PDDL, Conditional Effects and Axioms

Manuel Buser

03.05.2026

Abstract

Classical planning is the area of artificial intelligence concerned with finding a sequence of actions that transforms an initial state into one satisfying a given goal. The Planning Domain Definition Language (PDDL) provides a standardised formalism for specifying such planning problems, allowing the same problem description to be used by different planning systems. This report presents first the STRIPS fragment of PDDL in detail, covering its syntax, grounding process, and formal semantics. Building on this foundation, we then examine two additional PDDL constructs: conditional effects, which let a single action schema produce different outcomes depending on the current state, and axioms (derived predicates), which let properties be inferred from the state rather than maintained through explicit action effects. For both conditional effects and axioms, we describe the PDDL syntax, give a formal explanation of its semantics, and discuss how it relates to the base language in terms of compactness and compilation cost.

1 Introduction

Classical planning, as introduced in the abstract, requires a formalism in which initial states, goals, and the actions available to transform one into the other can all be expressed. The Planning Domain Definition Language (PDDL) was introduced for the first International Planning Competition in 1998 (McDermott, 2000) to provide such a formalism, with the goal of allowing planning problems to be shared across different planning systems (Haslum et al., 2019). Its simplest supported fragment is STRIPS (Fikes and Nilsson, 1971), in which every action effect applies unconditionally.

Many domains, however, could benefit from context-dependent effects or from properties that follow transitively from the current state. Conditional effects address the first case by allowing a single action schema to produce different outcomes depending on the state (Haslum et al., 2019). Axioms (derived predicates), introduced by Thiébaux et al. (2005), address the second: a property is inferred from the state rather than maintained through explicit action effects. Both yield more compact domain models, and both can be compiled back into STRIPS, though the cost of doing so differs substantially between the two constructs.

The goal of this report is to present these three fragments of PDDL in a unified way, covering the syntax and formal semantics of each and the compilation techniques that translate the two additional features, conditional effects and axioms, back to STRIPS. Section 2 covers the basics of PDDL and introduces the STRIPS fragment. Section 3 presents conditional effects and two compilation techniques for them. Section 4 covers axioms, their stratified semantics, and the cost of compiling them back into STRIPS.

2 The Planning Domain Definition Language

PDDL derives its syntax from the LISP programming language: every expression is enclosed in parentheses, and functions such as `and`, `not`, or `action` are written before their arguments rather than between them (Haslum et al., 2019). The next section covers the STRIPS fragment of PDDL, which excludes conditional effects, and derived predicates. These extensions are treated in Sections 3 and 4.

2.1 Structure and Syntax

A PDDL planning problem consists of two files. The *domain file* declares the vocabulary of the model (requirements, predicates and action schemas), while the *problem file* specifies a concrete instance by listing objects, the initial state, and a goal condition (Haslum et al., 2019).

Domain file

Requirements. The `:requirements` section declares which PDDL features the domain uses. For the STRIPS fragment, the keyword `:strips` is used. Most planners determine the required features by inspecting the domain directly, but a correct requirements declaration is considered good practice (Haslum et al., 2019).

Predicates. The `:predicates` section lists the predicate symbols with their parameters. The number of parameters a predicate takes is called its *arity*. PDDL also permits predicates with no parameters, corresponding to an arity of 0; such a predicate behaves like a propositional variable, true or false without referring to any objects.

Action schemas. An action schema consists of a name, a list of parameters, a precondition, and an effect. Parameter symbols in PDDL must begin with `?` and serve as placeholders for the objects the action operates on (Haslum et al., 2019). A symbol may appear in a precondition or effect only if it is declared in the `:parameters` section of that action (Haslum et al., 2019). The precondition specifies which conditions must hold for the action to be applicable.

The effect specifies how the state changes when the action is applied: facts that appear positive in the effect are added to the state, and facts that appear negated (within `not`) are removed from the state (Haslum et al., 2019).

Problem file

Objects. The `:objects` section of the problem file lists the concrete entities of the instance. When an action schema is grounded, its variable parameters are replaced by objects from this set; more on grounding in Section 2.3.

Initial state and goal. The `:init` section lists all facts true in the starting state. Every unlisted fact is false. This is called the *closed-world assumption* (Haslum et al., 2019). The `:goal` section specifies a condition that must hold at the end of a valid plan.

Running Example

Blocks World The Blocks World is a classic planning domain[1]. A set of blocks sits on a table, and a robot arm can pick up, put down, stack, and unstack one block at a time. The goal is to rearrange the blocks into a target configuration.

We now model this domain in PDDL to illustrate the components above. The domain file begins by naming the domain and declaring that it uses the STRIPS fragment:

```
(define (domain blocks)
  (:requirements :strips)
```

Next, the `:predicates` section declares five predicate symbols. The predicate `on` has arity 2 and represents that one block sits on top of another. The predicates `ontable`, `clear`, and `holding` each have arity 1: `ontable` means a block is directly on the table, `clear` means nothing is stacked on top of it, and `holding` means the arm is currently grasping it. The predicate `handempty` has arity 0 and is true whenever the arm holds no block.

```
  (:predicates
    (on ?x ?y) (ontable ?x) (clear ?x)
    (handempty) (holding ?x))
```

[1]The PDDL formulation used here is adapted from the benchmark collection at `https://github.com/aibasel/downward-benchmarks/blob/master/blocks/domain.pddl`.

The domain defines four action schemas. As an example we take `pick-up`, which takes a single parameter `?x` (the block to pick up). Its precondition requires that `?x` is clear, on the table, and the arm is empty; all three must hold simultaneously. The effect removes `?x` from the table, makes it no longer clear, sets the arm to no longer empty, and records that the arm now holds `?x`:

```
(:action pick-up
  :parameters (?x)
  :precondition (and (clear ?x) (ontable ?x)
                     (handempty))
  :effect (and (not (ontable ?x)) (not (clear ?x))
               (not (handempty)) (holding ?x)))
```

The problem file specifies a concrete instance. It references the domain, declares three objects (`A`, `B`, `C`), and lists the initial state: block `A` sits on `B`, block `C` is on the table beside them, and the arm is empty. The goal requires `A` on `B` and `B` on `C`, so block `B` must be moved underneath `C` while preserving `A` on top.

```
(define (problem blocks-1)
  (:domain blocks)
  (:objects A B C)
  (:init (clear A) (on A B) (ontable B)
         (clear C) (ontable C) (handempty))
  (:goal (and (on A B) (on B C)))
)
```

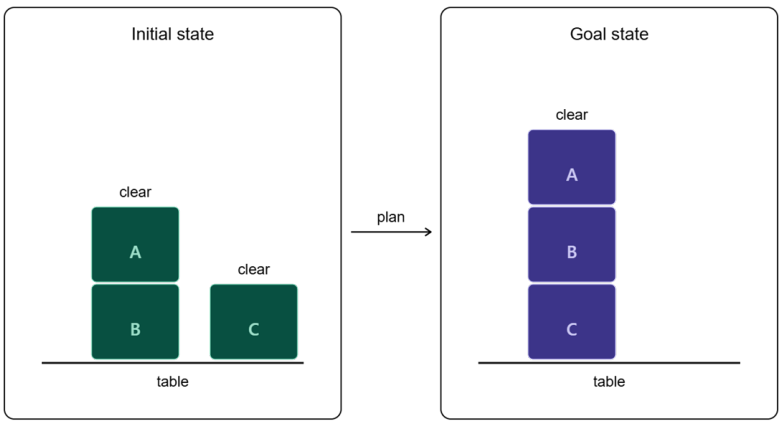

Figure 1: One Blocks World instance: initial state (left) and goal state (right).

The foundations of PDDL come from predicate-logic. Following definitions, taken from Helmert and Röger (2020), are important to understand its syntax and semantics.

Definition 1 (Signature)**.** *A* signature *in predicate logic is a tuple $\mathcal{S} = \langle \mathcal{V}, \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ consisting of the following four disjoint sets: $\mathcal{V}$ is a finite set of variable symbols, $\mathcal{C}$ is a finite set of constant symbols, $\mathcal{F}$ is a finite set of function symbols, and $\mathcal{P}$ is a finite set of predicate symbols. Every predicate symbol $\mathrm{P} \in \mathcal{P}$ has an associated* arity *$ar(\mathrm{P}) \in \mathbb{N}_1$, denoting the number of arguments it takes.*

Definition 2 (Term)**.** *A* term *over $\mathcal{S}$ is either a variable symbol $x \in \mathcal{V}$ or a constant symbol $c \in \mathcal{C}$.*

Definition 3 (Formula)**.** *For a signature $\mathcal{S} = \langle \mathcal{V}, \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$, the set of predicate logic* formulas *(over $\mathcal{S}$) is inductively defined as follows:*

- *If $t_1, \ldots, t_k$ are terms (over $\mathcal{S}$) and $\mathrm{P} \in \mathcal{P}$ is a k-ary predicate symbol, then the* atomic formula *(or* atom*) $\mathrm{P}(t_1, \ldots, t_k)$ is a formula over $\mathcal{S}$.*
- *If t_1 and t_2 are terms (over $\mathcal{S}$), then the* identity *$(t_1 = t_2)$ is a formula over $\mathcal{S}$.*
- *If $x \in \mathcal{V}$ is a variable symbol and φ a formula over $\mathcal{S}$, then the* universal quantification *$\forall x\, \varphi$ and the* existential quantification *$\exists x\, \varphi$ are formulas over $\mathcal{S}$.*
- *If φ is a formula over $\mathcal{S}$, then so is its* negation *$\neg\varphi$.*
- *If φ and ψ are formulas over $\mathcal{S}$, then so are the* conjunction *$(\varphi \wedge \psi)$ and the* disjunction *$(\varphi \vee \psi)$.*

Definition 4 (Ground Atom)**.** *A* ground atom *is an atom $\mathrm{P}(c_1, \ldots, c_k)$ in which every argument is a constant symbol rather than a variable symbol. Let F denote the set of all ground atoms for a given domain and problem instance.*

Definition 5 (Effect)**.** *Effects over F are defined inductively (Helmert and Röger, 2025):*

- *$\top$ is an effect (the* empty effect*).*
- *For $v \in F$, both v and $\neg v$ are effects (*atomic effects*).*
- *If e and e' are effects, then $(e \wedge e')$ is an effect (*conjunctive effect*).*

The PDDL components introduced above correspond to the elements of a signature $\mathcal{S}$. The predicates listed in `:predicates` are the predicate symbols $\mathcal{P}$, with arity determined by their number of parameters; PDDL extends the formal definition by also permitting arity 0. The objects declared in `:objects` serve as the constant symbols $\mathcal{C}$. The variable symbols $\mathcal{V}$ correspond to the parameter names that appear across the action schemas. In the STRIPS fragment there are no numeric functions, so $\mathcal{F} = \emptyset$. Preconditions and effects are then formulas over $\mathcal{S}$.

In the PDDL syntax, positive facts in an `:effect` correspond to atomic effects v, and facts wrapped in `not` correspond to atomic effects $\neg v$.

2.2 The STRIPS Fragment

STRIPS (Stanford Research Institute Problem Solver) refers both to an early planning system (Fikes and Nilsson, 1971) and to the restricted fragment of PDDL that mirrors its action schema format (Helmert and Röger, 2025). In the STRIPS fragment, preconditions are conjunctions of positive atoms only. Negation and disjunction are not permitted in preconditions, and quantifiers are excluded. Effects, by contrast, may contain both positive atoms (add effects) and negative literals (delete effects), but every effect applies unconditionally. A state is represented as the set of ground atoms that are currently true. Each grounded action can be described in STRIPS by three sets: its precondition, its add effects, and its delete effects. Grounding, which produces these sets from an action schema, is introduced in the following section. The STRIPS fragment does not support conditional effects or derived predicates; these constructs are discussed in Sections 3 and 4.

2.3 Formal Semantics and Grounding

States. A *state* s is a subset of F, the set of all ground atoms for the given domain and problem instance. The ground atoms contained in s are considered true, and all ground atoms not in s are considered false (Haslum et al., 2019).

Grounding. An action schema as written in PDDL contains variable parameters (e.g. `?x`) and is called a *lifted action*. For an action schema a with parameters $(?x_1, \ldots, ?x_k)$, a substitution $\sigma = \{?x_1 \mapsto o_1, \ldots, ?x_k \mapsto o_k\}$ replaces each variable with a concrete object, producing the ground precondition $\sigma(pre(a))$ and ground effect $\sigma(\mathit{eff}(a))$ (Haslum et al., 2019). Repeating this for every action schema and every valid combination of objects produces all ground actions of the planning task. Grounding the `pick-up` schema with $\sigma = \{\texttt{?x} \mapsto \texttt{A}\}$ yields the following ground action in STRIPS set-based notation, described with the three sets: precondition, add and delete set.

$$
\begin{aligned}
\mathit{pre}(\texttt{pick-up-A}) &= \{\texttt{clear(A)}, \texttt{ontable(A)}, \texttt{handempty}\} \\
\mathit{add}(\texttt{pick-up-A}) &= \{\texttt{holding(A)}\} \\
\mathit{del}(\texttt{pick-up-A}) &= \{\texttt{ontable(A)}, \texttt{clear(A)}, \texttt{handempty}\}
\end{aligned}
$$

Applicability, planning tasks, and plans. An action schema is *applicable* in a state s if every ground atom in its precondition belongs to s. When applied, the successor state is:

$$s' = (s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$$

If an atom appears in both the add and delete set, the add effect takes precedence; this is called *add-after-delete semantics* (Helmert and Röger, 2025).

Example. In the Blocks World instance from Section 2.1:

$$s_0 = \{\mathit{clear}(A),\ \mathit{on}(A,B),\ \mathit{ontable}(B),\ \mathit{clear}(C),\ \mathit{ontable}(C),\ \mathit{handempty}\}$$

The goal is $\{\mathit{on}(A,B),\ \mathit{on}(B,C)\}$ and a valid plan is:

$$\pi = \langle \texttt{unstack}(A,B),\ \texttt{put-down}(A),\ \texttt{pick-up}(B),\ \texttt{stack}(B,C),\ \texttt{pick-up}(A),\ \texttt{stack}(A,B)\rangle$$

As an example, for $\texttt{unstack}(A,B)$: all atoms in

$$\mathit{pre} = \{\mathit{on}(A,B),\ \mathit{clear}(A),\ \mathit{handempty}\}$$

belong to s_0, so the action is applicable, producing:

$$s_1 = \{\mathit{holding}(A),\ \mathit{clear}(B),\ \mathit{ontable}(B),\ \mathit{clear}(C),\ \mathit{ontable}(C)\}$$

The remaining actions are applied analogously until the goal is satisfied or no plan could be found.

3 Conditional Effects

As mentioned before, in the STRIPS fragment, every effect of an action schema applies unconditionally. Conditional effects, on the other hand, introduced as part of the ADL extension to PDDL (Haslum et al., 2019), lift this restriction by allowing a single action schema to express state-dependent outcomes.

3.1 Syntax

PDDL introduces conditional effects through the `when` keyword, which requires the `:conditional-effects` declaration in `:requirements`. A conditional effect has the form `(when` ⟨`condition`⟩ ⟨`effect`⟩`)`: the sub-effect applies only if the condition holds in the current state (Haslum et al., 2019).

3.2 Motivation and Running Example

Consider a robot that enters a room. Entering always moves the robot from outside to inside, but two additional outcomes depend on the room's current state: if fragile items are present, they are broken; if an alarm is active, it triggers. Conditional effects can capture these two scenarios within a single schema:

```
(:action enter-room
  :precondition (outside)
  :effect (and
    (inside)
    (not (outside))
    (when (fragile-items) (items-broken))
    (when (alarm-active) (alarm-triggered))))
```

This schema serves as the running example for the remainder of this section.

3.3 Formal Semantics

The following definitions follow Helmert and Röger (2025).

Effect syntax. The effect constructs from Definition 5 cover the STRIPS fragment. Conditional effects extend this with one further case: if χ is a formula over F and e is an effect, then $(\chi \triangleright e)$ is a *conditional effect*; sub-effect e is triggered only when χ holds in the current state (Helmert and Röger, 2025).

Effect conditions. Given an atomic effect ℓ and an effect e, the *effect condition* $\mathit{effcond}(\ell, e)$ is the formula under which e produces ℓ (Helmert and Röger, 2025):

- $\mathit{effcond}(\ell, \top) = \bot$
- $\mathit{effcond}(\ell, e) = \top$ if $e = \ell$
- $\mathit{effcond}(\ell, e) = \bot$ if $e = \ell' \neq \ell$
- $\mathit{effcond}(\ell, (e \wedge e')) = \mathit{effcond}(\ell, e) \vee \mathit{effcond}(\ell, e')$
- $\mathit{effcond}(\ell, (\chi \triangleright e)) = \chi \wedge \mathit{effcond}(\ell, e)$

The first four cases follow the structure of Definition 5 for the STRIPS effects. The fifth handles the conditional effect: ℓ is produced only when χ holds in the current state.

Applying effects. For a state $s \subseteq F$ and effect e, the resulting state $s[\![e]\!] \subseteq F$ is obtained by adding every $v \in F$ for which $s \models \mathit{effcond}(v, e)$, removing every v for which $s \models \mathit{effcond}(\neg v, e)$, and leaving all other atoms unchanged (Helmert and Röger, 2025).

3.4 Compilation to STRIPS

When a planner does not support conditional effects, they can be compiled away. Two classical approaches exist.

Gazen-Knoblock compilation. Gazen and Knoblock (1997) create one STRIPS action schema for each subset of conditions that simultaneously hold. For n independent conditions, this yields up to 2^n schemas. The two conditions in the `enter-room` example produce four schemas. Since STRIPS preconditions are positive only, the absence of a condition is encoded via auxiliary complement atoms:

```
(:action enter-fragile-alarm
  :precondition (and (outside) (fragile-items)
                     (alarm-active))
  :effect (and (inside) (not (outside))
               (items-broken) (alarm-triggered)))
(:action enter-fragile-only
  :precondition (and (outside) (fragile-items)
                     (not-alarm-active))
  :effect (and (inside) (not (outside))
               (items-broken)))
(:action enter-alarm-only
  :precondition (and (outside) (not-fragile-items)
                     (alarm-active))
  :effect (and (inside) (not (outside))
               (alarm-triggered)))
(:action enter-neither
  :precondition (and (outside) (not-fragile-items)
                     (not-alarm-active))
  :effect (and (inside) (not (outside))))
```

Plan length is preserved, but the number of schemas grows up to 2^n in the number of independent conditions (Haslum et al., 2019).

Nebel compilation. Nebel (2000) avoids the exponential blowup by introducing auxiliary atoms. For each conditional effect $(\chi_i \triangleright e_i)$, a fresh atom $pending_i$ is added. A single main action applies the unconditional effects and sets every $pending_i$; separate auxiliary actions then check each condition and apply e_i if χ_i holds, or clear $pending_i$ otherwise. For the `enter-room` example:

```
(:action enter-main
  :precondition (outside)
  :effect (and (inside) (not (outside))
               (pending-fragile) (pending-alarm)))
(:action resolve-fragile
  :precondition (and (pending-fragile) (fragile-items))
  :effect (and (items-broken) (not (pending-fragile))))
(:action skip-fragile
  :precondition (and (pending-fragile)
                     (not-fragile-items))
  :effect (not (pending-fragile)))
(:action resolve-alarm
  :precondition (and (pending-alarm) (alarm-active))
  :effect (and (alarm-triggered) (not (pending-alarm))))
(:action skip-alarm
  :precondition (and (pending-alarm) (not-alarm-active))
  :effect (not (pending-alarm)))
```

To force the auxiliary actions to be applied before the planner can continue, additional preconditions can be added to all remaining actions that require every `pending` atom to be cleared. This ensures that each conditional effect is checked, and either resolved or skipped, before any further action becomes applicable. Entering a room with both conditions active then requires the plan ⟨`enter-main`, `resolve-fragile`, `resolve-alarm`⟩ rather than a single step. For n conditional effects, the plan grows by up to n additional steps. The number of action schemas is $O(n)$ rather than the $O(2^n)$ of the Gazen-Knoblock compilation. Currently, no compilation can achieve both polynomial size and constant plan length increase simultaneously (Nebel, 2000).

4 Axioms (Derived Predicates)

Conditional effects handle state-dependent outcomes of action schemas, but they do not address a different modelling challenge: properties that follow from the current state without being directly changed by action schemas. Thiébaux et al. (2005) introduced axioms into PDDL to let such properties be defined declaratively, rather than maintained through explicit action effects. Despite their potential, axiom support among planners remains limited (Ivankovic and Haslum, 2015).

4.1 Syntax

PDDL introduces axioms through the `:derived` keyword, requiring the `:derived-predicates` declaration in `:requirements`. An axiom has the form `(:derived` ⟨`predicate`⟩ ⟨`condition`⟩`)`: when the condition holds, the predicate is inferred to be true (Haslum et al., 2019).

A predicate defined this way is called a *derived predicate*; action schemas cannot change it directly. If no axiom derives a predicate, it is assumed false, this is the *negation as failure* principle (Haslum et al., 2019). Axioms may reference their own predicate recursively, which makes them suited for expressing transitive properties such as graph reachability (Haslum et al., 2019):

```
(:derived (reachable ?x)
  (or (= ?x start)
      (exists (?y)
        (and (reachable ?y)
             (connected ?y ?x)))))
```

A node `?x` is reachable if it is the start node, or if some already reachable node `?y` is connected to it. The predicate `reachable` appears in both the head and the body, forming a recursive definition that computes the transitive closure of `connected`.

4.2 Formal Semantics via Stratification

Following Ivankovic and Haslum (2015), predicates are partitioned into two groups. *Primary predicates* are the standard predicates modified by action schemas (e.g. `on`, `connected`). *Secondary predicates* are derived predicates (e.g. `reachable`), whose values are computed from the current state via axioms. Secondary predicates default to F: a derived predicate is false unless some axiom makes it true.

Definition 6 (Axiom)**.** *An* axiom *is a rule $x \leftarrow \varphi$, where x is a secondary predicate and φ is a state condition in negation normal form. When φ holds, the truth of x is inferred (Ivankovic and Haslum, 2015).*

Since axioms can reference other derived predicates, they must be evaluated in the right order. This is ensured by *stratification*.

Definition 7 (Stratification)**.** *A* stratification *maps each secondary predicate x to a level $l(x) \in \{0, \ldots, m\}$ such that for every axiom $x \leftarrow \varphi$: if a secondary predicate y appears positively in φ then $l(y) \leq l(x)$; if $\neg y$ appears in φ then $l(y) < l(x)$ (Ivankovic and Haslum, 2015).*

The first condition ($l(y) \leq l(x)$) allows a derived predicate to reference itself, enabling recursive definitions such as the reachability axiom above. The second condition ($l(y) < l(x)$) requires that any negated derived predicate is fully evaluated before it is used. Whether a set of axioms is stratifiable can be tested in polynomial time (Thiébaux et al., 2005).

The values of derived predicates are computed by a *fixpoint procedure*: all secondary predicates start at F. The axioms are processed level by level; within each level, every applicable axiom is applied repeatedly until no predicate changes, after which the next level is processed (Ivankovic and Haslum, 2015).

4.3 Compilation to STRIPS

Axioms can be compiled away, but unlike conditional effects, where Nebel's compilation achieves polynomial plan-length blow-up, any axiom compilation incurs a blow-up exponential in the maximum fixed-point depth of the derived predicates, in either the size of the domain description or the length of the resulting plans (Thiébaux et al., 2005). The compilation replaces each axiom with explicit actions that propagate derived facts one step at a time, controlled by a flag predicate that sequences derivation against primary actions. Because basic state changes can invalidate previously derived facts, the full compilation also resets derived atoms (folded into each primary action's effect) so that the derivation cycle re-runs after every state change; this reset-and-re-derive structure is what drives the exponential plan-length blow-up. We illustrate here only the propagation core. Since STRIPS preconditions cannot contain negation (see Section 2.2), every negated condition is encoded via a complement atom maintained alongside the original. For the reachability axiom with nodes `s` (start), `a`, and `t`, and edges `(s,a)` and `(a,t)`, the compilation proceeds as follows. The initial state sets `(reachable s)`, `(currently-deriving)`, and the complement atoms `(not-reachable a)` and `(not-reachable t)` for every node not yet reached. A propagation action then extends reachability one step along any connected edge:

```
(:action propagate-reachable
  :parameters (?x ?y)
  :precondition (and (currently-deriving)
                     (reachable ?x)
                     (connected ?x ?y)
                     (not-reachable ?y))
  :effect (and (reachable ?y)
               (not (not-reachable ?y))))

(:action finish-deriving
  :parameters ()
  :precondition (currently-deriving)
  :effect (and (not (currently-deriving))
               (derivation-done)))
```

The flag `currently-deriving` is true while derivation is in progress; primary actions that use `reachable` as a precondition must additionally require the complement flag `(derivation-done)`, ensuring derivation completes before they fire, and must re-activate the full derivation cycle (reset, propagate, finish) after each basic state change. Because the derivation cycle re-runs after every primary action and may need to propagate through every derived atom, the resulting plan can grow exponentially with the size of the domain description (Thiébaux et al., 2005).

The construction shown above is only a simplified illustration of the propagation core; alternative compilation techniques exist that differ in how they sequence derivation, manage the reset cycle, and trade off domain size against plan length. More information, including the proof of the exponential lower bound, can be found in the paper by Thiébaux et al. (2005).

4.4 Impact on Planning

The practical benefit of axioms is that derived predicates remove unnecessary choices from the search space. Ivankovic and Haslum (2015) demonstrate this on Sokoban, a benchmark domain based on a puzzle game in which a player pushes boxes to target positions, where defining reachability as a derived predicate eliminates irrelevant movement decisions and reduces blind search node expansions by a factor of 17 compared to the STRIPS formulation. For controller verification problems, the STRIPS encoding could not be solved within 4 hours and 60 GB of memory, while the axiom formulation solved all instances in under a minute.

5 Conclusion

This report covered three fragments of PDDL: the STRIPS fragment, conditional effects, and axioms (derived predicates). The central question for the two non-STRIPS fragments was the same: each yields more compact domain models, and each can be compiled back to STRIPS, but at what cost?

The two fragments give different answers. For conditional effects, two classical techniques offer a choice of which dimension pays the cost: Gazen and Knoblock (1997) preserve plan length but can produce exponentially many action schemas, while Nebel (2000) keeps the number of schemas polynomial at the cost of polynomially longer plans. Nebel (2000) proves that no compilation can preserve plan size linearly, but at least one of the two dimensions can always be kept polynomial.

Axioms do not admit such a trade-off. Thiébaux et al. (2005) prove that any compilation back to STRIPS incurs an exponential blow-up in either the domain description or the plan length: at least one of the two dimensions is always exponential.

This asymmetry shows up in practice. Thiébaux et al. (2005) report that a planner with native axiom support outperforms the same planner run on the compiled, axiom-free version, with the compiled plans being an order of magnitude longer. Ivankovic and Haslum (2015) report that on Sokoban, blind search on the STRIPS formulation expands on average 17 times more nodes than on the axiom formulation. Both experiments support the view that handling axioms natively can be more efficient than paying the exponential cost in plan length, though the practical gains depend on the domain and the heuristic used.

Use of Digital Tools

The following digital resources were used in the preparation of this report. They were used solely as aids. The selection and analysis of the literature, the technical content and the final presentation of the arguments are my own work.

Tool	Provider	Purpose of use
Consensus (via ChatGPT)	OpenAI / Consensus	Used to identify potentially relevant papers, obtain high-level summaries, and support brainstorming.
DeepL	DeepL SE	Used for translation support and for improving sentence phrasing and overall language clarity.

References

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208.

Gazen, B. C. and Knoblock, C. A. (1997). Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *Proceedings of the Fourth European Conference on Planning (ECP)*, pages 221–233.

Haslum, P., Lipovetzky, N., Magazzeni, D., and Muise, C. (2019). *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Helmert, M. and Röger, G. (2020). Discrete mathematics in computer science—lecture notes. University of Basel. `https://ai.dmi.unibas.ch/_files/teaching/hs20/dmics/slides/dmics-e05-handout.pdf`. Accessed: 12 April 2026.

Helmert, M. and Röger, G. (2025). Planning and optimization—lecture notes. University of Basel. `https://ai.dmi.unibas.ch/_files/teaching/hs24/po/slides/po-b03.pdf`. Accessed: 12 April 2026.

Ivankovic, F. and Haslum, P. (2015). Optimal planning with axioms. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1580–1586.

McDermott, D. (2000). The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55.

Nebel, B. (2000). On the compilability and expressive power of propositional planning formalisms. *Journal of AI Research*, 12:271–315.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of PDDL axioms. *Artificial Intelligence*, 168(1–2):38–69.