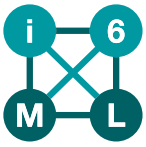

Chair of Machine Learning and Reasoning (Computer Science 6)

RWTH AACHEN UNIVERSITY

This thesis was submitted to the Chair of Machine Learning and Reasoning.

Generalized Planning with Lifted Temporal Goals

Master Thesis

Presented by

Chanjuan Huang
441933

Supervised by Till Hoffman, M.Sc.

1st Examiner Prof. Hector Geffner, Ph.D.

2nd Examiner Jun.-Prof. Dr. Christopher Morris

Aachen, July 6, 2026

Abstract

Classical AI planning produces plans for individual problem instances, but in practice, applications often require generalized policies that solve entire classes of problems regardless of the number of objects. Traditional generalized planning deals with only reachability goals but it becomes significantly harder when the objective involves extended temporal constraints. This paper focuses on temporally extended goals expressed in first-order linear temporal logic on finite traces (FO-LTL$_f$). Such goals, like "eventually transport all passengers to their destinations" or "keep all blocks clear until they are placed on the table", require policies that interleave multi-phase behaviors across a variable number of objects.

In order to achieve this, we investigated two approaches. The first, ASP-based policy synthesis, generates description logic features from a product automaton of PDDL states and NFA tracking states, then uses answer set programming to synthesize a policy that distinguishes states by feature values. This approach achieves provably correct policies for simple goal structures, but suffers from ASP grounding explosion that makes complex temporal goals intractable.

Our second approach, lifted rule extraction, overcomes this barrier. It extracts first-order policy rules directly from optimal plans in the product automaton. Each rule is a conjunction of predicates with variables that bind to specific objects at execution time, enabling action-argument selection. The trade off of this approach is that is is heavily dependent on the training set construction, and does not guarantee completeness. A key innovation is the use of existential context variables from NFA tracking that link navigation actions to goal-directed motivation without specifying which object to serve next. We evaluate the second approach across 11 planning domains and 47 temporal goals spanning universal and existential quantification, nested quantifiers, and complex temporal operators. The lifted rule approach achieves 100 % generalization on 34 out of 47 goals and solves 88.7 % of test instances (385/434), with training times of 0.6–207.3 seconds and memory usage under 1.5 GB. This is orders of magnitude faster and lighter than the ASP approach, which fails on complex temporal goals.

Contents

1 Introduction

1.1 Motivation

Classical AI planning addresses the problem of finding a sequence of actions that transforms an initial state into a goal state [12]. A domain description specifies the available actions and their effects, while a problem instance defines the initial state and goal. Given these inputs, a planner produces a plan, which is a specific sequence of actions for that specific instance. However, practical applications require a more general solution: a policy, there has been a lot of research in how a general policy should be presented, some presented the policies as algorithms [18], some even use deep neural networks as generalized reactive policies. [13] here we take the explicit rule set language based on concepts and roles [16] that specifies what to do in any state, for any problem instance in a given domain, regardless of the number of objects involved.

Consider an elevator controller. A classical planner might produce a plan to serve three specific passengers on four floors. The required solution is a general strategy: "go to each passenger's origin floor, board them, navigate to their destination, and depart." This strategy must generalize to arbitrary numbers of passengers and floors. Learning such generalized policies is the goal of generalized planning.

Beyond simple reachability goals that require reaching a specific state s_g, many tasks require temporally extended goals that constrain not just the final state but the whole execution trace. Examples include:

- $\forall p.\ \mathbf{F}(\mathit{served}(p))$. "Eventually serve every passenger." This is a liveness property requiring that every passenger eventually receives service.
- $\forall x.\ (\mathit{clear}(x)\ \mathbf{U}\ \mathit{ontable}(x))$. "Each block must remain clear until it is placed on the table." This combines safety and progress requirements.
- $\forall p.\ \exists a.\ \mathbf{F}(\mathit{in}(p, a))$. "Every person eventually boards some aircraft." This demonstrates nested quantification over multiple object types.

These goals can be expressed in first-order linear temporal logic on finite traces (FO-$\mathrm{LTL_f}$), which extends linear temporal logic on finite traces ($\mathrm{LTL_f}$) with universal and existential quantifiers over domain objects. A generalized policy for such goals must:

1. **Track temporal progress** per object. The policy must monitor which passengers have been served and which blocks are still waiting.

2. **Coordinate multi-phase behaviors.** For example, serving each passenger requires boarding, navigating to the destination, and departing in sequence.
3. **Generalize across instance sizes.** The same policy must work equally well for 2 passengers or 20 passengers.

Definition 1.1 (Generalized Policy Learning for FO-LTL$_f$ Goals)**.** *Given:*

- *A Planning Domain Definition Language (PDDL) domain $\mathcal{D}$ defining action schemas and predicates,*
- *A set of small training instances $\{P_1, \ldots, P_k\}$ with increasing numbers of objects,*
- *A first-order temporal goal φ in extended prenex normal form (EPNF),*

learn a generalized policy π such that for any unseen problem instance P over $\mathcal{D}$, executing π from the initial state of P produces a finite trace τ that satisfies φ.

The key challenges are:

1. **Quantified goals.** The goal formula contains quantifiers over a variable number of domain objects. Universal quantifiers require properties to hold for all objects, while existential quantifiers allow choice among alternatives.
2. **Multi-phase objectives.** Serving a passenger requires boarding, navigating, and departing. These three phases must be completed in sequence for each passenger independently.
3. **Action-argument selection.** The policy must decide not only which action schema to apply but also which specific objects to use as arguments. For example, it must select both the action "board" and the specific passenger and floor, such as "board passenger p_3 at floor f_2".

1.2 Contributions

This thesis makes three contributions:

1. **Systematic evaluation of answer set programming (ASP)-based policy synthesis for FO-LTL$_f$ goals.** We apply the DL-plan feature framework [8, 9] to the product automaton of PDDL state spaces and nondeterministic finite automaton (NFA) temporal tracking. We identify that while this approach achieves provably correct policies on simple goals, it suffers from ASP grounding explosion that makes complex multi-phase temporal goals intractable.
2. **Lifted rule extraction from optimal plans.** We propose a novel method that extracts first-order policy rules directly from breadth-first search (BFS)-optimal plans in the product automaton. Rules use variables that bind to specific objects at execution time, solving the action-argument problem. Rules also use existential

context variables, which encode why a navigation action is needed without specifying which object to serve next.

3. **Comprehensive experimental evaluation.** We evaluate the ASP approach on 11 goals where the product automaton remains tractable, demonstrating its ability to synthesize provably correct policies on simple temporal structures. We then evaluate the lifted rule approach across 11 planning domains and 47 temporal goals spanning universal quantification ($\forall$), existential quantification ($\exists$), nested quantifiers, and complex temporal operators (eventually, until, next, globally). The lifted rule approach achieves 100 % test generalization on 34 out of 47 goals and solves 385 out of 434 test instances (88.7 % overall), learning policies of 1–25 rules with training times ranging from 0.6 to 207.3 seconds. The two approaches use different training sets and are not directly comparable, but together they demonstrate the trade-off between provable correctness on simple goals and scalability to complex multi-phase objectives.

The remainder of this thesis is organized as follows. Chapter 2 introduces the necessary background on classical planning, temporal logic, NFA construction, and generalized planning. Chapter 3 describes the system architecture and the five-phase pipeline shared by both approaches. Chapter 4 presents the ASP-based approach, its experimental results, and a detailed analysis of its failure modes. Chapter 5 describes the lifted rule extraction method, its execution semantics, and its experimental results. Chapter 6 provides a comparative analysis, discusses limitations, and concludes. Chapter 6 concludes with a summary and directions for future work.

2 Background

This chapter discusses the background underlying this thesis, which includes classical planning and PDDL, temporally extended goals expressed in LTL_f and its first-order extension FO-LTL_f, automata-based temporal tracking, generalized planning with description logic (DL) features, and related work.

2.1 Classical Planning and PDDL

Definition 2.1 (PDDL Planning Problem)**.** *A PDDL planning problem is a tuple* $\langle \mathcal{D}, P \rangle$ *where:*

- $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{T} \rangle$ *is a domain with predicates* $\mathcal{P}$*, action schemas* $\mathcal{A}$*, and types* $\mathcal{T}$*.*
- $P = \langle \mathcal{O}, s_0, g \rangle$ *is a problem instance with objects* $\mathcal{O}$*, initial state* s_0*, and goal condition* g*.*

A state s is a set of ground atoms (instantiated predicates) that are true. An action schema $a(\bar{x})$ has typed parameters $\bar{x}$, preconditions $\text{pre}(a)$, and effects $\text{eff}(a)$. A ground action is obtained by substituting concrete objects for the parameters. A ground action a is applicable in state s if $\text{pre}(a) \subseteq s$. Applying a in s yields successor state $s' = (s \setminus \text{del}(a)) \cup \text{add}(a)$.

2.2 Temporally Extended Goals

Classical planning goals specify a target state. Temporally extended goals additionally constrain the path taken to reach it. We use LTL_f [7], which interprets temporal operators over finite execution traces.

2.2.1 Linear Temporal Logic on Finite Traces

Definition 2.2 (LTL_f Syntax)**.** *Given a set of propositional atoms AP,* LTL_f *formulas are defined by the grammar:*

$$\varphi ::= p \mid \mathit{final} \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi \,\mathbf{U}\, \psi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi$$

where $p \in AP$*, final is a special atom true only at the end of a finite trace,* $\mathbf{X}$ *is "next,"* $\mathbf{U}$ *is "until,"* $\mathbf{F} \equiv \top \,\mathbf{U}\, \varphi$ *is "eventually," and* $\mathbf{G} \equiv \neg\mathbf{F}\neg\varphi$ *is "globally/always."*

Definition 2.3 (LTL_f Semantics)**.** *A finite trace* $\tau = s_0, s_1, \ldots, s_n$ *satisfies* φ *(written* $\tau \vDash \varphi$*) according to:*

- $\tau, i \vDash p$ *iff* $p \in s_i$.
- $\tau, i \vDash$ *final iff* $i = n$ *(true only at the end of the trace).*
- $\tau, i \vDash \neg\varphi$ *iff* $\tau, i \nvDash \varphi$.
- $\tau, i \vDash \varphi \wedge \psi$ *iff* $\tau, i \vDash \varphi$ *and* $\tau, i \vDash \psi$.
- $\tau, i \vDash \mathbf{X}\varphi$ *iff* $i < n$ *and* $\tau, i{+}1 \vDash \varphi$.
- $\tau, i \vDash \varphi\, \mathbf{U}\, \psi$ *iff* $\exists j \geq i$ *such that* $\tau, j \vDash \psi$ *and* $\forall k \in [i, j) \colon \tau, k \vDash \varphi$.
- $\tau, i \vDash \mathbf{F}\varphi$ *iff* $\exists j \geq i$ *such that* $\tau, j \vDash \varphi$.
- $\tau, i \vDash \mathbf{G}\varphi$ *iff* $\forall j \geq i \colon \tau, j \vDash \varphi$.

A trace satisfies the formula if $\tau, 0 \vDash \varphi$.

2.2.2 First-Order Extension

To express goals over variable numbers of objects, we extend $\mathrm{LTL_f}$ with first-order quantifiers:

$$\varphi ::= \cdots \mid \forall x{:}\tau.\ \varphi \mid \exists x{:}\tau.\ \varphi$$

where x is a variable and τ is a type. The quantifiers range over all objects of type τ in the current problem instance.

Definition 2.4 (Extended Prenex Normal Form)**.** *An FO-LTL$_f$ formula is in EPNF if all quantifiers appear at the outermost level, i.e., the formula has the form:*

$$Q_1 x_1{:}\tau_1.\ Q_2 x_2{:}\tau_2.\ \cdots Q_k x_k{:}\tau_k.\ \psi$$

where each $Q_i \in \{\forall, \exists\}$ *and* ψ *is quantifier-free.*

EPNF is required by our NFA construction algorithm, which strips the quantifier prefix and constructs the automaton over the quantifier-free body ψ. Non-EPNF formulas like $\forall p.\ \mathbf{F}(\exists a.\ \mathit{in}(p, a))$ must be rewritten to $\forall p.\ \exists a.\ \mathbf{F}(\mathit{in}(p, a))$.

2.3 Nondeterministic Finite Automata for $\mathrm{LTL_f}$

The temporal goal formula is converted to an NFA that tracks progress toward goal satisfaction. We use the LGBA-style expansion algorithm [11], adapted for $\mathrm{LTL_f}$.

Definition 2.5 (State-Labelled NFA)**.** *An NFA is a tuple* $\mathcal{N} = \langle Q, q_0, \delta, F, L \rangle$ *where:*

- Q *is a finite set of states.*
- $q_0 \in Q$ *is the initial state.*
- $\delta \subseteq Q \times Q$ *is the transition relation.*
- $F \subseteq Q$ *is the set of accepting states.*
- $L \colon Q \to 2^{AP}$ *assigns labels (sets of propositional formulas) to states.*

Unlike standard NFAs where transitions are labelled, our NFA labels states. A transition from q to q' is enabled when the current planning state satisfies the labels of q'. This design simplifies the integration with the product automaton construction.

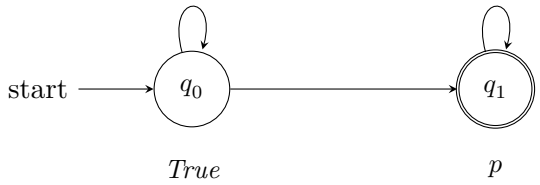

Figure 2.1: State-labeled NFA for $\mathbf{F}(p)$ ("eventually p"). State q_0 has label True (always satisfied), allowing the system to wait. Transition to q_1 is enabled when p holds. State q_1 is accepting and represents the goal sink.

The NFA for a typical "eventually" formula $\mathbf{F}(\mathit{pred}(x))$ has two states after minimization (Figure 2.1): a waiting state and an accepting goal sink.

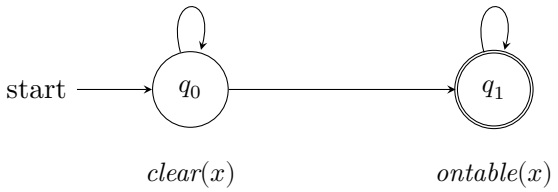

Figure 2.2: State-labeled NFA for $\mathit{clear}(x)\,\mathbf{U}\,\mathit{ontable}(x)$. State q_0 requires $\mathit{clear}(x)$ to hold (the "until" invariant). Transition to q_1 is enabled when $\mathit{ontable}(x)$ becomes true (the "until" goal). Compared to Figure 2.1, the waiting state q_0 has a non-trivial label that constrains behavior.

2.4 Generalized Planning

Definition 2.6 (Generalized Policy)**.** *A generalized policy π for a domain $\mathcal{D}$ maps states to actions such that for every problem instance P over $\mathcal{D}$, executing π from P's initial state reaches a goal state.*

2.4.1 Description Logic Features

The DL-plan framework [9] represents states using features derived from a DL concept grammar. Features are either Boolean or numerical:

Definition 2.7 (DL-plan Feature Grammar)**.** *Primitive concepts: $C_0 ::= p \mid p_G$ (current and goal predicates).*
Concept constructors:

$$C ::= C_0 \mid \neg C \mid C \sqcap C' \mid C \sqcup C' \mid \exists R.C \mid \forall R.C$$
$$R ::= r \mid r^{-1} \mid R^* \quad \textit{(role, inverse, transitive closure)}$$

Feature constructors:

$$b ::= b\_empty(C) \quad \textit{(concept has no instances)}$$
$$n ::= n\_count(C) \mid n\_concept\_distance(C_1, R, C_2)$$

The complexity k of a feature is its derivation depth in the grammar.

2.4.2 Policy Synthesis via Answer Set Programming

Given a set of features and a product automaton, Bonet and Geffner [4] formulate policy synthesis as an ASP optimization problem. The ASP program selects a subset of features and defines "good actions" for each state such that:

1. Every non-goal alive state has at least one good action.
2. Good actions lead to alive states.
3. States indistinguishable by selected features have the same good actions.
4. All alive states can reach a goal state via good transitions.

The optimization minimizes the total complexity of selected features, producing a compact policy.

2.5 Related Work

Lifted / First-Order Policy Representations. Yang et al. [20] (PG3) learn policies as lifted decision lists with variables, the representation closest to our second approach. Key differences are: PG3 iteratively improves policies through planning, while we extract rules once from optimal plans. PG3 also uses only domain predicates, whereas we add NFA predicates for temporal awareness. PG3 treats all variables equally, but we distinguish action parameters from context variables for goal-directed behavior. PG3 handles classical reachability goals, while we handle temporal goals with quantifiers. Finally, PG3 needs a classical planner during learning, whereas we only need optimal plans from the product automaton.

Learning from Plan Traces. Srivastava et al. [18] abstract concrete plans by replacing objects with variables to derive generalized plans (fixed action sequences with loops), whereas we derive reactive policies (ordered decision lists). Silver et al. [17] learns general policies from a small number of optimal demonstrations with Bayesian learning.

Neural Approaches to Action-Argument Selection. Groshev et al. [13] train deep neural networks end-to-end on execution traces to represent generalized reactive policies that map problem instances and states directly to actions, removing the need for handcrafted

domain features while also yielding transferable heuristic functions for search. Toyer et al. [19] introduce Action Schema Networks (ASNets), where they used a neural network for learning generalized policies for probabilistic planning problems.

Planning with Temporally Extended Goals. Camacho et al. [5] and Camacho and McIlraith [6] compile LTL_f goals to deterministic finite automaton (DFA) for planning. Baier and McIlraith [1] provide the first-order extensions of temporal goals in planning closest to our formulation.

3 System Architecture

This chapter describes the overall system architecture: the five-phase pipeline shared by both approaches, the NFA construction algorithm, the product automaton construction with binding trees for nested quantifiers, and the benchmark domains used in the evaluation.

3.1 Pipeline Overview

The system follows a five-phase pipeline (Figure 3.1), shared by both the ASP-based and lifted-rule approaches:

1. **Parse.** Load PDDL domain, problem instances, and temporal goal. Parse the FO-LTL$_f$ formula in EPNF.
2. **NFA Construction.** Convert the temporal goal to an NFA. Strip quantifier prefix; apply LGBA expansion; minimize via bisimulation.
3. **Product Automaton.** Construct the augmented product automaton $\mathcal{S} \times \mathcal{Q}$ by BFS over PDDL states cross NFA tracking states.
4. **Policy Learning.** Either (a) generate DL features and solve an ASP problem, or (b) extract lifted rules from optimal plans.
5. **Testing.** Execute the learned policy on held-out problem instances.

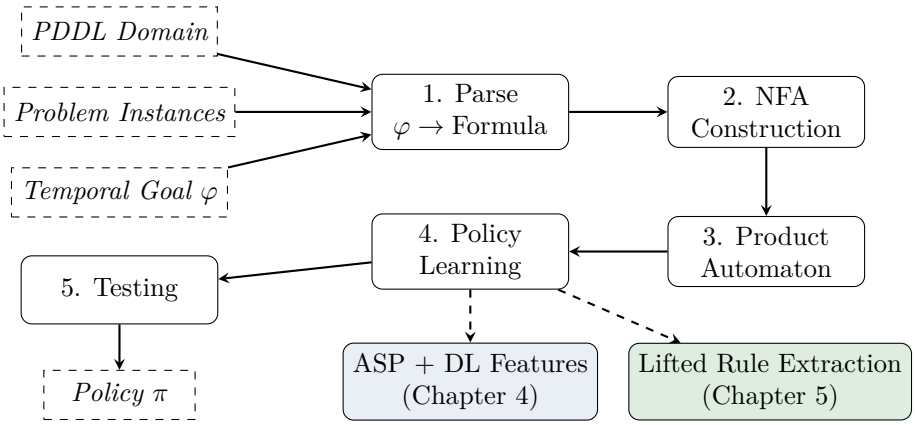

Figure 3.1: The overall pipeline. Phases 1–3 are shared by both approaches. Phase 4 branches into ASP-based synthesis or lifted rule extraction.

3.2 LTL Parsing and NFA Construction

3.2.1 LGBA Expansion Algorithm

The NFA is constructed using a variant of the Gerth–Peled–Vardi–Wolper (GPVW) expansion algorithm [11]. The algorithm processes the quantifier-free formula by building a graph where each node tracks three formula sets: New(unprocessed), Old (processed, becoming state labels), and Next (deferred to successors). The expansion proceeds recursively: literals and conjunctions are added to the current node, **X** formulas are deferred to successors, and disjunctive formulas ($\vee$, **U**, **R**) cause node splitting into branches. Contradictions ($\bot$ or both φ and $\neg\varphi$) are discarded. Nodes with identical Old and Next sets are merged.

The special atom final represents the end of a finite trace. A state is accepting if all temporal obligations are satisfied (Next $= \emptyset$) and final is not explicitly denied ($\neg$final $\notin$ Old).

Algorithm 1 shows the expansion procedure. Sugar operators are expanded before processing: $\mathbf{F}\varphi \equiv \top\,\mathbf{U}\,\varphi$ and $\mathbf{G}\varphi \equiv \bot\,\mathbf{R}\,\varphi$.

Algorithm 1 LGBA expansion: Expand(*node*, *nodes*).

```
1  if node.New = ∅ then
2      if ∃nd ∈ nodes with same Old and Next as node then
3          Merge incoming edges; return nodes
4      else if final ∈ node.Old and node.Next ≠ ∅ then
5          return nodes                          Discard: final with pending obligations
6      else
7          Add node to nodes
8          Create successor with New ← node.Next
9          return Expand(successor, nodes)
10 else
11     Pick η ∈ node.New; move η from New to Old
12     if η is a literal, ⊤, or ⊥ then
13         if η = ⊥ or ¬η ∈ node.Old then
14             return nodes                                        Contradiction
15         else
16             return Expand(node, nodes)
17     else if η = Xψ then
18         node.Next ← node.Next ∪ {ψ}
19         return Expand(node, nodes)
20     else if η = φ ∧ ψ then
21         node.New ← node.New ∪ {φ, ψ}
22         return Expand(node, nodes)
23     else if η ∈ {φ ∨ ψ, φ U ψ, φ R ψ} then
24         Split node into node_1 (with New_1(η)) and node_2 (with New_2(η))
25         nodes ← Expand(node_1, nodes)
26         return Expand(node_2, nodes)
```

The splitting functions for disjunctive operators are shown in Table 3.1.

Table 3.1: Splitting functions for disjunctive operators in the LGBA expansion.

η	$\text{New}_1(\eta)$	$\text{New}_2(\eta)$
$\varphi \vee \psi$	$\{\varphi\}$	$\{\psi\}$
$\varphi \,\mathbf{U}\, \psi$	$\{\varphi, \mathbf{X}(\varphi \,\mathbf{U}\, \psi)\}$	$\{\psi\}$
$\varphi \,\mathbf{R}\, \psi$	$\{\psi, \text{final} \vee \mathbf{X}(\varphi \,\mathbf{R}\, \psi)\}$	$\{\varphi, \psi\}$

After expansion completes, the graph nodes are converted to an NFA:

- **States**: Each node becomes an NFA state labeled by its Old formula set.
- **Transitions**: For each node n with Next $\neq \emptyset$, create transitions to all successors whose New $= n$.Next.
- **Accepting states**: $F = \{n \in \textit{nodes} \mid n.\text{Next} = \emptyset \wedge \neg\text{final} \notin n.\text{Old}\}$.

The NFA is then minimized using bisimulation-based state merging [15].

3.3 Product Automaton Construction

The product automaton combines the PDDL state space with NFA temporal tracking.

Definition 3.1 (Augmented State)**.** *An augmented state is a pair (s, T) where s is a PDDL state and T is a tracking structure encoding NFA progress. For quantified goals, T is a binding tree (Section 3.3.2).*

Definition 3.2 (Enriched Augmented State)**.** *An enriched augmented state s' is the PDDL state s augmented with NFA predicates derived from the tracking structure T:*

$$s' = s \cup \{\texttt{nfa\_at\_}q(o) \mid \textit{object } o \textit{ is at NFA state } q \textit{ in } T\}$$

Policy rules match against s', and action execution updates both s (via PDDL action effects) and T (via NFA progression). Algorithm notation uses (s, T) for the abstract augmented state node and s' for the enriched state used in rule matching.

Definition 3.3 (Product Automaton)**.** *The product automaton is a tuple $\mathcal{A} = \langle V, v_0, E, V_g \rangle$ where:*

- *V is a set of augmented state nodes.*
- *$v_0 = (s_0, T_0)$ is the initial node.*
- *$E \subseteq V \times \mathcal{A}_g \times V$ are transitions labelled by ground actions.*
- *$V_g \subseteq V$ are goal nodes (all per-object NFA copies in accepting states).*

Algorithm 2 describes the BFS construction.

Algorithm 2 Product automaton construction.

Require: PDDL problem P, NFA $\mathcal{N}$, quantifier prefix Q, state limit M
Ensure: Product automaton $\mathcal{A}$

$s_0 \leftarrow$ initial state of P
$T_0 \leftarrow \text{InitTracking}(\mathcal{N}, Q, P)$ *Create initial binding tree*
$v_0 \leftarrow (s_0, T_0)$; enrich v_0 with NFA predicates
queue $\leftarrow [v_0]$; *visited* $\leftarrow \{v_0\}$
while *queue* $\neq \emptyset$ **and** $|\textit{visited}| < M$ **do**
  $(s, T) \leftarrow$ *queue*.Dequeue()
  if (s, T) is a goal state **then**
    Mark as terminal; **continue** *No outgoing edges from goals*
  for each applicable ground action a in s **do**
    $s' \leftarrow \text{Apply}(a, s)$
    $T' \leftarrow \text{ProgressTracking}(T, s', \mathcal{N})$ *Advance NFA for all bindings*
    Enrich s' with NFA predicates from T'
    $v' \leftarrow (s', T')$
    if $v' \notin$ *visited* **then**
      visited $\leftarrow$ *visited* $\cup \{v'\}$; enqueue v'
    Add edge $(s, T) \xrightarrow{a} (s', T')$ to $\mathcal{A}$
return $\mathcal{A}$

3.3.1 NFA Predicates as Augmented State Fluents

We encode NFA progress as additional predicates in the augmented state. For each object o tracked at NFA state q, the predicate `nfa_at_q`(o) is added to the augmented state. This allows DL features and lifted rules to reason about temporal progress using the same predicate language as domain predicates.

Example 3.4. *In the elevators domain with goal* $\forall p.\, \mathbf{F}(\textit{served}(p))$*, the augmented state might contain:*

$$\{\textit{lift-at}(f_1), \textit{origin}(p_0, f_0), \ldots, \texttt{nfa\_at\_n0}(p_0), \texttt{nfa\_at\_n0}(p_1), \texttt{nfa\_at\_n1}(p_2)\}$$

Here p_0 and p_1 are still in NFA state n_0 (unserved), while p_2 has progressed to n_1 (served).

3.3.2 Binding Trees for Nested Quantifiers

For goals with nested quantifiers like $\forall x.\ \exists y.\ \mathbf{F}(\textit{on}(x, y))$, a flat set of per-object NFA states is insufficient. Each binding of the outer variable x requires independent tracking of the inner variable y's NFA state. Our implementation uses *binding trees*: a tree of depth k for quantifier prefix $Q_1 x_1.\ \cdots Q_k x_k$, where level i nodes represent bindings of variable x_i to objects and store the corresponding NFA state. Universal ($\forall$) nodes require all children to be in accepting states, while existential ($\exists$) nodes require at least one accepting

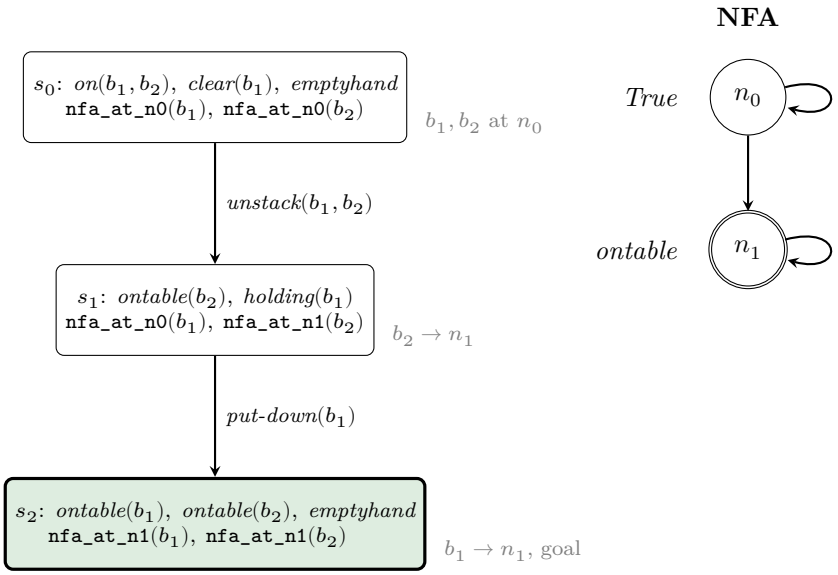

Figure 3.2: Product automaton fragment for $\forall x.\, \mathbf{F}(ontable(x))$ with 2 blocks. Each state contains PDDL facts and per-object NFA predicates. After *unstack*, block b_2 reaches the table so `nfa_at_n0`(b_2) transitions to `nfa_at_n1`(b_2). After *put-down*, both blocks are at accepting NFA state n_1.

child. This tree structure grows as $O(n^k)$ for k nested quantifiers over n objects, which is a fundamental scalability constraint for deeply nested quantifier goals.

3.4 Benchmark Domains

We evaluate our approaches on eleven benchmark domains summarized in Table 3.2. All domains are drawn from the classical planning benchmarks and augmented with temporal goal formulas.

Table 3.2: Benchmark domains with their types, predicates, actions, and key complexity characteristics.

Domain	**Types**	**Preds**	**Actions**	**Key Challenge**
Blocksworld	block	5	4	Simple sequential manipulation; no navigation
Childsnack	child, bread, sandwich, tray	8	4	Multi-step preparation and serving
Depot	place, truck, crate, hoist	8	5	Loading/unloading with spatial constraints
Elevators	passenger, floor	8	4	Goal-directed navigation with passenger context
Ferry	car, location	4	3	Single-capacity boat, multi-location routing
Gripper	ball, room, gripper	4	3	Dual-capacity back-and-forth transport
Logistics	package, truck, airplane, location	7	4	Multi-modal transportation (ground + air)
Robot	room, door, item, robot	8	5	Multi-room paths with door management
Spanner	location, man, spanner, nut	7	3	Consumable resources (single-use tools)
Visitall	location	2	1	Complete coverage with minimal actions
Zenotravel	aircraft, person, city, flevel	5	4	Fuel tracking and multi-resource coordination

4 Approach 1: ASP-Based Policy Synthesis

This chapter presents the first approach to learning generalized policies for FO-LTL$_f$ goals: generating DL features from the product automaton and using ASP to synthesize a policy. We describe the method, present experimental results, and analyze the scalability barrier that motivates the second approach.

4.1 Method

The ASP-based approach follows Bonet and Geffner [4]: given the product automaton, generate a pool of DL-plan features, encode the feature evaluations and state transitions as ASP facts, and use the Clingo solver [10] to find a minimal feature set and corresponding policy.

The DL-plan feature framework [8, 9] generates description logic features from states. Features are generated iteratively with increasing complexity k, starting from primitive concepts (one per predicate) and applying constructors such as intersection, union, negation, and existential/universal restrictions. Boolean features check concept emptiness, while numerical features count objects or compute concept distances. Features are evaluated on every state in the training product automaton, and the resulting valuations are encoded as ASP facts representing states, transitions, and feature evaluations.

4.1.1 ASP Policy Synthesis Procedure

The ASP program selects features and good actions to form a policyHofmann and Geffner [14]. The core constraints are:

1. **Feature selection**: $\{selected(F)\} :- feature(F)$.
2. **Good action selection**: At least one good action per alive non-goal state.
3. **Boolean distinguishability**: States with different good actions must be distinguishable by selected features.
4. **Transition consistency**: For equivalent states (same selected feature values), good actions must produce consistent feature value changes (deltas).
5. **Safety**: All alive states must reach a goal state via good transitions (inductive argument).
6. **Optimization**: Minimize total complexity of selected features.

4.2 Experimental Setup

We evaluate the ASP approach on 24 goals across 4 domains (blocksworld, elevators, robot, zenotravel). Training uses several small instances (typically instances with approximately 4-7 objects). Testing using held-out instances of increasing size.

The solver configuration uses Clingo with 8 threads, complexity k ranging from 2 to 8, and a maximum feature cost of 20.

4.3 Results

Table 4.1 summarizes the ASP-based approach results.

Table 4.1: ASP-based approach results. k^* is the complexity at which a solution was found. Grounding limit rows indicate failure due to Clingo's 32-bit atom ID restriction.

Domain	**Goal**	k^*	**Rules**	**Rate**	**Time**	**Memory**
Blocksworld	$\forall x.\ \mathbf{F}(\mathit{ontable}(x))$	3	3	100 %	1.5 s	53 MB
Blocksworld	$\forall x.\ \mathbf{F}(\mathit{clear}(x))$	3	3	100 %	0.8 s	48 MB
Blocksworld	$\forall x.\ \exists y.\ \mathbf{F}(\mathit{on}(x, y))$	6	7	100 %	22 s	312 MB
Elevators	$\forall p.\ \mathbf{F}(\mathit{served}(p))$	—	—	—	—	>73 GB
Elevators	$\forall p.\ \mathbf{F}(\mathit{boarded}(p))$	8	2	100 %	1190 s	107 GB
Robot	$\forall o.\ \mathbf{F}(\mathit{at\text{-}obj}(o, r_2))$	—	—	—	—	Grounding limit
Robot	$\forall d.\ \mathbf{F}(\mathit{opened}(d))$	5	3	100 %	5.8 s	416 MB
Zenotravel	$\forall p.\ \mathbf{F}(\mathit{at}(p, c_2))$	—	—	—	—	Grounding limit
Zenotravel	$\exists a.\ \mathbf{F}(\mathit{at}(a, c_2))$	6	4	100 %	30 s	1.8 GB

Successes. The ASP approach works well when training instances remain small (few objects, small product automata). Blocksworld goals succeed with $k^* = 3$–6, training in seconds to minutes with memory under 500 MB. Simpler goals across all domains (e.g., board-only, open-only) also succeed.

Failures. The approach fails when complex goal structures require training on instances with more objects to achieve adequate coverage. More objects lead to larger product automata, which cause exponential grounding explosion analyzed in Section 4.4.

4.4 Analysis of Failure Modes

The ASP approach's scalability is fundamentally limited by two compounding factors: state space blowup in the product automaton and quadratic grounding cost in the ASP solver. This section analyzes both sources of complexity and their interaction.

4.4.1 Product Automaton State Space Blowup

The product automaton construction (Section 3.3) combines the PDDL state space with NFA states for temporal tracking. We first characterize the theoretical and empirical blowup in augmented state count.

Theoretical Upper Bound

Definition 4.1 (State Space Blowup)**.** *Given a PDDL problem P with reachable state space $\mathcal{S}_{PDDL}$ and a temporal goal with NFA $\mathcal{N}$, the* blowup factor *of the product automaton is:*

$$\beta = \frac{|\mathcal{S}_{aug}|}{|\mathcal{S}_{PDDL}|}$$

For a goal with quantifier prefix $Q_1 x_1{:}\tau_1.\ Q_2 x_2{:}\tau_2.\ \cdots\ Q_m x_m{:}\tau_m$ and an NFA with $Q = |\mathcal{N}|$ states, each object of a quantified type maintains an independent NFA copy. The theoretical upper bound on the blowup factor is:

$$\beta \ \leq\ \left(\prod_{i=1}^{m} O_i\right) \cdot 2^Q \tag{4.1}$$

where $O_i = |\{o \in \mathrm{Obj}(P) \mid \mathrm{type}(o) = \tau_i\}|$ is the number of objects of type τ_i.

The 2^Q factor arises because augmenting the original state space with all possible NFA successors is equal to determinising the DFA using power set construction, and it has been proven that it leads to a worst case blow up of 2^Q. On top of that, we need to track configurations of all combinations of quantified objects, and where they are in the NFA setting, hence the $\prod O_i$ factor.

Empirical Analysis

To validate this bound and characterize the actual blowup in practice, we conducted an empirical study across 71 instances from 9 domains and 45 temporal goals, restricted to instances with at most 6 objects for tractable full state-space enumeration.

For each instance, we constructed both the plain PDDL reachable state space (via BFS without temporal tracking) and the full product automaton, then measured the blowup factor β.

Table 4.2 groups results by the two key factors from Equation (4.1): the product of quantified object counts ($\prod O_i$) and the number of NFA states (Q).

Table 4.2: State space blowup grouped by quantified object product ($\prod O_i$) and NFA size (Q). The theoretical upper bound ($\prod O_i) \cdot 2^Q$ is compared to actual measured blowup across 71 instances from 9 domains with ≤ 6 objects. Tightness = average ratio of actual to theoretical bound.

$\prod O_i$	Q	Theor. bound	Avg β	Min	Max	$\#I$	Representative domains
1	3	8	1.4×	1.0×	1.8×	10	Robot, Spanner
1	5	32	2.3×	1.9×	2.7×	4	Robot G10/G11
1	6	64	2.4×	1.9×	2.7×	3	Robot G6/G10
2	3	16	2.4×	1.0×	4.0×	14	Ferry, Gripper, Logistics
2	4	32	3.2×	1.2×	5.7×	3	Gripper, Blocksworld
2	5	64	4.9×	1.5×	8.1×	3	Elevators, Gripper, Ferry
2	8	512	3.8×	3.8×	3.8×	1	Zenotravel G10
3	3	24	2.0×	1.0×	4.7×	12	Elevators, Ferry
3	4	48	2.1×	1.0×	4.0×	5	Blocksworld, Elevators
3	5	96	6.1×	1.8×	10.4×	2	Elevators, Ferry
4	3	32	1.6×	1.1×	2.2×	3	Blocksworld, Visitall
4	4	64	4.5×	1.0×	8.0×	2	Blocksworld, Elevators
4	5	128	1.4×	1.3×	1.5×	3	Elevators G7/G8/G11
9	3	72	22.4×	13.1×	24.8×	5	Blocksworld G4/G5/G6

Key Findings

Across all 71 instances, the actual blowup uses on average only 11.7% of the theoretical maximum $(\prod O_i) \cdot 2^Q$. The looseness increases with NFA complexity: for $Q = 3$ the tightness ratio is ∼15%, while for $Q = 8$ it drops to 0.7%. This is because monotonic NFA progress (objects advance forward through states and do not revisit earlier states) prunes most of the 2^Q configuration space.

From the table we can observe that quantified object count is the dominant factor. For the same NFA size ($Q = 3$), the average blowup progresses as:

$$\prod O_i = 1\text{:}\quad 1.4\times$$
$$\prod O_i = 2\text{:}\quad 2.4\times$$
$$\prod O_i = 3\text{:}\quad 2.0\times$$
$$\prod O_i = 9\text{:}\quad 22.4\times$$

The jump to $\prod O_i = 9$ (from nested $\forall x.\ \exists y$ over 3 blocks, giving $3 \times 3 = 9$) produces the highest observed blowup.

NFA size does not in practive cause too much state space explosion. Comparing rows with $\prod O_i = 1$: increasing Q from 3 to 6 only raises the average blowup from 1.4× to 2.4×, far below the theoretical ratio of $2^6/2^3 = 8\times$. Similarly, zenotravel Goal 10 with $Q = 8$ and $\prod O_i = 2$ has a theoretical bound of 512 but actual blowup of only 3.8×. But, nested

quantifiers amplify through $\prod O_i$. The highest blowup instances above are blocksworld goals with nested quantifiers ($\forall x.\ \exists y$ or $\exists x.\ \exists y$), where $\prod O_i = 3 \times 3 = 9$ for 3 blocks. In contrast, elevators goals with nested $\forall v.\ \forall r$ but only $\prod O_i = 2 \times 2 = 4$ show modest 1.3–1.5× blowup, confirming that the object count product, not the nesting depth, drives the explosion.

4.4.2 ASP Grounding Explosion

The state space blowup directly compounds with Clingo's grounding cost, creating an exponential barrier. The dominant term in Clingo's ground program size is:

$$|\text{ground atoms}| \propto S^2 \times F \times A^2$$

where S is the number of augmented states, F the number of features, and A the number of ground actions. This arises from the transition-delta consistency constraint in the ASP program, which must compare every pair of states across every selected feature and every pair of actions.

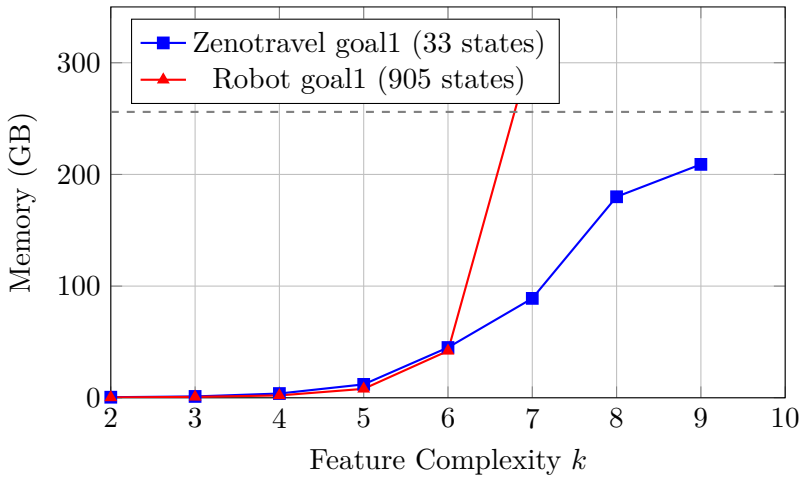

Figure 4.1: Memory usage vs. feature complexity for two representative goals. Both exceed available memory before finding a solution. The Clingo 32-bit atom ID limit (approximately 4.3 billion) is typically reached before physical memory is exhausted.

Clingo uses 32-bit identifiers for ground atoms, imposing a hard limit of approximately 4.3 billion atoms. As Figure 4.1 illustrates, complex domains hit this limit at moderate complexity levels ($k = 7$–9), long before a solution can be found.

Mitigation attempts included:

- **Seeded training**: Scout on a tiny instance (7 states) at high k to discover useful features, then inject those features into training on a larger instance at low k. This reduces F but not $S^2 \times A^2$.
- **Feature pool pruning**: Remove redundant or uninformative features. Reduces F but the savings are insufficient.

Neither mitigation overcomes the quadratic state-pair scaling. The fundamental issue is that complex goals require training instances with more objects to achieve adequate behavioral coverage, which directly increases S, A, and consequently the $O(S^2 \times F \times A^2)$ grounding cost.

4.4.3 Combined Effect and Limitations

The interaction between product automaton blowup and ASP grounding cost creates a multiplicative scalability barrier:

1. **State space blowup.** Even modest blowup ($5\times$) in the product automaton from quantified goals means $S = 5|\mathcal{S}_{\text{PDDL}}|$.
2. **Grounding explosion.** This blowup enters quadratically into the ASP grounding cost: $O(S^2 \times F \times A^2) = O(25|\mathcal{S}_{\text{PDDL}}|^2 \times F \times A^2)$, a $25\times$ increase in ground atoms.
3. **Training instance selection.** To keep blowup manageable, training must use small instances (3–6 objects, $\prod O_i \leq 3$). This limits the complexity and coverage of learned policies.
4. **Nested quantifier barrier.** The $\prod O_i$ factor fundamentally limits nested quantifiers: $\forall x.\ \exists y$ with 4 blocks gives $\prod O_i = 16$, potentially $> 100\times$ blowup, restricting training to 2–3 object instances where adequate behavioral coverage is impossible.

4.5 Discussion

The ASP approach succeeds when training instances remain small but fails when goals require larger training sets. The core limitation is not the goal structure itself but the training set complexity: more complex goals need instances with more objects for adequate coverage, which exponentially increases the product automaton size. The $O(S^2 \times F \times A^2)$ grounding explosion makes this approach intractable as training instances scale beyond a few objects.

This scalability limitation motivates the second approach presented in the next chapter: extracting lifted rules directly from optimal plans, avoiding the need for an ASP solver entirely.

5 Approach 2: Lifted Rule Extraction

This chapter presents the second and primary approach to learning generalized policies for FO-LTL$_f$ goals: extracting first-order policy rules directly from optimal plans in the product automaton. We describe the rule representation, the extraction and execution algorithms, and evaluate the approach on 11 planning domains and 46 temporal goals.

5.1 Motivation

The ASP-based approach succeeds on simple goals with small training instances but fails when goals require larger training sets (Section 4.4). Complex goals need instances with more objects to achieve adequate behavioral coverage, which causes the ASP grounding to explode quadratically in the number of states and features.

These limitations motivate a fundamentally different strategy: rather than searching directly in the space of policies, we instead learn policies from optimal demonstrations. We build upon the lifted decision list representation [20], where a policy is an ordered list of first-order rules. Each rule has parameters, state-preconditions, goal-preconditions, and an action. This representation was originally designed for classical reachability goals with a fixed goal set G.

5.1.1 Core Insight: Temporal Progress as State

The key challenge of temporal goals is that policy decisions depend not only on the current planning state but also on the temporal progress achieved so far. A goal like $\forall p.\ \mathbf{F}(\mathit{served}(p))$ requires the policy to track which passengers have been served and to coordinate board, navigate, and depart actions for each passenger independently.

Unlike classical planning, which searches for paths from initial state s_0 to any state satisfying goal G:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n \models G$$

For temporal goals, there is no fixed G. Instead, we need to track per-object temporal progress:

- Which passengers have been served (completed their temporal requirement)
- Which passengers are in progress (boarded but not yet served)
- Which passengers have not started (not yet boarded)

The key insight is to encode temporal progress directly in the planning state. The product automaton construction (Section 5.3) combines PDDL states with NFA states that track each object's temporal progress:

$$(s, nfa_p) \text{ where } s \in S \text{ is a PDDL state}, \quad nfa_p : O \to Q \text{ maps objects to NFA states}$$

This reduces temporal goals to reachability: find paths to states where all tracked objects are in accepting NFA states. Rules extracted from such paths encode the temporal ordering required to satisfy the goal.

5.1.2 Adapting Lifted Decision Lists for Temporal Goals

We adapt the lifted decision list representation with four changes:

1. **Plans from the product automaton.** The base representation generates plans in the PDDL state space. We run BFS in the product automaton (PDDL state × NFA state) and extract rules from all shortest paths. The product automaton construction ensures that every accepting path satisfies the temporal specification because the automaton state encodes the temporal progress required for correctness. This allows standard BFS over the product automaton to generate plans satisfying temporal constraints. The policy is extracted from training plans in a single pass.
2. **NFA predicates in place of goal-preconditions.** Classical lifted rules use `:goal-preconditions` to check against the goal set G. We replace this with NFA predicates in the rule body. A predicate `nfa_at_n0`($?a1$) means that object $?a1$ is currently at NFA state n_0. The rule body contains both domain predicates and NFA predicates, enabling rules to check temporal progress and prevent redundant actions.
3. **Existential context variables.** In the base representation, all variables in a rule are action parameters. We add existential variables ($?c0$, $?c1$, …) that appear in the rule body but not in the action. For example, `nfa_at_n0`($?c0$) in a navigation rule requires that some unserved object exists, but does not specify which one. Existential context variables enable navigation rules to verify that unserved objects exist ($\exists ?c0 : $ `nfa_at_n0`$(?c0)$) without specifying which object, preventing navigation loops after all goals are achieved.
4. **Priority from NFA awareness and advancement.** The base representation determines rule ordering through search. We assign priorities based on whether the rule contains NFA state predicates (ensuring rules with temporal guards fire first) and how the rule changes the NFA state (goal-advancing, progressing, or navigation). Rules with temporal guards receive priority 0–99; rules without NFA predicates receive priority 100–199. The two-tier priority system ensures temporally-aware

rules (priority 0–99) are attempted before generic navigation rules (priority 100–199), guaranteeing that goal-completion status is checked before navigation actions.

5.2 Rule Representation

5.2.1 Lifted Rules

Definition 5.1 (Lifted Rule)**.** *A lifted rule is a tuple* $r = (body, schema, params, exist, priority)$ *where:*

- *body is a conjunction of lifted atoms (positive or negative predicates with variables),*
- *schema is a PDDL action schema name (e.g., board),*
- $params = (?a0, ?a1, ...)$ *are action parameter variables,*
- $exist = \{?c0, ?c1, ...\}$ *are existential context variables,*
- $priority \in \mathbb{N}$ *determines the rule's position in the decision list (lower = tried first).*

A lifted atom is either a positive predicate $p(?a0, ?a1)$ or a negative predicate $NOT\_p(?a0)$. NFA progress predicates `nfa_at_n0`$(?a0)$ are treated as regular atoms, encoding which NFA state a tracked object currently occupies.

Example 5.2 (Elevators goal1: serve all passengers)**.** *The following rule departs a passenger at their destination:*

$$\begin{aligned}[priority = 12] \quad & boarded(?a1),\ destin(?a1, ?a0),\ lift\text{-}at(?a0),\\ & \texttt{nfa\_at\_n0}(?a1),\ not\text{-}served(?a1)\\ & \longrightarrow depart(?a0, ?a1)\end{aligned}$$

The body specifies: the elevator is at floor $?a0$*, passenger* $?a1$ *is boarded, has destination* $?a0$*, is tracked at NFA state* n_0 *(not yet served), and has not been served.*

5.2.2 Existential Context Variables

Context variables allow navigation rules to be goal-directed without over-specifying the target object.

Definition 5.3 (Rule Matching Semantics)**.** *A rule* $r = (body, schema, params, exist, priority)$ *matches state* s *if there exists a grounding* $\sigma : params \to Objects$ *such that:*

1. *For all body atoms* ℓ *with variables only from params:* $\sigma(\ell) \in s$*,*
2. *For all body atoms* ℓ *with variables from exist: there exists an extension* $\sigma' : exist \to Objects$ *such that* $(\sigma \cup \sigma')(\ell) \in s$*.*

The rule fires by applying schema with arguments $\sigma(params)$*. Existential context variables witness goal-relevant predicates but do not constrain action arguments.*

Consider the elevator navigation rule:

$$[\textit{priority}{=}20] \quad \textit{NOT\_lift-at}(?a1),\ \textit{above}(?a0, ?a1),\ \textit{lift-at}(?a0),\ \texttt{nfa\_at\_n0}(?c0)$$
$$\longrightarrow \textit{up}(?a0, ?a1) \quad [\exists ?c0]$$

The variable $?c0$ does not appear in the action parameters. Instead, $\texttt{nfa\_at\_n0}(?c0)$ asserts that there exists some passenger still at NFA state n_0 (unserved), providing goal-directed motivation for the navigation action. Without this existential guard, the rule would fire even when all passengers have been served, causing infinite loops.

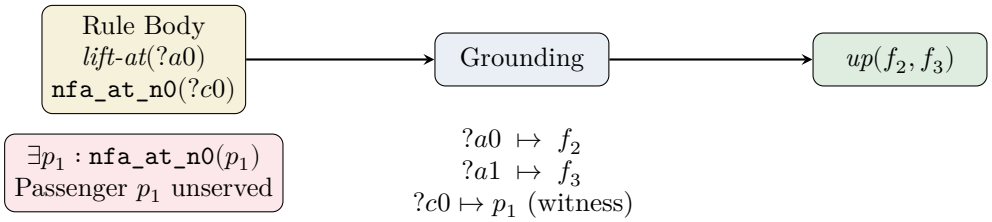

Figure 5.1: Existential context variables provide goal-directed motivation for navigation. The variable $?c0$ is bound to a witness object (an unserved passenger) but is not used as an action argument.

5.2.3 Decision List Semantics

A lifted policy $\pi = [r_1, r_2, \ldots, r_n]$ is an ordered decision list of lifted rules sorted by priority. At each execution step, the policy tries rules in priority order. The first rule whose body can be satisfied (a valid variable grounding exists and the grounded action's preconditions hold) is fired.

5.3 Temporal Progress Tracking

Before presenting the extraction algorithm, we formalize how temporal progress is tracked in the product automaton.

5.3.1 Product Automaton Structure

For a goal $\forall x \in X.\ \varphi(x)$ where X is the set of tracked objects and φ is an LTL_f formula, the product automaton state is:

$$(s, \vec{n}) \text{ where } s \in S \text{ (PDDL state)}, \quad \vec{n} : X \to Q \text{ (NFA state per object)}$$

Each tracked object has an independent NFA tracking its temporal progress. The product state combines PDDL facts with NFA states, encoded as predicates

5.3 Rule Extraction Algorithm

5.3.2 Overview

Given a product automaton $\mathcal{A}$ built from training instances, the extraction algorithm (Algorithm 3) proceeds in four stages:

1. Find all BFS-shortest paths from the initial state to goal states.
2. For each action in each path, extract its lifted context (relevant predicates, NFA progress, negative conditions).
3. Group extracted contexts by action schema, intersect predicate patterns across occurrences.
4. Assign priorities based on NFA awareness and advancement classification.

Algorithm 3 Lifted Rule Extraction

Require: Product automaton $\mathcal{A}$, domain $\mathcal{D}$, goal constants $\mathcal{G}$
Ensure: Lifted policy π

```
paths ← BfsAllShortestPaths(A)
contexts ← {}                                   Map: schema → list of contexts
for each path (s_0, a_1, s_1), ..., (s_{n-1}, a_n, s_n) in paths do
    for each step (s_i, a_{i+1}, s_{i+1}) do
        ctx ← ExtractActionContext(s_i, a_{i+1}, s_{i+1}, D, G)
        contexts[ctx.schema] += ctx
rules ← []
for each schema σ and context list C in contexts do
    body ← IntersectPredicates(C)
    body ← PruneRedundant(body)
    p ← ComputePriority(C)                      Based on NFA advancement
    rules += MakeRule(body, σ, p)
return π = SortByPriority(rules)
```

5.3.3 Extraction Pipeline Overview

The extraction process consists of four distinct stages, illustrated in Figure 5.2:

1. **BFS shortest paths:** Run BFS on the product automaton to find up to 10 shortest paths to goal states.
2. **Context extraction:** For each action in each path, extract predicates from the source state and lift to first-order by replacing concrete objects with variables.
3. **Intersection:** Group extracted contexts by action schema and intersect predicate sets to remove instance-specific details.
4. **Priority assignment:** Classify rules by NFA awareness (0 vs 100 base priority) and advancement type (goal-advancing, progressing, navigation).

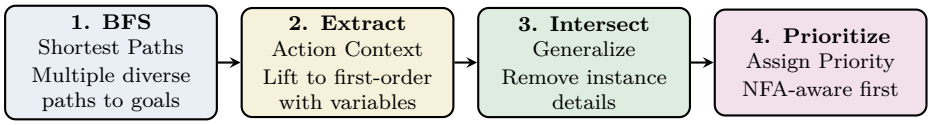

Figure 5.2: Rule extraction pipeline. BFS provides diverse optimal paths; extraction lifts ground actions to first-order; intersection generalizes by removing instance-specific predicates; priority assignment ensures NFA-aware rules are tried first.

5.3.4 BFS Shortest Paths

The algorithm begins by running BFS on the product automaton to find all shortest paths from the initial state (id 0) to any goal state. Up to 10 paths of optimal length are collected. Using multiple paths increases the diversity of action contexts, which leads to more robust rules after intersection.

5.3.5 Action Context Extraction

Context extraction operates on each plan step (s_i, a_{i+1}, s_{i+1}). First, action parameters are mapped to typed variables: the first parameter becomes $?a0$, the second $?a1$, and so on. Relevant predicates—those mentioning at least one action parameter—are collected from s_i and lifted by replacing concrete objects with their corresponding variables.

Objects appearing in NFA predicates (`nfa_at_n0`(x)) but not as action parameters are identified as context objects and mapped to existential variables ($?c0$, $?c1$, …). These witness goal-relevant progress without constraining which specific object the action targets.

Negative conditions are added systematically: for each unary predicate p whose type matches an action parameter, if p does not hold for that parameter in s_i, the atom NOT_p is added to the rule body. This is type-checked to avoid spurious negations.

Finally, constants are handled based on their role: goal-relevant constants (e.g., `city2` in $\forall p.\ \mathbf{F}(at(p, city2))$) are preserved, while other constants are generalized to fresh variables. The NFA predicates in s_i and s_{i+1} are compared to classify whether the action advances the goal (object reaches accepting state), makes progress (NFA state changes), or serves a support role (navigation without NFA change).

5.3.6 Predicate Intersection and Pruning

When multiple occurrences of the same action schema are observed across plans, their predicate sets are intersected: only atoms present in all occurrences are retained. This generalizes away instance-specific details while preserving structural invariants.

After intersection, redundant atoms are pruned: For example, if both $p(?a0, ?a1)$ and $p(?a1, ?a0)$ appear (e.g., bidirectional *connects*), only one is kept.

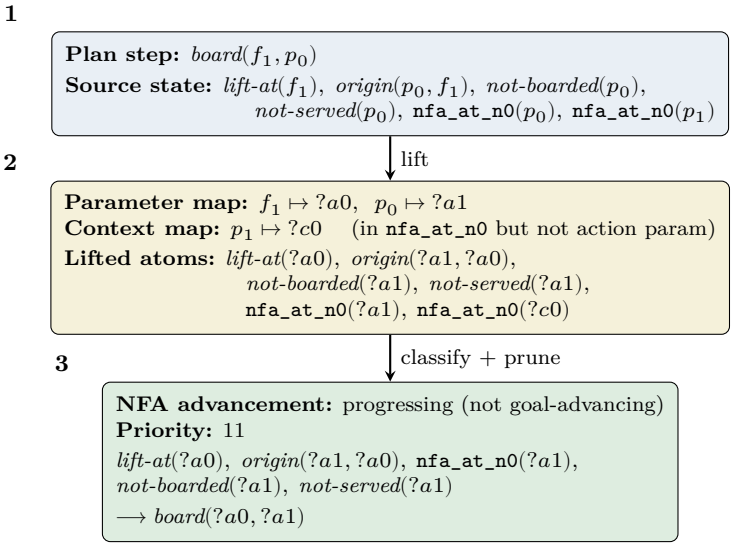

Figure 5.3: Rule extraction walkthrough for a *board* action in the elevators domain. Step 1: a ground action from the BFS-optimal plan. Step 2: action parameters and context objects are mapped to variables; relevant predicates are lifted. Step 3: NFA advancement is classified, priority is assigned, and redundant atoms are pruned to produce the final rule.

Correctness of Intersection-Based Generalization

The intersection operation preserves correctness by eliminating instance-specific predicates while retaining structural invariants. A predicate that appears in some training instances but not others correlates with the particular configuration rather than the action's applicability conditions. By retaining only predicates present across all occurrences, the method identifies the minimal structural requirements for action application.

Example: elevators goal1. Consider two training instances for $\forall p.\ \mathbf{F}(\mathit{served}(p))$:

Instance p2: 2 floors, passenger p_0 with origin f_0 and destination f_1.

- Optimal plan includes: $\mathit{board}(f_0, p_0)$, $\mathit{up}(f_0, f_1)$, $\mathit{depart}(f_1, p_0)$
- *board* precondition set: $\{\mathit{lift\text{-}at}(f_0), \mathit{origin}(p_0, f_0), \texttt{nfa\_at\_n0}(p_0), \mathit{above}(f_0, f_1)\}$

Instance p3: 3 floors, passenger p_1 with origin f_2 and destination f_0.

- Optimal plan includes: $\mathit{board}(f_2, p_1)$, $\mathit{down}(f_2, f_1)$, $\mathit{down}(f_1, f_0)$, $\mathit{depart}(f_0, p_1)$
- *board* precondition set: $\{\mathit{lift\text{-}at}(f_2), \mathit{origin}(p_1, f_2), \texttt{nfa\_at\_n0}(p_1), \mathit{above}(f_1, f_2)\}$

The intersection yields $\{\mathit{lift\text{-}at}(?a0), \mathit{origin}(?a1, ?a0), \texttt{nfa\_at\_n0}(?a1)\}$. The specific floor constants and *above* relations, which vary across instances, are eliminated. The retained predicates encode the structural requirements: the elevator and passenger must be co-located at the passenger's origin floor, and the passenger must not yet be served. The *above*

predicate is instance-dependent (reflecting navigation topology) rather than a precondition for boarding.

Limitation. Intersection may overgenerate if training instances employ structurally distinct strategies rather than merely differing in object bindings. This is mitigated by using BFS-optimal plans, which ensures all training paths have identical structure and length, differing only in the specific objects involved.

5.3.7 Priority Assignment

Priorities determine the order in which rules are tried during execution. The assignment uses two criteria in sequence.

First, rules are classified by whether they contain NFA state predicates in their body:

- **NFA-aware rules (base priority 0–99):** Rules with at least one NFA predicate (e.g., `nfa_at_n0`($?a0$), `nfa_at_n1`($?c0$)). These rules check temporal progress and only match when the current NFA state satisfies their conditions.
- **Generic rules (base priority 100–199):** Rules without any NFA predicates. These receive a +100 priority penalty to ensure they only fire when no NFA-aware rule applies.

Second, within each category, rules are further refined by how they change the NFA state:

1. **Goal-advancing actions** (base priority 0): Actions that push an object to an accepting NFA state (e.g., *depart*, *debark*, *release*).
2. **Progressing actions** (base priority 1): Actions that advance NFA states but not to accepting states (e.g., *board*, *grasp*).
3. **Navigation/support actions** (base priority 2): Actions that do not change NFA states but support goal achievement (e.g., *up*/*down*, *move*, *fly*).

The final priority is computed as: $priority = base\_priority \times 10 + position\_offset + NFA\_penalty$, where $base\_priority \in \{0, 1, 2\}$ indicates the NFA advancement type, $position\_offset \in \{0, \dots, 4\}$ reflects the action's normalized position in the optimal plan (computed as $\lfloor \frac{step\_idx}{plan\_length} \times 5 \rfloor$), and $NFA\_penalty = 100$ if the rule body contains no NFA predicates, otherwise 0. For multi-phase goals, the priority is further adjusted by multiplying by 10 and adding a phase index to ensure proper ordering. The position offset ensures that actions appearing earlier in optimal plans receive higher priority, preserving the temporal ordering observed during training.

Example priorities (elevators goal1):

- *depart* with `nfa_at_n0`($?a1$): base priority $1 \times 10 + 2 = 12$ (progressing, phase 2/3)
- *board* with `nfa_at_n0`($?a1$): base priority $2 \times 10 + 1 = 21$ (navigation, phase 2/2)

- *up* with `nfa_at_n0`(?$c0$): base priority $2 \times 10 + 0 = 20$ (navigation with context)
- *down* without NFA predicates: $12 + 100 = 123$ (generic navigation with penalty)

This prioritization ensures that rules with temporal guards are tried before rules that check only domain predicates.

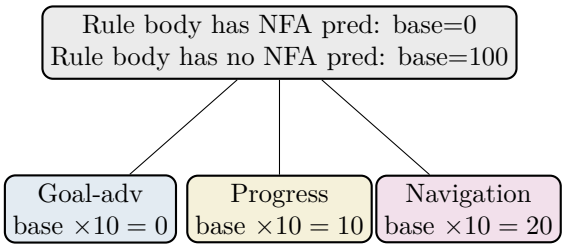

Figure 5.4: Priority assignment decision tree. Rules are first classified by NFA predicate presence (determining the NFA penalty), then by NFA advancement type (determining base priority). Position offsets (not shown) add 0–4 to each computed value based on the action's normalized position in the optimal plan. For example, a goal-advancing action with NFA predicates at the start of the plan receives base priority $0 \times 10 = 0$, position offset 0, and NFA penalty 0, yielding final priority 0.

Rules without NFA predicates receive a 100-point penalty because they can match even when temporal goals are satisfied. For instance, a rule $at(x, f) \rightarrow board(x, f)$ without `nfa_at_n0`(x) would board passengers that have already been served. The penalty ensures such rules only fire when no temporally-aware rule applies.

5.3.8 Constant Generalization

The rule extractor must decide which constants to preserve and which to generalize. The policy is:

- **Goal constants** (e.g., `city2`, `room2`, `loc2`) that appear in the temporal goal formula are preserved as literals, since they define the goal semantics.
- **Domain constants** (e.g., `robot1`, specific floor names) are generalized to fresh variables, enabling the rule to apply across different object configurations.

Goal constants are extracted by parsing the temporal goal string and identifying arguments that are not quantifier-bound variables. For instance, in $\forall p.\ \mathbf{F}(at(p, city2))$, the variable p is bound by $\forall$ while `city2` is a constant.

5.4 Policy Execution

5.4.1 Execution Algorithm

Algorithm 4 describes the execution procedure. At each step, the current PDDL state is augmented with NFA progress predicates, and rules are tried in priority order.

Algorithm 4 Lifted Policy Execution

Require: Domain $\mathcal{D}$, problem P, policy π, NFA $\mathcal{N}$, max steps T
Ensure: **true** if temporal goal satisfied, **false** otherwise

```
s ← InitialState(P)
tree ← BuildBindingTree(N, P)                          NFA tracking
recent ← []                                            Recent states for cycle breaking
for t = 1, ..., T do
    s⁺ ← s ∪ FlattenNFAPredicates(tree)                Augmented state
    if CheckGoal(tree, N) then
        return true
    idx ← BuildPredicateIndex(s⁺)
    for each rule r in π (by priority) do
        G ← FindGroundings(r, idx)
        for each assignment θ ∈ G do
            a ← GroundAction(r.schema, θ)
            if Preconditions(a, s) and Apply(a, s) ≠ s then
                if Apply(a, s) ∉ recent then
                    s ← Apply(a, s)
                    tree ← ProgressBindingTree(tree, s, N)
                    update recent; goto next step
    return false                                       No rule applicable
return false                                           Max steps exceeded
```

5.4.2 Variable Grounding via Backtracking

The FindGroundings procedure uses backtracking search to find all variable assignments that satisfy a rule's body. The algorithm:

1. Splits body atoms into positive and negative sets.
2. Sorts positive atoms by selectivity (fewest matching instances first) for efficient pruning.
3. Performs depth-first backtracking over positive atoms, extending the assignment at each step.
4. After finding assignments from positive atoms, filters by negative conditions (checking absence of NOT_p predicates).
5. For existential variables, only requires that at least one satisfying grounding exists.

5.4.3 Negative-Only Variable Binding

A subtle issue arises when an action variable appears only in negative atoms. For example, in the gripper domain:

$$NOT\_at\text{-}robby(?a1),\ at\text{-}robby(?a0) \longrightarrow move(?a0, ?a1)$$

The variable $?a1$ (the destination room) appears only in $NOT\_at\text{-}robby(?a1)$. Since backtracking only binds variables through positive atoms, $?a1$ would remain unbound. The solution involves:

1. Accepting partial groundings from backtracking (negative-only variables stay unbound).
2. Expanding unbound action variables by enumerating all domain objects.
3. Filtering expanded groundings against the negative conditions.

This mechanism is essential for domains where reachability is implicit rather than encoded through explicit connectivity predicates.

5.4.4 Cycle Breaking

Navigation rules can cause infinite loops if the policy repeatedly visits the same states. The execution algorithm maintains a sliding window of the 8 most recently visited PDDL states. When a grounded action would produce a state already in this window, the grounding is skipped in favor of an alternative. If no non-recent alternative exists, execution halts.

5.5 Experimental Setup

We evaluate the lifted rule approach on 47 temporal goals across 11 domains. The domains and goals are adapted from the FO-LTL$_\text{f}$ benchmark [2] and the PLAN4Past benchmark [3]: blocksworld (7 goals), elevators (8 goals), robot (7 goals), zenotravel (9 goals), gripper (4 goals), ferry (4 goals), spanner (2 goals), childsnack (2 goals), depot (1 goal), logistics (2 goals), and visitall (1 goal). Some goals are from the original benchmark, while others were created for this work to explore different temporal patterns and quantifier structures expressible in FO-LTL$_\text{f}$.

5.5.1 Training Instance Selection

Training instances were selected to balance computational feasibility with adequate coverage of goal-relevant configurations. The selection process followed these principles:

1. **Start small**: Begin with the smallest instance (typically p2 with 2–4 objects) to minimize product automaton size and training time.

2. **Incremental scaling**: Add progressively larger instances (p3, p4, etc.) with increasing object counts.
3. **Diversity for complex goals**: For goals requiring coordinated action sequences, include instances with diverse object configurations (e.g., different initial positions, multiple object arrangements).
4. **Computational constraints**: Stop adding instances when product automaton size approaches the maximum state limit (typically 10,000 states for training) or when additional instances provide diminishing coverage.

This process resulted in 1–6 training instances per goal (median 2). Simple monotonic goals (e.g., "all blocks on table") typically required 2 instances (p2, p3), while complex goals requiring sequential coordination (e.g., "inspect all items at room 1, then deliver to room 2") required 4–6 carefully selected instances covering different action orderings and object configurations.

Testing uses 4–16 held-out instances of increasing size (more than 30 objects in some cases, with 500-step execution limits). All experiments run on a single core with at most 8 GB of memory; training completes in 0.6–207.3 seconds depending on domain complexity.

5.6 Results

Table 5.1 presents the complete results from the latest experiment. The lifted rule approach achieves 100 % generalization on 34 out of 47 goals across 11 domains, with an overall test instance solving rate of 88.7 % (385 out of 434 test instances).

5.7 Analysis of Learned Policies

We show examples of learned policies for several domains.

5.7.1 Simple Universal Goals: Blocksworld

The simplest policies emerge for blocksworld goals with universal quantification. For $\forall x.\, \mathbf{F}(\mathit{ontable}(x))$ (put all blocks on the table), three rules suffice (Figure 5.5). The policy unstacks towers top-to-bottom, placing each block on the table.

5.7.2 Multi-Phase Goals: Elevators

The elevators goal1 ($\forall p.\, \mathbf{F}(\mathit{served}(p))$) requires a 4-phase policy: navigate to origin → board → navigate to destination → depart. The learned policy contains 5 rules organized by priority (Figure 5.6). The lifted rules bind $?a1$ to a specific passenger and $?a0$ to the correct floor.

```
(:rule rule-1
:parameters (?a0 ?a1)
:preconditions (and
(on ?a0 ?a1)
)
:action (unstack ?a0
?a1)
)
```

```
(:rule rule-2
:parameters (?a0 ?a1)
:preconditions (and
(on ?a0 ?a1)
(nfa_at_n0 ?a0)
(nfa_at_n0 ?a1)
)
:action (unstack ?a0
?a1)
)
```

```
(:rule rule-3
:parameters (?a0)
:preconditions (and
(holding ?a0)
(nfa_at_n0 ?a0)
)
:action (put-down ?a0)
)
```

Figure 5.5: Learned policy for blocksworld $\forall x.\ \mathbf{F}(\mathit{ontable}(x))$. Rule 1 unstacks any block; rule 2 is an NFA-aware variant for unserved blocks; rule 3 puts a held block on the table.

```
(:rule depart [priority 12]
:parameters (?a0 ?a1)
:preconditions (and
(boarded ?a1)
(destin ?a1 ?a0)
(lift-at ?a0)
(nfa_at_n0 ?a1)
(not-served ?a1)
)
:action (depart ?a0 ?a1)
)
```

```
(:rule up-nfa [priority 20]
:parameters (?a0 ?a1)
:context (exists (?c0))
:preconditions (and
(not (lift-at ?a1))
(above ?a0 ?a1)
(lift-at ?a0)
(nfa_at_n0 ?c0)
)
:action (up ?a0 ?a1)
)
```

```
(:rule board [priority 21]
:parameters (?a0 ?a1)
:preconditions (and
(lift-at ?a0)
(nfa_at_n0 ?a1)
(not-boarded ?a1)
(not-served ?a1)
(origin ?a1 ?a0)
)
:action (board ?a0 ?a1)
)
```

```
(:rule down [priority 123]
:parameters (?a0 ?a1)
:preconditions (and
(not (lift-at ?a1))
(above ?a1 ?a0)
(lift-at ?a0)
)
:action (down ?a0 ?a1)
)
```

Figure 5.6: Learned policy for elevators $\forall p.\ \mathbf{F}(\mathit{served}(p))$ (4 of 5 rules shown; up without NFA guard omitted). Priority 12 (depart) fires when a passenger reaches destination. Priority 21 (board) boards unserved passengers at origin. Priority 20 (up-nfa) navigates when unserved passengers exist (existential context $\exists ?c0$). Priority 123 (down) is generic navigation without NFA guard.

5.7.3 Existential Goals: Gripper

The gripper goal2 ($\exists b.\ \mathbf{F}(\mathit{at}(b, \mathit{roomb}))$) requires moving some ball to *roomb*. The learned policy has 4 rules (Figure 5.7).

Key observations: (1) The goal constant *roomb* is preserved in the drop and move-to-goal actions. (2) Drop and pick place `nfa_at_n0`($?a0$) on the ball parameter directly, since

```
(:rule drop [priority 3]
:parameters (?a0 ?a2)
:preconditions (and
(not (free ?a2))
(at-robby roomb)
(carry ?a0 ?a2)
(nfa_at_n0 ?a0)
)
:action (drop ?a0 roomb ?a2)
)
```

```
(:rule move-to-goal [priority 21]
:parameters (?a0)
:context (exists (?c0))
:preconditions (and
(not (at-robby roomb))
(at-robby ?a0)
(nfa_at_n0 ?c0)
)
:action (move ?a0 roomb)
)
```

```
(:rule pick [priority 10]
:parameters (?a0 ?a1 ?a2)
:preconditions (and
(at ?a0 ?a1)
(at-robby ?a1)
(free ?a2)
(nfa_at_n0 ?a0)
)
:action (pick ?a0 ?a1 ?a2)
)
```

```
(:rule move [priority 21]
:parameters (?a0 ?a1)
:context (exists (?c0))
:preconditions (and
(not (at-robby ?a1))
(at-robby ?a0)
(nfa_at_n0 ?c0)
)
:action (move ?a0 ?a1)
)
```

Figure 5.7: Learned policy for gripper $\exists b.\ \mathbf{F}(at(b, roomb))$. Drop and pick rules have NFA guards on the ball parameter $?a0$ (priority 3, 10). Move rules use existential context $\exists ?c0$ to navigate only when unfinished balls exist (priority 21). The goal constant *roomb* appears directly in actions.

the tracked object is an action parameter. (3) Move rules cannot check NFA state on the ball (robby does not know which ball to track), so they use an existential context variable $\exists ?c0 : \texttt{nfa\_at\_n0}(?c0)$ to verify some unfinished ball exists. (4) For $\exists$ goals, execution terminates as soon as any ball reaches *roomb*.

5.7.4 Nested Quantification: $\forall\exists$

The zenotravel goal4 ($\forall p\ \exists a.\ \mathbf{F}(at(p, city2) \vee in(p, a))$) requires every person to reach *city2* or board some aircraft. The learned policy has 2 rules (Figure 5.8).

The board rule places $\texttt{nfa\_at\_n0}(?a0)$ on the person parameter $?a0$ because the outer $\forall p$ quantifier requires tracking each person's progress. The inner $\exists a$ means any aircraft suffices, so the aircraft parameter $?a1$ has no NFA condition: person $?a0$ boards whichever co-located aircraft $?a1$ is available.

The fly rule has no NFA guard and receives priority 101 (no NFA predicates). It navigates aircraft without reference to persons. Board has higher priority (0 vs. 101), so whenever a person can board, board fires first. The disjunctive goal $(at(p, city2) \vee in(p, a))$ is satisfied by either outcome; the NFA advances to an accepting state for person p upon boarding. Once all persons satisfy the goal, the binding tree is complete and execution terminates.

```
(:rule board [priority 0]
:parameters (?a0 ?a1 ?a2)
:preconditions (and
(at ?a0 ?a2)
(at ?a1 ?a2)
(nfa_at_n0 ?a0)
(notin ?a0 ?a1)
)
:action (board ?a0 ?a1 ?a2)
)
```

```
(:rule fly [priority 101]
:parameters (?a0 ?a1 ?a2 ?a3 ?a4)
:preconditions (and
(at ?a0 ?a1)
(fuel-level ?a0 ?a3)
(next ?a4 ?a3)
)
:action (fly ?a0 ?a1 ?a2 ?a3 ?a4)
)
```

Figure 5.8: Learned policy for zenotravel $\forall p\, \exists a.\ \mathbf{F}(at(p, city2) \vee in(p, a))$. The board rule checks `nfa_at_n0`($?a0$) on the person parameter (priority 0). The fly rule has no NFA guard (priority 101) and navigates aircraft without tracking persons. Once all persons are boarded or at *city2*, execution terminates.

5.8 Training Instance Diversity

Training instance diversity is critical for generalization. Rules are extracted from optimal plans, so if a training instance does not require a certain action sequence (e.g., downward navigation), the corresponding rule will not appear in the policy.

The recommended training strategy includes:

- Instances of varying sizes (few objects, many objects).
- Instances of varying shapes (tall/narrow, wide/shallow configurations).
- At least one instance with objects spread across different locations, to ensure navigation rules appear in optimal plans.
- For nested $\forall\exists$ goals: keeping training instances small due to state space explosion.

All four new domains (childsnack, depot, logistics, visitall) required careful selection of training instances to ensure navigation and action diversity.

5.9 Discussion

5.9.1 Failure Analysis

Three goals in Table 5.1 achieve less than 100 % generalization. We analyze the causes.

Pattern 1: Nested quantifiers. Blocksworld goal6 requires pairing each block x with some block y. With n blocks, there are $n(n-1)$ possible pairings. Training instances cannot exhaustively cover all pairings. Intersection generalizes predicates but cannot infer pairing constraints absent from training.

Pattern 2: Rare temporal sequences. Zenotravel goal7 ($\mathbf{F}(\neg at(p, c_1) \wedge \mathbf{F}(at(p, c_1)))$) requires leaving and returning to the same location. Optimal plans in small training instances (2–3 cities) rarely exhibit this pattern. The extracted policy lacks rules for the return phase.

Pattern 3: Undertrained configurations. Zenotravel goal9 has 80 % success. The failures occur on test instances with city configurations not represented in training. Two additional training instances would likely address this.

All three failure modes stem from insufficient training coverage. The approach does not fail due to representational limits or algorithmic issues. Increasing training diversity (more instances, larger instances, or manually designed edge cases) addresses these failures.

5.9.2 Complexity

Training. Given n training instances with average product automaton size $|S|$ and branching factor $|A|$, the extraction algorithm has the following costs:

1. Product automaton construction: $O(n \cdot |S|)$
2. BFS to find k shortest paths: $O(n \cdot |S| \cdot |A|)$
3. Context extraction from k paths of length ℓ: $O(n \cdot k \cdot \ell)$
4. Intersection across k contexts for m action schemas: $O(m \cdot k^2 \cdot |B|)$ where $|B|$ is body size

Total: $O(n \cdot |S| \cdot |A| + m \cdot k^2 \cdot |B|)$. The first term dominates.

Observed values from Table 5.1:

- $n \in [1, 6]$ training instances
- $|S|$ varies widely across domains (from tens to tens of thousands of states)
- $k = 10$ paths, $\ell \approx 20$ steps, $m \approx 10$ schemas, $|B| \leq 10$ atoms
- Training time: 0.6–207.3 seconds
- Memory: up to 1.5 GB, typically under 100 MB

The training time varies significantly based on domain complexity and product automaton size.

Execution. Let $|R|$ = number of rules, $|O|$ = objects in test instance, $|B|$ = max rule body size.

Per execution step:

- Try rules in priority order until one matches
- For each rule: backtracking search over body atoms to find groundings
- Worst case per step: $O(|R| \cdot |O|^{|B|})$

- Typical case: high-priority rules match early, $O(|O|^{|B|})$

Observed values:

- $|R| \in [2, 83]$ rules (median 6)
- $|B| \in [2, 8]$ body atoms (median 4)
- NFA predicates reduce branching: few objects in each NFA state
- Test instances with 30+ objects execute in under 500 steps

Execution is dominated by grounding search. Indexing on predicates reduces the effective search space. The two-tier priority system ensures NFA-aware rules (which are more selective) are tried first.

Table 5.1: Lifted rule extraction results across 11 domains and 47 temporal goals. Ops = temporal operators used (F=eventually, U=until, G=globally, X=next). Q = quantifier pattern. NFA = number of NFA states. Rules = number of extracted rules. Train = training instances used. Solved = number of test instances solved. Rate = test success rate. Time = total training time.

Domain	Goal	Ops	Q	NFA	Rules	Train	Solved	Rate	Time
blocksworld	goal1	F	∀	3	4	p2, p3	16/16	**100%**	95.1s
blocksworld	goal2	F	∀	3	5	p2, p3, p4	7/12	58%	28.5s
blocksworld	goal3	F	∀	4	5	p13, p2	13/13	**100%**	50.1s
blocksworld	goal6	F	∀-∃	3	9	p2, p3, p3b	4/13	31%	180.5s
blocksworld	goal7	U	∀	3	3	p2, p3	13/13	**100%**	79.0s
blocksworld	goal11	F+G	∀	3	3	p2, p3	16/16	**100%**	125.3s
blocksworld	goal12	F+G+X	∀	4	4	p2, p3	16/16	**100%**	137.9s
childsnack	goal1	F	∀	3	6	p2, p3	3/3	**100%**	64.3s
childsnack	goal2	F	∃	3	5	p2, p3	3/3	**100%**	35.1s
depot	goal1	F	∀	3	3	p2, p3	3/3	**100%**	34.2s
elevators	goal1	F	∀	3	5	p3, p2	8/8	**100%**	36.6s
elevators	goal2	F	∀	3	3	p3, p4	14/14	**100%**	40.4s
elevators	goal3	F	∃	3	5	p2, p3, p4	7/13	54%	76.2s
elevators	goal4	F	∀	4	3	p2, p3	11/14	79%	91.0s
elevators	goal5	F	∀	3	2	p2, p3	14/14	**100%**	53.0s
elevators	goal6	U	∀	3	4	p3, p4	10/14	71%	61.9s
elevators	goal11	U	∀-∀	5	3	p2, p3	3/3	**100%**	24.3s
elevators	goal12	F+X	∀	5	4	p_minimal, p2, p3	0/0	0%	0.0s
ferry	goal1	F	∀	3	3	p2, p3	8/8	**100%**	54.5s
ferry	goal2	F	∃	3	3	p2, p3	5/8	62%	51.9s
ferry	goal3	F	∀	3	4	p2, p3, p4	7/7	**100%**	48.6s
ferry	goal4	F+X	∀	5	3	p_minimal, p2	8/8	**100%**	47.2s
gripper	goal1	F	∀	3	4	p2, p3	9/9	**100%**	58.2s
gripper	goal2	F	∃	3	4	p2, p3, p_robot_at_b	6/6	**100%**	60.1s
gripper	goal3	F	∀	3	4	p2, p3, p7	8/8	**100%**	86.5s
gripper	goal5	F+X	∀	5	9	p2, p3, p4, p_mixed	0/6	0%	88.3s
logistics	goal1	F	∀	3	4	p2, p3	3/3	**100%**	32.1s
logistics	goal2	F	∃	3	4	p2, p3	2/2	**100%**	19.5s
robot	goal1	F	∀	3	12	p3, p4	8/8	**100%**	101.9s
robot	goal2	F	∃	3	6	p3, p2b	7/8	88%	57.9s
robot	goal3	F	∀	3	8	p3, p7	8/8	**100%**	107.6s
robot	goal4	F	∀	3	2	p2, p3	6/8	75%	50.8s
robot	goal9	F+G	∀	3	2	p2, p3	8/8	**100%**	44.7s
robot	goal10	F	∀	6	14	p2, p5, p11	1/7	14%	64.0s
robot	goal11	F+X	∀	5	25	p1, p2, p12, p11, p14, p13	8/8	**100%**	89.2s
spanner	goal1	F	∀	3	3	p1, p2	8/8	**100%**	66.7s
spanner	goal2	F	∃	3	3	p1, p2	8/8	**100%**	32.5s
visitall	goal1	F	∀	3	1	p2, p3	1/3	33%	34.2s
zenotravel	goal1	F	∀	3	5	p2, p3, p_3city, p_2aircraft	13/13	**100%**	75.5s
zenotravel	goal2	F	∀-∃	3	2	p2, p3, p_3city, p_2aircraft	13/13	**100%**	179.2s
zenotravel	goal3	F	∃	3	4	p2, p3, p_3city	6/6	**100%**	47.1s
zenotravel	goal4	F	∀-∃	4	2	p2, p3, p_3city, p_2aircraft	13/13	**100%**	108.1s
zenotravel	goal5	F	∀-∃-∃	3	2	p2, p3, p_3city, p_2aircraft	13/13	**100%**	207.3s
zenotravel	goal6	F	∃	3	1	p2, p3, p_3city	11/11	**100%**	92.7s
zenotravel	goal11	F	∀	5	8	p_train1, p_train2,	15/15	**100%**	147.0s

Table 5.2: Effect of training instance diversity on elevators generalization.

Goal	**p2+p3 only**	**Diverse training**
goal1 ($\forall p.\, \mathbf{F}(\mathit{served})$)	40–75 %	100 %
goal2 ($\forall p.\, \mathbf{F}(\mathit{boarded})$)	12.5 %	100 %
goal5 ($\forall f.\, \mathbf{F}(\mathit{lift\text{-}at})$)	25 %	100 %
goal6 ($\forall p.\, (\neg \mathit{served}\, \mathbf{U}\, \mathit{boarded})$)	12.5 %	100 %

Table 5.3: Goals with partial success

Domain	**Formula**	**Rate**	**Analysis**
Blocksworld	$\forall x\, \exists y.\, \mathbf{F}(\mathit{on}(x, y))$	15 %	Nested quantifier creates $O(n^2)$ object pairings. 3 training instances cover subset of configurations; test instances require uncovered pairings.
Zenotravel	$\forall p.\, \mathbf{F}(\neg \mathit{at}(p, c_1) \wedge \mathbf{F}(\mathit{at}(p, c_1)))$	53 %	Requires flying away from c_1 then returning. This pattern is rare in small training instances. 8/15 test instances solvable with extracted rules.
Zenotravel	$\forall p\, \exists a.\, \mathbf{F}(\mathit{in}(p, a) \wedge \mathbf{XF}(\mathit{at}(p, c_2)))$	80 %	Requires boarding then debarking at c_2. Some test instances need intermediate stops not present in training paths. 8/10 test instances solved.

6 Conclusion

6.1 Summary

This thesis addressed the problem of learning generalized policies for first-order temporally extended goals expressed in FO-LTL$_f$. Given a PDDL domain, small training instances, and a temporal goal formula, we developed methods to produce policies that generalize to unseen instances with arbitrary numbers of objects.

We investigated two distinct approaches:

ASP-based policy synthesis. We applied the DL feature framework to the product automaton of PDDL states and NFA tracking states, using ASP to synthesize a minimal policy. While this approach produces high-quality policies when applicable, achieving 100 % success on 7 out of 11 evaluated goals, it suffers from a fundamental scalability barrier: the ASP grounding explosion makes complex goals with large product automata intractable.

Lifted rule extraction. We proposed a novel method that extracts first-order policy rules directly from BFS-optimal plans in the product automaton. Rules use variables that bind to specific objects at execution time, avoiding the ASP solver entirely and eliminating the grounding explosion. The key innovation is existential context variables that link navigation actions to goal-directed motivation without specifying which object to serve next.

Our evaluation across 11 planning domains and 47 temporal goals demonstrates that the lifted rule approach achieves 100 % generalization on 34 out of 47 goals and solves 88.7 % of test instances (385 out of 434), with training times of 0.6–207.3 seconds and memory usage under 1.5 GB, orders of magnitude faster and lighter than the ASP approach.

6.2 Limitations and Future Work

While the lifted rule extraction approach demonstrates strong generalization across diverse temporal goals, several limitations suggest directions for future work:

EPNF restriction. The current approach requires all quantifiers at the outermost scope. Goals with nested quantifiers like $\forall p.\ \mathbf{F}(\exists a.\ in(p, a))$ cannot be directly expressed, as the binding tree tracks per-object NFA progress only for objects bound at preprocessing.

Future work could investigate Skolemization or runtime witness introduction to handle nested quantification.

Training instance selection. The manual selection of training instances does not guarantee complete coverage of test configurations. As shown in Table 5.1, 13 out of 47 goals achieve only partial generalization due to insufficient training diversity. An active learning approach that identifies coverage gaps and automatically selects instances could eliminate manual tuning and provide systematic generalization guarantees.

Intersection over-generalization. Rule body intersection (Section 5.3.6) may inadvertently remove predicates essential for correctness on certain test configurations. Developing principled criteria to distinguish genuinely instance-specific atoms from correctness-critical ones would improve generalization reliability.

Multi-agent coordination. Extending the framework to multi-agent settings would require distributed binding trees and coordination primitives, enabling generalized policies for multi-robot systems with shared temporal goals.

A Appendix

A.1 ASP Encoding

The ASP policy synthesis program (`solve.lp`) encodes the policy learning problem as a Clingo logic program. The key constraint groups are:

1. **Feature selection.** Choice rules select a subset of features:

```
{selected(F)} :- feature(F).
```

2. **Good action selection.** Each alive non-goal state must have at least one "good" action:

```
{good(I,S,A)} :- trans(I,S,A,S'), alive(I,S).
:- alive(I,S), not goal(I,S),
   #count{A : good(I,S,A)} < 1.
```

3. **Boolean distinguishability.** States with different good actions must differ on at least one selected Boolean feature:

```
bool_dist(I1,S1,I2,S2) :-
  selected(F), boolean(F),
  eval(I1,S1,F,V1), eval(I2,S2,F,V2), V1!=V2.
```

4. **Transition-delta consistency.** For states with identical selected feature values, good actions must produce consistent feature value changes:

```
trans_delta(I,S,F,D) :-
  good(I,S,A), trans(I,S,A,S'),
  selected(F), eval(I,S,F,V), eval(I,S',F,V'),
  D = V'-V.
```

5. **Safety constraint.** Inductive argument that all alive states reach a goal via good transitions.
6. **Optimization.** Minimize total feature complexity:

```
#minimize{C,F : selected(F), cost(F,C)}.
```

The dominant term in the ground program size is the transition-delta consistency constraint, which generates $O(S^2 \times F \times A^2)$ ground atoms (see Section 4.4.2).

A.2 Complete Learned Policies

This section lists the complete learned policies for selected goals.

A.2.1 Elevators Goal 1: $\forall p.\, \mathbf{F}(\textbf{\textit{served}}(p))$

```
1  [12] boarded(?a1), destin(?a1, ?a0), lift-at(?a0),
2       nfa_at_n0(?a1), not-served(?a1)
3         → depart(?a0, ?a1)
4
5  [20] NOT_lift-at(?a1), above(?a0, ?a1),
6       lift-at(?a0), nfa_at_n0(?c0)
7         → up(?a0, ?a1)  [∃?c0]
8
9  [21] lift-at(?a0), nfa_at_n0(?a1), not-boarded(?a1),
10      not-served(?a1), origin(?a1, ?a0)
11        → board(?a0, ?a1)
12
13 [123] NOT_lift-at(?a1), above(?a1, ?a0),
14       lift-at(?a0)
15         → down(?a0, ?a1)
16
17 [132] NOT_lift-at(?a1), above(?a0, ?a1),
18       lift-at(?a0)
19         → up(?a0, ?a1)
```

A.2.2 Robot Goal 1: $\forall o.\, \mathbf{F}(\textbf{\textit{at-obj}}(o, \mathit{room2}))$

```
1  [10] at(?a0, ?a2), at-obj(?a1, ?a2),
2       handempty(?a0), nfa_at_n0(?a1)
3         → grasp(?a0, ?a1, ?a2)
4
5  [22] NOT_closed(?a1), NOT_handempty(?a0),
6       at(?a0, ?a2), connects(?a1, ?a2, room2),
7       connects(?a1, room2, ?a2),
8       holding(?a0, ?c0), nfa_at_n0(?c0)
9         → move(?a0, ?a1, ?a2, room2)  [∃?c0]
10
11 [40] at(?a0, ?a2), closed(?a1),
12      connects(?a1, ?a2, room2),
13      connects(?a1, room2, ?a2)
14        → open(?a0, ?a1, ?a2, room2)
15
16 [43] NOT_handempty(?a0), at(?a0, room2),
```

```
      holding(?a0, ?a1), nfa_at_n0(?a1)
        → release(?a0, ?a1, room2)

% ... (8 additional navigation and door-opening rules omitted)
```

A.2.3 Spanner Goal 1: $\forall n.\, \mathbf{F}(\textit{tightened}(n))$

```
[41] at(?a0, ?a1), link(?a1, ?a2), link(?a2, ?a1)
       → walk(?a0, ?a1, ?a2)

[41] at(?a0, ?a2), at(?a1, ?a2), useable(?a1)
       → pickup-spanner(?a0, ?a1, ?a2)

[43] NOT_tightened(?a2), at(?a0, ?a3),
     at-nut(?a2, ?a3), carrying(?a0, ?a1),
     loose(?a2), nfa_at_n0(?a2), useable(?a1)
       → tighten-nut(?a0, ?a1, ?a2, ?a3)
```

A.3 Domain Specifications

All PDDL domain and problem files used in this work are available in the accompanying code repository: `https://github.com/CHaJungn/genTEGFO.git`.

A.4 Temporal Goal Specifications

Table A.1 lists all 47 temporal goals used in the lifted rules evaluation, with their EPNF formulas and quantifier patterns across 11 planning domains.

Table A.1: Complete set of 47 temporal goals used in the lifted rules evaluation.

Domain	**Goal**	**Formula**	**Q**
blocksworld	goal1	$\forall x.\ \mathbf{F}(\mathit{ontable}(x))$	$\forall$
blocksworld	goal2	$\forall x.\ \mathbf{F}(\mathit{clear}(x))$	$\forall$
blocksworld	goal3	$\forall x.\ \mathbf{F}(\mathit{ontable}(x) \vee \mathit{holding}(x))$	$\forall$
blocksworld	goal6	$\forall x\ \exists y.\ \mathbf{F}(\mathit{on}(x, y))$	$\forall$-$\exists$
blocksworld	goal7	$\forall x.\ (\mathit{clear}(x)\ \mathbf{U}\ \mathit{ontable}(x))$	$\forall$
blocksworld	goal11	$\forall x.\ \mathbf{F}(\mathbf{G}(\mathit{ontable}(x)))$	$\forall$
blocksworld	goal12	$\forall x.\ \mathbf{F}(\mathit{holding}(x) \wedge \mathbf{X}(\mathbf{F}(\mathit{ontable}(x))))$	$\forall$
childsnack	goal1	$\forall c.\ \mathbf{F}(\mathit{served}(c))$	$\forall$
childsnack	goal2	$\exists c.\ \mathbf{F}(\mathit{served}(c))$	$\exists$
depot	goal1	$\forall c.\ \mathbf{F}(\mathit{clear\text{-}crate}(c))$	$\forall$
elevators	goal1	$\forall p.\ \mathbf{F}(\mathit{served}(p))$	$\forall$
elevators	goal2	$\forall p.\ \mathbf{F}(\mathit{boarded}(p))$	$\forall$
elevators	goal3	$\exists p.\ \mathbf{F}(\mathit{served}(p))$	$\exists$
elevators	goal4	$\forall p.\ \mathbf{F}(\mathit{boarded}(p) \vee \mathit{served}(p))$	$\forall$
elevators	goal5	$\forall f.\ \mathbf{F}(\mathit{lift\text{-}at}(f))$	$\forall$
elevators	goal6	$\forall p.\ (\neg \mathit{served}(p)\ \mathbf{U}\ \mathit{boarded}(p))$	$\forall$
elevators	goal11	$\forall v\ \forall r.\ (\mathit{vip}(r) \vee \neg \mathit{vip}(v) \vee (\neg \mathit{boarded}(r)\ \mathbf{U}\ \mathit{boarded}(v)))$	$\forall$-$\forall$
elevators	goal12	$\forall p.\ \mathbf{F}(\mathit{boarded}(p) \wedge \mathbf{X}(\mathbf{F}(\mathit{served}(p))))$	$\forall$
ferry	goal1	$\forall c.\ \mathbf{F}(\mathit{at}(c, \mathit{loc2}))$	$\forall$
ferry	goal2	$\exists c.\ \mathbf{F}(\mathit{at}(c, \mathit{loc2}))$	$\exists$
ferry	goal3	$\forall c.\ \mathbf{F}(\mathit{on}(c))$	$\forall$
ferry	goal4	$\forall c.\ \mathbf{F}(\mathit{on}(c) \wedge \mathbf{X}(\mathbf{F}(\mathit{at}(c, \mathit{loc2}))))$	$\forall$
gripper	goal1	$\forall b.\ \mathbf{F}(\mathit{at}(b, \mathit{roomb}))$	$\forall$
gripper	goal2	$\exists b.\ \mathbf{F}(\mathit{at}(b, \mathit{roomb}))$	$\exists$
gripper	goal3	$\forall b.\ \mathbf{F}(\mathit{carry}(b, \mathit{left}))$	$\forall$
gripper	goal5	$\forall b.\ \mathbf{F}(\mathit{carry}(b, \mathit{left}) \wedge \mathbf{X}(\mathbf{F}(\mathit{at}(b, \mathit{roomb}))))$	$\forall$
logistics	goal1	$\forall p.\ \mathbf{F}(\mathit{at\text{-}pkg}(p, \mathit{depot}))$	$\forall$
logistics	goal2	$\exists p.\ \mathbf{F}(\mathit{at\text{-}pkg}(p, \mathit{depot}))$	$\exists$
robot	goal1	$\forall o.\ \mathbf{F}(\mathit{at\text{-}obj}(o, \mathit{room2}))$	$\forall$
robot	goal2	$\exists o.\ \mathbf{F}(\mathit{at\text{-}obj}(o, \mathit{room2}))$	$\exists$
robot	goal3	$\forall o.\ \mathbf{F}(\mathit{holding}(\mathit{robot1}, o))$	$\forall$
robot	goal4	$\forall d.\ \mathbf{F}(\mathit{opened}(d))$	$\forall$
robot	goal9	$\forall d.\ \mathbf{F}(\mathbf{G}(\mathit{opened}(d)))$	$\forall$
robot	goal10	$\forall o.\ \mathbf{F}(\mathit{at\text{-}obj}(o, \mathit{room2}) \wedge \mathbf{F}(\mathit{at\text{-}obj}(o, \mathit{room1})))$	$\forall$
robot	goal11	$\forall o.\ \mathbf{F}(\mathit{at\text{-}obj}(o, \mathit{room1}) \wedge \mathbf{X}(\mathbf{F}(\mathit{at\text{-}obj}(o, \mathit{room2}))))$	$\forall$
spanner	goal1	$\forall n.\ \mathbf{F}(\mathit{tightened}(n))$	$\forall$
spanner	goal2	$\exists n.\ \mathbf{F}(\mathit{tightened}(n))$	$\exists$
visitall	goal1	$\forall p.\ \mathbf{F}(\mathit{visited}(p))$	$\forall$
zenotravel	goal1	$\forall p.\ \mathbf{F}(\mathit{at}(p, \mathit{city2}))$	$\forall$
zenotravel	goal2	$\forall p\ \exists a.\ \mathbf{F}(\mathit{in}(p, a))$	$\forall$-$\exists$
zenotravel	goal3	$\exists p.\ \mathbf{F}(\mathit{at}(p, \mathit{city2}))$	$\exists$
zenotravel	goal4	$\forall p\ \exists a.\ \mathbf{F}(\mathit{at}(p, \mathit{city2}) \vee \mathit{in}(p, a))$	$\forall$-$\exists$
zenotravel	goal5	$\forall p\ \exists a\ \exists c.\ \mathbf{F}(\mathit{in}(p, a) \wedge \mathit{at}(a, c))$	$\forall$-$\exists$-$\exists$
zenotravel	goal6	$\exists a.\ \mathbf{F}(\mathit{at}(a, \mathit{city2}))$	$\exists$
zenotravel	goal11	$\forall p.\ \mathbf{F}(\neg \mathit{at}(p, \mathit{city1}) \wedge \mathbf{F}(\mathit{at}(p, \mathit{city1})))$	$\forall$
zenotravel	goal12	$\forall a.\ (\mathit{at}(a, \mathit{city1})\ \mathbf{U}\ (\mathit{at}(a, \mathit{city2}) \wedge (\mathit{at}(a, \mathit{city2})\ \mathbf{U}\ \mathit{at}(a, \mathit{city1}))))$	$\forall$
zenotravel	goal13	$\forall p\ \exists a.\ \mathbf{F}(\mathit{in}(p, a) \wedge \mathbf{X}(\mathbf{F}(\mathit{at}(p, \mathit{city2}))))$	$\forall$-$\exists$

List of Acronyms

ASP	Answer Set Programming
BFS	Breadth-First Search
DFA	Deterministic Finite Automaton
DL	Description Logic
EPNF	Extended Prenex Normal Form
FO-LTL$_\mathbf{f}$	First-Order Linear Temporal Logic on Finite Traces
LTL$_\mathbf{f}$	Linear Temporal Logic on Finite Traces
NFA	Nondeterministic Finite Automaton
PDDL	Planning Domain Definition Language

List of Symbols

Planning

$\mathcal{D}$	a PDDL planning domain
P	a problem instance
s	a planning state (set of ground atoms)
s_0	the initial state of a problem instance
$\mathcal{O}$	set of all objects in a problem instance
$\mathcal{P}$	set of all predicates in a domain
$\mathcal{A}$	set of all action schemas in a domain

Temporal Logic

φ	a temporal goal formula
F	eventually (temporal operator)
G	globally / always (temporal operator)
U	until (temporal operator)
X	next (temporal operator)
R	release (temporal operator)
τ	a finite execution trace

Automaton

Q	set of NFA states
δ	NFA transition relation
F	set of accepting NFA states
L	NFA state labelling function
$\mathcal{A}$	the product automaton
T	NFA tracking structure (binding tree)

Policy

π	a generalized policy
r	a lifted rule
B	rule body (conjunction of lifted atoms)
ρ	rule priority (lower is higher)
k	feature complexity parameter

List of Figures

List of Tables

List of Algorithms

List of References

[1] J. A. Baier and S. A. McIlraith, "Planning with first-order temporally extended goals using heuristic search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2006.

[2] J. A. Baier and S. A. McIlraith, "Planning with first-order temporally extended goals using heuristic search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, Benchmark domains, 2006.

[3] L. Bonassi, G. De Giacomo, M. Favorito, F. Fuggitti, A. E. Gerevini, and E. Scala, "Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 33, no. 1, pp. 61–69, Jul. 1, 2023, issn: 2334-0843, 2334-0835. doi: 10.1609/icaps.v33i1.27179. Accessed: Jul. 5, 2026. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/27179.

[4] B. Bonet and H. Geffner, "Learning general policies from small examples without supervision," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[5] A. Camacho, J. A. Baier, C. Muise, and S. A. McIlraith, "Finite LTL synthesis as planning," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2017.

[6] A. Camacho and S. A. McIlraith, "Strong fully observable non-deterministic planning with LTL and LDL goals," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[7] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2013, pp. 854–860.

[8] D. Drexler, G. Francès, and J. Seipp, *DLPlan*, 2022. doi: 10.5281/zenodo.5826139. [Online]. Available: https://doi.org/10.5281/zenodo.5826139.

[9] G. Francès, B. Bonet, and H. Geffner, "Effective generation of DL-based features for generalized planning," in *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 2021.

[10] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Answer set solving in practice," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2012.

[11] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification, Testing and Verification XV*, Springer, 1995, pp. 3–18.

[12] M. Ghallab, D. S. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[13] E. Groshev, M. Goldstein, A. Tamar, S. Srivastava, and P. Abbeel. "Learning Generalized Reactive Policies using Deep Neural Networks." arXiv: 1708.07280 [cs.AI], Accessed: Jul. 5, 2026. [Online]. Available: http://arxiv.org/abs/1708.07280, pre-published.

[14] T. Hofmann and H. Geffner, "Learning Generalized Policies for Fully Observable Non-Deterministic Planning Domains," in *Proceedings of the Thirty-ThirdInternational Joint Conference on Artificial Intelligence*, Jeju, South Korea: International Joint Conferences on Artificial Intelligence Organization, Aug. 2024, pp. 6733–6742, isbn: 978-1-956792-04-1. doi: 10.24963/ijcai.2024/744. Accessed: Apr. 14, 2025. [Online]. Available: https://www.ijcai.org/proceedings/2024/744.

[15] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*, Academic Press, 1971, pp. 189–196.

[16] M. Martín and H. Geffner, "Learning Generalized Policies from Planning Examples Using Concept Languages," *Applied Intelligence*, vol. 20, no. 1, pp. 9–19, Jan. 1, 2004, issn: 1573-7497. doi: 10.1023/B:APIN.0000011138.20292.dd. Accessed: Mar. 23, 2025. [Online]. Available: https://doi.org/10.1023/B:APIN.0000011138.20292.dd.

[17] T. Silver, K. R. Allen, A. K. Lew, L. P. Kaelbling, and J. Tenenbaum. "Few-Shot Bayesian Imitation Learning with Logical Program Policies." arXiv: 1904.06317 [cs.AI], Accessed: Jul. 5, 2026. [Online]. Available: http://arxiv.org/abs/1904.06317, pre-published.

[18] S. Srivastava, N. Immerman, and S. Zilberstein, "A new representation and associated algorithms for generalized planning," *Artificial Intelligence*, vol. 175, no. 2, pp. 615–647, 2011.

[19] S. Toyer, S. Thiébaux, F. Thirunavukarasu, and L. Ong, "Action schema networks: Generalised policies with deep learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[20] R. Yang, T. Silver, A. Curtis, T. Lozano-Pérez, and L. P. Kaelbling, "PG3: Policy-guided planning for generalized policy generation," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2022.