In this thesis, we suggest an approach based on symbolic models Silver *et al.* [26, 27] and version space learning to identify feasible critical regions, or more precisely, action-feasibility regions, within the robot's configuration space. Using a unified feature grammar of geometric and relational features, a set of candidate hypotheses can be generated, and iteratively pruned via specific feasibility checks, our suggested approach can learn a set of interpretable constraints or rules, which can be utilized to generate continuous action parameters during the low-level phase. These rules can be used to define the critical region or feasibility region of an action, which can be integrated into the TAMP framework and realized to yield a more efficient planning process with less backtracking.

1.2 Outline

Now that we have established the motivation for our work, we will provide an overview of the structure of this thesis. In Chapter 2, we will provide the necessary background information that is crucial for the later chapters. We introduce the basic concept of spatial representations in robotics and then dive into the details of TAMP and its challenges. Furthermore, we discuss the concept of critical regions, the topic of this thesis. We also discuss works in decomposition, symbolic models, and version space learning. Chapter 3 handles the related work in the field of TAMP and especially action feasibility in planning. In Chapter 4, we introduce our approach, starting with a solid problem formulation and problem setting, and then present the research questions that we tackle. We then introduce our method for learning critical regions, as well as the implementation. Then, we discuss the integration of our approach into a TAMP framework. Some implementation details are discussed in Chapter 5, and we also introduce the framework that we use. In Chapter 6, we present the results of our experiments and evaluate the performance of our approach. Finally, we conclude the thesis in Chapter 7 with future work and a summary of our work.

2 Background

2.1 Spatial Representation in Robotics

In this subsection, we will introduce the foundational concepts of spatial representation in robotics, which are essential for planning problems, as they describe the locations, movements, and relations of objects in the environment. We start by defining the coordinate frames used in robotics, followed by the geometric representation and transformations in the first subsection. The second subsection shows a basic description of the workspace and the configuration space, which are important for planning problems.

2.1.1 Geometric Representation, Transformation and Coordinate Frames

In robotics, a robot is composed of multiple rigid bodies or links that are connected by joints. The first link is often called the base, and the last link is called the end effector, which is typically a gripper. The end-effector plays an important role in robot planning, which is also central to this work.

Generally, there is a need to systematically represent the positions and orientations of all objects in the environment. This is where coordinate frames come into play [14, 25]. A coordinate frame consists of an origin and three mutually orthogonal axes, all fixed within a body or object. The position and orientation of that body or object can then be described with respect to that coordinate frame. A universal approach in planning is to define a fixed coordinate frame, which we refer to as the World Frame (WF), which provides a global reference for all objects in the environment. A local coordinate frame is typically attached to every robot link, as well as to every object in the environment. These local frames are also called moving frames, as they move with the object they are attached to.

Translation and rotation are used to describe these motions in the environment [18]. A translation is a movement along a straight line, in which all points of the object move to a new position without changing their orientation. A rotation, on the other hand, changes the orientation of an object by some angle θ while at least one point remains fixed [14, 25]. The combination of both rotation and translation is called a *rigid body transformation*, which is expressed using homogeneous transformation matrices to describe the movement of an object from one reference frame to another.

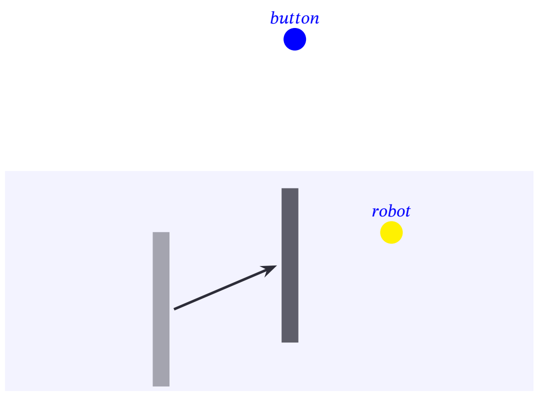

Figure 2.1: Translation of the stick in the Stick-Button domain [27].

2D Transformations In a 2D environment, a robot $\mathcal{A}$ in a position $p = (x, y)$ is translated by a vector $d = \left(d_x, d_y\right)$ to a new position

$$p' = (x', y') = \left(x + d_x, y + d_y\right). \tag{2.1}$$

The rotation by an angle θ on the other hand, is done by using a rotation matrix

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \tag{2.2}$$

which gives the new position transformed by

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = R(\theta) \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{pmatrix}. \tag{2.3}$$

Now both translation and rotation can be combined [18]. Consider the reference frames RF_1 and RF_2 in a 2D environment. If we want to change the point (x, y) from the RF_1 to RF_2, we can do this by first rotating the point by θ and then translating it by $d = \left(d_x, d_y\right)$. This can be expressed as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -sin\theta \\ \sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}. \tag{2.4}$$

Thus, we get the following 3×3 homogeneous transformation matrix

$$T = \begin{pmatrix} \cos\theta & -\sin\theta & d_x \\ \sin\theta & \cos\theta & d_y \\ 0 & 0 & 1 \end{pmatrix}, \tag{2.5}$$

which represents a rotation followed by a translation in a 2D environment [18].

3D Transformations The 3D environment follows the same principles as the 2D environment, but the rotation is more complex [18]. A robot $\mathcal{A}$ in a position $p = (x, y, z)$ is translated by a vector $d = (d_x, d_y, d_z)$ to a new position

$$p' = (x', y', z') = (x + d_x, y + d_y, z + d_z). \tag{2.6}$$

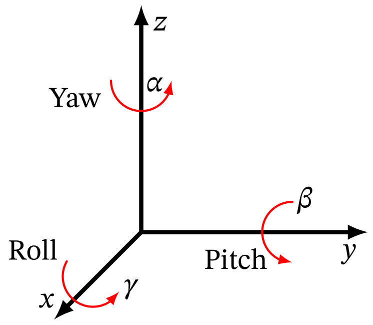

Figure 2.2: yaw, pitch, and roll in 3D.

The rotation in 3D can be done by a sequence of rotations around the three axes, which are called yaw, pitch, and roll, as shown in 2.2. The books [25] and [18] provide detailed explanations of these rotations, which we will summarize here. The rotation of a roll is given by

$$R_x(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix}, \tag{2.7}$$

which is a counterclockwise rotation around the x-axis. A pitch, on the other hand, is a counterclockwise rotation around the y-axis, given by

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}. \tag{2.8}$$

Finally, the rotation matrix of a yaw is counterclockwise rotation around the z-axis, given by

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.9}$$

Now, we can obtain a single rotation matrix by multiplying these three matrices:

$$\begin{aligned} R(\alpha,\beta,\gamma) &= R_z(\alpha)\cdot R_y(\beta)\cdot R_x(\gamma) \\ &= \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma-\sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma+\sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma+\cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma-\cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{pmatrix}. \end{aligned} \tag{2.10}$$

Combining the rotation matrix with the translation vector $d = (d_x, d_y, d_z)$ results in the 4×4 homogeneous transformation matrix [1]

$$T = \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma-\sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma+\sin\alpha\sin\gamma & d_x \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma+\cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma-\cos\alpha\sin\gamma & d_y \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.11}$$

2.1.2 Workspace and configuration space

Now that we have introduced the basic concepts of spatial representation, we can describe the workspace that a robot operates in. Generally in robotics, the workspace $\mathcal{W} = \mathbb{R}^n$ is the total volume of space the end-effector can reach [14]. The workspace obstacle region $\mathcal{W}_{obs} \subset \mathcal{W}$ represents the set of states of all obstacles within the workspace $\mathcal{W}$. $\mathcal{W}_{free}$ represents the free workspace. Task and Motion Planning traditionally operates in the *configuration space* or C-space C, the set of all possible configurations q a robot can attain [18]. A configuration q of a robot $\mathcal{A}$ is a complete specification of the positions of all parts of the robot. Mathematically, q is a point in the C-space, representing a unique arrangement of the robot's joints and links [12, 25]. It includes all the parameters necessary to describe the robot's pose, considering its kinematic structure. The set of points occupied by the robot when at configuration $q \in C$ is denoted as the closed set $\mathcal{R}(q) \subset \mathcal{W}$. The *configuration space obstacle region* C_{obs} can be defined as

$$C_{obs} = \{q \in C \mid \mathcal{R}(q) \cap \mathcal{W}_{\mathcal{O}} \neq \emptyset\}. \tag{2.12}$$

The *free configuration space*, the set of configurations where the robot is not in collision with any obstacle is denoted as $C_{free} = C \setminus C_{obs}$. Thus, a configuration q is free, if $q \in C_{free}$ [18].

2.2 Task and Motion Planning

Generally in robotics, planning problems can be categorized into task planning and motion planning, which are integrated into the (TAMP) framework.

[1]Note that $alpha$, $beta$, and $gamma$ are the yaw, pitch, and roll angles, respectively.

Task Planning (TP) or AI planning in robotics refers to the high-level decision-making process in which a sequence of symbolic actions is generated to achieve specific predefined goals [7]. It operates within a discrete domain, focusing on a state space $\mathcal{S}$ (set of all possible states), transitions $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ (actions between states), an initial state $s_0 \in \mathcal{S}$ and a set $\mathcal{S}_* \subseteq \mathcal{S}$ of goal states. Actions are defined with preconditions that must be satisfied for the execution to occur, and effects that describe how the action changes the state of the environment [7]. The objective of the planner is to find a task plan τ_{TP}, a valid sequence of transitions that leads from s_0 to a goal state $s_* \in \mathcal{S}_*$. Essentially, the problem is formulated as a search over the space of possible action sequences until a valid plan is found [7], satisfying the goal conditions. Traditional frameworks such as Stanford Research Institute Problem Solver (STRIPS) [5] and Planning Domain Definition Language (PDDL) [8] formalize these problems using symbolic representations, preconditions, and effects.

Motion Planning (MP), on the other hand, deals with the low-level continuous aspects of planning in the robot's configuration space [18]. For an initial configuration q_s and a goal configuration q_g, the planner computes a collision-free trajectory $\tau_{MP} : [0, 1] \rightarrow C_{free}$, such that $\tau_{MP}(0) = q_s$ and $\tau_{MP}(1) = q_g$ [7, 19]. This includes considerations such as the robot's kinematic constraints or obstacles. The problem is challenging due to the curse of dimensionality of C, which increases the computational demands. It is shown in [7, 25] that the computation costs for C_{obs} and C_{free} increase with the increase of the robot's Degree of Freedom (DoF), since the general motion planning problem is PSPACE-hard. Thus, researchers try to develop methods that focus on specific parts of this problem to enhance the speed and efficiency [12, 19], guiding the exploration of C to areas that are more likely to yield valid and effective results.

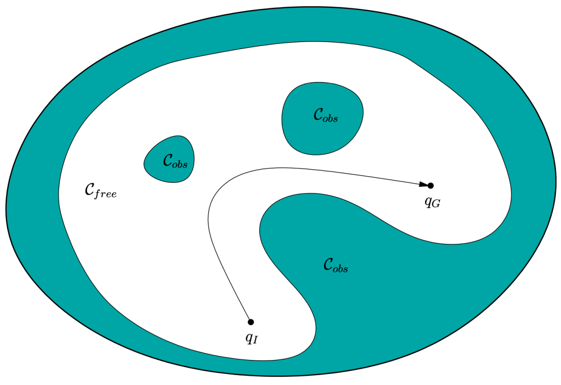

Figure 2.3: The motion planning problem in the configuration space. q_1 represents q_s in our problem definition. (Source: LaValle [18])

Multimodal Motion Planning (MMMP) lays another foundation of my work. It is an extension of Motion Planning that considers the robot $\mathcal{A}$ moving between a finite set of modes Σ [9, 16]. Ultimately, this transfers our problem to a so-called *hybrid problem* [7, 9], in which the space includes discrete modes $\sigma \in \Sigma$ and the robot's state is represented as a hybrid state $\langle q, \sigma \rangle$

with $q \in C$. Each mode σ has a corresponding feasible space $\mathcal{F}_\sigma \subset C$, which consists of all configurations that satisfy certain constraints related to σ.

The objective is to find a collision-free path τ_{MP} from an initial state $\langle q_s, \sigma_s \rangle$ to a goal state $\langle q_g, \sigma_g \rangle$ [7]. This involves a mode sequence $[\sigma_0, \sigma_1, ..., \sigma_n]$ with $\sigma_0 = \sigma_s$ and $\sigma_n = \sigma_g$, that the robot needs to transition through to reach the goal, within-mode transitions, where the planner computes a continuous path $\tau_{MP_i} : [t_i, t_{i+1}] \rightarrow \mathcal{F}_{\sigma_i}$ for each mode σ_i, $i \in \{0, ..., n\}$, and mode transitions, where the switch from σ to σ' is determined by reaching a configuration $q \in \mathcal{F}_\sigma \cap F'_\sigma$ that satisfies the **constraints of both modes** [7]. Finally, the mode transitions and within-mode transitions are combined to complete the continuous trajectory $\tau_{MP} : [0, 1] \rightarrow C_{free}$, which satisfies all constraints and the conditions $\tau_{MP}(0) = q_s$ and $\tau_{MP}(1) = q_g$.

As indicated in [7] identifying configurations that are in the intersection of the set of constraints for two modes is challenging. When dealing with constraints of two different modes, the dimensionality of the intersection tends to be of a lower dimension. Hauser and Latombe claim in their work [9] that the existence of such configurations or so-called *transitions* is a good indication of the existence of a feasible path between a configuration q in mode σ and mode σ'. In the case of Σ being infinite, it can be partitioned into a finite set of disjoint mode families $\Sigma_1, \Sigma_2, ..., \Sigma_k$, with each mode $\sigma \in \Sigma_f$ being identifiable by a real-valued parameter $\eta \in \mathrm{H}_f$. Thus, we denote $\sigma = \langle \Sigma_f, \eta \rangle$ [7, 10]. Switching between different modes in the same family requires the system first to move to another mode family, as the families do not overlap.

In [1] authors introduced mode feasibility map that is used in a search-based Task and Motion Planning to evaluate if it is possible to transition to the other modes from the current continuous state. This is in particular used for agile driving on a slippery road where the planner employs different models to plan the motion.

TAMP integrates these elements [7], combining symbolic decisions from Task Planning with the geometric and kinematic considerations of Motion Planning. In addition to preconditions and effects, TAMP introduces constraints over continuous parameters, ensuring that each action satisfies both symbolic and geometric feasibility. The process typically involves generating a plan skeleton (a sequence of actions) and assigning continuous parameters, with feasibility checks performed iteratively to ensure constraints like collision avoidance and kinematic limitations are satisfied. This process is generally done in a feedback loop. If a motion fails, the planner revisits the symbolic plan or generates a new one. This continues until a feasible plan is found that satisfies both the symbolic and the geometric constraints.

2.3 Critical Regions

Critical regions are subsets of the configuration space C, that can be essential for finding feasible paths, as they represent specific and important areas in C, where a lot of changes to the environment occur [2]. The goal is to identify a set $\mathcal{X}_{critical} \subset C_{free}$[2], which can be seen as connecting different configurations in C_{free}. This can leave out unnecessary samples in less relevant areas, which reduce the efficiency of pathfinding.

One of the first ideas came from Gordon Wilfong [31], who addressed the importance of partitioning C into different regions for easier planning for robots that interact with movable objects. The idea is to construct a polygon called the *Trace* by tracking a reference point, known as the origin, on a robot Z. The origin is a fixed point on Z and is used to represent its position as it generally moves an object B, while maintaining contact without overlap. Then, the boundary of the trace can be formed by tracing the path of the origin of Z, and the critical regions can be identified based on interactions involving this point, such as when it aligns with a feature of B.

Several approaches were suggested to make use of critical regions in Sampling-Based Motion Planning (SBMP). Brian Ichter et al. used a graph-theoretic measure called *betweenness centrality* to estimate how often a sampled state, depicted as a node, lies on the shortest paths between other nodes in a Probabilistic Roadmap (PRM) [2, 13]. Samples with high betweenness centrality are considered critical, since they connect different regions in the space. Then they use a trained neural network to predict how critical a state is based on some local environment features, which is supposed to make the algorithm generalizable to new, unseen environments. Felipe Felix Arias et al. extended the work of Brian Ichter et al. to dynamic environments. In order to identify areas where the robot has to wait to avoid collisions with obstacles, they propose the notion of *avoidance criticality* [2], which is more challenging due to the dynamic feature of the environment. They also train a neural network to identify these avoidance critical regions based on local occupancy grids. They incorporate this into their planner to construct an Avoidance Critical Probabilistic Roadmap (ACPRM), that uses the model to sample in these identified critical regions.

Shah and Srivastava [24] pursue an approach that identifies critical regions by using deep learning to bootstrap high-level abstractions during the hierarchical planning process for completely new environments. They observe that typical methods rely upon environment-dependant state and action abstractions that have to be hand-designed by experts. They demonstrate their idea on a room navigation example, in which a robot has to reach the kitchen from another room. Rather than handcrafting the regions, they train a neural network that identifies which points lie on bottleneck regions or passages that connect different rooms, which should represent the critical regions. During planning, these learned passages are used to guide the Task and Motion Planning planner to find a feasible solution for a given task.

[2]Note that, although we stated $R \subset C$, we cannot ensure the critical regions to always be collision free. The goal however is to find the feasible (collision-free) part

2.4 Decomposition

In this work, we also refer to decomposition techniques, which are also used in both Motion Planning and Computational geometry.

2.4.1 Decomposition in Motion Planning

The complexity of dynamic workspaces in Motion Planning has driven researchers to seek for other ways to reduce the overall problem into simpler forms. The idea of decomposition in motion planning shares similarities with critical regions, in that both aim to simplify the search within the workspace or configuration space. Yet, whereas critical regions focus is on specific areas, decomposition typically involves dividing the entire workspace into regions. We explore certain aspects of decomposition, which could be relevant to our work.

One approach, called *cell decomposition*, was introduced by Jur P. van den Berg et al. [29], where the entire free workspace is partitioned into smaller cells to enhance sampling in probabilistic roadmap planners. The approach operates in two phases: the data structure phase and the sampling phase. In the data structure phase, the free workspace W_{free} is divided into a set of cells X using octrees or binary space partitions. The union of all cells results in a set $D = \bigcup\{x \in X\}$ that approximates W_{free}. Then, cells are grouped into regions based on some local properties by labeling each cell with a labeled region. A weight is assigned to each labeled region, which affects the probability of sampling within that region. Therefore, higher weights are assigned to regions like narrow passages. The sampling phase selects a random cell within a selected region, and the robot's position within that cell is uniformly selected to generate a sample. Such samples are incorporated into the probabilistic roadmap. Erion Plaku et al. introduce a planning algorithm called *SyCLoP* in [23], which combines discrete search with SBMP. It applies workspace decomposition using uniform grid partitioning or triangulation, which uses characteristics from computational geometry. Starting from the initial state in a critical region, *SyCLoP* iteratively explores neighboring regions by using an exploration tree, and regions are assigned a cost, which decreases based on the frequency of searches in a region. It then uses a specific cost-based evaluation to check the need for further search in a region.

2.4.2 Decomposition in Computational Geometry

Decomposition also plays a crucial role in *computational geometry* by partitioning complex geometric structures into simpler forms for simpler use. There are several decomposition problems researched, which include *polygon partitioning*, *space partitioning*, *mesh generation* [6, 15, 21]. One of the methods used for polygon partitioning is *convex decomposition*, which breaks down a polygon into a set of convex components, which are easier to handle in areas like motion planning. Building upon this concept, the paper [21] introduces an improved version of convex decomposition, which addresses the cost drawback, as this decomposition can be computationally expensive, especially for polygons with holes. The method uses a pre-defined *non-concavity tolerance* τ, which allows for a few components that are ”approximately convex”. In other words, a certain level of concavity is allowed within the components, which results in fewer components while still holding most of the convex advantage. To yield consistent results, the method also produces multiple different decompositions or approximations through hierarchical decomposition.

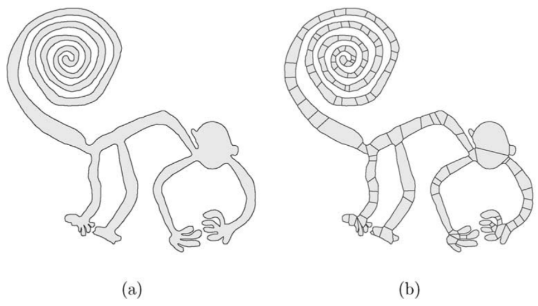

Figure 2.4: (a) Initial monkey to be decomposed into multiple cells. (b) approximate convex decomposition with 126 cells. (Source: Lien and Amato [21])

2.5 Symbolic Models in TAMP

The concept of symbolic models as outlined in [27] is an integral part of our proposed method. In this work, symbolic models are denoted as predicates. Essentially, they are symbolic representations of properties among objects in the environment. These predicates play a crucial role in TP, since they serve as the basis to define the preconditions and effects. Formally, each predicate is defined by an ordered list of object types and a binary classifier that evaluates to true or false given the current state and the specific objects involved [3]. In a typical TAMP problem, they are formulated as lifted predicates and lifted atoms, which can be grounded by specifying objects to obtain the ground predicates and ground atoms. These grounded predicates can be evaluated in a state to determine if a condition holds. A simple example of a lifted predicate would be ONSURFACE(?OBJECT, ?SURFACE) in a tabletop environment, which evaluates to true if the object is on the surface. This lifted predicate can be grounded using specific objects to obtain ground predicates like BLOCK as OBJECT and TABLE as SURFACE. The ground predicate ONSURFACE(BLOCK, TABLE) can then be evaluated to determine if the block is on the table. Tom silver et al. use predicates to generate samplers, which are essentially functions that generate the continuous parameters necessary to satisfy the preconditions. This form of coupling predicates and samplers allows the planner to refine abstract actions into concrete actions with continuous parameters that meet the required constraints and thus generate feasible motions.

[3]More details are specified later in section 4.1.1

2.6 Version Spaces Learning

As we use Version Space Learning (VSL) in our method, it is useful to outline its core idea. Version Space Learning is a logical learning method in binary classification, in which the learner maintains a set of hypotheses from the hypothesis space that are consistent with the training data. A hypothesis is considered *consistent* [22], if it correctly classifies all observed instances of the dataset, labeling positive instances as positive and negative instances as negative. The method operates by maintaining two sets:

1. S the **most specific** hypothesis set: This set contains hypotheses that are consistent with the training data while being as specific as possible.
2. G the **most general** hypothesis set: This set contains hypotheses that are consistent with the training data while being as general as possible.

Essentially, the difference between the two sets is that the hypotheses in S explain the positive instances as narrowly as possible, excluding any space that might contain negative instances. Both hypothesis sets serve as boundaries in the hypothesis space. Given a finite set of hypotheses and a training dataset, the method refines and narrows down the hypothesis space by removing the hypotheses inconsistent with the data. The goal is to refine S and G until the learner converges to the set of hypotheses that lie between the most specific boundary S and the most general boundary G, yet still remaining consistent with the training data, which is referred to as the *version space* [11].

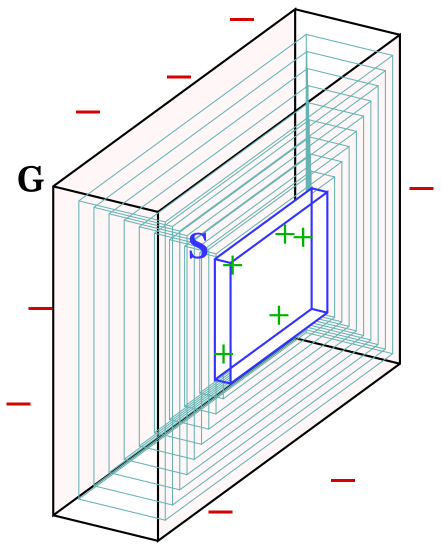

Figure 2.5: 3-D perspective of the version space. The green and red points represent positive and negative instances, respectively. The blue box represents the set S and the black box represents the set G. The version space is the area between S and G, that is made up of consistent hypotheses.

3 Related Work

Wells *et al.* [30] published a learning-based approach to feasibility checking for TAMP in tabletop environments. The goal, as they state, is to reduce the total number of motion planning attempts, which is a common problem in TAMP approaches. They train a motion feasibility SVM classifier using geometric features from different problem instances of the tabletop environment. This classifier is then integrated into the planning loop to classify high-level plan actions into feasible or infeasible ones. When the TAMP framework generates a high-level task plan, actions classified as infeasible are pruned from the plan. This pruning in the early phase of the planning process reduces the time used on motion planning attempts. Although the classifier can sometimes misclassify feasible actions as infeasible, the analysis in their paper shows that the approach is still robust.

Driess *et al.* [4] tackled a similar problem of reducing the high computational cost of infeasible motion planning attempts. They formulate the TAMP problem as a Mixed Integer Program (MIP), where the discrete variable represents the high-level decision, like which robot arm to use, and the continuous geometric part corresponds to a nonlinear trajectory optimization problem. More importantly, they mention their decision to decide the feasibility of the geometric problem in the sensor space rather than in the feature space, due to the fixed dimensionality of the sensor space. To predict the feasibility of MIPs, they feed a multi-channel image into a deep neural network along with the discrete action. The image consists of a depth image of the scene and object masks that highlight the target object. They claim that this setup allows for a generalizable approach across different environments. When the learned model predicts a given MIP as infeasible, the TAMP framework can skip that action and prioritize more promising alternatives, leading to a more efficient planning process.

Yang *et al.* [32] introduced a sequence-based feasibility checking approach that is aimed at long-horizon TAMP problems. Rather than performing feasibility checks on individual actions, they suggest an approach to evaluate the feasibility of an entire sequence of actions. This is a step towards addressing the challenge of sequential feasibility, where the feasibility of one action depends on the feasibility of the previous action. This could also be one of the drawbacks of other approaches to feasibility prediction. To implement this idea, Yang *et al.* introduce a plan feasibility predictor f that takes as input a task plan and other relevant information, such as the initial state and the goal state, and outputs a feasibility score. A plan is scored based on the likelihood that it can satisfy all the geometric constraints. Thus, plans with high scores are more likely to be feasible and are therefore more prioritized in the planning process.

4 Approach

In this chapter, we introduce the proposed approach to the challenges outlined in previous chapters. We start by formally defining the problem and the research questions that we want to address. After addressing the problem along with the research questions, we introduce the formal problem setting that we use in our TAMP framework. This problem setting is derived from Silver *et al.* [27], and is also used as a baseline in their TAMP framework, which is called Predicators. We then introduce the *feature grammar* that we use in our implementation to generate the hypotheses. Building upon this grammar, we present the learning framework based on Version Space Learning, which we will use in the TAMP framework. Finally, we describe implementation details, providing insights into how the approach is integrated into the Predicators [27] framework, which is used as a base for our work.

4.1 Problem Formulation

To precisely address the challenges outlined in the introduction, it is essential to have a clear definition of the problem we are tackling. Specifically, we need a solid problem formulation of the concept we are investigating and the research questions we are trying to answer.

This work is centered around the critical regions, as introduced in Section 2.3, we assume that critical regions are based on the feasibility of the different robot actions. That is, each action in a task has an associated region within the configuration space, where it can be executed successfully. We refer to these regions as *action-feasibility regions*.

By connecting the concept of critical regions to the feasibility of actions, we obtain a clearer understanding of the problem statement. The objective is to learn the regions in the robot's configuration space, where all actions are feasible. For a given task, typically represented as a sequence of actions, we aim to learn the feasibility regions corresponding to each action in that sequence. The aim is to learn the constraints under which each action is feasible. This definition allows us to decompose the problem into smaller subproblems, where we can focus on the learning at the action level.

Now that we have a clear definition of the problem, we can focus on the two main research questions addressed in this work:

1. What is a generalizable representation for action-feasibility regions in TAMP?
2. How can we efficiently learn these action-feasibility regions across different tasks and environments?

To tackle both challenges, we use a decomposition method based on object-centric geometric grammar, which will be discussed in detail later. Regarding the second question of learning these regions, we propose a method based on *version space learning*, which iteratively refines a set of hypotheses that we use to detect potential action feasibility regions in the robot's configuration space.

4.1.1 Problem Setting

The problem formulation in this work is inspired by Silver *et al.* [27] and we extend this setting for the direction of this work. The integrated Task and Motion Planning problems in this setting include a fully-observable and deterministic environment. The workspace $\mathcal{W}$ is static, and an *environment* is a tuple $\langle \Lambda, d, \mathcal{C}, f, \Psi \rangle$ with a distribution Ω over tasks, where each task $T \in \mathcal{T}$ is a tuple $\langle \mathcal{O}, x_0, g \rangle$. Λ is a finite set of object types and $d : \Lambda \rightarrow \mathbb{N}$ defines the dimensionality of the real-valued feature vector of each type. The set $\mathcal{C}$ denotes the set of low-level controllers or action templates, the agent can perform. Each controller $c \in \mathcal{C}$ is represented as a pair $c = ((\lambda_1, ..., \lambda_v), \theta)$, where $(\lambda_1, ..., \lambda_v) \in \Lambda$ are the discrete parameters that the controller operates on, and θ is a real-valued vector that specifies continuous aspects, such as poses or joint angles. For instance, in a tabletop environment, a controller could be a grasping action, where the discrete parameter could be the object to be grasped, and the continuous parameter could be the grasping pose.

Each task $T = \langle \mathcal{O}, x_0, g \rangle$ contains an object set $\mathcal{O}$, each with a type from Λ, the initial state $x_0 \in \mathcal{X}$ of the environment and a set of ground atoms g over predicates in Ψ and objects in $\mathcal{O}$ that specifies the desired end condition of the task. The state space $\mathcal{X}$ is induced by the object set $\mathcal{O}$. Each $x \in \mathcal{X}$ is a mapping from objects $o \in \mathcal{O}$ to a feature vector in $\mathbb{R}^{d(type(o))}$, which describes the features of all objects at the beginning. The object can have static features (e.g. length, width, etc.) and dynamic features (e.g. position in Euclidean space). Static features are fixed and don't change throughout the planning process, while dynamic features can change based on the actions performed on the objects by the robot. The robot is also considered an object in the environment, and its features represent its configuration space (e.g. the joint angles).

The action space $\mathcal{A}$ is induced by the controllers $\mathcal{C}$ and $\mathcal{O}$. Each action $a = ((o_1, ..., o_v), \theta) \in \mathcal{A}$ is a controller $c \in \mathcal{C}$ with θ specifying continuous parameters for the controller c and the objects $(o_1, ..., o_v)$ are assigned to the controller's discrete parameters, ensuring type compatibility. This formulation allows both discrete and continuous aspects of the action to be represented. $f : \mathcal{X} \times \mathcal{A} \rightarrow \mathcal{X}$ is a deterministic transition function that governs how actions cause transitions between states in the environment. In this case, the Configuration Space (C-space), the set of all possible configurations of the robot, is represented as a subset of $\mathcal{X}$.

To represent the abstract state of the environment, the set of predicates Ψ is used. A predicate ψ is defined by an ordered list of types $(\lambda_1, ..., \lambda_m)$ and a lifted binary classifier $c_\psi : \mathcal{X} \times \mathcal{O}^m \rightarrow \{true, flase\}$ that determines whether the predicate holds in a particular state, only when each object o_i in $c_\psi(x, (o_1, ..., o_m))$ has a type λ_i. This is an extension of the setting in Section 2.2, as predicates can represent the preconditions and effects of an action. We differentiate between lifted and ground atoms. A lifted atom is a predicate with typed variables (e.g., Free(?surface)). A ground atom $\underline{\psi}$ is a predicate with specific objects assigned to its variables (e.g., Free(Table1)). In the case of ground atoms, the state classifier $c_\psi : \mathcal{X} \times \rightarrow \{true, flase\}$ checks if the predicate holds in a particular state. A goal holds in a state x if, for all ground atoms $\underline{\psi} \in g$, the classifier c_ψ returns *true*.

It is important to examine the form of the solution to a task, as it not only defines the end condition of the task, but also plays a crucial role in the learning process. The form of the solution to a task T here is a plan τ_{tp}, a sequence of actions $\tau_{tp} = (a_1, ..., a_n)$. Starting from x_0, the transition function f should result in a final state x_n, and the goal g is satisfied, if all atoms hold in x_n.

To finalize the formalization of the TAMP framework, the problem setting is extended incorporating state abstraction and lifted operators. These elements establish a structured interface between symbolic reasoning on the task-level with the geometric reasoning on the low-level. This is a core aspect of the 'Predicators' framework from Silver *et al.* Thus, the planning is done in both stages, which they call *Bilevel Planning*. Abstractions are defined as mappings from the state space $\mathcal{X}$ and the action space $\mathcal{A}$ to discrete symbolic states. The notion of abstract state space $\mathcal{S}_\Psi$ is introduced, which is induced by the predicates in Ψ. We follow the following definition of the abstract state.

Definition 1 (**Abstract state**). *Silver* et al.
An abstract state $s_\psi \in \mathcal{S}_\Psi$ is the set of all ground atoms $\underline{\psi}$ under Ψ, that hold in x:

$$s_\psi = \textit{ABSTRACT}(x, \Psi) \triangleq \{\psi : c_\psi(x) = true, \forall \psi \in \Psi\}.$$

$$\begin{tikzcd}
 & \text{Abstraction} & \\
\mathcal{S}_\Psi \arrow[rr, "F"] & & \mathcal{S}_\Psi \\
 & \Omega \arrow[d, dash] & \\
 & \Sigma \arrow[d] & \\
\mathcal{X} \arrow[uuu, "\Psi"] \arrow[rr, "\mathcal{A}", "f"'] & & \mathcal{X} \arrow[uuu, "\Psi"'] \\
 & \text{Task} &
\end{tikzcd}$$

Figure 4.1: Abstraction in the TAMP framework (Source: Silver *et al.* [27])

This abstraction reduces the complexity of the planning problem, as it enables the planner to reason about discrete relationships between objects and the environment. To apply transitions between abstract states, the transition function $F : S_\Psi \times \Omega \rightarrow S_\Psi$, which uses the abstract action space Ω to transition between abstract states. Ω contains a set of lifted operators ω, which define transitions in S_Ψ.

Definition 2 (**Operator**). *Silver* et al.
An operator is a tuple $\omega = \langle PAR, PRE, EFF^+, EFF^-, CON \rangle$, where:

- *PAR is an ordered list of parameters, with types from Λ.*
- *PRE is a set of preconditions, a set of lifted atoms over Ψ and PAR.*
- *EFF^+ is a set of add effects, a set of lifted atoms over Ψ and PAR.*
- *EFF^- is a set of delete effects, a set of lifted atoms over Ψ and PAR.*
- *$CON = (\mathcal{C}, PAR_{CON})$ contains a low-level controller $\mathcal{C}$ and an ordered list of controller arguments PAR_{CON}, which are taken from PAR.*

These lifted operators can be grounded for a given task in the environment. More specifically, for a given task with object set $\mathcal{O}$, a ground operator $\underline{\omega}$ is a tuple $\langle \omega, \delta \rangle$, which comprises the lifted operator ω and a mapping $\delta : PAR \rightarrow \mathcal{O}$, which assigns objects from $\mathcal{O}$ to the parameters in PAR and $\underline{PRE}$, $\underline{EFF^+}$, $\underline{EFF^-}$ and $\underline{PAR_{CON}}$ are the grounded preconditions, add effects, delete effects and controller arguments, respectively. The controller $\mathcal{C}$ is important for the operator, as it bridges the symbolic and geometric reasoning. Since the grounding already determines the discrete parameters for the controller by the substitution δ, for example, which objects the robot should interact with, the only remaining task is to specify the continuous parameters for the controller, such that it satisfies the low-level constraints. Finding a suitable θ, more specifically suitable values for θ involves using a sampler, which generates candidate samples for the controller given the state of the environment. For instance, in the already mentioned stick-button environment [27], the sampler that generates candidate samples for the action *pickStick* would randomly sample along the long dimension of the stick (its length). A candidate continuous sample, in this case a candidate vector, is validated in $\mathcal{X}$ to check if the action is feasible. With this operator sampler combination, the grounding of an operator does not only defines the discrete objects for the interaction, but also sets the stage for continuous geometric checks. If the sampler does not find a suitable θ, it constantly samples for new values until it finds a suitable candidate sample θ. This is the main aspect, of the introduced framework, which Silver *et al.* [27] introduced in a series of papers [3, 17, 20, 26–28].

4.1.2 Learning Action Feasibility Regions

Based on our problem setting, the goal of our learning framework is to learn the regions within the state space for each action, where the action is feasible. For a given task $T \in \mathcal{T}$ with a state space $\mathcal{X}$ and a set of actions $\mathcal{A}$, each grounded action $a \in \mathcal{A}$ transitions the system from an abstract state $s \in \mathcal{S}$ to a state $s' \in \mathcal{S}$. We define the feasible region $\mathcal{X}^a_{critical}$ for an action a as the subset of $\mathcal{X}$, such that for each state x from this feasible region, there exists at least one assignment μ of continuous parameters under which action a is feasible.

$$\mathcal{X}^a_{critical} = \{x \in \mathcal{X} \mid \exists \mu \text{ such that } a(o_1, \ldots, o_v, \mu) \text{ is feasible}\}.$$

Essentially, $\mathcal{X}^a_{critical}$ characterizes the subset of the state space from which action a can be reliably executed. Identifying the feasibility region for each action is central to our goal of replacing blind uniform sampling in $\mathbb{R}^T$ with more efficient sampling inside the feasibility region, which can successfully lead to the feasibility of the action and how the feasibility of all actions a for task T can lead to the feasibility of the entire task.

4.2 Feature Grammar

The proposed method is based on feasibility feedback from the environment and the generation of hypotheses through a set of features. The approach can determine the essential circumstances in which an action is feasible by enumerating potential constraints that capture object geometries and relationships. In order to accomplish this, an expressive domain-representation is required that can be applied to different domains with varying object types. The concept of a feature grammar addresses this issue by generating and maintaining a set of derived geometric features, such as distances, poses, or key points, and enabling the combination of these features using arithmetic and logical operators. This grammar is central to our key idea of generating hypotheses for action feasibility regions.

To organize the features for our approach, we rely on three main categories:

- **Environment features** - describes each object's feature, which includes x, y, z positions, orientation θ and other features found in the environment, such as workspace dimensions and boundaries, robot radius, etc.
- **Geometric features** - describes the geometric relationships between objects, such as distances, angles, surface areas, etc.
- **Virtual objects** - objects created by translation or rotation of existing objects, which can be certain key points on the object.

Environment features The environment features capture the fundamental features of the environment, which are used in the planning process or generally define the environment itself. Concretely, these features cover:

- **general features (static)** - these features are used to set boundaries for the workspace, or set other constraints used in the environment, with some not being object-specific.

  - Workspace limits with e.g. rectangular bounds $(x_{min}, x_{max}, y_{min}, y_{max})$ for bounded environments[1]
  - bounds for specific regions inside the workspace, where the robot can operate and cannot leave
  - static features of certain objects like the radius of the robot, the radius of the gripper, robot arm length, etc.
- **Object features (dynamic)** - each object in the environment has a set of features, which are defined by the object type. In our framework, these are mainly dynamic features. They include:
  - Object pose, consisting of x, y or x, y, z coordinates, depending on the dimensionality of the environment, as well as the orientation θ of the object.
  - In our TAMP framework, certain objects also include features, that are high-level scalars, but are still used in the low-level geometry. For instance, the stick-button environment includes the object button, which has a feature `'pressed'`, a continuous values between 0 and 1, where a value bigger than 0.5 indicates that the button is pressed, or the object stick, which has a feature `'held'`, where again a continuous value bigger than 0.5 indicates that the stick is held by the robot.

These environment features form the first layer of environment information, upon which other features are built. They are extracted from the environment and remain available throughout the learning process. Each environment or domain that we handle in the predicators framework has a different set of objects and workspace information. This means that the entire list of features here is environment-specific. Thus, all the other feature groups are built on top of these features. Virtual features can be seen as extensions of these features, and the geometric features use environment objects and object features to build the geometric features or relationships. This also implies that the number of geometric and virtual features is dependent on the number of environment features. The more features we have, the more geometric and virtual features can be generated.

Geometric features Now that we have established the environment features, we continue with the geometric features, which describe the relationships between the environment objects and the workspace as well. The critical part here is to make sure that the geometric features that we define must be expressive enough for the learning process. This is important, as the hypothesis generation is exponential in the number of features. In our learning approach, we allow for the calculation of distance features between objects in the Euclidean space, which is defined as the norm of a difference. More specifically, we include the following **distance features** for two objects o_1 and o_2:

[1]This will be the case for all the domains that we consider in the predicators framework.

- $f_{dis^x}(o_1, o_2) =| o_1.x - o_2.x |$, $f_{dis^y}(o_1, o_2) =| o_1.y - o_2.y |$ and $f_{dis^z}(o_1, o_2) =| o_1.z - o_2.z |$ - 1D distance
- $f_{dist}(o_1.p, o_2.p) = ||o_1 - o_2||_2 = \sqrt{(o_1.x - o_2.x)^2 + (o_1.y - o_2.y)^2}$ - 2D distance with $o_1.p = (0_1.x, o_1.y)$ and $o_2.p = (o_2.x, o_2.y)$
- $f_{dist}(o_1.p, o_2.p) = ||o_1 - o_2||_3 = \sqrt{(o_1.x - o_2.x)^2 + (o_1.y - o_2.y)^2 + (o_1.z - o_2.z)^2}$ - 3D distance for $o_1.p = (o_1.x, o_1.y, o_1.z)$ and $o_2.p = (o_2.x, o_2.y, o_2.z)$

Virtual Objects In this work, we refer to virtual objects as key points that are derived from the environment objects. This can be important, as the learning approach can require reasoning about certain points of an object as well, which can result in a more detailed learning process. This is especially important in case the action feasibility of certain actions depends on where the object is interacted with. A typical example in the stick-button domain would be the tip of the stick, which can be helpful to know, as pressing the button with the stick requires the tip of the stick to be in contact with the button.

The most common transformation to create virtual objects is the translation of an object by an offset in the Euclidean space. Translating an object $o^1 = (o_1, ..., o_m)$ by a constant c would result in a new object $T(o^1) = (o_1, ..., o_m)$ with c being e.g. an environment feature represented as a constant. For better use of these translations, it is ideal to construct some key points. An example for an object o_i with width w and height h would be the following:

- $o_i^L = T_{x_L}(o_i) = \left(o_i.x - \frac{o_i.w}{2}\right)$ - point on the left boundary of the object
- $o_i^R = T_{x_R}(o_i) = \left(o_i.x + \frac{o_i.w}{2}\right)$ - point on the right boundary of the object
- $o_i^T = T_{z_T}(o_i) = (o_i.z + o_i.h)$ - point on the top boundary of the object
- $o_i^B = T_{z_B}(o_i) = (o_i.z - o_i.h)$ - point on the bottom boundary of the object

These points can be useful for contact detection between objects. For instance, a placement action might require the bottom boundary of a block to be in contact with the surface of another object or the table.

With environment features, geometric features and virtual objects now in place, the hypothesis generation can be done by combining these features. To perform this, we rely on the following chosen set of operators:

- $<, \leq, \geq, >$ - inequality symbols
- $+, -$ - arithmetic operators
- $\wedge, \vee, \neg$ logical operators

We also allow for the multiplication of features with the constants 0.25, 0.5, 0.75, 1.25 and 1.5 which can be used to scale the features. The hypotheses are generated as combinations of the features and the operators. Typical results e.g. in the stick-button domain are:

```
stick.x < robot.x - 0.5 * robot.w
$f_{dist^x}$(stick, button) < 0.5 * button.radius + robot.radius
robot.radius >= $f_dist$(robot, button)
```

Chaining the hypotheses with logical operators can result in more complex hypotheses, which can be more expressive.

```
button.y > robot.y + robot.w ∧ robot.x < button.x - 0.5 * button.w
```

Since the hypotheses are built exclusively from the features and the fixed operators, they are interpretable in the respective domain.

Constructing this object-centric feature grammar, comprising environment, geometric, and relational features, can help us express any candidate action-feasibility region as logical combinations over those features, which can be applied regardless of the domain. This approach directly answers our first research question and simultaneously provides us with a basis for the hypotheses generation in the learning process.

4.3 Learning Framework

The proposed learning framework builds upon the problem setting from Section 4.1.1 and the feature grammar from Section 4.2. The learned action-feasibility regions for a given environment would be integrated into the planner in the TAMP framework to improve the planning. Figure 4.2 illustrates the learning method in a simplified way, which is divided into multiple steps. In the paragraphs below, we step through each phase and explain the details of the learning process.

Initialization Given an environment and a set of training tasks $\mathcal{T}_{train}$ with $T \in \mathcal{T}_{train}$ with an initial task $T = \langle \mathcal{O}, x_0, g \rangle$, the learning algorithm starts by initializing the empty hypothesis space $\mathcal{H}$. At this initial stage, the learning algorithm makes the assumption that any sample of continuous parameters in any reference frame is provisionally assumed to succeed. Next, to introduce the learning algorithm to varying scenarios, we introduce a set of problem instances $\mathcal{P}$ for the learning. Each problem instance is essentially a copy of the same task T, but with different dynamic object features (object positions, orientations, etc.). By providing several such problem instances, we ensure that the hypotheses are tested and learned under diverse scenarios rather than sticking to a specific one. Finally, we provide two continuous manual samples for the action, a positive sample z^+, under which the action is feasible, and a negative sample z^-, under which the action is infeasible. We refer to them as manual samples, as we have total control over which values we specify, as these samples are used to initialize the learning process. Further details regarding the use of these manual samples will be revealed later.

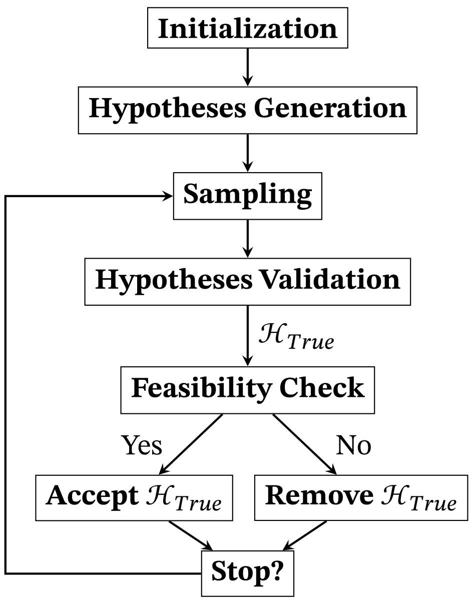

Figure 4.2: Learning method - simplified

Hypotheses Generation Once the initialization is complete, we turn to the hypotheses generation step. The goal of this step is to construct a set of candidate hypotheses $\mathcal{H}$, in which we enumerate possible combinations of the features in the entire environment, and the operators implied by the feature grammar 4.2. Concretely, all the generated hypotheses are of the form

$$h: \quad r_1 \circ r_2, \tag{4.1}$$

where r_1 and r_2 can be features from the environment (e.g. an object's x-coordinate, a 1D distance, a surface area) or even scaled features, and $\circ \in \{<, \leq, >, \geq\}$ is one of the four inequality symbols. By iterating over all feature pairs (r_1, r_2), all permitted scale factors that we define in Section 4.2 and every inequality symbol, we obtain the full candidate set

$$\mathcal{H} = \{h_1, h_2, ..., h_M\}. \tag{4.2}$$

One important point to address is the size of the generated hypothesis space. The number of generated hypotheses is, in principle, exponential in the number of objects and features. However, in typical TAMP domains, the number of objects and features is relatively small. Thus, M is manageable.

Sampling With the candidate hypotheses $\mathcal{H}$ generated, the next step is to sample the continuous parameters for the action. This sample will derive both the hypotheses validation and the feasibility check. Generally, the sample takes the form

$$z = (RF_o, \theta) \quad o \in \mathcal{O}, \theta \in \mathbb{R}^d, \tag{4.3}$$

where RF_o is the reference frame of the object o and θ is a vector of continuous parameters with dimensionality d for the action. In this case, the sampling is done in the reference frame of the object o that is being acted upon.

Hypotheses Validation At each learning iteration, the algorithm validates the hypotheses, which works as follows. Given the generated hypotheses $\mathcal{H}$, the algorithm iterates over each hypothesis $h \in \mathcal{H}$ and instantiates it with a sample z and a problem instance p_i. To be more precise, given a candidate hypothesis h, the algorithm substitutes the variables in h with the values from the sample z and the problem instance p_i. The algorithm checks if the hypothesis is satisfied, i.e. if the inequality holds. The validation for each hypothesis h is defined as

$$h(z) = \begin{cases} 1 & \text{if } h \text{ is satisfied on } z \\ 0 & \text{otherwise} \end{cases}. \tag{4.4}$$

To illustrate the hypothesis validation, consider the following example. Given the stick-button environment with one button, we get the candidate hypothesis

$$h := stick.x < robot.radius + robot.x \tag{4.5}$$

for the pick stick action. The algorithm gets the value of the robot radius, as well as the position of the stick from the environment as:

$$\text{robot.radius} = 1, \quad \text{stick.x} = 2.0. \tag{4.6}$$

The algorithm instantiates the hypothesis with these values and the sample
$z = (RF_{stick}, (0.3, 0.5))$, and the hypothesis is removed, as it is not satisfied:

```
stick.x < robot.radius + robot.x
2.0 < 1 + 0.3
2.0 < 1.3
```

Ultimately, this should result in two sets of hypotheses $\mathcal{H}_{True}$ and $\mathcal{H}_{False}$, where $\mathcal{H}_{True}$ is defined as

$$\mathcal{H}_{True} = \{h \in \mathcal{H} \mid h(z) = 1\}, \tag{4.7}$$

contains the hypotheses that are valid in p_i and $\mathcal{H}_{False} = \mathcal{H} \setminus \mathcal{H}_{True}$ contains the hypotheses that are not valid in p_i. The provisional set $\mathcal{H}_{True}$ could eventually form the feasibility region.

Feasibility Check Following the hypotheses validation, the algorithm checks the feasibility of the action using the same continuous sample z to confirm the results of the hypotheses validation. To accomplish this, the learning algorithm runs the planner, which simulates the low-level action and checks if both the add-effects and delete-effects of the corresponding grounded operator hold. If the simulation was successful and the effects hold, then all hypotheses h in $\mathcal{H}_{True}$ remain as valid. Conversely, if the simulation fails, then every hypothesis in $\mathcal{H}_{True}$ is removed, as they were too permissive and were satisfied by a sample that was not feasible.

Given the sample s and the feasibility outcome $\phi(z) \in \{0, 1\}$ of the simulation, the algorithm updates the sets as follows:

- If $\phi(z) = 1$ (the sample is feasible):
  1. All hypotheses in $\mathcal{H}_{True}$ remain valid and are initially marked as consistent for the problem instance p_i and are kept in $\mathcal{H}$ for the next iterations.
  2. All hypotheses in $\mathcal{H}_{False}$ are removed, as they did not hold for the feasible sample z during the validation.

$$\mathcal{H} = \mathcal{H} \setminus \mathcal{H}_{False}. \tag{4.8}$$

- If $\phi(z) = 0$ (the sample is infeasible):
  1. All hypotheses in $\mathcal{H}_{True}$ are removed, as they were too permissive and were satisfied by a sample that was not feasible.

$$\mathcal{H} = \mathcal{H} \setminus \mathcal{H}_{True}. \tag{4.9}$$

  2. There is no information yet on the validity of the hypotheses in $\mathcal{H}_{False}$. Thus, we keep them as well for the next iterations.

By using the same continuous sample for both the symbolic hypotheses validation and the low-level feasibility check, we ensure that the hypotheses do not only respect the symbolic feature constraints, but also capture the robot's low-level capabilities in the same problem instance. The surviving hypotheses can be seen as a set of constraints that define a region from which further parameters can be sampled for the active learning loop.

Manual iterations and Active Learning Loop Initially during learning, the second step of hypotheses generation would generate many hypotheses. We propose to limit the number of hypotheses to a manageable size, specified by the user. More importantly, by generating this many hypotheses, many hypotheses will be irrelevant for the learning process, as some would always be inconsistent and some would always be valid. To counter this, the negative manual sample z^- is initially used to filter out the irrelevant hypotheses. The algorithm applies the same learning step of hypotheses validation and feasibility check with the negative sample z^-. As we already know that the sample is infeasible, any hypothesis that is valid in this sample would not be beneficial for the learning process. Thus, after the hypotheses validation provides the set of hypotheses $\mathcal{H}_{True_{z-}}$, the algorithm removes all hypotheses h in $\mathcal{H}_{True_{z-}}$ after the failed action simulation, since they were satisfied by a sample that was not feasible. This initial pruning step is crucial to ensure that the learning process is not overwhelmed by irrelevant hypotheses and only focuses on hypotheses that could potentially lead to the true action feasibility region of an action. Following this step, $\mathcal{H}$ is updated to only include hypotheses from $\mathcal{H}_{False_{z-}}$, which are the hypotheses that were not satisfied by the negative sample z^- and are used for the next iterations. The algorithm proceeds with the positive manual sample z^+ to activate the learning loop. Applying the same steps as before, the algorithm generates the set of hypotheses $\mathcal{H}_{True_{z+}}$ and $\mathcal{H}_{False_{z+}}$, and then checks the feasibility of the action. The feasibility of the action would result in keeping all hypotheses in $\mathcal{H}_{True_{z+}}$ as initially consistent for the tested problem instance. It is important to mention that the first two manual iterations are only performed on one problem instance $p_0 \in \mathcal{P}$. Below is the pseudocode of the learning algorithm.

Algorithm 1 Action-feasibility learning

```
Require: problem instances $\mathcal{P}$, set of hypotheses $\mathcal{H}$, positive sample $z^+$,
         negative sample $z^-$, ground operator $\underline{\omega}$, maximum iterations $K$
                                                        1. Manual initialization
for all z in $\{z^-, z^+\}$ do
    $\mathcal{H}_{True} \leftarrow \emptyset$
    $\mathcal{H}_{False} \leftarrow \emptyset$
    $p_0 \leftarrow \mathcal{P}$
    for all hypothesis $h \in \mathcal{H}$ do
        if validate($h$, $z$, $p_0$) = True then
            $\mathcal{H}_{True} \leftarrow \mathcal{H}_{True} \cup \{h\}$
        else
            $\mathcal{H}_{False} \leftarrow \mathcal{H}_{False} \cup \{h\}$
    success ← simulate($\underline{\omega}$, $z$, $p_0$)
    $\mathcal{H}$ ←Update($\mathcal{H}_{True}$, $\mathcal{H}_{False}$, success, $z$)
                                                        2. Start Active learning loop
for all $k$ in $K$ do
    $z \leftarrow$ SampleFromRegion($\mathcal{H}$)
    $p \leftarrow$ SelectInstance($\mathcal{P}$)
    $\mathcal{H}_{True} \leftarrow \emptyset$
    for all $h \in \mathcal{H}$ do
        if validate($h$, $z$, $p$) = True then
            $\mathcal{H}_{True} \leftarrow \mathcal{H}_{True} \cup \{h\}$
        else
            $\mathcal{H}_{False} \leftarrow \mathcal{H}_{False} \cup \{h\}$
    success ← simulate($\underline{\omega}$, $z$, $p$)
    $\mathcal{H}$ ←Update($\mathcal{H}_{True}$, $\mathcal{H}_{False}$, success)
return $\mathcal{H}$
```

Note that the function Update on line 12 needs the continuous sample z as an input, since the update in the initial runs differs between the positive and negative samples, as explained above. The second part of the learning after line 14 starts active learning loop, which is described below in details. Here, the parameter K is used as a stopping criterion. In this case, we use the number of loop iterations. To start the active learning loop, the algorithm uses the set of hypotheses $\mathcal{H}$, that was updated in the manual run which would capture our initial region $\mathcal{X}^a_{critical} \subset \mathcal{X}$.

Within this initial region, we store the hypotheses, as inequalities of features over numerical values from the problem instance. This is important to make sure that our sampling algorithm in the active learning phase considers these values as bounds.

Once the initial region is specified, the learning algorithm can perform the active learning loop to further refine the hypotheses and finally converge to the target hypotheses. We provide the pseudocode of the active learning loop, which only differentiates from the initial learning step by the sampling of the continuous parameters.

Since we rely on an active approach, the algorithm will have control over the sampling through the region. It actively samples continuous parameter values from the region $\mathcal{X}^a_{critical}$ and then applies the same steps of hypotheses validation and action simulation as before. The sample value can be obtained by

$$\theta \sim Uniform(\mathcal{X}^a_{critical}), \tag{4.10}$$

where it uniformly samples from within the current region, which means it should resepct the current hypotheses in the region. Each active learning iteration would prune the hypotheses in the region until the algorithm reaches the stopping criterion K. During each iteration, we also perform a dominance check on the hypotheses. We introduce the following notion of dominance ($\preceq$). For hypotheses of the form:

$$h_1 : g \leq k_1, \quad h_2 : g \leq k_2, \tag{4.11}$$

where g is the same feature and k_1 and k_2 are real-valued constants, we say that hypothesis h_1 dominates hypothesis h_2 if the following condition holds:

$$h_1 \preceq h_2 \iff k_1 \leq k_2. \tag{4.12}$$

The final result of this learning process is a set of hypotheses, which represent the final action feasibility region for the action a in the environment. Ideally, as the active learning loop proceeds with the pruning of the hypotheses by the validation and simulation outcomes, the learning algorithm converges toward a minimal set of inequality constraints that characterize the true action-feasibility region. Mathematically, this means that at convergence, the set of hypotheses $\mathcal{H}$ will contain one or a very small number of hypotheses, such that they define the region

$$\mathcal{X}_{critical} = \{\theta \mid h(\theta) = 1 \text{ for all } h \in \mathcal{H}_{True}\}, \tag{4.13}$$

under which the action should always succeed.

5 Implementation

This chapter describes the implementation of the proposed learning framework and the integration into the planning process in the framework 'Predicators' [27]. Before we dive into the learning, it is helpful to recall how this framework works.

5.1 Predicators

This framework uses the problem setting Section 4.1.1 that we use in this work as a baseline and revolves around the concept of *Bilevel Planning*, which is a two-level planning process, that combines the high-level abstract planning with the low-level geometric planning.

Objects, Features and States The framework works with a set of objects, which are defined by their types. A type declares a fixed, finite list of real-valued features. For example, the stick-button environment has the following object types:

```
robot_type = Type("robot", ["x", "y", "theta"])
button_type = Type("button", ["x", "y", "pressed"])
stick_type = Type("stick", ["x", "y", "theta", "held"])
holder_type = Type("holder", ["x", "y", "theta"])
```

These types of features are identified by the current state of the object. At runtime a continuous state is a mapping of numerical values to the features of the object. A typical state in the stick-button environment would look like this:

```
State(data={button:button: array([5.14687554, 2.63375593, 0]),
robot:robot: array([5.65362596, 0.98865843, 1.57079633]),
stick:stick: array([2.09518428, 1.38631947, 1.57079633, 0]),
holder:holder: array([2.13268428, 1.81874084, 1.57079633])})
```

Planning On the symbolic level, the framework follows the general tradition of using symbolic models to represent atoms that can serve as preconditions and effects of an operator. Each predicate has its own classifier, which returns a boolean value indicating whether the predicate is satisfied or not by reading the current state of the environment. For example, the predicate `pressed` for a button has the classifier `pressed_holds`, which checks if the numerical value of the feature `pressed` is bigger than 0.5. These predicates are used to define the operators, or

NSRTs, as they are called, which stand for *Neuro-Symbolic Relational Transition Systems*. The continuous values are drawn from the NSRT's sampler, which uniformly samples normalized values in the reference frame of an object. NSRTs that do not need continuous parameters are assigned the so-called `null_samplers`, which do not provide any values. Each NSRT is defined by a set of preconditions, a set of add-effects, a set of delete-effects, and a set of parameters. Each NSRT is also assigned a low-level parameterized option, which is the low-level controller that defines how the robot interacts with the environment by specifying the exact motions the robot should take. Each option contains the following components:

- **Name** - the name of the option as an identifier used by NSRTs to reference the option
- **Parameter types** - the objects that the option can act
- **Parameter space** - the continuous parameters that the option can take
- **Policy** - A function that maps a state to an action
- **Termination condition** - a function that checks if the option has terminated

A complete planning process involves a planning phase and an execution phase. The planning phase is responsible for generating a plan by applying a search algorithm over the NSRTs to create a sequence of NSRTs that will lead to the goal of a task. For each NSRT, the corresponding option calls its policy to create a normalized action array for the robot, which contains the calculated movements. As an example, the action array for a press action that involves the stick looks like this:

```
    action = [dx, dy, theta, press]
```

where dx and dy are the normalized movements in the x and y directions, θ is the rotation of the stick, and `press` is a boolean value indicating whether the stick is pressed or not. Once this action array is created, the planner can simulate the action. The simulation function takes the action array and the current state of the environment as inputs and returns the next state after the execution is finished. If the simulation fails, new continuous parameters are sampled from the NSRT's sampler, and the simulation is repeated until the planner performs a successful simulation and the goal is reached.

5.2 Learning Algorithm

Having introduced the TAMP framework, that we work on, we now move to the implementation of the learning algorithm. Our learning algorithm is supposed to replace the samplers entirely by generating the continuous samples from the feasibility region instead. The implementation is split into three parts: feature extraction, hypotheses generation and learning.

We start by providing the algorithm with the environment or domain that it should work on. The algorithm first extracts features from this environment. As already explained in Section 4.2, the extracted environment features are divided into the object features as well as other general features stored as constants in the environment. We store all objects in the environment as object roles in a dictionary, and we create for each role one `ObjectAttributeFeature` for each attribute defined in the object's type description. General features, which are stored as constants in the environment, are wrapped in `ConstantFeature`. Interestingly, some object features, such as the radius of the robot or the radius of the button, are not defined in the object type, but as constants in the environment. At the end of this step, we should have all the features from the environment to create further features.

Once, we have our main features, we define the other features mentioned in Section 4.2, such as `DistanceFeature`, `ScaledFeature`, or `ArithmeticFeature`. Each of these features has a dedicated `evaluate` function. This simplifies the hypotheses validation, as we do not have to validate the full hypotheses, but we rather evaluate all the features independently in the hypotheses. Once we have all of these single features, we can generate the candidate hypotheses by using the inequality symbols. The general approach is to enumerate all possible combinations of these features and inequality symbols to get hypotheses that have a right and a left feature. We also extend both feature sides with logical operators to get more extended and expressive hypotheses. These hypotheses are stored in a list and are used in the learning process.

As explained in Chapter 4, we run the learning in two phases, a manual phase that filters hypotheses and introduces the learning algorithm to an initial region, and an active learning phase that refines the hypotheses until convergence. Some environments, such as the stick-button environment, have chained actions, such as `pickStickFromNothing`, which includes not only picking but also moving. The logic is that its parameterized option would move with a fixed speed until it reaches the position of the stick specified by the `pick_stick_sampler`, and then pick the stick. This approach simplifies the planning problem by allowing the option to handle the movement and free the symbolic planner from this complexity. However, this might lead to a problem when accounting the feasibility of multiple actions in a sequence. To handle this, we learn the regions for each action independently. We create a new task plan generator, which generates task plans consisting of one NSRT, whose preconditions hold in the initial state of the environment. This approach to task plan generation is ideal for our aim of learning feasibility regions for each NSRT independently. Once we get this task plan the learning can be performed on the candidate hypotheses and the generated problem instances.

The learning algorithm is implemented as specified in Algorithm 1 and **??**. During the hypotheses validation, we substitute the values of the objects from the given problem instance into the hypotheses. Whenever a feature boils down to a feature of one of the NSRT's parameter objects, we use the value of the sample instead of the value from the problem instance, guaranteeing that we truly get hypotheses that were valid for the given sample, which can then be confirmed after the feasibility check. After each learning step, the hypotheses are stored, such that the right side is the numerical value of the feature from the problem instance that was confirmed to be valid. This way, we always get hypotheses of the form:

```
stick.x < 0.75
dist_x_y >= 1.5
```

that simplify the sampling step. The sampling from such regions would have to satisfy the constraints of the region.

During each active learning iteration, we make sure to use the dominance check on the hypotheses to avoid storing many hypotheses over the same object feature. For example, if at any iteration, we get a valid hypothesis `stick.x < 0.75` from one tested problem instance and another valid `stick.x < 0.6` from another tested problem instance, we utilize a conservative dominance check that accepts the hypothesis with the smaller value and removes the other one.

Integration into the planner To integrate the learning algorithm into the framework, we create a new approach class `RegionLearningApproach`, which replaces the usual ”oracle” approach used in the framework. As soon as this approach is invoked on a new environment, it first executes the learning algorithm over a user-defined number of problem instances. It runs both manual and active learning stages and outputs a region consisting of the final consistent hypotheses across all provided problem instances. Once this region is learned, the planner uses the standard bilevel planner with the exception of using the learned region instead of the usual samplers. If a region was successfully learned for an NSRT, then the sample is uniformly drawn from the region, otherwise, the planner falls back to the NSRT’s original sampler.

This implementation ensures that the planning process is not interrupted and will always succeed. If, for any reason, the learned region is empty, we make sure not to sacrifice the completeness of the planning process and use the original sampler.

6 Evaluation

In this chapter, we evaluate our region-learning approach in the stick-button environment from Predicators [3]. We begin by reviewing the environment's structure and its sampling mechanism, then present our results. Finally, we compare our approach with the original sampling method and discuss the implications of our findings.

6.1 Stick-button-environment

The stick-button environment features a simple 2D tabletop scenario in which a robot arm is tasked with pressing existing buttons on the tabletop. There is a visible region where the robot can operate freely. A typical initial state places the robot at one end of the table, the stick in a holder directly in front of it inside the reachable region, and potentially multiple buttons in the workspace. For scenarios where the button is above this reachable region, the robot must pick the stick, move it to the button, and press it with the tip of the stick touching the button.

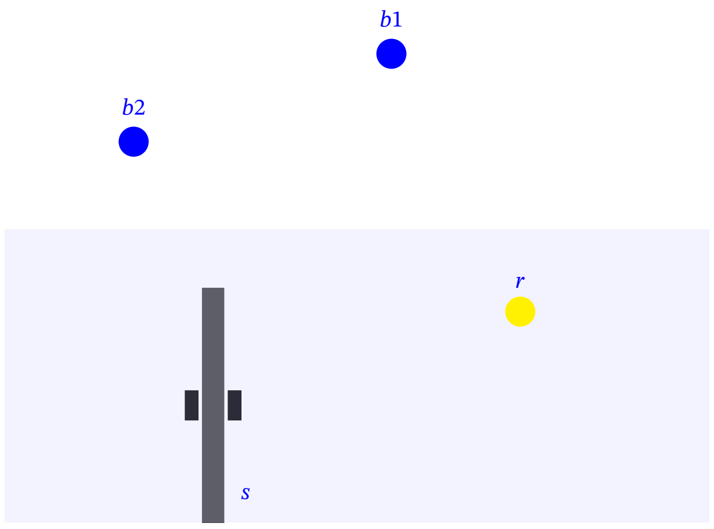

Figure 6.1: Stick-button environment

Since our learning algorithm relies on single NSRT learning by generating single-NSRT task plans, the only NSRT we learn is `PickStickFromNothing`, as its preconditions are satisfied in the initial state. The `pick_stick_sampler` generates a single value which, is specified as the normalized position along the long dimension of the stick in the center of the short dimension.

The long dimension here is x and the short dimension is y. This value is uniformly sampled from the range $[0,1]$. These restrictions to sampling are used to reduce the backtracking steps during planning, which means that this approach already defines a hand-crafted region, as the planner does not have to sample in the entire action space, but only focuses on sampling in the reference frame of the stick. The failure of the simulation can only be caused when the sampler returns a value that contacts the stick's holder, which is not a valid grasping point. This sampler implementation simplifies our testing, as we can validate our learned region against the hand-crafted sampler.

6.2 Learning Objectives

Before presenting the learning setup and results, it is helpful to exactly clarify what the learning algorithm should ideally discover in the stick-button environment and why. For the `PickStickFromNothing` NSRT, the only continuous decision lies in choosing the exact grasping point on the stick. Our goal is to learn a compact set of hypotheses that define a subset

$$\mathcal{X}_{critical} \subset [0,1], \tag{6.1}$$

such that any grasping point x in $\mathcal{X}_{critical}$ is a feasible point that avoids collision with the stick's holder.

At the end of the learning, we ideally expect to learn two hypotheses that define the maximum and minimum boundaries of the feasibility region, which correspond to the sets G and S, respectively. That is, the sample μ would be restricted by the following ideal hypotheses:

```
    h₁ := μ =< k₁
    h₂ := μ >= k₂
```

We showcase this in Figure 6.2, where the green area represents the feasible region for the action.

With a sufficient number of candidate hypotheses, the ideal region could even contain only one hypothesis that contains both boundaries.

```
    h := μ =< k₁ ∧ μ >= k₂
```

This would result in the following region:

$$\mathcal{X}^{pick}_{critical} = \{\mu \mid k_2 \leq \mu \leq k_1\}, \tag{6.2}$$

where k_1 and k_2 represent the most general hypothesis space G and the most specific hypothesis space S, respectively in the context of Version Space Learning [11].

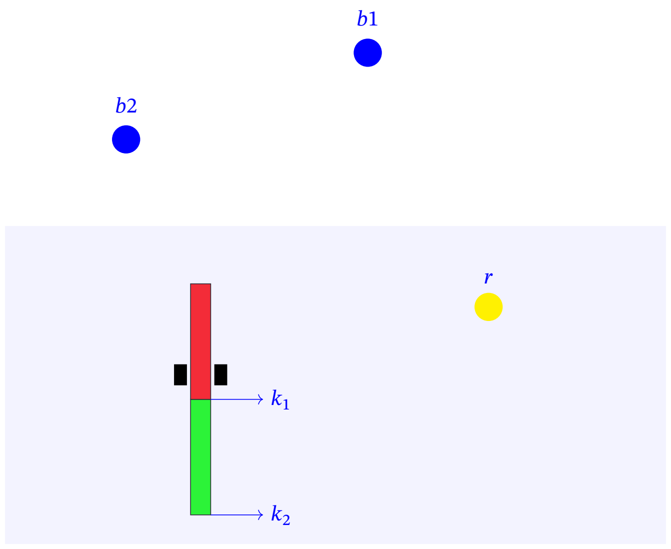

Figure 6.2: feasibility region (green) for `PickStickFromNothing`

Our goal is to maximize the size of $\mathcal{X}^{pick}_{critical}$ to include as many feasible grasping points as possible. Once learned, sampling

$$\mu \sim \text{Uniform}\left(\mathcal{X}^{pick}_{critical}\right), \tag{6.3}$$

will guarantee the feasibility of pick action.

6.3 Experimental Setup

To train our region-learning algorithm, we use four problem instances and generate 20 000 candidate hypotheses, covering combinations of the object's features, as well as important environment constants (`robot_radius`, `stick_width`, `stick_length`, etc.) We employ a positive sample $z^- = 0.95$ to prune out unnecessary hypotheses, followed by a positive manual sample $z^+ = 0.5$ to start the initial learning process. Following the two manual iterations, the active learning loop is invoked, that runs for 100 iterations. Once the learning is complete, the implemented `region_learning_approach` runs the planner on a new problem instance with our feasibility region sampler.

6.4 Results

Applying the learning algorithm with the described setup from Section 6.3 yields interesting results. The initial manual runs do manage to get an initial region for the active learning. Most notable is the number of hypotheses removed after these two runs. Following the pruning of the set of candidate hypotheses as well as the update mechanism of the positive manual sample run, roughly 50% of the candidate hypotheses are removed. This is a good sign, as it shows that the algorithm is able to learn from the manual samples and prune out unnecessary hypotheses before any active learning occurs. Once the active learning loop is invoked, further pruning occurs in response to each drawn sample and failed feasibility check. After 100 iterations, the algorithm converges to the following hypotheses:

```
    robot.x ≤ 0.5
    robot.y ≤ 0.4704908466265516
    robot.theta ≤ 0.5
```

These hypotheses define a three-dimensional bound in the robot's action space. However, only the first hypothesis (robot.x $\leq$ 0.5) is relevant in our stick-button scenario, as the sampling for the `PickStickFromNothing` NSRT only occurs along the x-dimension. Thus, the learning algorithm updates the parameter space to include this region.

With this learned region, which contains the three mentioned hypotheses, we can compare the planning performance over 50 new test instances using the original Predicators sampler and our learned region-sampler. The framework already provides a set of metrics for evaluation, and the metrics are computed after a complete test run. The following Table 6.1 summarizes the metrics we used for evaluation.

Table 6.1: Performance analysis between the original sampler and the learned region sampler.

Metric	**Original Sampler**	**Learned Region Sampler**
Successes	50/50	50/50
Average success time (s)	0.00694	0.00726
Minimum samples in any run	1.0	1.0
Maximum samples in any run	230.0	17.0
Average sample count	7.76	2.64

The results show that both approaches complete all tasks successfully. This stability is to be expected, as the planning approach in this framework continues to sample until a feasible point is found. However, the learned region sampler significantly reduces the number of samples needed to find a feasible point. The average sample count for the original sampler is 7.76, while the learned region sampler only requires 2.64 samples on average, which is a 66% reduction. The maximum sample count for the original sampler is 230, while the learned region sampler only requires 17 samples at most. These are huge improvements in terms of efficiency on paper, as the learned region sampler is able to find feasible points much faster than the original sampler. However, as we can observe from the table, the average success time is about the same for both approaches. This is due to the fact that most of the difference in planning time occurs during the task plan generation, where a search algorithm is used to find a feasible sequence of NSRTs for the task skeleton. The sampling of the continuous parameters is only a small part of the overall planning time.

It is important to note that, since our learning approach relies on active learning and random uniform sampling, the exact set of learned hypotheses, and hence the learned region, may vary between runs. In a number of trials, the learning algorithm may converge to a set of hypotheses, that does not yield a reliable feasibility region. In these cases, the metrics remain unchanged from the approach with the original sampler. There are further cases where we get regions that include unnecessary hypotheses, that were not pruned out. However, in these cases, the sampling efficiency is still improved and is comparable to our best results in Table 6.1.

To summarize, although the learning algorithm can, in some cases, produce a feasibility region, which increases the sampling efficiency of the planner, there are still threats to validity due to the randomness of the active learning. However, even when we do not obtain the ideal region, the planning never performs worse than the original sampler. We leave the exploration of the learning algorithm's robustness to future work.

7 Conclusion

7.1 Summary

In this thesis, we tackled the challenge of learning critical regions in the context of TAMP. By redefining critical regions in the configuration space as action-feasibility regions, we proposed a generalizable learning framework that combines high-level symbolic models with version space learning. These action-feasibility regions are compact sets of continuous action parameters, under which a given action can be executed successfully, respecting both kinematic and collision constraints.

The key component to our learning algorithm, as well as to a generalizable definition of the action-feasibility regions, is the concept of feature grammar. We introduced an object-centric feature grammar comprising environment, geometric, and relational features. Generalization was a crucial aspect to ensure the proper integration and testing of our approach into different environments. Building on Version Space Learning, we implemented our learning framework that works in two phases. An initial manual phase that uses user-provided positive and negative samples to prune the set of generated candidate hypotheses and provide a proper initialization for the second phase. The second phase is an iterative active learning process, where the candidate hypotheses are validated on a problem instance and a given continuous sample from the provisional feasibility region of the first phase, and then confirmed or rejected based on the feasibility of the action simulation. We demonstrated our approach in the stick-button domain by learning the feasibility region for picking along a stick. Across 50 test instances, our learned region sampler managed to reduce the average sampling attempts by 66% in some cases compared to the standard sampler approach without sacrificing planning time. The reason, a reduction of 100% in backtracking attempts was not achieved, is most likely due to two reasons: (1) the sampled point is in collision with the holder, which leads to an infeasible pick action, and (2) the feasible pick position is not feasible for the later actions (e.g. pressing the button). However, our results highlight the potential efficiency benefits of learning explicit action-feasibility regions in TAMP, yielding a more efficient planning process with less backtracking. We believe this learning framework offers a promising direction for future research in TAMP.

7.2 Future Work

During the implementation and evaluation of our learning approach, we encountered several challenges and limitations that could be addressed in future work. We provide a list of potential future work directions that could enhance the performance and applicability of our approach.

The current feasibility regions that can be learned are limited to the implementation of the environments and samplers in Predicators. For instance, the `pick_stick_sampler` already defines a region where the sampling only occurs along the stick's longer dimension, which already enodes bias in sampling, and this makes the tested domain less ideal for learning. Our learning algorithm only manages to learn a subset of this region, as shown in Section 6.4. Although our feasibility region did improve the sampling attempts, backtracking was still needed in some test instances, possibly due to the sampling on the holder's position. To avoid this, a detailed exploration of the object features with a more complex grammar in the training problem instances could be beneficial. For geometric features, such as the distance between two objects, the learning algorithm can learn relative distances with respect to different reference frames. For example, during the active learning phase for the pick action, the algorithm can learn the distance of the sampled point to the holder's position in the reference frame of the holder or the stick. This would require a more complex feature grammar, where we introduce distances over relative constants (e.g. from 0.1 to 0.5). Learning these relative distances could help to learn hypotheses that provide us with a bigger region or perhaps even two regions for sampling.

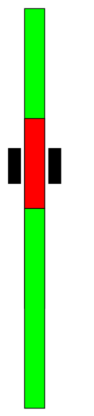
Figure 7.1: ideal feasibility region (green) for `PickStickFromNothing`

If we solely focus on the feasibility of just the pick action, a better feasibility region would be the green region in Figure 7.1. This would be the ideal feasibility region for the pick action, where it is defined by a single hypothesis, that indicates that the distance between the holder and the sample in the reference of the holder should be greater than 0.1. Thus, we extend our previously learned feasibility region, such that we can pick the action above the holder as well.

Sequential Feasibility Since we currently only learn the feasibility of individual actions, an interesting extension would be to use these initial regions to learn the feasibility of a sequence of actions to account for long-horizon tasks. Thus, the learning algorithm would carry forward the feasibility region for step i into the learning of step $i + 1$. The initially learned region can be used to sample continuous parameters for the next action. Iterating this process should theoretically lead to a chain of feasibility regions, where the intersection could precisely yield a backtracking-free planning process. In our stick-button environment, we can also learn the feasibility of the NSRT `RobotPressButtonWithStick`. To do so, we initially learn the region for the pick action and then use this region to sample continuous action parameters for the place action with the goal of learning the intersection of both NSRT's regions. The intersection of both regions would result in a new region, which ensures that the initial sample for picking would lead to a valid pressing action as well, thus leading to a successful task completion without any need for backtracking.

List of Acronyms

ACPRM	Avoidance Critical Probabilistic Roadmap
C-space	Configuration Space
DoF	Degree of Freedom
MIP	Mixed Integer Program
MMMP	Multimodal Motion Planning
MP	Motion Planning
PDDL	Planning Domain Definition Language
PRM	Probabilistic Roadmap
SBMP	Sampling-Based Motion Planning
STRIPS	Stanford Research Institute Problem Solver
SVM	Support Vector Machine
TAMP	Task and Motion Planning
TP	Task Planning
VSL	Version Space Learning
WF	World Frame

List of Symbols

General

$\mathcal{R}$	robot
$\langle \cdot, \cdot, \cdot \rangle$	tuple of multiple variables
$R(\theta)$	rotation by angle θ
T	transofmration matrix
RF_o	reference frame of object o

Task and Motion Planning

$\mathcal{W}$	workspace
$\mathcal{W}_{free}$	free workspace
$\mathcal{W}_{obs}$	workspace obstacle region
$C, q \in C$	configuration space
C_{free}	free configuration space
C_{obs}	obstacle configuration space
$\mathcal{R}(q)$	set of points occupied by robot $\mathcal{R}$ at configuration q
$\mathcal{S}, s \in \mathcal{S}$	set of all states
$\mathcal{A}, a \in \mathcal{A}$	set of all actions
τ_{tp}	task plan
τ_{mp}	collision-free trajectory
$\Sigma, \sigma \in \Sigma$	set of mode families
$\mathcal{F}_\sigma \subset C$	feasible space of mode σ
$\mathcal{X}^a_{critical}$	critical region for action a

TAMP framework

$\mathcal{T}, T \in \mathcal{T}$	set of tasks
$\mathcal{O}, o \in \mathcal{O}$	set of objets
$\Lambda, \lambda \in \Lambda$	set of object types
$d : \Lambda \to \mathbb{N}$	dimensionality of the real-valued feature vector of each object type
$\mathcal{C}, c \in \mathcal{C}$	set of controllers
θ	real-valued vector for a controller c
$\mathcal{X}, x \in \mathcal{X}$	continuous state space
f	deterministic transition function
$\Psi, \psi \in \Psi$	set of predicates
c_ψ	binary classifier for predicate ψ

$\mathcal{S}_\psi, s_\psi \in \mathcal{S}_\psi$	abstract state space
F	transition function between abstract states
ω	operator
$\underline{\omega} = (\omega, \delta)$	grounded operator with mapping δ of operator parameters to objects
$\mathcal{P}, p \in \mathcal{P}$	set of problem instances

Version Space Learning

$\mathcal{H}, h \in \mathcal{H}$	hypothesis set
S	most specific hypothesis set
G	most general hypothesis set
H_{True}	set of valid hypotheses
H_{False}	set of invalid hypotheses

List of Figures

List of References

[1] Z. Ajanović, E. Regolin, B. Shyrokau, H. Ćatić, M. Horn, and A. Ferrara, “Search-based task and motion planning for hybrid systems: Agile autonomous vehicles,” *Engineering Applications of Artificial Intelligence*, vol. 121, p. 105 893, May 1, 2023, ISSN: 0952-1976. DOI: `10.1016/j.engappai.2023.105893`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0952197623000775` (visited on 02/11/2023).

[2] F. F. Arias, B. Ichter, A. Faust, and N. M. Amato, “Avoidance critical probabilistic roadmaps for motion planning in dynamic environments,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021, pp. 10 264–10 270.

[3] R. Chitnis, T. Silver, J. B. Tenenbaum, T. Lozano-Perez, and L. P. Kaelbling, “Learning neuro-symbolic relational transition models for bilevel planning,” in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022, pp. 4166–4173.

[4] D. Driess, O. Oguz, J.-S. Ha, and M. Toussaint, “Deep visual heuristics: Learning feasibility of mixed-integer programs for manipulation planning,” in *2020 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2020, pp. 9563–9569.

[5] R. E. Fikes and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[6] M. Filipiak, “Mesh generation,” *Edinburgh parallel computing centre, the University of Edinburgh, Edinburgh*, 1996.

[7] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, “Integrated task and motion planning,” *Annual review of control, robotics, and autonomous systems*, vol. 4, no. 1, pp. 265–293, 2021.

[8] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, “Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning,” in *Proceedings of the international conference on automated planning and scheduling*, vol. 30, 2020, pp. 440–448.

[9] K. Hauser and J.-C. Latombe, “Multi-modal motion planning in non-expansive spaces,” *The International Journal of Robotics Research*, vol. 29, no. 7, pp. 897–915, 2010.

[10] K. Hauser and V. Ng-Thow-Hing, “Randomized multi-modal motion planning for a humanoid robot manipulation task,” *The International Journal of Robotics Research*, vol. 30, no. 6, pp. 678–698, 2011.

[11] H. Hirsh, “Generalizing version spaces,” *Machine learning*, vol. 17, pp. 5–46, 1994.

[12] D. Hsu, J.-C. Latombe, and R. Motwani, “Path planning in expansive configuration spaces,” in *Proceedings of international conference on robotics and automation*, IEEE, vol. 3, 1997, pp. 2719–2726.

[13] B. Ichter, E. Schmerling, T.-W. E. Lee, and A. Faust, “Learned critical probabilistic roadmaps for robotic motion planning,” in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 9535–9541.

[14] R. N. Jazar, *Theory of applied robotics*. Springer, 2010.

[15] S. Katz and A. Tal, “Hierarchical mesh decomposition using fuzzy clustering and cuts,” *ACM transactions on graphics (TOG)*, vol. 22, no. 3, pp. 954–961, 2003.

[16] Z. Kingston and L. E. Kavraki, “Scaling multimodal planning: Using experience and informing discrete search,” *IEEE Transactions on Robotics*, vol. 39, no. 1, pp. 128–146, 2022.

[17] N. Kumar, W. McClinton, R. Chitnis, T. Silver, T. Lozano-Pérez, and L. P. Kaelbling, “Learning efficient abstract planning models that choose what to predict,” in *Conference on Robot Learning*, PMLR, 2023, pp. 2070–2095.

[18] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[19] S. M. LaValle, “Motion planning: Wild frontiers,” *IEEE Robotics Automation Magazine*, vol. 18, no. 2, pp. 108–118, 2011.

[20] A. Li and T. Silver, “Embodied active learning of relational state abstractions for bilevel planning,” *arXiv preprint arXiv:2303.04912*, 2023.

[21] J.-M. Lien and N. M. Amato, “Approximate convex decomposition of polygons,” in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 17–26.

[22] T. M. Mitchell, “Generalization as search,” *Artificial intelligence*, vol. 18, no. 2, pp. 203–226, 1982.

[23] E. Plaku, L. E. Kavraki, and M. Y. Vardi, “Motion planning with dynamics by a synergistic combination of layers of planning,” *IEEE Transactions on Robotics*, vol. 26, no. 3, pp. 469–482, 2010.

[24] N. Shah and S. Srivastava, “Using deep learning to bootstrap abstractions for hierarchical robot planning,” *arXiv preprint arXiv:2202.00907*, 2022.

[25] B. Siciliano, O. Khatib, and T. Kröger, *Springer handbook of robotics*. Springer, 2008, vol. 200.

[26] T. Silver, A. Athalye, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, “Learning neuro-symbolic skills for bilevel planning,” *arXiv preprint arXiv:2206.10680*, 2022.

[27] T. Silver, R. Chitnis, N. Kumar, W. McClinton, T. Lozano-Pérez, L. Kaelbling, and J. B. Tenenbaum, “Predicate invention for bilevel planning,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 12 120–12 129.

[28] T. Silver, R. Chitnis, J. Tenenbaum, L. P. Kaelbling, and T. Lozano-Pérez, “Learning symbolic operators for task and motion planning,” in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 3182–3189.

[29] J. P. Van den Berg and M. H. Overmars, “Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners,” *The International Journal of Robotics Research*, vol. 24, no. 12, pp. 1055–1071, 2005.

[30] A. M. Wells, N. T. Dantam, A. Shrivastava, and L. E. Kavraki, “Learning feasibility for task and motion planning in tabletop environments,” *IEEE robotics and automation letters*, vol. 4, no. 2, pp. 1255–1262, 2019.

[31] G. Wilfong, “Motion planning in the presence of movable obstacles,” in *Proceedings of the fourth annual symposium on Computational geometry*, 1988, pp. 279–288.

[32] Z. Yang, C. R. Garrett, T. Lozano-Pérez, L. Kaelbling, and D. Fox, “Sequence-based plan feasibility prediction for efficient task and motion planning,” *arXiv preprint arXiv:2211.01576*, 2022.