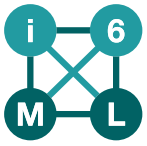


Chair of
Computer Science 6
(Machine Learning
and Reasoning)

RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair of Machine Learning and Reasoning.

From pixels to plans: Converting image observations of grid domains into plans

Bachelor Thesis

Presented by

Louis Maiworm
434159

Supervised by Jonas Reiter, M.Sc.

1st Examiner Univ.-Prof. Ph. D. Hector Geffner

2nd Examiner Univ.-Prof. Dr. rer. nat. Christopher Morris

Aachen, January 12, 2026

With sincere thanks to my family for their constant encouragement
and support throughout my entire studies.

Abstract

Vision-to-planning is a challenging problem, as it requires bridging raw perceptual input and symbolic planning without relying on handcrafted intermediate representations. This thesis investigates an object-centric vision-to-planning pipeline that induces symbolic planning models directly from visual observations in grid-based environments. The approach combines object-centric visual abstraction with the novel algorithms SYNTH and SIFT, using SYNTH to recover implicit action arguments and SIFT for predicate induction and domain learning, producing lifted action schemas and a complete PDDL representation.

The pipeline was evaluated on the Blocksworld and Delivery domains. While it robustly induces lifted action schemas and complete domains in Delivery, the complexity of Blocksworld prevented SIFT from fully inducing a complete domain within practical constraints. The results highlight both the potential and current limitations of object-centric vision-to-planning approaches, particularly regarding generalization and the completeness of induced symbolic representations.

Contents

1 Introduction

Reasoning and planning are central challenges in Artificial Intelligence (AI). Autonomous agents are expected to perceive their environment, understand the dynamics of objects, and generate action sequences that transform an initial state into a desired goal state [8]. Classical planning methods achieve this by relying on explicit, human-designed models, typically expressed in the Planning Domain Definition Language (PDDL) [6]. However, constructing such models manually is labor-intensive, requires domain-specific knowledge, and does not scale to previously unseen or dynamic environments.

A natural question arises: can an agent learn planning domains directly from raw perceptual data, such as sequences of images, without relying on annotated object labels or predefined action schemas? This question is at the core of this thesis. The ability to extract structured symbolic information from raw visual traces would allow agents to plan in unknown domains, opening the door to fully autonomous learning and reasoning.

Existing approaches have addressed parts of this problem but remain limited. Classical planners require fully specified PDDL domains and cannot operate directly on images. Methods such as ROSAME [21] demonstrate that symbolic planning models can be induced from visual traces using object-centric representations. However, prior approaches often rely on partial supervision or limited feature induction, which constrains generalization. To date, no method has demonstrated the fully automated induction of lifted, symbolic Stanford Research Institute Problem Solver (STRIPS) domains directly from raw visual traces.

This thesis proposes a novel, object-centric vision-to-planning pipeline that combines the SYNTH [12] and SIFT [9] algorithms. SYNTH infers implicit action arguments from object traces, while SIFT induces symbolic features and learns actions along with their preconditions and effects. Together, these modules enable the automatic construction of PDDL domains and instances from sequences of images annotated only with high-level action labels. By leveraging object-centric representations, the pipeline generalizes across instances and domains while providing interpretable symbolic models suitable for classical planning. We evaluate the pipeline on the Blocksworld and Delivery domains, showing its ability to learn planning domains, generate valid plans, and analyze the challenges encountered. A limitation of the current approach is that it operates on grid-like environments, which constrains its applicability to more general spatial settings.

Developing such a pipeline presents several challenges. Objects must be accurately extracted and consistently tracked across visual sequences and the system must correctly associate

objects with their actions to induce meaningful symbolic representations. Integrating these components into a coherent pipeline capable of producing valid plans further increases the complexity. Systematically studying these difficulties forms the core research question of this work: *Which challenges arise when learning symbolic planning domains from raw visual traces, and how can they be addressed?*

The remainder of this thesis is structured as follows. Chapter 2 presents the necessary background on the relevant computer vision techniques and symbolic planning with an introduction to SYNTH and SIFT. Chapter 3 reviews related work in vision-based planning and automated domain learning. Chapter 4 describes the design and implementation of the proposed pipeline. Chapter 5 presents experimental results and analyzes the challenges observed in the Blocksworld and Delivery domains. Finally, Chapter 6 concludes with a summary of the main findings and potential directions for future work.

2 Background

For our pipeline, certain background concepts are required. First, we need a method to process and extract information from image observations. Next, we must introduce the concept of planning, along with the algorithm SIFT, which plays a role in connecting the visual data to the symbolic representations used in the planning process.

2.1 ResNet

ResNet (Residual Network) is a deep Convolutional Neural Network (CNN) architecture [10]. It solves the problem of training very deep networks by using skip connections that let layers learn residual functions instead of direct mappings. This helps prevent the vanishing gradient problem and enables training of much deeper models, improving performance in tasks like image classification. ResNet uses a stack of convolutional layers, batch normalization, and Rectified Linear Unit (ReLU) activations, making it a standard architecture in computer vision.

2.2 Guided Backpropagation

To achieve this, we replace all ReLU activations in the network with a modified version called Guided ReLU [19]. In the forward pass, the network behaves as usual. The activation function is the standard ReLU :

$$f_i^{l+1} = \text{ReLU}\big(f_i^l\big) = \max\big(f_i^l, 0\big)$$

Here, f_i^l denotes the activation of the i-th neuron in layer l, and f_i^{l+1} is the corresponding output in the next layer.

For a selected network output, we compute the gradient of the corresponding activation with respect to the input tensor. This output may correspond to a class score or, more generally, to a high-level network activation. The gradients are then propagated backward through the network using a specific rule.

Standard backpropagation applies the following rule:

$$R_i^l = \big(f_i^l > 0\big) \cdot R_i^{l+1}, \quad \text{where } R_i^{l+1} = \frac{\partial f^{\text{out}}}{\partial f_i^{l+1}}$$

- R_i^l is the relevance or gradient signal received by the i-th neuron at layer l during the backward pass.

- f^{out} denotes the selected network output used to compute the gradient.

The 'Deconvnet' backward pass filters negative gradients [22]:

$$R_i^l = \left(R_i^{l+1} > 0\right) \cdot R_i^{l+1}$$

Guided backpropagation combines both constraints [19]:

$$R_i^l = (ReLU > 0) \cdot \left(R_i^{l+1} > 0\right) \cdot R_i^{l+1}$$

This formulation ensures that only positive contributions, those where both the activation and the gradient are positive, are allowed to pass through.

The result is a saliency map that highlights image regions with strong positive influence on the selected network activation. Since the gradient has three channels (RGB), it is reduced to a single-channel saliency map by taking the maximum value across all channels for each pixel. The final output is an image $I(x, y) \in [0, 255]$, where each pixel value indicates the relative visual relevance of that region.

2.3 Planning

We aim to formalize a representation that allows us to reliably solve complex tasks. One common approach in Artificial Intelligence is planning. For this purpose, we adopt a widely used planning model called the STRIPS formalism [8], defined as:

$$P = \langle F, I, O, G \rangle$$

where

- F - set of all grounded atoms (boolean variables)
- I - initial state of the problem
- O - actions with preconditions, add, and delete effects
- G - goal description

In the lifted representation, atoms and actions are described with variables rather than concrete objects. For example, the lifted atom:

```
at(?x)
```

expresses that "some object ?x is at a certain location", without specifying which object. This makes the domain description general and reusable and it can be applied to any number of objects by later substituting variables with constants.

In contrast, the grounded representation instantiates all variables with specific objects from the problem instance. If the domain contains the objects `A` and `B`, the grounded version of `at(?x)` would be:

```
at(A),  at(B)
```

A domain is defined by the lifted representation of predicates F and operators O. It provides a general model of the world, describing the types of objects, possible actions, and how these actions affect the state. A problem is a concrete instance of this domain. It specifies the actual objects present, the initial state I, and the goal state G. Grounded predicates for the objects in the initial state form the `(:init)` section of the PDDL problem [17], while the goal state defines the `(:goal)` section.

2.4 SYNTH

SYNTH [12] introduces an extension of the classical STRIPS formalism, referred to as *STRIPS*$^+$. This variant relaxes the requirement that all action parameters must be explicitly declared. Instead, action preconditions are formulated in a restricted first-order logic fragment that allows only conjunctive predicates and enforces structural constraints to ensure unique object bindings. Such objects are implicitly determined by the state via the structure of the precondition, enabling more abstract action descriptions. Within this formalism, the goal of SYNTH is to infer lifted *STRIPS*$^+$ action models from partially observed state-action-state transitions (s,a,s'). Each transition captures the execution of an action a in a state s and the resulting successor state s'.

In this representation, the precondition of an action is expressed as a conjunctive query Q(x,y,z), where the variables are partitioned into three disjoint sets: x contains the explicit arguments, y introduces additional free variables, and z comprises implicit variables whose values are uniquely determined by x. For efficient computation, queries are constrained to be simple, with each y occurring at most once, and stratified, meaning they can be decomposed into layers $Q_1, \dots, Q_n$ such that each z_i appears only in its corresponding layer and no alternative prefix assignment produces conflicting bindings.

The learning procedure consists of three stages. First, SYNTH identifies implicit action arguments by constructing a binding precondition Q(x,y,z) that deterministically selects the values of z from the observed traces. This construction proceeds incrementally, decomposing Q into conjunctive subqueries $Q_1(x, y, z_1), \dots, Q_n(x, y, z_n)$, where each Q_i determines a single implicit variable z_i as a conjunction of lifted atoms $q_{i,j}(x, y, z_i)$. The index i orders the implicit variables according to the sequence in which they are determined, while j enumerates the atoms within a subquery. A lifted atom may reference the explicit arguments y, free variables y, and the current z_i, but may not reference any future implicit variable z_k with $k > i$. Atoms are

added incrementally and only if the resulting query satisfies all three binding precondition criteria. First, validity requires that the query remains satisfiable for every ground action *a(o)* that occurs in the trace. Second, stratification enforces that the query adheres to the stratified form defined above. Third, maximality ensures that all deterministically inferable variables are included and that no two variables z_i are merged when they refer to different objects in any action application.

Once the conjunction of atoms in Q_i satisfies these conditions and uniquely determines z_i for all relevant traces, the variable is fixed and the construction proceeds to z_{i+1}, terminating when maximality is reached.
The resulting binding precondition $Q(x, y, z) = Q_1(x, y, z_1) \wedge \cdots \wedge Q_n(x, y, z_n)$ provides a unique grounding for each implicit variable, allowing z to be treated as implicit action arguments in the learned *STRIPS*$^+$ schema.

In the second stage, after all implicit variables are identified, the precondition is extended with additional atoms $Q'(x, y, z)$ that hold whenever the action is executed but do not further constrain the implicit variables. These atoms capture contextual regularities, i.e., relations that hold in all observed executions but do not contribute to determining implicit variables, ensuring that the learned precondition consistently reflects the environmental constraints present in the traces.

Finally, the effects of each action are determined by comparing predecessor and successor states in the traces. Atoms that appear in the successor state but were absent in the predecessor are classified as add effects, while atoms that disappear are classified as delete effects. Only changes that are consistent across all observations of the action are retained, ensuring that the learned action effects are robust, generalizable, and capture the functional impact of the action without overfitting to noise in individual traces. The resulting lifted *STRIPS*$^+$ domain D_L contains one action schema per observed action type, including explicit arguments, deterministically inferred implicit arguments, the complete precondition $Q \wedge Q'$, and all stable add and delete effects. SYNTH thus extends classical STRIPS by encoding functional dependencies between action arguments and implicitly determined variables, enabling accurate reconstruction of lifted actions even under partial observability.

2.5 SIFT

With SIFT [9], the goal is to learn an unknown domain D_T solely from experience from a trace T, without requiring prior knowledge about the domain structure. This experience consists of observed action transitions of the form (*s*,*a*,*s*'), where an agent executes an action glsaction in state *s* and transitions to a new state *s*'. Such transitions can be collected from sequential traces or represented as graphs, where nodes correspond to states and edges represent actions. Notably,

SIFT relies solely on the observed action glsaction and does not require the corresponding states *s* or *s'*.

Traditionally, each action in a symbolic planning domain, or action schema, is defined by a set of preconditions and effects, as introduced in Section 2.3. SIFT addresses the challenge of learning these action schemas directly from data. Rather than directly observing changes in symbolic atoms, SIFT infers which atom changes would be consistent with each observed state-action-state transition. In this way, action effects are captured implicitly as latent explanations of the observed transitions. This approach enables the learning of both action effects and preconditions directly from data, without requiring a predefined symbolic vocabulary.

A key step in this process is the construction of types, following a procedure similar to that used in LOCM [3]. Initially, each argument position of every action is assigned its own type. Then, whenever an object appears in multiple argument positions across different actions in the traces, the corresponding types are merged. This merging process is repeated until no further merges are possible, yielding a final set of disjoint types that partition the objects found in the traces. These types serve two purposes: they provide a meaningful abstraction of object roles in actions, and they dramatically reduce the number of features that must be considered during learning.

After inferring the types, SIFT proceeds to construct typed action patterns, denoted as $a[t]$, where t is a tuple representing the positions of the objects involved in the action. These patterns encode the potential interactions between actions and domain atoms, constrained by type compatibility: only object positions whose inferred types match those expected by a candidate atom are considered valid for binding.

Based on the constructed action patterns, SIFT generate candidate features, defined as $f = \langle k, B \rangle$, where k denotes the arity and B is a non-empty set of action patterns of arity k. These features are formed by enumerating the powerset of all type-consistent argument positions in each action pattern.

The next step is to determine which of these features are admissible. A feature is considered admissible if it is consistent with the observed action traces. This is assessed using a set of pattern constraints, specifically the Consecutive Pattern and the Fork Pattern. These constraints capture regularities that must hold if a feature genuinely corresponds to a changing (or preserved) logical condition in the domain. Only those features that satisfy all constraints are retained as admissible features. Note that the truth assignment of each admissible feature is determined only up to negation, since the observed flips in the traces indicate changes in a property but do not reveal whether the feature itself or its negation is represented.

After identifying the admissible features, SIFT reconstructs both the domain and the instance directly from the observed traces. Each admissible feature $f = \langle k, B \rangle$ defines a predicate of arity k, and every action observed in the traces becomes a lifted action schema with the

corresponding arity. Effects are obtained from the action patterns in B, which specify whether an action makes a feature true or false. Preconditions are inferred by intersecting all literals $f(t[o'])$ that hold consistently before each execution of the corresponding grounded action *a(o)*. Only literals that are always true (or always false) before every occurrence of the action are retained as preconditions.

To derive the instance, SIFT considers only ground atoms whose values change in the observed state-action-state transitions. Their truth values over all states in the traces follow uniquely from the learned preconditions, effects, and action order. The initial truth values of these atoms in the first state of the trace define the learned initial state I_T, yielding the complete learned planning problem $P_T = \langle D_T, I_T \rangle$.

3 Related Work

In recent years, several approaches have explored the derivation of symbolic planning models directly from visual data. The presented methods illustrate that visual input has the potential to be raised to a level suitable for symbolic reasoning. However, existing approaches differ substantially in the assumptions they impose and in the type of symbolic structure they are able to recover.

3.1 Scene Graph Generation

Scene Graph Generation (SGG) [16] refers to the process of deriving a structured, relational representation of a scene from an image or a visual observation. A scene is modeled as a graph consisting of objects (nodes) and relations (edges) between these objects: $G = (O, R)$, where $R \subseteq O \times P \times O$ (e.g., (BoxA, on, BoxB)). Here, O denotes the set of objects, and P is a predefined vocabulary of relations. The central insight is that a visual scene is better described by relationships between objects than by isolated objects alone. While early work focused on static images, more recent approaches extend SGG to dynamic scene graphs that capture the temporal evolution of objects and relations [13].

Despite these advances, SGG has limitations when applied to previously unseen domains. Most methods rely on supervised learning and require extensive object and relation annotations, which makes learning new relations difficult. Furthermore, while changes between successive graphs can indicate possible actions, existing approaches, both supervised [13] and unsupervised [18], typically produce latent action clusters or representations rather than fully extractable action models [5, 14]. Consequently, deriving explicit planning rules, such as action preconditions and effects, directly from scene graphs remains challenging.

3.2 Neuro-Symbolic Planning

Neuro-symbolic planning is one approach for combining learning from data with symbolic reasoning to support decision making from sensory information. In this paradigm, neural components process raw inputs and learn internal representations, while symbolic components use these representations to perform reasoning and planning. There are multiple ways to implement this idea, depending on how the interface between the neural and symbolic parts is designed and how planning is performed. Two representative approaches will be described in the following.

LatPlan

LatPlan [1] is an early system that learns symbolic planning models directly from image sequences, without requiring prior knowledge of states, objects, or actions. It takes as input both a sequence of images showing state transitions and a pair of images representing the initial and goal states.

To represent states, LatPlan employs a State Autoencoder (SAE) that learns a bidirectional mapping between raw images and symbolic states. The SAE is implemented as a Variational Autoencoder (VAE) with Gumbel-Softmax, producing discrete, differentiable latent variables that can serve directly as symbolic propositions. The Encode function maps images to binary vectors, while Decode reconstructs images from these vectors, ensuring that the latent representation captures the essential structure of the environment rather than just pixel values. Actions and their models are acquired in an unsupervised manner through the Action Model Acquisition (AMA) methods. AMA1 generates a PDDL model directly from all observed transitions, demonstrating compatibility with classical planners but without generalization. AMA2 jointly learns action symbols and models from a small subset of transitions without predefined symbols; since it does not output PDDL, planning must be performed using a search algorithm such as A*. For planning, initial and goal states are converted into symbolic representations, allowing the planner to operate entirely in latent space. Intermediate states can be decoded back into images to visualize the plan, bridging the gap between perception and symbolic reasoning. LatPlan is particularly notable for its ability to learn a fully symbolic planning model directly from raw images.

However, the approach has inherent limitations. The learned states are fixed propositional vectors without semantic grounding, and actions are defined solely through observed state-to-state differences, making them parameter-free and specific to the latent states seen during training. The system primarily memorizes observed transitions rather than inducing general domain knowledge, and any change in problem size, object count, or environment structure invalidates the representation. Without explicit object identities or relational information, generalization to unseen configurations is fundamentally impossible.

ROSAME

ROSAME [21] is a neuro-symbolic approach that learns lifted symbolic action models from visual execution traces under partial state observability. Each trace consists of images interleaved with fully observable actions, while only the final state is provided as a fully grounded symbolic state. All intermediate states are observed exclusively through images, allowing multiple symbolic execution traces to correspond to the same visual observation. Observed images are encoded into probabilistic state vectors representing marginal probabilities over all grounded propositions. Instead of inferring discrete states, ROSAME enforces consistency

between successive probabilistic states through a differentiable relaxation of the symbolic successor function. Action models are learned using Probabilistic Action Models (PAMs), which classify each relevant predicate–action pair into one of four cases: irrelevant, add effect, precondition, or precondition with delete effect. Each action symbol is associated with a PAM network, enabling lifted, typed action models that are independent of concrete object instances and transferable across planning problems within the same domain.

ROSAME-I extends this framework to an end-to-end setting by integrating a computer vision model that predicts probabilistic states directly from images. Training is driven by consistency losses between visually inferred states and PAM-based successor predictions, with supervision provided only by the fully observable final state. This allows ROSAME-I to learn reusable symbolic planning domains directly from raw image sequences without supervision of intermediate symbolic states.

Despite its advantages, ROSAME is constrained by its reliance on fully observable actions and a fully grounded final state, which serves as an anchor for constructing a human-readable lifted domain. All predicates and objects must be known in advance, limiting its applicability to unknown or dynamically changing domains.

3.3 Research Gap

Existing approaches demonstrate that symbolic planning models can, to some extent, be derived from visual observations, but they differ substantially in the assumptions they impose and in the symbolic structure they recover. SGG provides rich object-relation representations from images, yet typically relies on supervised learning and does not yield explicit action models with preconditions and effects required for planning. LatPlan operates under minimal assumptions and learns symbolic representations directly from pixels, but produces grounded, instance-specific models without relational or object-centric structure. ROSAME, in contrast, learns lifted and typed action models, but depends on fully observable actions and prior knowledge of predicates and objects. Together, these limitations indicate that deriving generalizable, object-centric planning domains from visual data under weaker assumptions remains an open challenge.

4 Methods

We propose a pipeline that transforms raw pixel-based input representations $(x_i, y_j) \in \mathbb{N}$ of an unknown domain d, together with action labels, into a symbolic planning domain model. This learned domain can subsequently be combined with concrete planning problem instances to generate executable plans π using classical planning algorithms Figure 4.1 provides an overview of the complete pipeline.

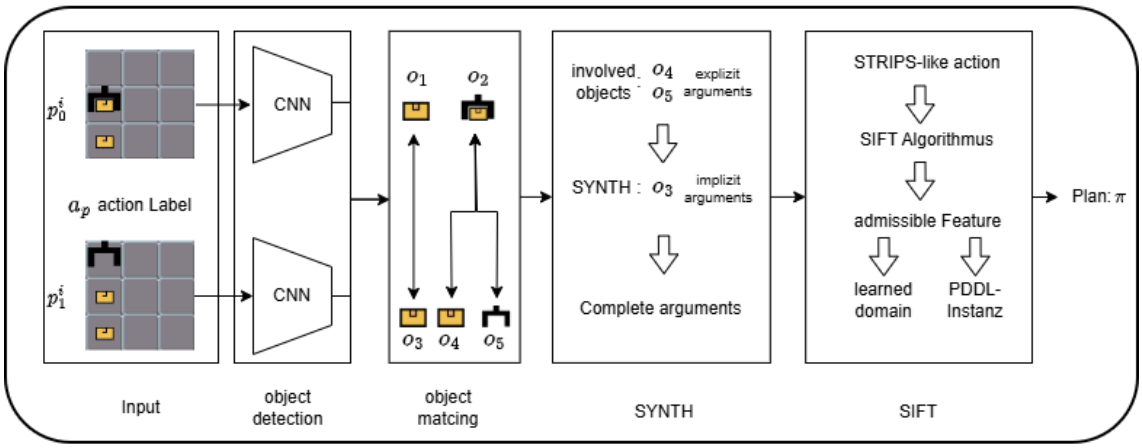


Figure 4.1: Overview of the pipeline. The system processes trace frames with action labels. Two consecutive frames (p_0, p_1) illustrate the flow: a CNN detects objects, an object-matching module links detections across frames, and SYNTH derives implicit arguments. The explicit and implicit arguments yield STRIPS-like action traces for SIFT, which extracts admissible features and constructs the PDDL domain and instance. A planner then produces the final action plan π.

To learn a symbolic domain model using the SIFT algorithm [9], we require complete STRIPS-like action traces over the entire image traces. Constructing such traces requires explicit knowledge of the objects involved in each action and consistent object identities across time steps. Object detection is therefore used to identify the observable objects in each frame, which form the explicit arguments of the actions. Since actions span multiple frames, Multi-Object Tracking (MOT) is required to ensure that these objects maintain consistent identities over time. However, not all action arguments are directly decductible from the visual input. To recover such implicit arguments, we employ the SYNTH algorithm [12], which completes the action descriptions. Together, object detection, MOT, and SYNTH transform raw visual observations into fully specified STRIPS-like action traces, which provide the structured input required by SIFT to induce a lifted PDDL domain model.

We now describe how these STRIPS-like action traces are constructed from image traces. The process begins with the extraction of semantically meaningful objects $(o_1, \dots, o_n),\ n \in \mathbb{N}$ from RGB images, $I \in [0, 255]^{H \times W \times 3}$. Each detected object is localized using a bounding box representation:

$$b = [x_{\min}, y_{\min}, w, h],$$

where $(x_{\min}, y_{\min})$ denotes the coordinates of the top-left corner, and $w, h \in \mathbb{N}$ specify the width and height, respectively. To capture temporal dynamics, consecutive images $I_i(x, y)$ and $I_{i+1}(x, y)$ are compared to detect object-level state changes. From these differences, raw transition traces are extracted, containing only the explicitly observable objects and predicates involved in each change. These incomplete traces are then provided to the SYNTH algorithm [12], which induces the corresponding lifted action models and recovers any implicit objects and arguments not directly visible in the image differences. The resulting completed action schemas allow us to derive STRIPS-like action traces, which are then processed by SIFT [9] to extract symbolic, lifted features and construct the PDDL domain. Finally, a classical planner computes a plan that solves a given instance.

Assumptions

This work makes the following assumptions:

1. The input consists of a finite set of traces:

   $$[s_0, \dots, s_{l-1}] \quad \text{with } l \in \mathbb{N}$$

   where each trace s_i contains a series of overlapping image-action-image triplets:

   $$s_i = \left[p_0^i,\ a_p,\ p_1^i,\ a_p{}',\ p_2^i,\ \dots,\ p_{m-1}^i\right] \quad \text{with } m \in \mathbb{N},\ 0 \leq i < l,\ a_p \in \text{String}$$

   Each triplet $\left(p_j^i, a_p, p_{j+1}^i\right)$ captures a transition from image p_j^i to p_{j+1}^i caused by action a_p. Here, p_j^i denotes the j-th image in the i-th trace, and a_p is the action label applied between the images.
2. The images are discrete grid-based environments where:
   - Each image represents an $N \times M$ grid with known dimensions.
   - The grid structure remains consistent across the image traces.
3. Objects cannot appear or vanish throughout the trace; they can only merge with other objects.
4. Objects are always located within a grid cell, there is only one object or multiple merge objects within a cell.

4.1 Object Detection

In order to derive action arguments from image data, we must first reliably recognize all objects in the image $I(x, y)$. This requires identifying the bounding boxes b of objects within the image.

The challenge lies in detecting all relevant objects in the grid-based image without any prior fine-tuning on the specific objects. This requires the use of unsupervised or pretrained object detection methods capable of recognizing object instances purely based on visual appearance.

4.1.1 Saliency Map Generation

Our approach uses a pretrained ResNet-50 model originally trained on the ImageNet1K dataset for image classification. Although the model is trained to classify images, we do not use it for classification. Instead, we extract saliency maps, which visualize the regions of the image that most influence the model's output.

Following the Guided Backpropagation approach described in Section 2.2, all ReLU activations are replaced by Guided ReLU functions during the backward pass. This modification restricts gradient propagation to positive gradients passing through positively activated neurons, resulting in sharp and spatially localized saliency responses. The backward gradients with respect to the input image yield a three-channel saliency representation corresponding to the RGB channels. To obtain a single-channel saliency map $I(x, y)$, we aggregate the gradients by taking the maximum value across channels at each pixel. The resulting saliency image is normalized to the range $[0, 255]$, where higher values indicate stronger visual relevance (Figure 4.2a).

4.1.2 Refinement of the Saliency Map

Next, we apply a grid mask to zero out the regions between cells in the saliency map (see Figure 4.2b). This step is crucial, as edges between grid cells often produce strong gradients, even though no objects should appear there. Failing to remove them could lead to distorted object boundaries and incorrect bounding boxes. The mask can be applied beacuse the grid dimensions are known.

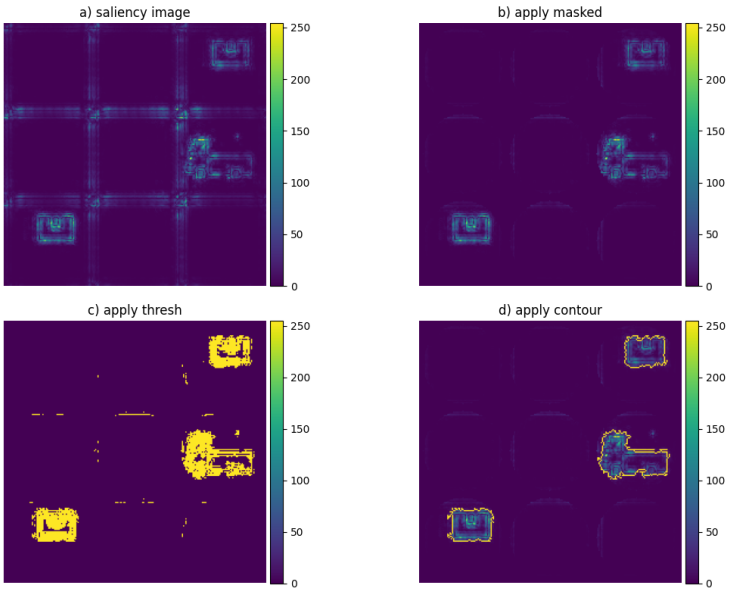


Figure 4.2: Pipeline for Object Detection. The process consists of four stages:
a) saliency image generation highlighting regions of interest,
b) applying a mask to filter relevant areas,
c) threshold application to isolate high-intensity regions (yellow),
d) contour detection to identify object boundaries with colored outlines.

Additionally, we apply a threshold-based binary mask. Let T_p denote the p-th percentile of all values in $I(x, y)$. We define a binary mask $M(x, y)$ as:

$$M(x,y) = \begin{cases} 255 & \text{if } I(x,y) \geq T_p \\ 0 & \text{otherwise} \end{cases}$$

The resulting masked saliency map $I'(x, y)$ is given by

$$I'(x,y) = I(x,y) \cdot M(x,y)$$

This binary map highlights only the most likely object regions (see Figure 4.2c). The choice of T_p is important. If it is too high, many relevant pixels are set to zero, increaing the likelihood of false negative detections. If it is too low, too many pixels remain, raising the chance of false positives detections.

We then group adjacent pixels using the Suzuki–Abe algorithm [20]. The output is a list:

$$C = [C_1, C_2, \ldots, C_n], \quad n \in \mathbb{N}$$

where each component

$$C_k = \{(i, j) \mid I_{i,j} = 255 \text{ and } (i, j) \text{ is connected to other pixels in } C_k\}.$$

For each C_k (see Figure 4.2d), we extract its outer boundary to form a bounding box b. We further refine the result by removing nested or overlapping boxes and discarding C_k with fewer than $m \in \mathbb{N}$ pixels, which helps exclude noise (e.g., 1×1 pixel boxes).

At this stage, we assign a unique ID to each bounding box. These IDs serve as the basis for the next step, where each unique object across consecutive images is given a consistent identifier.

4.1.3 Object embeddings

For object matching, we compute embeddings for each object region. These embeddings are obtained by passing the cropped image regions through a pretrained ResNet-50 model and extracting the output of the last layer before the classification layer. The final fully connected layer is omitted, as we are not interested in class predictions but in a meaningful representation of the image content suitable for object comparison and clustering.

To make the embeddings more reliable for clustering, we normalize the bounding box sizes and assign each bounding box to a corresponding grid cell. Assuming that each cell contains at most one object, and given the image dimensions and the number of grid cells per row and column, each bounding box can be uniquely mapped to a grid cell. These assignments ensure that embeddings are computed from consistent object regions, which is critical for obtaining accurate and stable clusters in the object matching step.

4.2 Multi-Object Tracking

MOT ensures that each object is consistently identified across consecutive images. Maintaining this consistency is essential because unique object identifiers are required throughout the entire trace for both SYNTH [12] and SIFT [9] algorithms.

Our analysis is based on the image-action-image triplets $\left(p_j^i, a_p, p_{j+1}^i\right)$. At this stage, the bounding boxes of all objects in both images have already been detected, providing a foundation for reliably tracking each object across frames.

4.2.1 Local Matching

In consecutive frames, matches between objects may involve one-to-many or many-to-one relationships. This makes it insufficient to rely solely on the Hungarian algorithm [15], which can only handle strict 1:1 matches.

To address this, we model the tracking problem as a graph: each object is represented as a node, and potential matches between objects in consecutive images are represented as edges. This graph-based formulation naturally accommodates one-to-many or many-to-one matches and allows us to assign probabilities to potential matches.

Each edge in the graph is weighted by a probability representing the likelihood that the connected objects correspond. These probabilities allow the graph to capture potential one-to-many or many-to-one matches between objects. By propagating information in both forward (from frame p_j^i to p_{j+1}^i) and backward (from frame p_{j+1}^i to p_j^i) directions, we ensure consistent matching across frames while respecting one-to-many or many-to-one matches.

To compute the edge probabilities, we first construct a cost matrix. The matrix C is defined as $C \in \mathbb{R}^{m \times n}$, where m and n are the number of objects in images p_j^i and p_{j+1}^i. Each entry $C_{i,k}$ quantifies the dissimilarity between object i in the first image and object k in the second image. Higher values indicate greater dissimilarity. Formally, each object is associated with a row or column of the cost matrix:

$$o_i^j \in O^j \leftrightarrow \text{row } i \text{ of } C, \qquad o_k^{j+1} \in O^{j+1} \leftrightarrow \text{column } k \text{ of } C.$$

A key indicator for object similarity is spatial proximity, since many objects either do not move or move only slightly during an action. Spatial proximity is measured using the Manhattan distance between the objects corresponding grid cells:

$$d\left(o_i^j, o_k^{j+1}\right) = |x_i^j - x_k^{j+1}| + |y_i^j - y_k^{j+1}|,$$

Here $\left(x_i^j, y_i^j\right)$ and $\left(x_k^{j+1}, y_k^{j+1}\right)$ are the coordinates of the grid cells that contain objects $o_i^j \in O^j$ and $o_k^{j+1} \in O^{j+1}$, respectively. The Manhattan distance is well-suited for grid domains as it counts the steps needed to move vertically and horizontally between cells.

Another important similarity indicator is the visual content within each cell. The visual content is measured using cosine similarity between the corresponding feature vectors. The feature vectors are from the object embedding that were computed in section 4.1.3. Formally, for objects $o_i^j \in O^j$ and $o_k^{j+1} \in O^{j+1}$ with embeddings A_i and B_k, the cosine similarity is defined as:

$$\text{cosine\_similarity}(A_i, B_k) = \frac{A_i \cdot B_k}{\|A_i\| \cdot \|B_k\|},$$

where

- $A_i \cdot B_k$ denotes the dot product of the embeddings, and
- $\|A_i\|$ and $\|B_k\|$ are the Euclidean norms of the embeddings.

Spatial proximity and visual similarity are combined into a single cost value using a weighting parameter α:

$$C_{i,k} = \alpha \cdot d\left(o_i^j, o_k^{j+1}\right) + (1 - \alpha) \cdot (1 - \text{cosine\_similarity}(A_i, B_k)),$$

where the first term captures spatial proximity and the second term captures visual similarity.

After constructing the cost matrix, we convert the costs into probabilities using a row-wise softmax. To ensure that smaller costs correspond to higher probabilities, we use the negative of the cost values. This way, objects that are more similar (have a smaller cost) are assigned a higher probability of correspondence in the softmax.

$$P(k \mid i) = \frac{\exp\left(-C_{i,k}\right)}{\sum_l \exp\left(-C_{i,l}\right)}.$$

Each row is normalized independently, so for a fixed object i in frame p_j, the probabilities $P(k \mid i)$ sum to 1 across all candidate objects l in frame p_{j+1}. Column-wise normalization (as in the Sinkhorn algorithm [4]) is not applied, because we only retain the maximal-probability correspondence per direction instead of enforcing a bistochastic structure.

Each object in one frame proposes a single match in the other frame by selecting the object with the highest probability. Formally:

$$k^*(i) = \arg\max_k P(k \mid i), \qquad i^*(k) = \arg\max_i P(k \mid i),$$

where $k^*(i)$ is the forward match of object i in frame p_j to frame p_{j+1}, and $i^*(k)$ is the backward match of object k in frame p_{j+1} to frame p_j. Both directions are computed independently.

The final set of local matches is obtained by taking the union of the forward and backward maximum-probability matches, with duplicates removed:

$$E = \{(i, k^*(i)) \mid i \in p_j\} \cup \{(i^*(k), k) \mid k \in p_{j+1}\},$$

With this, we complete the local matching. The result is a set of pairwise matches between consecutive frames. These matches are then integrated in the global matching step to produce consistent identity traces for all objects across the full trace.

4.2.2 Global Matching

The goal of global matching is to consolidate the local matches into consistent object identities across the entire trace. Some objects may still have ambiguous matches because not all matches are one-to-one. As described in the local matching part, let O^j denote the set of objects in frame j, and $E^{j,j+1} \subseteq O^j \times O^{j+1}$ the set of local matches.

An object $o_i^j \in O^j$ is considered ambiguous if it participates in multiple matches, either as a source or as a target:

$$|\{k : \left(o_i^j, o_k^{j+1}\right) \in E^{j,j+1}\}| > 1 \quad \text{or} \quad |\{k : \left(o_k^{j-1}, o_i^j\right) \in E^{j-1,j}\}| > 1.$$

We first identify nodes with a unique incoming or a unique outgoing edge in the matching graph. Consecutive traces of such nodes are concatenated to form deterministic identity chains. Each deterministic identity chain is extracted as a tracklet, which is defined as a trace of object instances across consecutive frames that are confidently linked to the same physical object:

$$T = \left(o_i^j, o_k^{j+1}, \dots, o_l^{j+n}\right).$$

Tracklets represent a consistent object trajectory in the image trace, providing a stable identifier for each object over time.

Objects with multiple incoming or outgoing candidate matches are marked as ambiguous and deferred for conflict resolution, which is handled separately to link them into the global matching.

Although many-to-many matches can occur in principle, in practice the dominant ambiguity patterns are one-to-many (splits) and many-to-one (merges). We therefore illustrate the resolution procedure using this common merge–split structure, which already captures the essential difficulty of global disambiguation.

Consider a merge in which multiple objects in frame j correspond to a single object in frame $j+1$:

$$\{o_1^j, \dots, o_n^j\} \longrightarrow o^{j+1}.$$

The object o^{j+1} may then be part of a tracklet that continues across several frames. Suppose the last element of this tracklet, denoted o^{j+t}, again splits into multiple candidates in the next frame:

$$o^{j+t} \longrightarrow \{o_1^{j+t+1}, \dots, o_n^{j+t+1}\}.$$

This creates a merge–split pattern in which two equally sized sets of objects must be matched consistently:

$$\underbrace{\{o_1^j, \dots, o_n^j\}}_{\text{pre-merge}} \rightarrow \underbrace{o^{j+1} \rightarrow \dots \rightarrow o^{j+t}}_{\text{tracklet}} \rightarrow \underbrace{\{o_1^{j+t+1}, \dots, o_n^{j+t+1}\}}_{\text{post-split}}.$$

To restore a one-to-one assignment across this structure, we compute a matching between the two boundary sets:

$$\{o_1^j, \dots, o_n^j\} \quad \text{and} \quad \{o_1^{j+t+1}, \dots, o_n^{j+t+1}\}.$$

We apply the Hungarian algorithm [15] using the same cost function as in local matching but restricted to these objects and with stricter thresholds on cosine similarity and the weight parameter α.

For the hungarian algorithm we also need a cost matrix. The cost matrix that we use is the same as we use to get the propabilities. The only difference is that we only use the m objects and the hyperparameters. We know that the objects should look alike so we apply a threshold that we only consider a matching if the cosine similarity is very high. The same applies to the alpha value in the cost matrix, which is set high to emphasize cosine similarity.

The algorithm is applied as follows: First, a row reduction is performed by subtracting the smallest value in each row from all elements of that row. Next, a column reduction is applied by subtracting the smallest value in each column from all elements of that column. After these reductions, we cover all zeros in the matrix using the minimum number of horizontal or vertical lines. If the number of covering lines equals the number of rows or columns, an optimal assignment has been found. If not, we adjust the matrix by performing another reduction step. This time based on the smallest uncovered value without modifying the positions of the zeros. This process is repeated until an optimal assignment becomes possible.

This yields a unique and globally consistent identity assignment across the merge-split pattern, providing stable object trajectories throughout the full trace. With the global matching completed, we can now focus on validating these matches to ensure that all edges are reliable before proceeding with the SYNTH algorithm [12].

4.2.3 Global Assignment Refinement

The global matching provides a preliminary assignment of consistent object identities across frames. While effective in many cases, it can produce ambiguities in situations involving visually similar objects, merges, or splits. These ambiguities, if left unresolved, can propagate through the trace and compromise downstream analyses. To mitigate this, we introduce a refinement procedure that combines object embeddings with action-specific correspondence statistics. The refinement aims to correct unlikely assignments while preserving the overall structure imposed by the global matching. This ensures consistency in both the visual and structural aspects of the trace.

Cluster Construction

To reduce ambiguity in object matches, we first construct clusters of visually similar objects. Each object is represented by an embedding derived from its normalized cell representation (Section 4.1.3). Similar objects are grouped into clusters to form a type-level abstraction that captures the notion of object identity beyond individual instances. Let $U = \{U_1, \dots, U_m\}$ denote the set of resulting clusters. Each object o_i is assigned to the cluster with the highest cosine

similarity. If no existing cluster exceeds a similarity threshold τ, a new cluster is created for o_i. The clustering process stabilises object identities and provides a foundation for analysing consistency in matches at a higher level. This enables the system to detect anomalies in the initial global matching.

Detecting Suspicious matches

The clusters from the previous step provide a type-level abstraction of objects. This allows us to evaluate matches beyond individual instances. Suspicious matches are pairs of objects whose cluster-level mappings rarely occur for a given action.

For each transition $(j, j+1)$ with action a_p, object-level matches $\left(o_i^j, o_k^{j+1}\right)$ are mapped to cluster-level pairs $\left(u_i^j, u_k^{j+1}\right)$. Ambiguous matches result in ambiguous cluster-level combinations, which we collect across the trace in the multiset $E_{a_p}^{\text{amb}}$. We measure how often each cluster pair occurs under action a_p using its relative frequency.

$$\text{freq}\big(U_i, U_k \mid a_p\big) = \frac{\#\{(U_i, U_k) \in E_{a_p}^{\text{amb}}\}}{\#E_{a_p}^{\text{amb}}}.$$

Cluster pairs with frequencies below a threshold θ are deemed suspicious, and all frame pairs containing such combinations are collected into the set $\mathcal{F}_{\text{suspicious}}$. This step detects potentially incorrect matches and provides a basis for corrective actions.

Generating and Evaluating Alternative Assignments

For each suspicious frame pair $(j, j+1)$, we generate alternative object-level assignments that comply with the set of non-suspicious cluster pairs $\mathcal{C}_{a_p}$. Candidate assignments $\mathcal{O}'$ are restricted so that every object mapping satisfies

$$\left(u\left(o_i^j\right), u\left(o_k^{j+1}\right)\right) \in \mathcal{C}_{a_p},$$

ensuring consistency with common patterns observed for the action a_p. The evaluation of each candidate involves recomputing the forward and backward transition probabilities, as described in Section 4.2.1, with the candidate mappings treated as fixed. The score of a candidate assignment is

$$\text{score}(\mathcal{O}') = \sum_{(o_i \to o_k) \in \mathcal{O}'} P_{\text{forward}}(o_i \to o_k) + P_{\text{backward}}(o_k \to o_i),$$

and the best candidate is selected as $\mathcal{O}^* = \arg\max_{\mathcal{O}'} \text{score}(\mathcal{O}')$. To prevent unnecessary changes, we replace the original assignment only if the new candidate is nearly as likely:

$$\text{score}\big(\mathcal{O}^{\text{original}}\big) - \text{score}(\mathcal{O}^*) < \delta.$$

This ensures that refinements are adopted only when they are supported by both local evidence and global action-conditioned statistics.

Outcome and Consistency

After refinement, each object obtains a unique and stable identity throughout the trace. By integrating type-level regularities, action-conditioned statistics, and local transition likelihoods, the procedure produces a globally consistent mapping that reflects both visual similarity and structural patterns of the environment. This refined assignment forms a reliable foundation for upcoming SYNTH [12] and SIFT [9] algorithm, ensuring that object identities remain coherent even in challenging scenarios with ambiguous transitions.

4.3 SYNTH

SYNTH [12] constructs lifted action models in domains where some STRIPS arguments and predicates are fully observable, while others are not directly observable. Its input is a grounded trace composed of fully observable predicates and explicit action arguments extracted from observed state-action-state transitions. In our grid-based setting, the grounded trace includes both visible objects and the underlying grid structure. To this end, in addition to detecting visible objects, each grid cell is assigned a unique identifier so that spatial relations can be represented symbolically. Let the grid be represented as a matrix $C \in \mathbb{R}^{m \times n}$, where m denotes the number of rows and n the number of columns. A cell at row $r \in \{0, \dots, m-1\}$ and column $c \in \{0, \dots, n-1\}$ is assigned the unique identifier $\text{cell}_{id} = r \cdot n + c + \text{offset}$, where offset corresponds to the total number of globally assigned object identifiers. This ensures that cell identifiers do not overlap with object identifiers in the symbolic representation. Fully observable predicates in the grounded trace include, for example, the position of each object at every timestep, represented via an *at*-relation linking objects to cell identifiers. The grid topology is encoded using adjacency predicates, separated into horizontal and vertical relations. Explicit action arguments are derived from observable interactions such as movements between cells or interactions with other objects. Aggregating these predicates and explicit arguments across the entire image trace yields the grounded trace provided as input to SYNTH.

Given this input, SYNTH incrementally infers the implicit arguments of each action instance by constructing a binding precondition {Qquery}. Here, x denotes the explicit arguments. For example, in Blocksworld, x could be the block being moved (block A). The free variable y serves as a temporary placeholder connecting x to the implicit argumtens in SYNTH. The variable z denotes implicit variables. For instance, z could be the block or cell on which A is placed (block B or the table). These implicit variables can be deterministically inferred from x. The construction proceeds in layers $Q_1, \dots, Q_n$. Each layer Q_i determines a single implicit variable z_i through a conjunction of lifted atoms $q_{i,j}(x, y, z_i)$. For example, a lifted atom could express

that z_i (the destination block) is clear, or that A is currently on another block C. Each atom may reference explicit arguments x, free variables y, and the current implicit variable z_i, but never any future implicit variable z_k with $k > i$. Atoms are added incrementally and only included if the resulting query satisfies the conditions of validity, stratification, and maximality (see Section 2.4). Once a variable z_i is uniquely determined across all relevant traces—for instance, the block B that A is consistently placed on-it is fixed, and the construction proceeds to the next implicit variable z_{i+1}. This process continues until all implicit arguments of the action instance, such as target positions or indirectly affected blocks, have been identified.

The output of SYNTH is a complete PDDL action schema *a(o)* for every observed action type, including explicit arguments x, inferred implicit arguments z, a full precondition $Q \wedge Q'$, and consistent add and delete effects extracted by comparing predecessor and successor states in the trace. These schemas are then used to update the grounded trace, producing a STRIPS-like action trace in which observed effects are consistently aligned with the corresponding action instances. Implicit arguments are expressed in the lifted schema through the variables z, which are deterministically grounded whenever the action applies, thereby enabling accurate action reconstruction even under partial observability.

The output of SYNTH is a lifted PDDL action schema *a(o)* for each observed action type. Each schema specifies the explicit arguments x, inferred implicit arguments z, a full precondition $Q \wedge Q'$, and consistent add and delete effects derived from observed state transitions. To reconstruct action instances for a given trace, the implicit variables z are grounded by matching the schema's effects to the changes observed between predecessor and successor states. This produces a STRIPS-like grounded action trace in which all action arguments (explicit x and implicit z) are consistently instantiated to match the observed state changes, enabling accurate reconstruction of action arguments even under partial observability.

4.4 SIFT

Before applying the SIFT algorithm [9], the STRIPS-like action trace must be transformed into a transition graph. States become nodes in this graph, and each edge corresponds to an action instance annotated with its arguments. The resulting transition graph is then provided to SIFT, which derives the lifted domain as described in the Background Section 2.5. This yields both a PDDL domain and a corresponding PDDL problem instance.

To obtain consistent results, we compress the created domain to its minimal form, retaining only the necessary predicates. SIFT provides a minimally constrained set that identifies which predicates are actually required. Using an ASP solver such as Clingo [7], we then determine the minimal set of features needed to correctly reproduce the observed traces. This minimal domain is subsequently used to generate the PDDL domain and instances.

In order to continue with the planning, we need to modify the static predicates derived by SIFT. Static predicates are predicates that do not change during planning and primarily control the grounding of actions. Many domains use static predicates to encode invariant properties, such as the topology of a grid or fixed relationships between objects. SIFT handles static predicates by introducing them directly from traces: a static predicate is set to true if and only if the corresponding grounded action appears in the observed trace. As a consequence, the induced domain cannot generalize to unseen action instantiations, which limits applicability in novel environments. To overcome this limitation, we modify the static predicates derived by SIFT. Since we always have a known grid structure in our domain, we can compute the values of adjacency predicates directly rather than relying on the trace. Specifically, for predicates representing the adjacency relation, we compute their values independently using the information obtained in the SYNTH step. By doing so, we slightly adjust the SIFT domain and instance, but we are no longer constrained to the specific steps observed in the trace. This allows the grounded action schemas to be applied more generally within the trace, while still correctly reflecting the grid structure and barriers in the environment

4.5 Planning

From the output of SIFT, we construct a PDDL problem instance by extracting the grounded predicates of the initial state. These predicates populate the `(:init)` section of the PDDL problem together with the corresponding object declarations. Combined with the learned PDDL domain, this yields a fully specified planning problem. Given the learned PDDL domain, the constructed PDDL problem instance, and the grounded STRIPS-like action trace, symbolic planning can be performed. The initial state is obtained from the initial state in the image trace, as grounded by SIFT. By applying the learned PDDL action schemas, all subsequent states in the trace can be grounded accordingly, yielding a consistent symbolic representation of the observed execution.

At present, planning is restricted to the set of states contained within the observed image trace. This limitation arises from the absence of a classifier capable of mapping arbitrary visual observations to symbolic feature valuations. Consequently, planning outside the observed trace is not yet supported. Nevertheless, the learned symbolic representation can be extended to previously unseen states under the assumption of a fixed grid size and a fixed number of objects. In particular, an arbitrary image can be mapped to a symbolic state description by learning a classifier that predicts the truth value of symbolic features extracted by SIFT. Training data for this classifier is obtained from the grounded STRIPS-like action trace. The initial state is grounded directly from the first image, and all subsequent states are grounded by applying the learned PDDL action schemas. This process yields a dataset of image states paired with fully grounded symbolic feature valuations. Using this dataset, we train a collection of binary classifiers, one per symbolic feature extracted by SIFT, each predicting whether the corresponding feature

holds in a given state. The input to these classifiers is derived from the observable structure of the image. Objects are extracted using the same object detection and tracking pipeline, and their spatial relations are computed, including object-to-cell *at*-relations, adjacency relations induced by the grid topology, and object type information. These observable relations form the feature vectors for classification. Any standard binary classification model, such as logistic regression, support vector machines, or neural networks, can be employed in this setting. For a previously unseen image that satisfies the same structural assumptions—namely identical grid dimensions and the same number of objects—the observable relations are extracted and evaluated by each binary classifier over all relevant object and cell combinations. The resulting feature truth assignments are combined to form a complete symbolic state description. This allows arbitrary visual states to be lifted into symbolic representations compatible with the learned PDDL domain, thereby enabling planning beyond the original image trace once such classifiers are available.

With a fully grounded domain and problem instance, standard PDDL planners can then be applied to generate a plan. The resulting plan consists of a sequence of actions that transitions the system from the initial state to the goal state defined within the trace.

5 Results

This chapter evaluates the proposed vision-to-planning pipeline and analyzes its performance across all major components. Since learning symbolic planning models directly from visual input under partial supervision is still a largely unexplored setting, there exist no directly comparable end-to-end baselines. As a result, the evaluation focuses on assessing the pipeline in a structured, component-wise manner, examining both the correctness of the induced symbolic representations and their suitability for classical planning.

While SYNTH successfully reconstructs implicit action arguments in the Blocksworld domain, the current SIFT implementation does not produce a stable lifted planning domain for this environment. The complexity of Blocksworld results in a large number of inferred features and potential predicate combinations, which prevents SIFT from generating a complete domain. Consequently, Blocksworld is used solely to evaluate implicit argument reconstruction, while the full end-to-end evaluation of domain induction and planning is conducted on the Delivery domains.

5.1 Evaluation Setup

Due to the lack of directly comparable approaches in vision-to-planning that learn symbolic planning domains from visual input under similar assumptions, we evaluate our method in a component-wise manner. We first introduce the symbolic planning domains used in our experiments. These domains are obtained by generating a dataset of symbolic planning problems and applying our complete vision-to-planning pipeline to recover their underlying planning models. We then describe the evaluation metrics used to assess each stage of the pipeline individually, allowing us to isolate the contributions of object detection, MOT, SYNTH and SIFT. The experimental results are then analyzed with respect to the structural correctness of the learned planning domains and the solvability of planning tasks derived from the induced models.

5.2 Domains

We evaluate our vision-to-planning pipeline on a set of grid-based planning domains implemented in Pygame [2]. As stated in Section 4, we restrict our evaluation to grid domains to provide a controlled and well-defined setting for analyzing the individual components of the pipeline. Each domain represents a discrete, fully observable environment rendered as RGB

images, which serve as the sole input to the perception module, while action labels are provided separately as supervision for action model induction.

For each domain, we evaluate the pipeline on a single problem instance and a single execution trace. The trace consists of a sequence of state-action-state transitions generated by executing a randomized action sequence within this fixed environment. Although the object set and spatial layout remain constant, the trace contains diverse interactions between objects over time. This setup allows us to assess whether the pipeline can recover a coherent lifted, object-centric planning model from a single visual trajectory, rather than memorizing individual states. Generalization across multiple problem instances or object configurations is left for future work.

5.2.1 Delivery Domain

The delivery domain contains two object types, truck and package. The planning goal is for the truck to reach a package, load it, transport it to a designated target cell and unload it there.

In our implementation, each cell can contain at most one object. Because of this restriction, the load action is implemented as a combined move and load, where the truck enters the package's cell and immediately loads it, forming a single combined object representing the loaded truck. The unload action separates the package from the truck by leaving the package in the cell while the truck moves out.

These modifications preserve standard planning semantics while ensuring that state changes are clearly observable by the perception module. The full PDDL domain specification is provided in the Appendix.

5.2.2 Delivery Domain with Fences

This variant of the delivery domain introduces static fence objects that block specific cells in the grid. Fences prevent trucks and packages from occupying certain locations, creating additional constraints on movement and object interactions. The purpose of this modification is to evaluate how the perception and symbolic induction modules, particularly SYNTH and SIFT, respond to changes in the environment that restrict action applicability and alter spatial relations. All other aspects of the domain remain the same as the original delivery domain. The full PDDL specification for this domain variant is provided in the Appendix.

5.2.3 Blocksworld Domain

The Blocksworld domain contains two object types, blocks and grippers. The planning goal is for the gripper to move the blocks into a desired configuration.

Following the same assumption as in the delivery domain, each cell can hold at most one object. Because of this restriction, some actions are implemented as combined moves and manipulations. Pickup and unstack actions move a gripper to a block while simultaneously grasping it, forming a single combined object. Putdown and stack actions leave the block in a target cell while the gripper moves away, making state changes clearly observable. The main distinction between the two pairs of actions lies in what the gripper interacts with. In stack and unstack, the gripper explicitly places a block on top of another block or removes it from another block, directly manipulating block-to-block relations. In contrast, pickup and putdown operate only on free cells: the gripper picks up a block from an empty cell or places a held block onto an empty cell, without directly involving other blocks. The full PDDL domain specification is provided in the Appendix.

Domain Overview

Table 5.1 summarizes key domain statistics and trace characteristics. It reports the number of object types, the size of the action set, the average number of instantiated objects per trace and the fixed trace length in frames.

Domain	Object Types	Action Set Size	Avg. Objects per Frame	Trace Length (Frames)	Grid (rows × columns)
Delivery	3	3	6.86	1000	5×5
Delivery (Fences)	4	3	9.72	1000	5×5
Blocksworld	3	5	4.46	2000	5×3

Table 5.1: Domain statistics, trace characteristics, and workspace grids used for evaluation. The "Avg. Objects per Frame" column reports the typical number of objects present in an image along the trace.

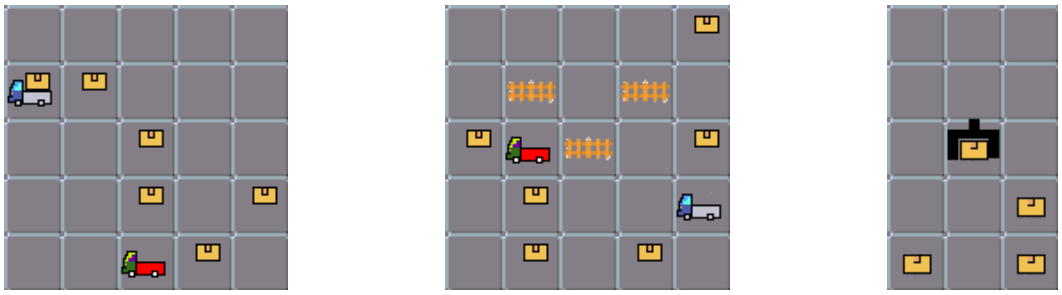
Table 5.1 provides a numerical overview of the objects, actions, and workspace grids in each domain. All three domains share a common structure: the workspace is organized as a discrete grid, and each cell can hold at most one object. Objects belong to distinct categories depending on the domain and actions manipulate these objects. Each simulation trace records a sequence of frames representing the states of all objects, providing input for perception and planning.

The Delivery domain contains 3 object types and an action set of size 3. Adding fences introduces an additional static object type, increasing the number of types to 4 and the average number of objects per trace from 6.86 to 9.72, while the action set size and trace length remain constant at 1000 frames.

The Blocksworld domain contains 3 object types (blocks and grippers) and 5 actions. Its traces are longer, consisting of 2000 frames. The workspace grid is smaller (5×3) compared to the Delivery domains (5×5). This smaller grid is chosen to focus the gripper's movements and

avoid excessive wandering, ensuring that the simulation generates meaningful interactions with the blocks.

Figure 5.1 shows visual examples of the domains. These images illustrate the object layouts and spatial arrangements within a single state from each trace, highlighting the grid structure and object types.



(a) Delivery (b) Delivery with Fences (c) Blocksworld

Figure 5.1: Example images of the evaluation domains. Each grid shows object positions and types for a single state in the trace.

5.3 Evaluation Strategy for Pipeline Components

We evaluate the pipeline in a component-wise manner under a fixed configuration. Each trace is instantiated according to the domain statistics and trace characteristics summarized in Table 5.1. The hyperparameters listed in Table 5.2 were selected to balance sensitivity and robustness: the saliency threshold and minimum object size ensure reliable object detection while filtering out spurious detections, the border margin prevents edge artifacts, and the minimum matching ratio defines the lower limit for considering candidate edges in the matching procedure. To isolate the behavior of individual submodules, we assume correct inputs at their interfaces. If an upstream component produces invalid output, it is replaced with ground-truth information, provided that the ground-truth data is available. This allows us to analyze the functionality and limitations of each pipeline stage independently, without confounding effects from error propagation.

Component	Parameter	Value
Object Detection	Saliency percentile threshold	92
	Minimum object width (pixels)	7
	Minimum object height (pixels)	7
	Border cell margin (pixels)	8
Object Matching	Minimum matching ratio	0.10
	Matching confidence weight (α)	0.7

Table 5.2: Fixed nominal parameter configuration used for component-wise evaluation.

5.3.1 Object Detection Evaluation Metrics

We evaluate the object detection module using standard metrics based on the correspondence between predicted and ground-truth objects. Let TP, FP, and FN denote:

$$\begin{aligned}
\text{TP} &= \text{Number of predicted objects that correctly match a ground-truth object,}\\
\text{FP} &= \text{Number of predicted objects that do not match any ground-truth object,}\\
\text{FN} &= \text{Number of ground-truth objects that are not detected by the model.}
\end{aligned}$$

These quantities are counted for each frame in the trace and then aggregated across all frames. From these, we compute:

$$\begin{aligned}
\text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}},\\
\text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}},\\
\text{F1-score} &= 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.
\end{aligned}$$

To evaluate the quality of the predicted bounding boxes, we report the Intersection-over-Union (IoU) for each matched object pair:

$$\text{IoU} = \frac{\text{Area}(\text{BoundingBox}_{\text{predicted}} \cap \text{BoundingBox}_{\text{ground truth}})}{\text{Area}(\text{BoundingBox}_{\text{predicted}} \cup \text{BoundingBox}_{\text{ground truth}})}.$$

For each trace, we compute the IoU for all matched object pairs and report the mean as well as the 1st, 2nd (median), and 3rd quartiles, providing a more comprehensive view of bounding box quality across predictions.

5.3.2 Multi-Object Tracking Evaluation Metrics

We evaluate object matching using the same metrics as object detection (Precision, Recall, F1), but without IoU, because object connections over time are discrete relationships, not bounding boxes. True positives (TP) correspond to correctly matched edges, false positives (FP) to edges incorrectly predicted, and false negatives (FN) to ground-truth edges that were not matched. Evaluation is performed both before and after edge recovery, as described in Section 4.2.3, to quantify the impact of recovering missing edges. This approach allows us to measure not only the raw matching performance of the object pairs but also the contribution of the recovery step.

5.3.3 SYNTH Evaluation Metrics

We evaluate the SYNTH module by examining its ability to recover implicit action arguments from partially specified input traces. For each action schema, we distinguish between explicit arguments, provided directly in the trace, and implicit arguments, which must be inferred by SYNTH based on the explicit parameters and available predicates.

For every action instance, we inspect the inferred implicit parameters and describe how they correspond to the intended action schema. When an implicit parameter is missing or incorrectly inferred, we analyze the underlying reasons. This procedure allows us to systematically document which parameters are successfully reconstructed and to characterize the conditions under which reconstruction failures occur.

5.3.4 SIFT Evaluation Metrics

We evaluate the SIFT module by comparing the features, preconditions, and effects it generates with those of a manually constructed reference domain intended to correspond to the target environment. Instead of relying solely on aggregate metrics, we analyze each generated element in the context of the reference domain. This allows us to determine which elements are correctly captured, which are partially inferred, and which are missing or incorrectly induced. Metrics include the fraction of preconditions and effects correctly recovered, the number of spurious or missing elements, and an action-level analysis highlighting where SIFT succeeds or fails. By combining quantitative and qualitative evaluation, we systematically assess the reliability and limitations of SIFT in generating accurate symbolic planning models from visual input.

5.3.5 Planning Evaluation Metrics

To evaluate the planning module, we use the classical Fast Downward planner [11] to generate plans from the induced PDDL domains. For each trace, we randomly select an initial and a goal state from the sequence of observed states, grounding them with the objects and predicates extracted from the trace. The planner is then allowed to run for up to one hour, and the best

plan found within this time is used for comparison, rather than enforcing plan optimality. The resulting plan is compared to the plan derived from the handcrafted domain. Evaluation considers both plan equivalence and plan differences. Plan equivalence measures whether the planner produces the same sequence of actions as in the plan produced by the handcrafted domain. When plans differ, we analyze deviations such as missing or additional actions and variations in action ordering. Metrics include the fraction of traces with identical plans, the average number of differing actions per trace, and qualitative observations on common sources of divergence. This methodology allows us to assess both the functional correctness and generalization capabilities of the induced planning models.

5.4 Evaluation Results

In this section, we present the results obtained from applying the evaluation strategy described in Section 5.3. We follow a component-wise analysis, reporting performance metrics for object detection, MOT, SYNTH, SIFT, and planning. For each component, we summarize quantitative metrics, highlight qualitative observations, and discuss common failure modes.

All results are reported under the fixed nominal configuration defined in Table 5.2. Where relevant, we analyze the impact of environmental variations, such as the introduction of fences in the delivery domain, on the performance of perception and symbolic inference modules.

5.4.1 Object Detection Results

Table 5.3 summarizes the object detection performance for the Delivery domain, its fenced variant, and the Blocksworld domain.

Domain	Recall	Precision	F1	Mean IoU	IoU Q1	IoU Median	IoU Q3
Delivery	1.000	1.000	1.000	0.523	0.412	0.458	0.699
Delivery + Fences	0.999	1.000	0.999	0.702	0.633	0.686	0.778
Blocksworld	1.000	0.834	0.909	0.423	0.388	0.404	0.478

Table 5.3: Object detection performance and IoU distribution for the Delivery and Blocksworld domains.

In the base Delivery domain, the detector achieves perfect recall and precision, correctly identifying all objects. Despite this, the mean IoU of 0.523 indicates moderately loose bounding boxes, with the median IoU at 0.458. This level of localization is sufficient for symbolic abstraction, particularly in grid-world domains, where the environment itself imposes a coarse spatial resolution and symbolic reasoning depends on object presence and relative spatial relations rather than pixel-accurate boundaries.

Adding fences increases the number of objects per image from 6.93 to 9.78 (see Table 5.1). Detection performance remains high, with recall slightly decreasing to 0.999 while precision stays perfect. Notably, the mean IoU rises to 0.702 and the median to 0.686. This improvement is caused by percentile-based thresholding of the saliency maps: more objects lead to more high-intensity regions, which shifts the global percentile threshold. Peripheral weak activations are suppressed, while the cores of objects remain, producing tighter and more stable bounding boxes.

By contrast, the Blocksworld domain shows lower localization quality, with a mean IoU of 0.423 and a median of 0.404. Two factors contribute to this decrease: the workspace is smaller (5 rows by 3 columns), increasing object proximity, and fewer objects per trace (4.46 on average), which reduces the number of strong saliency activations. As a result, percentile-based thresholding preserves relatively larger peripheral regions, producing looser bounding boxes. To achieve better localization in Blocksworld, the global percentile threshold hyperparameter would have to be increased to account for the lower number of objects.

Overall, these results show that the object detection component scales robustly with increasing object density and scene complexity, while smaller and sparser environments like Blocksworld require hyperparameter adjustment to achieve comparable localization quality.

5.4.2 Mulit-Object Tracking Results

We evaluate the object matching component separately from object detection, focusing on the correctness of edge correspondences between consecutive frames. Ground-truth objects are associated with idealized bounding boxes, whereas detected objects that achieve a perfect detection score (F1 = 1.000) may still exhibit geometric inaccuracies, as reflected in the IoU distributions in Table 5.3. To avoid penalizing the matching component for detection errors while still accounting for imperfect localization, detected bounding boxes are used only when the detected object set is complete; otherwise, ground-truth positions are applied. Table 5.4 summarizes the MOT performance for the Delivery and Blocksworld domains.

Domain	Correct Matches	Recall	Precision	F1	Recovered Edges	Final F1
Delivery	7097	0.999	0.999	0.999	10	1.000
Delivery + Fences	9973	1.000	1.000	1.000	0	1.000
Blocksworld	8993	1.000	1.000	1.000	0	1.000

Table 5.4: MOT performance for the Delivery and Blocksworld domains. ‘Recovered Edges‘ indicates the number of edges corrected using refinement, and ‘Final F1‘ reports the F1 score after recovery.

Matching operates exclusively on edges between detected objects and assumes a correct set of object nodes per frame (see Section 4.2.1). No explicit mechanism handles missing or spurious

objects; such cases automatically induce at least two incorrect edges. Thus, the evaluation isolates the matching component from detection-level errors.

In the Delivery domain, matching performance is high. Across all sequences, 7097 edges are correctly matched, yielding recall, precision, and F1 of 0.999. Most errors occur in a small fraction of frames with ambiguous multi-object configurations during *Load* and *Unload* actions.

Table 5.5 breaks down the types of object transitions observed. The majority of frames correspond to common transitions, specifically clusters `[[0,1],[3]]` and `[[0,2],[4]]`, where the cluster numbers indicate object types: 0: package; 1 and 2: two distinct truck objects; 3 and 4: the corresponding loaded states of these trucks carrying a package. The distinction between the two trucks (and their loaded counterparts) is preserved because objects are normalized by their cell positions during clustering, allowing visually similar sprites to be treated as separate instances (see Section 4.2.3). These transitions, representing interactions between a package and a truck or a truck carrying a loaded package, account for over 97% of frames in both *Load* and *Unload* actions.

Action	Cluster-Type	Count	Percent	Qualitative Note	Outlier
Unload	[[3],[0,1]]	98	54.1%	package and truck / truck loaded with package	No
	[[4],[0,2]]	78	43.1%	package and truck / truck loaded with package	No
	[[1],[1,2]]	4	2.2%	two trucks interact with a third	Yes
	[[2],[1,2]]	1	0.55%	two trucks interact with a third	Yes
Load	[[0,1],[3]]	97	53.3%	package and truck / truck loaded with package	No
	[[0,2],[4]]	80	43.96%	package and truck / truck loaded with package	No
	[[1,2],[2]]	3	1.65%	two trucks interact with a third	Yes
	[[1,2],[1]]	2	1.10%	two trucks interact with a third	Yes

Table 5.5: Recovery statistics in the Delivery domain separated by action type (Load / Unload). Each ”Cluster-Type” represents a cluster-level object abstraction based on normalized cell positions. Outlier types correspond to rare interactions, such as multiple trucks interacting in consecutive frames.

Outlier frames contain rare multi-object interactions, which are the primary source of matching uncertainty. While these cases are infrequent, they exhibit reduced matching confidence and are thus assessed with respect to the minimum acceptance threshold of the matching procedure (minimum matching ratio of 0.10; see Table 5.2).

Ambiguous outlier frames are addressed via the global assignment refinement procedure (Section 4.2.3), which evaluates alternative cluster-level assignments consistent with observed action patterns. This step corrects all ambiguous edges, including cases with multiple incoming or outgoing edges per object, and raises the Final F1 score to 1.000.

For the Delivery domain with fences, the reported matching performance reaches perfect precision and recall, with 9973 correctly matched edges. As with the Blocksworld domain, ground-truth bounding boxes are used for evaluation to ensure fully correct object correspondences. This approach isolates the matching procedure from potential errors in the detection stage, resulting in a Final F1 score of 1.000.

In the Blocksworld domain, a total of 8993 correctly matched edges are obtained, also yielding recall, precision, and F1 scores of 1.000. Here too, ground-truth object positions are used as input to the matching algorithm, since object detection is not perfect. The results indicate that, given accurate object localizations, the matching algorithm reliably produces correct correspondences between consecutive frames. This ensures that symbolic states derived from Blocksworld traces are consistent and suitable for downstream action inference and planning.

Overall, these results demonstrate that the matching algorithm is robust across different domains, object densities, and scene complexities. Using ground-truth object positions for evaluation confirms that the algorithm itself can achieve perfect matching when provided with correct inputs.

5.4.3 SYNTH Results

We aim to systematically evaluate which implicit action arguments SYNTH can reliably reconstruct and under which structural conditions reconstruction fails. These arguments are crucial for the next step, where SIFT generates a complete STRIPS-like action trace; without them, the domain cannot be constructed. For this analysis, we focus on the Blocksworld domain and the standard Delivery domain. The extended Delivery variant, which introduces additional fences, exhibits identical behavior with respect to implicit argument reconstruction and is therefore omitted to avoid redundancy. By limiting the analysis in this way, we can highlight the core patterns of argument reconstruction without the added complexity of redundant structures.

In the following, we analyze how action arguments are specified in the Blocksworld domain. Specifically, we distinguish between arguments that are provided explicitly as input parameters and arguments that are defined indirectly via relational constraints. Table 5.6 summarizes this distinction for the actions move, pickup, putdown, stack, and unstack, including the semantic interpretation of each argument.

Action	Parameter	Description	Argument Semantics
move	x0	given as input	Cell object of the gripper *after* the action
	x1	given as input	Gripper object
	x2	(relation_at, (1,-1))	Cell object of the gripper *before* the action
pickup	x0	given as input	Cell object of the gripper *after* the action
	x1	given as input	Gripper object
	x2	(relation_at, (1,-1))	Cell of gripper *before* the action
	x3	(relation_at, (-1,0))	Block object
	x4	(relation_vertical, (2,-1))	free cell above
	x5	(relation_vertical, (4,-1))	free cell above
	x6	(relation_vertical, (5,-1))	free cell above
putdown	x0	given as input	Cell object of the gripper *after* the action
	x1	given as input	Gripper object
	x2	(relation_at, (1,-1))	Cell of gripper *before* the action
	x3	(relation_vertical, (0,-1))	free cell above
	x4	(not-relation_at, (-1, None)) (not-relation_horizontal, (None, -1)) (not-relation_horizontal, (-1, None))	Block object
	x5	(relation_vertical, (3,-1))	free cell above
	x6	(relation_vertical, (5,-1))	free cell above
stack	x0	given as input	Cell object of the gripper *after* the action
	x1	given as input	Gripper object
	x2	(relation_at, (1,-1))	Cell object of the gripper *before* the action
	x3	(not-relation_at, (-1, None)) (not-relation_horizontal, (None,-1)) (not-relation_horizontal, (-1,None))	Block object that was with the gripper
	x4	(relation_vertical, (-1,2))	Block object below the stacked object
	x5	(relation_at, (-1,4))	Cell of the Block below
unstack	x0	given as input	Cell object of the gripper *after* the action
	x1	given as input	Gripper object
	x2	(relation_at, (1,-1))	Cell object of the gripper *before* the action
	x3	(relation_at, (-1,0))	Cell of target block
	x4	(relation_vertical, (-1,0))	Block object below the stacked object
	x5	(relation_at, (-1,4))	Cell of the Block below

Table 5.6: Parameter characterization and semantic interpretation of all Blocksworld actions.

Note on Description: For a predicate of arity k, we associate a tuple of length k that specifies, for each predicate argument, how its value is obtained. Each entry in this tuple follows the same convention:

1. any integer ≥ 0: the predicate argument is bound to the corresponding explicit action argument (corresponding to an explicit variable in x),
2. -1: the predicate argument is not explicitly given and is instead implicitly inferred from the current state (corresponding to an implicit variable in z),
3. `None`: the predicate argument is a free variable that is not used to deterministically infer any implicit argument (corresponding to a free variable in y).

For example, $(relation\_at, (1, -1))$ specifies a binary predicate where the first argument refers to the second explicit action argument, while the second argument is implicitly inferred from

the state, corresponding to the start cell of the gripper. In contrast, $(relation\_at, (-1, 0))$ denotes a relation between an implicitly inferred object and the first explicit action argument, such as the cell associated with a target block. This notation directly corresponds to the *STRIPS*$^+$ formalism described in SYNTH [12] and is further discussed in the background section 2.4.

For all actions, "the cell after the action" (x_0) and the gripper (x_1) are given explicitly, serving as input to SYNTH. All other parameters are inferred based on relational constraints. A detailed examination reveals that SYNTH correctly reconstructs all relevant implicit arguments in the more complex actions. For the move, stack, and unstack actions, "the pre-action cell of the gripper" (x_2) is consistently identified. In addition, for stack and unstack, the block being held (x_3) as well as the block and corresponding cell directly beneath it (x_4 and x_5) are correctly inferred. This demonstrates that SYNTH can reliably capture both vertical and horizontal relational dependencies in the domain. For pickup and putdown, the pre-action cell (x_2) is also correctly identified. However, additional inferred arguments appear: in pickup, the cells x_4, x_5, x_6, and in putdown, x_3, x_5, x_6, correspond to free cells above the gripper. In the 5×3 grid used for Blocksworld, these cells are indeed empty, but they do not represent semantically relevant arguments for the action. This illustrates a minor over-reconstruction by SYNTH: the algorithm infers all cells that are uniquely determined under the current predicate representation, even if some are not strictly required for the action.

For Blocksworld SYNTH reliably reconstructs the key implicit arguments. For stack and unstack, it recovers both the manipulated block and the supporting block and cell. In pickup and putdown, minor over-reconstruction can occur when uniquely identifiable but semantically irrelevant objects (e.g., free cells above the gripper) are inferred. These extra arguments do not affect correctness but highlight that SYNTH may over-specify action parameters in configurations with predictable uniqueness.

Next, we now observe how these same mechannism behave in the standard Delivery domain. Table 5.7 summarizes this distinction for the actions move, load, and unload. It additionally reports the semantic interpretation of each argument.

Action	Parameter	Description	Argument Semantics
move	x0	given as input	Cell object of the truck *after* the action
	x1	given as input	Truck object
	x2	(relation_at, (1,-1))	Cell object of the truck *before* the action
load	x0	given as input	Cell object of the truck *after* the action
	x1	given as input	Truck object
	x2	(relation_at, (1,-1))	Cell object of the truck *before* the action
	x3	(relation_at, (-1,0))	Package object
unload	x0	given as input	Cell object of the truck *after* the action
	x1	given as input	Truck object
	x2	given as input	Package object
	x3	(relation_at, (1,-1))	Cell object of the truck *before* the action

Table 5.7: Parameter characterization and semantic interpretation of all delivery actions.

For all three actions, the pre-action cell of the truck (x_2 for move and load, x_3 for unload) can be uniquely reconstructed from the relational constraints. In addition, for the load action, the package object (x_3) is also recoverable. These are the only arguments in the standard Delivery domain that can be inferred from relational information and in each case the reconstruction is unambiguous. Consequently, SYNTH is able to correctly identify the truck's pre-action cell and the package in the load action as implicit arguments, while no other implicit arguments can be recovered without ambiguity.

The question of why the remaining arguments are ambiguous becomes particularly relevant for the unload action in scenarios with multiple trucks. When multiple trucks each carry a package, the package to unload cannot be uniquely determined from the predicates. Without a functional dependency between trucks and packages, SYNTH cannot assign packages to trucks, leaving the argument ambiguous. This is a natural consequence of indistinguishability in the predicate space, not an implementation error.

The Delivery domain is restricted to a single-truck scenario. Table 5.8 shows the unload action arguments recovered by SYNTH. With at most one loaded package at a time, the package argument (x_3) becomes unambiguous, enabling correct reconstruction. Ambiguities present in multi-truck scenarios disappear, similar to the Blocksworld domain where a single gripper ensures a unique object association.

The Delivery domain is restricted to a scenario with only a single truck. Table 5.8 lists only the arguments of the unload action, showing which arguments can be successfully recovered by SYNTH.

Action	Parameter	Description	Argument Semantics
unload	x0	given as input	Cell object of the truck *after* the action
	x1	given as input	Truck object
	x2	(relation_at, (1, -1))	Cell object of the truck *before* the action
	x3	(not-relation_horizontal, (None,-1))	Package object
		(not-relation_horizontal, (-1,None))	
		(not-relation_at, (-1,None))	

Table 5.8: Parameter characterization and semantic interpretation of the `unload` action with a single truck in the delivery domain.

It should be noted, however, that argument uniqueness can also depend on the structure and number of actions in the domain. One possible approach to resolving this issue would be a structural modification of the action definitions. Specifically, the move action could be split into two separate actions: one variant for when the truck has a loaded package and a second variant for when it does not. This would create actions of different arity—for example, a three-argument move action without a package and a four-argument variant with an explicit package argument. In such a model, the package would be explicitly bound during movement, allowing SYNTH to make a unique assignment even for the unload action. However, this approach requires a fundamental change in the domain modeling and is therefore not pursued further in this work.

Overall, SYNTH demonstrates strong performance in reconstructing semantically relevant implicit action arguments in both the Blocksworld and standard Delivery domains, provided that these arguments are uniquely identifiable through the available predicates. In Blocksworld, SYNTH consistently recovers the pre-action cell of the gripper and, in more complex actions (stack and unstack), also identifies the manipulated block and the supporting block and cell beneath it. Minor over-reconstruction can occur when uniquely identifiable but irrelevant objects are inferred, reflecting limitations of the predicate representation rather than the reconstruction method itself. In the Delivery domain, argument reconstruction is similarly reliable when uniqueness is guaranteed; the pre-action cell of the truck and, in the load action, the package are correctly inferred. Ambiguities arise only in multi-truck scenarios, where multiple packages satisfy the same relational constraints, preventing unique reconstruction. In single-truck settings, these ambiguities disappear, mirroring the uniqueness conditions observed in Blocksworld. In summary, SYNTH is robust in reconstructing implicit arguments under uniquely-determined conditions.

5.4.4 SIFT Results

In this section, we evaluate the Delivery domain extracted by SIFT domain. The goal is to analyze the correctness of features, parameters, preconditions, and effects, and to identify any

differences. Furthermore, we examine the behavior of SIFT when additional static objects, such as fences, are introduced.

For interpreting the features generated by SIFT, we map the types to the real objects in the *Delivery* domain:

Object Type	Delivery Domain	Description
Type0	cell	Grid cell in which a truck or package can be located
Type1	truck	Truck that transports packages
Type2	package	Package that can be transported

Table 5.9: Mapping of SIFT types to objects in the *Delivery* domain

In the following, we use the actual object names instead of the type names for better readability.

Predicates

SIFT cannot distinguish between True and False, as described in Section (2.5). In the following analysis of the mapping between SIFT predicates and the corresponding domain, we initially consider only the positive predicates from the SIFT domain.

SIFT Feature	SIFT Feature Arguments	Semantic Meaning	Semantic Equivalent
Feature_0_v0_s1	(?Arg0 - cell)	Cell is empty / available	clear(?c)
Feature_1_v0_s1	(?Arg0 - truck)	Truck is loaded	isLoaded(?t)
Feature_2_v0_s1	(?Arg0 - cell, ?Arg1 - truck)	Truck is at a cell	at(?c, ?t)
Feature_3_v0_s1	(?Arg0 - cell, ?Arg1 - package)	Package is at a cell	at(?c, ?p)
Feature_4_v0_s1	(?Arg0 - truck, ?Arg1 - package)	Truck carries a package	carrying(?t, ?p)
Adjacent	(?Arg0 - cell, ?Arg1 - cell)	Two cells are adjacent	adjacent(?c1, ?c2)

Table 5.10: Positive SIFT predicates and their equivalents in the Delivery domain.

As shown in Table 5.10, all ground-truth predicates are correctly matched by SIFT features. The static predicate *adjacent*, as described in Section 4.4, was additionally inserted manually. Furthermore, there are two *at* predicates because no aggregation of packages and trucks was performed for the types.

Action Parameters

The action parameters listed in Table 5.11 show the semantic correspondence between the parameters recognized by SIFT and the corresponding parameters in the Delivery domain.

Action	SIFT Arguments	Delivery Domain Arguments
move	?Arg0 - cell, ?Arg1 - truck, ?Arg2 - cell	?t - truck, ?x - cell, ?y - cell
load	?Arg0 - cell, ?Arg1 - truck, ?Arg2 - cell, ?Arg3 - package	?t - truck, ?p - package, ?x - cell, ?y - cell
unload	?Arg0 - cell, ?Arg1 - truck, ?Arg2 - package, ?Arg3 - cell	?t - truck, ?p - package, ?x - cell, ?y - cell

Table 5.11: Comparison of SIFT action parameters with the corresponding parameters in the Delivery domain.

As can be seen, the action parameters are semantically identical. They only differ in order. The individual parameters of the actions can be mapped from the SIFT domain to the Delivery domain as follows:

move: ?Arg0 → ?Arg2, ?Arg1 → ?Arg0, ?Arg2 → ?Arg1

load: ?Arg0 → ?Arg3, ?Arg1 → ?Arg0, ?Arg2 → ?Arg2, ?Arg3 → ?Arg1

unload: ?Arg0 → ?Arg3, ?Arg1 → ?Arg0, ?Arg2 → ?Arg1, ?Arg3 → ?Arg2

This mapping is derived directly from the domain generated by SYNTH. Each argument and its corresponding assignment are discussed in detail in Section 5.4.3.

Preconditions

Next, we examine the preconditions of the individual actions. Table 5.12 lists all relevant predicates in positive and negative form. The numbers in square brackets indicate which action parameters the predicate affects. Predicates with multiple arguments, such as (*at* ?c ?t) or (*carrying* ?t ?p), list all affected parameters. Here, we use the predicates from SIFT, but with the semantically equivalent names described in Table 5.10. For correct interpretation, the previously described mapping of SIFT action parameters to the Delivery domain is crucial, as the parameters are ordered differently in the actions. Only through this mapping can the affected arguments be correctly assigned and the preconditions properly interpreted.

Predicate	move		load		unload	
	SIFT	Delivery	SIFT	Delivery	SIFT	Delivery
(clear ?c1)	[0]	[2]	—	—	[0]	[3]
not (clear ?c1)	[2]	—	[2]	—	—	—
(isLoaded ?t)	—	—	—	—	[1]	[0]
not (isLoaded ?t)	—	—	[1]	[0]	—	—
(at ?c ?t)	[2,1]	[1,0]	[2,1]	[2,0]	[3,1]	[2,0]
not (at ?c ?t)	[0,1]	—	[0,1]	—	[0,1]	—
(at ?c ?p)	—	—	[0,3]	[3,1]	—	—
not (at ?c ?p)	—	—	—	—	[3,2]	—
(carrying ?t ?p)	—	—	—	—	[1,2]	[0,1]
not (carrying ?t ?p)	—	—	[1,3]	—	—	—
(adjacent ?x ?y)	[2,0]	[1,2]	[2,0]	[2,3]	[3,0]	[2,3]
(adjacent ?y ?x)	[0,2]	—	[0,2]	—	[0,3]	—

Table 5.12: Comparison of action preconditions between SIFT and the Delivery domain. Numbers in square brackets indicate the indices of action parameters involved in the predicate. A '—' denotes that the predicate is not present in the respective action.

Table 5.12 shows that SIFT uses both positive and negative predicates, whereas the delivery domain only uses negative predicates when necessary for semantic consistency—for example, to check before executing the load action whether the truck is already carrying a package. SIFT simply reveals more information without causing errors. Apart from that, all features match the ground-truth domain when the action parameter mapping is taken into account. Thus, the preconditions from SIFT are well covered.

Effects

Similarly to the preconditions, we next compare the effects of the actions in the SIFT-learned domain with those in the ground-truth domain. Table 5.13 shows the state changes caused by the actions. A positive sign (+) indicates predicates added to the state, while a negation symbol (¬) denotes predicates removed from the state. The numbers in square brackets indicate which action parameters are affected by each effect. Due to the different ordering of action parameters in both domains, the previously introduced mapping is necessary for a correct comparison of effects.

Predicate	move		load		unload	
	SIFT	Delivery	SIFT	Delivery	SIFT	Delivery
(clear ?c1)	+[2],¬[0]	+[1], ¬[2]	+[2]	+[2]	¬[0]	¬[3]
not (clear ?c1)	+[0], ¬[2]	—	¬[2]	—	+[0]	—
(isLoaded ?t)	—	—	+[1]	+[0]	¬[1]	¬[0]
not (isLoaded ?t)	—	—	¬[1]	—	+[1]	—
(at ?c ?t)	+[0,1], ¬[2,1]	+[2,0], ¬[1,0]	+[0,1], ¬[2,1]	+[3,0], ¬[2,0]	+[0,1], ¬[3,1]	+[3,0], ¬[2,0]
not (at ?c ?t)	+[2,1], ¬[0,1]	—	+[2,1], ¬[0,1]	—	+[3,1], ¬[0,1]	—
(at ?c ?p)	—	—	¬[0,3]	—	+[3,2]	+[2,1]
not (at ?c ?p)	—	—	+[0,3]	—	¬[3,2]	—
(carrying ?t ?p)	—	—	+[1,3]	—	¬[1,2]	¬[0,1]
not (carrying ?t ?p)	—	—	¬[1,3]	—	+[1,2]	—
(adjacent ?x ?y)	—	—	—	—	—	—
(adjacent ?y ?x)	—	—	—	—	—	—

Table 5.13: Comparison of action effects between SIFT and the Delivery domain. Numbers in square brackets indicate the indices of action parameters involved in the predicate. A '—' denotes that the predicate is not affected by the action.

As Table 5.13 shows, SIFT models the effects of actions completely and explicitly, specifying both positive and negative state changes. The delivery domain uses a more compact representation but describes the same semantic effects. Considering the action parameter mapping, the effects learned by SIFT match those of the delivery domain. Thus, SIFT correctly and consistently learns the effects of actions.

Handling Static Obstacles via Domain Extension

After analyzing the behavior of SIFT in the original Delivery domain, we now examine its handling of static obstacles. For this purpose, we extend the Delivery domain with fences, which block individual cells and thereby restrict the movement possibilities of the truck. The introduction of static obstacles directly affects the traces extracted by SIFT and the resulting symbolic representation. Cells blocked by fences cannot be entered by the truck and therefore do not appear in the extracted traces. Accordingly, SIFT completely ignores these cells, and they are not included in the initial instance of the generated planning problem. This applies both to individual blocked cells and to fully enclosed areas that are unreachable by the truck. Thus, only objects that actually appear in the traces are included in the initial instance. SIFT implicitly models only the reachable part of the environment, while static obstacles are not explicitly represented as objects or predicates in the learned domain.

The domain found by SIFT largely captures the semantic structure of the Delivery domain. The true features are correctly extracted, the action parameters match the ground truth, and the precondition and effect features mostly correspond to the real domain. Deviations mainly arise for static objects that do not appear in the observations.

5.4.5 Planning Results

We evaluate the quality of plans generated from the induced domains by comparing them against plans obtained from the handcrafted *Delivery* domain. For each experiment, a single execution trace was selected as the source trace. From this trace, a random initial state and a corresponding goal state were sampled, and a planning problem was constructed accordingly. The planner was allowed to run for up to one hour, and the best plan found within this time limit was used for evaluation.

Table 5.14 summarizes the results. For each planning problem, the table reports the plan length as well as the number of *move*, *load*, and *unload* actions. Each cell contains two values in the form *SIFT* | *Handcrafted*, allowing for a direct comparison between the induced domain and the handcrafted reference. The row index (*No.*) refers to the corresponding detailed plans listed in the appendix.

No.	Len	#Move	#Load	#Unload
1	29 | 29	16 | 18	6 | 5	7 | 6
2	37 | 37	24 | 24	7 | 7	6 | 6
3	32 | 32	22 | 18	4 | 6	6 | 8

Table 5.14: Comparison of plans generated by the induced SIFT model (left value) and the handcrafted *Delivery* domain (right value).

While SIFT frequently produces plans with the same overall length as the handcrafted domain, differences in action composition can be observed. In particular, deviations often manifest as additional or missing *load* and *unload* actions, even when the total plan length remains unchanged.

It is important to note that SIFT cannot execute all plans that are valid in the handcrafted *Delivery* domain. Although the induced domain supports both positive and negative predicates, only predicates for which SIFT has sufficient confidence are included in the generated problem instance. Consequently, some negative predicates that are implicitly assumed in the handcrafted domain may be missing from the initial state. When planning problems deviate from the original execution trace, this can result in violated preconditions and render certain handcrafted plans infeasible under the induced model. anning problems deviate from the original execution trace, this can result in violated preconditions and render certain handcrafted plans infeasible under the induced model.

Despite these limitations, the results demonstrate that plans generated by SIFT can be mapped to the ground-truth *Delivery* domain and correctly executed there.

6 Conclusion

This thesis addressed the challenge of learning symbolic planning domains directly from raw visual traces. A central research question was: *Which challenges arise when learning symbolic planning domains from visual sequences, and how can they be addressed?* The main difficulties lie in accurately detecting objects in each frame, maintaining consistent object identities across consecutive states, correctly inferring the arguments of actions, and extracting relational features to construct a coherent symbolic domain. Each of these steps is critical: errors in object detection or matching can propagate through the pipeline, misgrounding action arguments, while inconsistencies in object identities or relational features can compromise the resulting symbolic domain and any generated plans.

To tackle these challenges, we proposed a fully object-centric vision-to-planning pipeline that automatically induces complete PDDL domains and planning instances from sequences of images annotated only with action labels. The pipeline integrates multiple stages, including object detection, tracking objects across frames to maintain consistent identities, inferring implicit action arguments, and constructing a symbolic domain from relational features. By structuring the system around object-centric representations, the approach bridges perception and symbolic reasoning, enabling fully automated planning without requiring manually designed domain models.

Experimental evaluation on the Blocksworld and Delivery domains demonstrated the feasibility of this approach. Object extraction and tracking provide stable and consistent object traces, supporting reliable grounding of action arguments. The inferred action schemas and relational features allow the pipeline to reconstruct symbolic domains that are semantically equivalent to manually constructed reference domains. Plans generated from these domains are structurally consistent with reference plans, providing evidence for the validity and practical utility of the pipeline.

Despite these successes, the pipeline has several limitations. It currently relies on assumptions such as the restriction to grid-based domains (see Section 4), which constrain its applicability. Object detection forms a critical single point of failure, as downstream components cannot correct errors. Multi-object matching can recover missing links but cannot eliminate false positives, and edge-recovery methods are highly sensitive to the chosen cell representation. In SYNTH, inferring implicit action arguments remains challenging, with some ambiguities unresolved. Finally, SIFT currently struggles to generate complete domains for more complex

settings with many actions and parameters, and the computation time grows significantly in such cases.

6.1 Future Work

While the proposed pipeline demonstrates that symbolic planning domains can be learned from visual traces, its robustness and generalization remain limited. Future work should focus on improving object detection, resolving ambiguities in action argument inference, and extending the approach to handle unseen problem instances. In particular, an iterative pipeline that grounds states from single observations could increase reliability in more complex and dynamic environments. These directions provide a practical path toward more generalizable vision-to-planning systems.

Taken together, these improvements outline a practical path toward more robust and generalizable vision-to-planning systems, providing a foundation for future work on object-centric, learning-based planning pipelines.

A Appendix

A.1 Domains

A.1.1 Delivery Domain

```
(define (domain delivery)
  (:requirements :negative-preconditions :typing)
  (:types cell locatable package truck - object)
  (:predicates
    (at ?c - cell ?x - locatable) (clear ?c - cell) (adjacent ?c1 ?c2 - cell)
    (isLoaded ?t - truck) (carrying ?t - truck ?p - package)
  )
  (:action move
    :parameters (?t - truck ?from ?to - cell)
    :precondition (and (at ?from ?t) (adjacent ?from ?to) (clear ?to))
    :effect (and (not (at ?from ?t)) (not (clear ?to)) (at ?to ?t) (clear ?from))
  )
  (:action load
    :parameters (?t - truck ?p - package ?from ?to - cell)
    :precondition (and (at ?from ?t) (at ?to ?p) (adjacent ?from ?to) (not (isLoaded ?t)))
    :effect (and (not (at ?from ?t)) (not (at ?to ?p)) (isLoaded ?t) (carrying ?t ?p)
    (at ?to ?t) (clear ?from))
  )
  (:action unload
    :parameters (?t - truck ?p - package ?from ?to - cell)
    :precondition (and (at ?from ?t) (carrying ?t ?p) (clear ?to) (adjacent ?from ?to) (isLoaded ?t))
    :effect (and (not (at ?from ?t)) (not (carrying ?t ?p)) (not (clear ?to))
    (not (isLoaded ?t)) (at ?to ?t) (at ?from ?p))
  )
)
```

A.1.2 Delivery Domain with Fences

```
(define (domain delivery-fences)
  (:requirements :negative-preconditions :typing)
  (:types cell locatable package truck fence - object)
  (:predicates
    (at ?c - cell ?x - locatable) (clear ?c - cell) (adjacent ?c1 ?c2 - cell)
    (isLoaded ?t - truck) (carrying ?t - truck ?p - package)
  )
  (:action move
    :parameters (?t - truck ?from ?to - cell)
    :precondition (and (at ?from ?t) (adjacent ?from ?to) (clear ?to))
    :effect (and (not (at ?from ?t)) (not (clear ?to)) (at ?to ?t) (clear ?from))
  )
  (:action load
    :parameters (?t - truck ?p - package ?from ?to - cell)
    :precondition (and (at ?from ?t) (at ?to ?p) (adjacent ?from ?to) (not (isLoaded ?t)))
```

```
    :effect (and (not (at ?from ?t)) (not (at ?to ?p)) (isLoaded ?t)
    (carrying ?t ?p) (at ?to ?t) (clear ?from))
  )
  (:action unload
    :parameters (?t - truck ?p - package ?from ?to - cell)
    :precondition (and (at ?from ?t) (carrying ?t ?p) (clear ?to) (adjacent ?from ?to) (isLoaded ?t))
    :effect (and (not (at ?from ?t)) (not (carrying ?t ?p)) (not (clear ?to))
    (not (isLoaded ?t)) (at ?to ?t) (at ?from ?p))
  )
)
```

A.1.3 Blocksworld

```
(define (domain blocksworld)
  (:requirements :typing :negative-preconditions)
  (:types cell locatable block gripper - object)
  (:predicates
    (adjacent ?c1 ?c2 - cell) (at ?o - locatable ?c - cell)
    (on ?b1 ?b2 - block) (holding ?b - block) (free ?o - locatable)
  )
  (:action MOVE
    :parameters (?g - gripper ?c1 ?c2 - cell)
    :precondition (and (at ?g ?c1) (adjacent ?c1 ?c2))
    :effect (and (not (at ?g ?c1)) (at ?g ?c2))
  )
  (:action PICKUP
    :parameters (?g - gripper ?b - block ?c_from ?c_to - cell)
    :precondition (and (at ?g ?c_from) (at ?b ?c_from) (adjacent ?c_from ?c_to) (free ?g) (free ?b))
    :effect (and (holding ?b) (not (free ?g)) (not (free ?b)) (not (at ?b ?c_from))
    (not (at ?g ?c_from)) (at ?g ?c_to))
  )
  (:action PUTDOWN
    :parameters (?g - gripper ?b - block ?c_from ?c_to - cell)
    :precondition (and (holding ?b) (at ?g ?c_from) (adjacent ?c_from ?c_to))
    :effect (and (at ?b ?c_from) (free ?b) (free ?g) (not (holding ?b)) (not (at ?g ?c_from)) (at ?g ?c_to))
  )
  (:action UNSTACK
    :parameters (?g - gripper ?b ?b2 - block ?c_start ?c_block ?c_next - cell)
    :precondition (and (at ?g ?c_start) (at ?b2 ?c_block) (on ?b ?b2) (free ?b) (free ?g)
    (adjacent ?c_start ?c_block) (adjacent ?c_block ?c_next))
    :effect (and (holding ?b) (free ?b2) (not (on ?b ?b2)) (not (free ?g)) (not (free ?b))
    (not (at ?g ?c_start)) (not (at ?g ?c_block)) (at ?g ?c_next))
  )
  (:action STACK
    :parameters (?g - gripper ?b ?b2 - block ?c_start ?c_block ?c_next - cell)
    :precondition (and (holding ?b) (at ?g ?c_start) (at ?b2 ?c_block) (free ?b2)
    (adjacent ?c_start ?c_block) (adjacent ?c_block ?c_next))
    :effect (and (on ?b ?b2) (free ?b) (not (free ?b2)) (free ?g) (not (holding ?b))
    (not (at ?g ?c_start)) (not (at ?g ?c_block)) (at ?g ?c_next))
  )
)
```

A.1.4 Delivery Domain by SIFT

```
(define (domain xyz)
```

```
(:requirements :typing :strips)

(:types
 Type0 Type1 Type2 - object
)
(:predicates
 (Feature_0_v0_s0 ?Arg0 - Type0) (Feature_0_v0_s1 ?Arg0 - Type0)
 (Feature_1_v0_s0 ?Arg0 - Type1) (Feature_1_v0_s1 ?Arg0 - Type1)
 (Feature_2_v0_s0 ?Arg0 - Type0 ?Arg1 - Type1) (Feature_2_v0_s1 ?Arg0 - Type0 ?Arg1 - Type1)
 (Feature_3_v0_s0 ?Arg0 - Type0 ?Arg1 - Type2) (Feature_3_v0_s1 ?Arg0 - Type0 ?Arg1 - Type2)
 (Feature_4_v0_s0 ?Arg0 - Type1 ?Arg1 - Type2) (Feature_4_v0_s1 ?Arg0 - Type1 ?Arg1 - Type2)
 (Adjazent ?Arg0 - Type0 ?Arg1 - Type0)
)

(:action load
 :parameters (?Arg0 - Type0 ?Arg1 - Type1 ?Arg2 - Type0 ?Arg3 - Type2)
 :precondition (and
  (Feature_2_v0_s1 ?Arg2 ?Arg1) (Feature_4_v0_s0 ?Arg1 ?Arg3)
  (Feature_3_v0_s1 ?Arg0 ?Arg3) (Feature_1_v0_s0 ?Arg1)
  (Feature_2_v0_s0 ?Arg0 ?Arg1) (Feature_0_v0_s0 ?Arg2)
  (Adjazent ?Arg0 ?Arg2) (Adjazent ?Arg2 ?Arg0)
 )
 :effect (and
  (Feature_0_v0_s1 ?Arg2) (Feature_3_v0_s0 ?Arg0 ?Arg3)
  (Feature_2_v0_s0 ?Arg2 ?Arg1) (Feature_2_v0_s1 ?Arg0 ?Arg1)
  (Feature_4_v0_s1 ?Arg1 ?Arg3) (Feature_1_v0_s1 ?Arg1)
  (not (Feature_2_v0_s1 ?Arg2 ?Arg1)) (not (Feature_4_v0_s0 ?Arg1 ?Arg3))
  (not (Feature_3_v0_s1 ?Arg0 ?Arg3)) (not (Feature_1_v0_s0 ?Arg1))
  (not (Feature_2_v0_s0 ?Arg0 ?Arg1)) (not (Feature_0_v0_s0 ?Arg2))
 )
)
(:action move
 :parameters (?Arg0 - Type0 ?Arg1 - Type1 ?Arg2 - Type0)
 :precondition (and
  (Feature_2_v0_s0 ?Arg0 ?Arg1) (Feature_2_v0_s1 ?Arg2 ?Arg1)
  (Feature_0_v0_s1 ?Arg0) (Feature_0_v0_s0 ?Arg2)
  (Adjazent ?Arg0 ?Arg2) (Adjazent ?Arg2 ?Arg0)
 )
 :effect (and
  (Feature_2_v0_s1 ?Arg0 ?Arg1) (Feature_0_v0_s0 ?Arg0)
  (Feature_0_v0_s1 ?Arg2) (Feature_2_v0_s0 ?Arg2 ?Arg1)
  (not (Feature_2_v0_s0 ?Arg0 ?Arg1)) (not (Feature_2_v0_s1 ?Arg2 ?Arg1))
  (not (Feature_0_v0_s1 ?Arg0)) (not (Feature_0_v0_s0 ?Arg2))
 )
)
(:action unload
 :parameters (?Arg0 - Type0 ?Arg1 - Type1 ?Arg2 - Type2 ?Arg3 - Type0)
 :precondition (and
  (Feature_3_v0_s0 ?Arg3 ?Arg2) (Feature_4_v0_s1 ?Arg1 ?Arg2)
  (Feature_2_v0_s0 ?Arg0 ?Arg1) (Feature_2_v0_s1 ?Arg3 ?Arg1)
  (Feature_1_v0_s1 ?Arg1) (Feature_0_v0_s1 ?Arg0)
  (Adjazent ?Arg0 ?Arg3) (Adjazent ?Arg3 ?Arg0)
 )
 :effect (and
  (Feature_3_v0_s1 ?Arg3 ?Arg2) (Feature_2_v0_s1 ?Arg0 ?Arg1)
  (Feature_1_v0_s0 ?Arg1) (Feature_4_v0_s0 ?Arg1 ?Arg2)
```

```
   (Feature_0_v0_s0 ?Arg0) (Feature_2_v0_s0 ?Arg3 ?Arg1)
   (not (Feature_3_v0_s0 ?Arg3 ?Arg2)) (not (Feature_4_v0_s1 ?Arg1 ?Arg2))
   (not (Feature_2_v0_s0 ?Arg0 ?Arg1)) (not (Feature_2_v0_s1 ?Arg3 ?Arg1))
   (not (Feature_1_v0_s1 ?Arg1)) (not (Feature_0_v0_s1 ?Arg0))
  )
 )
)
```

A.2 Comparison of Plans

The following tables presents the plans, showing actions generated by the SIFT-induced model alongside the corresponding actions with interpretable *Delivery* domain labels. Each row corresponds to a different plan variant as listed in Table 5.14.

Plan Details for No. 1

#	SIFT Domain Plan	Delivery Domain Plan
1	unload(type0_obj33, type1_obj7, type2_obj2, type0_obj32)	load(t4, p3, c29, c30)
2	load(type0_obj30, type1_obj4, type0_obj29, type2_obj3)	move(t4, c30, c31)
3	move(type0_obj31, type1_obj4, type0_obj30)	move(t7, c32, c27)
4	unload(type0_obj26, type1_obj4, type2_obj3, type0_obj31)	move(t7, c27, c26)
5	move(type0_obj21, type1_obj4, type0_obj26)	unload(t4, p3, c31, c32)
6	load(type0_obj16, type1_obj4, type0_obj21, type2_obj5)	move(t4, c32, c33)
7	move(type0_obj17, type1_obj4, type0_obj16)	move(t7, c26, c21)
8	move(type0_obj18, type1_obj4, type0_obj17)	unload(t7, p2, c21, c20)
9	load(type0_obj28, type1_obj7, type0_obj33, type2_obj1)	move(t7, c20, c15)
10	move(type0_obj33, type1_obj7, type0_obj28)	load(t7, p5, c15, c16)
11	unload(type0_obj28, type1_obj7, type2_obj1, type0_obj33)	move(t7, c16, c17)
12	move(type0_obj23, type1_obj4, type0_obj18)	move(t7, c17, c22)
13	unload(type0_obj18, type1_obj4, type2_obj5, type0_obj23)	unload(t7, p5, c22, c17)
14	load(type0_obj23, type1_obj7, type0_obj28, type2_obj5)	load(t7, p6, c17, c12)
15	move(type0_obj28, type1_obj7, type0_obj23)	move(t7, c12, c11)
16	unload(type0_obj27, type1_obj7, type2_obj5, type0_obj28)	move(t7, c11, c10)
17	load(type0_obj32, type1_obj7, type0_obj27, type2_obj2)	move(t7, c10, c15)
18	move(type0_obj27, type1_obj7, type0_obj32)	move(t7, c15, c20)
19	move(type0_obj22, type1_obj7, type0_obj27)	move(t7, c20, c25)
20	move(type0_obj21, type1_obj7, type0_obj22)	unload(t7, p6, c25, c24)
21	unload(type0_obj16, type1_obj7, type2_obj2, type0_obj21)	load(t4, p1, c33, c28)
22	move(type0_obj11, type1_obj7, type0_obj16)	move(t4, c28, c33)
23	load(type0_obj12, type1_obj7, type0_obj11, type2_obj6)	unload(t4, p1, c33, c28)
24	move(type0_obj17, type1_obj7, type0_obj12)	move(t4, c28, c23)
25	move(type0_obj22, type1_obj7, type0_obj17)	load(t4, p5, c23, c22)
26	move(type0_obj27, type1_obj7, type0_obj22)	move(t4, c22, c23)
27	move(type0_obj26, type1_obj7, type0_obj27)	move(t4, c23, c28)
28	move(type0_obj25, type1_obj7, type0_obj26)	unload(t4, p5, c28, c23)
29	unload(type0_obj24, type1_obj7, type2_obj6, type0_obj25)	move(t4, c23, c18)

Table A.1: Comparison of the generated plan for Trace 1 using SIFT-induced object labels (left) and interpretable *Delivery* domain labels (right).

Plan Details for No. 2

#	SIFT Object Labels	Interpreted Delivery Labels
1	load(type0_obj20, type1_obj7, type0_obj15, type2_obj6)	load(t7, p6, c15, c20)
2	move(type0_obj15, type1_obj7, type0_obj20)	move(t7, c20, c15)
3	move(type0_obj10, type1_obj7, type0_obj15)	move(t7, c15, c16)
4	move(type0_obj11, type1_obj7, type0_obj10)	move(t7, c16, c11)
5	unload(type0_obj12, type1_obj7, type2_obj6, type0_obj11)	load(t4, p5, c27, c28)
6	load(type0_obj13, type1_obj7, type0_obj12, type2_obj8)	move(t4, c28, c27)
7	load(type0_obj28, type1_obj4, type0_obj27, type2_obj5)	move(t4, c27, c26)
8	move(type0_obj27, type1_obj4, type0_obj28)	unload(t7, p6, c11, c12)
9	unload(type0_obj32, type1_obj4, type2_obj5, type0_obj27)	move(t4, c26, c25)
10	load(type0_obj33, type1_obj4, type0_obj32, type2_obj1)	move(t4, c25, c30)
11	move(type0_obj18, type1_obj7, type0_obj13)	unload(t4, p5, c30, c25)
12	move(type0_obj23, type1_obj7, type0_obj18)	load(t7, p8, c12, c13)
13	move(type0_obj32, type1_obj4, type0_obj33)	move(t7, c13, c18)
14	move(type0_obj28, type1_obj7, type0_obj23)	move(t7, c18, c23)
15	move(type0_obj33, type1_obj7, type0_obj28)	unload(t7, p8, c23, c28)
16	unload(type0_obj28, type1_obj7, type2_obj8, type0_obj33)	move(t4, c25, c26)
17	load(type0_obj27, type1_obj7, type0_obj28, type2_obj5)	load(t4, p3, c26, c31)
18	move(type0_obj26, type1_obj7, type0_obj27)	move(t4, c31, c32)
19	move(type0_obj27, type1_obj4, type0_obj32)	move(t4, c32, c27)
20	move(type0_obj25, type1_obj7, type0_obj26)	move(t4, c27, c22)
21	move(type0_obj30, type1_obj7, type0_obj25)	move(t4, c22, c17)
22	unload(type0_obj25, type1_obj7, type2_obj5, type0_obj30)	unload(t4, p3, c17, c22)
23	move(type0_obj26, type1_obj7, type0_obj25)	load(t7, p1, c28, c33)
24	load(type0_obj21, type1_obj7, type0_obj26, type2_obj2)	move(t7, c33, c32)
25	move(type0_obj26, type1_obj4, type0_obj27)	move(t7, c32, c31)
26	move(type0_obj25, type1_obj4, type0_obj26)	load(t4, p8, c22, c23)
27	move(type0_obj24, type1_obj4, type0_obj25)	move(t4, c23, c28)
28	unload(type0_obj25, type1_obj4, type2_obj1, type0_obj24)	move(t4, c28, c33)
29	move(type0_obj26, type1_obj4, type0_obj25)	unload(t4, p8, c33, c28)
30	load(type0_obj31, type1_obj4, type0_obj26, type2_obj3)	move(t7, c31, c26)
31	move(type0_obj26, type1_obj7, type0_obj21)	move(t7, c26, c25)
32	move(type0_obj32, type1_obj4, type0_obj31)	move(t7, c25, c24)
33	move(type0_obj27, type1_obj4, type0_obj32)	unload(t7, p1, c24, c25)
34	move(type0_obj22, type1_obj4, type0_obj27)	move(t7, c25, c26)
35	move(type0_obj17, type1_obj4, type0_obj22)	move(t4, c28, c27)
36	unload(type0_obj22, type1_obj4, type2_obj3, type0_obj17)	load(t7, p2, c26, c21)
37	move(type0_obj27, type1_obj4, type0_obj22)	move(t7, c21, c26)

Table A.2: Comparison of the generated plan for Trace 2 using SIFT-induced object labels (left) and interpretable *Delivery* domain labels (right).

Plan Details for No. 3

#	SIFT Domain Plan	Delivery Domain Plan
1	unload(type0_obj31, type1_obj4, type2_obj2, type0_obj26)	unload(t4, p2, c26, c27)
2	move(type0_obj19, type1_obj7, type0_obj24)	move(t4, c27, c28)
3	move(type0_obj32, type1_obj4, type0_obj31)	move(t7, c24, c19)
4	load(type0_obj33, type1_obj4, type0_obj32, type2_obj1)	load(t4, p1, c28, c33)
5	move(type0_obj32, type1_obj4, type0_obj33)	move(t4, c33, c32)
6	move(type0_obj31, type1_obj4, type0_obj32)	move(t4, c32, c31)
7	move(type0_obj30, type1_obj4, type0_obj31)	move(t4, c31, c30)
8	move(type0_obj29, type1_obj4, type0_obj30)	move(t4, c30, c29)
9	move(type0_obj24, type1_obj4, type0_obj29)	move(t4, c29, c24)
10	unload(type0_obj25, type1_obj4, type2_obj1, type0_obj24)	unload(t4, p1, c24, c25)
11	load(type0_obj20, type1_obj4, type0_obj25, type2_obj6)	load(t4, p6, c25, c20)
12	move(type0_obj15, type1_obj4, type0_obj20)	move(t4, c20, c21)
13	unload(type0_obj20, type1_obj7, type2_obj3, type0_obj19)	unload(t7, p3, c19, c20)
14	move(type0_obj10, type1_obj4, type0_obj15)	unload(t4, p6, c21, c16)
15	move(type0_obj11, type1_obj4, type0_obj10)	load(t7, p6, c20, c21)
16	move(type0_obj12, type1_obj4, type0_obj11)	move(t7, c21, c22)
17	unload(type0_obj17, type1_obj4, type2_obj6, type0_obj12)	move(t7, c22, c17)
18	load(type0_obj18, type1_obj4, type0_obj17, type2_obj5)	move(t7, c17, c12)
19	move(type0_obj21, type1_obj7, type0_obj20)	unload(t7, p6, c12, c17)
20	move(type0_obj22, type1_obj7, type0_obj21)	load(t7, p5, c17, c18)
21	move(type0_obj23, type1_obj7, type0_obj22)	move(t7, c18, c17)
22	move(type0_obj17, type1_obj4, type0_obj18)	unload(t7, p5, c17, c18)
23	move(type0_obj16, type1_obj4, type0_obj17)	load(t7, p8, c18, c13)
24	move(type0_obj15, type1_obj4, type0_obj16)	move(t7, c13, c18)
25	move(type0_obj20, type1_obj4, type0_obj15)	move(t7, c18, c23)
26	unload(type0_obj15, type1_obj4, type2_obj5, type0_obj20)	move(t7, c23, c28)
27	move(type0_obj18, type1_obj7, type0_obj23)	unload(t7, p8, c28, c33)
28	load(type0_obj13, type1_obj7, type0_obj18, type2_obj8)	load(t4, p5, c16, c17)
29	move(type0_obj18, type1_obj7, type0_obj13)	move(t4, c17, c16)
30	move(type0_obj23, type1_obj7, type0_obj18)	move(t4, c16, c15)
31	move(type0_obj28, type1_obj7, type0_obj23)	move(t4, c15, c20)
32	unload(type0_obj33, type1_obj7, type2_obj8, type0_obj28)	unload(t4, p5, c20, c15)

Table A.3: Comparison of the generated plan for Trace 1 (No. 2 in Table 5.14) using SIFT-induced object labels (left) and interpretable *Delivery* domain labels (right).

List of Acronyms

AI	Artificial Intelligence
AMA	Action Model Acquisition
CNN	Convolutional Neural Network
IoU	Intersection-over-Union
MOT	Multi-Object Tracking
PAM	Probabilistic Action Model
PDDL	Planning Domain Definition Language
ReLU	Rectified Linear Unit
SAE	State Autoencoder
SGG	Scene Graph Generation
STRIPS	Stanford Research Institute Problem Solver
VAE	Variational Autoencoder

List of Symbols

Object Detection

f_i^l	Activation of the i-th neuron in layer l
R_i^l	Gradient/relevance signal received by the i-th neuron in layer l during backward pass
f^{out}	Selected network output used to compute the gradient
$I(x, y)$	Input image used for object detection and saliency computation
$I'(x, y)$	Masked saliency image highlighting relevant object regions
T_p	Threshold value defined as the p-th percentile of saliency map values
b	Bounding box enclosing an object region in the image
$M(x, y)$	Binary mask applied to the saliency image to isolate high-activation regions
C_k	Connected component of adjacent high-activation pixels in the masked saliency image
p_j^i	The j-th image in the i-th trace
p_{j+1}^i	The $(j+1)$-th image in the i-th trace
a_p	Action applied between images p_j^i and p_{j+1}^i in trace s_i
$\left(p_j^i, a_p, p_{j+1}^i\right)$	Image-action-image triplet $\left(p_j^i, a_p, p_{j+1}^i\right)$ in trace s_i

Multi-Object Tracking (MOT)

o_i^j	Object i in frame j
O^j	Set of objects in frame j
C	Cost matrix between objects in consecutive frames
T	Tracklet of a single object across consecutive frames
$E^{j,j+1}$	Local match between objects in consecutive frames
$E_{a_p}^{\text{amb}}$	Ambiguous edge set for a given action
$\{U_1, \ldots, U_m\}$	Set of clusters of visually similar objects
$\left(u_i^j, u_k^{j+1}\right)$	Cluster-level pair of objects for action a_p

Planning

F	Set of all grounded atoms (boolean variables)
I	Initial state of the planning problem
O	Set of operators (actions) with preconditions, add and delete effects
G	Goal description of the planning problem
P	Lifted STRIPS planning problem represented as a tuple of atoms, initial state, operators, and goal

(*s*,*a*,*s'*)	Observed transition consisting of a predecessor state, an action, and a successor state
a(o)	(*a(o)*) Grounded action applied to objects
a	Action executed in a state
s	State in a state-action-state transition
s'	Resulting successor state after executing an action

SYNTH

$STRIPS^+$	($STRIPS^+$) Extended STRIPS formalism introduced by SYNTH
x	explicit action arguments
y	free variables
z	implicit variables
Q(*x*,*y*,*z*)	Binding precondition conjunctive query over explicit, free, and implicit variables

SIFT

$f = \langle k, B \rangle$	Feature defined by arity k and a non-empty set of action patterns B
$a[t]$	Typed action pattern with action a and type tuple t
D_T	Learned planning domain induced from trace T
I_T	Learned initial state induced from trace T
$P_T = \langle D_T, I_T \rangle$	Learned planning problem consisting of learned domain D_T and initial state I_T

List of Figures

List of Tables

List of References

[1] M. Asai and A. Fukunaga, *Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary*, 2017. arXiv: 1705.00154 [cs.AI]. [Online]. Available: https://arxiv.org/abs/1705.00154.

[2] P. Community, *Pygame: Python Game Development Library*, https://www.pygame.org, Accessed: 2025-12-08, 2000–2025.

[3] S. N. Cresswell, T. L. McCluskey, and M. M. West, "Acquiring planning domain models using LOCM," *The Knowledge Engineering Review*, vol. 28, no. 2, pp. 195–213, Jun. 2013, ISSN: 0269-8889, 1469-8005. DOI: 10.1017/S0269888912000422. [Online]. Available: https://www.cambridge.org/core/journals/knowledge-engineering-review/article/acquiring-planning-domain-models-using-locm/139F7A0986CD481F88A778B08891DC49# (visited on 07/10/2025).

[4] M. Cuturi, "Sinkhorn distances: Lightspeed computation of optimal transport," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26, Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2013/file/af21d0c97db2e27e13572cbf59eb343d-Paper.pdf.

[5] S. Feng, H. Mostafa, M. Nassar, S. Majumdar, and S. Tripathi, "Exploiting long-term dependencies for generating dynamic scene graphs," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, Jan. 2023, pp. 5130–5139.

[6] M. Fox and D. Long, "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, Dec. 1, 2003, ISSN: 1076-9757. DOI: 10.1613/jair.1129. [Online]. Available: https://jair.org/index.php/jair/article/view/10352 (visited on 07/10/2025).

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Clingo = asp + control: Preliminary report*, 2014. arXiv: 1405.3694 [cs.PL]. [Online]. Available: https://arxiv.org/abs/1405.3694.

[8] H. Geffner, "Computational models of planning," *WIREs Cognitive Science*, vol. 4, no. 4, pp. 341–356, 2013, ISSN: 1939-5086. DOI: 10.1002/wcs.1233. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcs.1233 (visited on 06/27/2025).

[9] J. Gösgens, N. Jansen, and H. Geffner. “Learning Lifted STRIPS Models from Action Traces Alone: A Simple, General, and Scalable Solution.” arXiv: 2411.14995 [cs]. (May 2, 2025), [Online]. Available: http://arxiv.org/abs/2411.14995 (visited on 06/22/2025), pre-published.

[10] K. He, X. Zhang, S. Ren, and J. Sun. “Deep Residual Learning for Image Recognition.” arXiv: 1512.03385 [cs]. (Dec. 10, 2015), [Online]. Available: http://arxiv.org/abs/1512.03385 (visited on 06/22/2025), pre-published.

[11] M. Helmert, “The fast downward planning system,” *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, Jul. 2006, ISSN: 1076-9757. DOI: 10.1613/jair.1705. [Online]. Available: http://dx.doi.org/10.1613/jair.1705.

[12] N. Jansen, J. Gösgens, and H. Geffner, “Learning lifted action models from traces of incomplete actions and states,” *CoRR*, vol. abs/2508.21449, 2025. DOI: 10.48550/ARXIV.2508.21449. arXiv: 2508.21449. [Online]. Available: https://doi.org/10.48550/arXiv.2508.21449.

[13] J. Ji, R. Krishna, L. Fei-Fei, and J. C. Niebles, “Action genome: Actions as compositions of spatio-temporal scene graphs,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020.

[14] P. Kochakarn, D. D. Martini, D. Omeiza, and L. Kunze, *Explainable action prediction through self-supervision on scene graphs*, 2023. arXiv: 2302.03477 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2302.03477.

[15] H. W. Kuhn, “The Hungarian method for the assignment problem,” *Naval Research Logistics Quarterly*, vol. 2, no. 1–2, pp. 83–97, 1955, ISSN: 1931-9193. DOI: 10.1002/nav.3800020109. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109 (visited on 07/01/2025).

[16] H. Li, G. Zhu, L. Zhang, Y. Jiang, Y. Dang, H. Hou, P. Shen, X. Zhao, S. A. A. Shah, and M. Bennamoun, “Scene graph generation: A comprehensive survey,” *Neurocomputing*, vol. 566, p. 127 052, 2024, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2023.127052. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092523122301175X.

[17] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, *PDDL - The Planning Domain Definition Language*, 1998.

[18] K. Soomro and M. Shah, “Unsupervised action discovery and localization in videos,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[19] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. “Striving for Simplicity: The All Convolutional Net.” arXiv: 1412.6806 [cs]. (Apr. 13, 2015), [Online]. Available: http://arxiv.org/abs/1412.6806 (visited on 06/22/2025), pre-published.

[20] S. Suzuki and K. be, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, Apr. 1, 1985, ISSN: 0734-189X. DOI: 10.1016/0734-189X(85)90016-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0734189X85900167 (visited on 06/26/2025).

[21] K. Xi, S. Gould, and S. Thiébaux, "Neuro-symbolic learning of lifted action models from visual traces," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 34, 2024, pp. 653–662.

[22] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks." arXiv: 1311.2901 [cs]. (Nov. 28, 2013), [Online]. Available: http://arxiv.org/abs/1311.2901 (visited on 06/23/2025), pre-published.