CHAPTER 3

More Expressive Classical Planning

Thus far, we have only considered a simple form of PDDL modelling. While we can represent complex scenarios and behaviour, as evidenced by Section 2.4, the language has been extended in key ways which we discuss here in this chapter. The modifications empower the modeller to succinctly represent complex but common phenomenon in the planning setting, but do not necessarily change the theoretical expressiveness of the language. We discuss these details on expressiveness further in Section 3.5. Sections 3.1–3.2 detail additions to the language that allow for more compact and logical representations of action preconditions and effects (including the ramifications some actions may have), and Sections 3.3–3.4 cover temporal specifications of both soft and hard constraints.

3.1 CONDITIONAL AND QUANTIFIED CONDITIONS AND EFFECTS

Arguably the most pervasive extension to the core PDDL language is the Action Description Language (ADL) [Pednault, 1989]. The core features this introduces include conditional effects (updating the state of the world only in certain situations), universally quantified effects (allowing modellers to refer to a set of objects collectively), disjunctive conditions (allowing a generalised view of state conditions), and both universally and existentially quantified preconditions (allowing for complex action pre-requisites). We will demonstrate each of these in turn through the common running example of an elevator controller.

3.1.1 BASIC ELEVATOR MODEL

In the basic elevator example, we model one or more lifts that must transport people between floors. We start with a basic encoding that uses the features already introduced in Chapter 2, and then progressively add ADL extensions to demonstrate the modelling possibilities. Consider the following domain:

```
(define (domain elevators)
    (:requirements :typing)
    (:types elevator passenger num - object)
    (:predicates
            (passenger-at ?person - passenger ?floor - num)
            (boarded ?person - passenger ?lift - elevator)
            (lift-at ?lift - elevator ?floor - num)
            (next ?n1 - num ?n2 - num)
    )

    (:action move-up
        :parameters (?lift - elevator ?cur ?nxt - num)
        :precondition (and (lift-at ?lift ?cur) (next ?cur ?nxt))
        :effect (and (not (lift-at ?lift ?cur)) (lift-at ?lift ?nxt))
    )

    (:action move-down
        :parameters (?lift - elevator ?cur ?nxt - num)
        :precondition (and (lift-at ?lift ?cur) (next ?nxt ?cur))
        :effect (and (not (lift-at ?lift ?cur)) (lift-at ?lift ?nxt))
    )

    (:action board
        :parameters (?person - passenger ?floor - num ?lift - elevator)
        :precondition (and (lift-at ?lift ?floor)
                           (passenger-at ?person ?floor))
        :effect (and (not (passenger-at ?person ?floor))
                     (boarded ?person ?lift))
    )

    (:action leave
        :parameters (?person - passenger ?floor - num ?lift - elevator)
        :precondition (and (lift-at ?lift ?floor)
                           (boarded ?person ?lift))
        :effect (and (passenger-at ?person ?floor)
                     (not (boarded ?person ?lift)))
    )
)
```

PDDL Example 9: Domain definition for a simple elevator problem.

In this simple domain model, we have types that correspond to elevators, passengers, and floors, and predicates that indicate if the passenger is at a particular floor (`passenger-at`), the passenger has boarded a lift (`boarded`), the location of a lift (`lift-at`), and finally one to represent the order of floors (`next`).

Four actions make up the domain: `move-up`, `move-down`, `board`, and `leave`. While the first two move the elevator in expected ways, the second two moves a passenger to or from the elevator (respectively).

Finally, the following example shows the initial configuration for the problem, with five floors, three passengers (on floors 2, 2, and 4), and two lifts (on floors 1 and 5). The goal is to bring all of the passengers to floor 1.

```
(define (problem elevators-problem)
    (:domain elevators)

    (:objects
        n1 n2 n3 n4 n5 - num
        p1 p2 p3 - passenger
        e1 e2 - elevator
    )

    (:init
        (next n1 n2) (next n2 n3) (next n3 n4) (next n4 n5)
        (lift-at e1 n1) (lift-at e2 n5)
        (passenger-at p1 n2) (passenger-at p2 n2) (passenger-at p3 n4)
    )

    (:goal (and
        (passenger-at p1 n1)
        (passenger-at p2 n1)
        (passenger-at p3 n1)
    ))
)
```

PDDL Example 10: Problem definition for a simple elevator problem.

Tip: Constants in the Domain PDDL

In many PDDL benchmarks, it is common to see objects that would otherwise be specified in a problem file as constants in the domain file. While this goes against the philosophy of separating the domain and problem aspects, it is sometimes required in order to avoid using more complex PDDL expression in the domain file. When this occurs, you will typically see a unique domain PDDL for each and every problem (instead of one domain for all of the problems).

A possible plan for the example elevator problem is as follows:

```
(move-up e1 n1 n2)
(board p1 n2 e1)
(move-down e1 n2 n1)
(leave p1 n1 e1)
(move-up e1 n1 n2)
(board p2 n2 e1)
(move-down e1 n2 n1)
(leave p2 n1 e1)
(move-up e1 n1 n2)
(move-up e1 n2 n3)
(move-up e1 n3 n4)
(board p3 n4 e1)
(move-down e1 n4 n3)
(move-down e1 n3 n2)
(move-down e1 n2 n1)
(leave p3 n1 e1)
```

3.1.2 CONDITIONAL EFFECTS

The first extension we will introduce is *conditional effects*. This allows us to model an effect on the world *only under certain conditions*. The format of a conditional effect is

```
(when <head> <body>)
```

The interpretation of this form is "when *<head>* holds in the current state, *<body>* should hold in the state reached by applying this action". For our running example, we will first introduce a set of constants to refer to the passengers as part of the domain:

```
(:constants p1 p2 p3 - passenger)
```

Now that we have the ability to refer to the individual passengers in the domain file itself, we can replace the `board` and `leave` actions with `load` and `unload`: while the former pair of actions moved individuals, the latter pair can load and unload the entire elevator. The two new actions are as follows:

```
(:action load
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor))
    :effect (and (when (passenger-at p1 ?floor)
                       (and (not (passenger-at p1 ?floor))
                            (boarded p1 ?lift)))
                 (when (passenger-at p2 ?floor)
                       (and (not (passenger-at p2 ?floor))
                            (boarded p2 ?lift)))
                 (when (passenger-at p3 ?floor)
                       (and (not (passenger-at p3 ?floor))
                            (boarded p3 ?lift))))
)

(:action unload
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor))
    :effect (and (when (boarded p1 ?lift)
                       (and (passenger-at p1 ?floor)
                            (not (boarded p1 ?lift))))
                 (when (boarded p2 ?lift)
                       (and (passenger-at p2 ?floor)
                            (not (boarded p2 ?lift))))
                 (when (boarded p3 ?lift)
                       (and (passenger-at p3 ?floor)
                            (not (boarded p3 ?lift)))))
)
```

Note that while this will force *every* passenger to either embark or disembark the lift (thus creating erroneous plans in some situations; an issue we will rectify in subsequent examples), it serves as an example of how multiple actions can be combined into a single one that relies on the current state of the world for choosing the effects that result. An example plan for the newly formatted domain is as follows:

```
(move-up e1 n1 n2)
(load n2 e1)
```

```
  (move-up e1 n2 n3)
  (move-up e1 n3 n4)
  (load n4 e1)
  (move-down e1 n4 n3)
  (move-down e1 n3 n2)
  (move-down e1 n2 n1)
  (unload n1 e1)
```

Notice how the plan first loads every passenger into the lift `e1` prior to unloading them in a single final step on the first floor.

3.1.3 UNIVERSALLY QUANTIFIED EFFECTS

While we have a more concise model with the conditional effects in the previous example, it comes at the cost of explicitly writing the passengers as constants in the domain file. To remedy this, and to drastically simplify the domain model itself, we can use *universal quantification* in the effects of the `load` and `unload` actions. A universally quantified effect has the following form:

```
(forall <parameters> <effect>)
```

The *<parameters>* section is precisely the same as an action's `:parameters` clause: a list of (possibly typed) variables. The *<effect>* is the same as any action effect we have already seen. The following example shows an updated version of the `load` and `unload` actions that takes advantage of this. Note that the domain constants for passengers are no longer required; they are simply represented as objects in the problem definition again:

```
(:action load
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor))
    :effect (and (forall (?person - passenger)
                    (when (passenger-at ?person ?floor)
                        (and (not (passenger-at ?person ?floor))
                            (boarded ?person ?lift)))))
)

(:action unload
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor))
    :effect (and (forall (?person - passenger)
                    (when (boarded ?person ?lift)
                        (and (passenger-at ?person ?floor)
                            (not (boarded ?person ?lift))))))
)
```

Note further that the universally quantified effect has a conditional effect nested within it. In general, conditional effects can only contain a conjunction of atomic effects, while quantified effects may contain both atomic effects and conditional effects. A more general representation that reverses the nesting (i.e., having a quantified effect inside of a conditional effect) can always be re-written without changing the representation size or interpretation.

3.1.4 DISJUNCTIVE AND EXISTENTIALLY QUANTIFIED PRECONDITIONS

Even though modern planners are *generally* good at avoiding redundant or useless actions, the example so far allows for a lift to stop at any floor regardless of whether there are any passengers there or on the lift itself. To disallow this, we will restrict the lift from officially stopping (i.e., using the `load` or `unload` actions) on a floor in this situation. As a condition, we would like to say "stop only if there is someone on the lift *or* someone waiting to enter it".

The key part of this statement is the disjunction in both the two parts of the statement, and in the notion of "someone". To handle these, we introduce *disjunctive preconditions* and *existentially quantified preconditions*, respectively. The form of a disjunctive precondition follows the same format as an `and` clause:

```
(or <condition1> <condition2> ···)
```

The interpretation is that only one of *<condition1>*, *<condition2>*, ··· must hold in order for this disjunctive condition to be satisfied. Similar to the universal quantification in syntax (and the disjunctive condition in spirit), this is the format for an existential condition:

```
(exists <parameters> <condition>)
```

The *<parameters>* clause also follows the syntax of the action's parameter list, and *<condition>* is simply any well-formed formula. The interpretation is that *at least* one of the valid object assignments for the *<parameters>* must cause *<condition>* to hold.

One further improvement that we will make with this iteration is to combine the `load` and `unload` actions into a single `stop` action (ultimately, a lift only opens its doors once while on a floor!). The updated action is as follows:

```
(:action stop
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor)
                       (exists (?person - passenger)
                          (or (passenger-at ?person ?floor)
                              (boarded ?person ?lift))))
    :effect (forall (?person - passenger)
                 (and (when (passenger-at ?person ?floor)
                           (and (not (passenger-at ?person ?floor))
```

Tip: Remodelling

When planners cannot handle a specific feature of the PDDL language, there may be ways to remodel the problem to avoid specific features. If an action's precondition is made up of a single or clause with k terms, then a way to remodel the problem is to create k copies of the action and use one term for each as a regular precondition. For existential preconditions, a similar approach can be followed after the grounding has occurred.

While in principle this does allow for a wider range of planners to work with the same problem, it can come at the expense of reduced efficiency given the increased number of actions required.

```
                    (boarded ?person ?lift)))
           (when (boarded ?person ?lift)
              (and (passenger-at ?person ?floor)
                    (not (boarded ?person ?lift))))))
)
```

Again, notice that we are able to nest the different condition types of `exists`, `or`, etc., as long as it makes sense and follows the above patterns of combining the various operators.

3.1.5 UNIVERSALLY QUANTIFIED PRECONDITIONS AND GOALS

The final PDDL feature we will introduce in this section is universally quantified preconditions or goals. We focus on the former in our running example, as the latter follows the exact same syntax and principle.

The syntax for a universally quantified precondition is the same as a universally quantified effect, with the exception that it appears as a term in an action's precondition (similar nesting to the other conditions introduced thus far is allowed). The interpretation is that an action can only be applied if every combination of objects that match the parameters in a `forall` condition lead to the nested condition being satisfied.

As a grounded example, we will extend the elevator domain one more time so that the final action will be for the elevator to enter a "maintenance-mode", and can only do so when all of the passengers have arrived at their floor and the elevator is empty. The first change to our domain is to introduce two new predicates:

```
(requested ?person - passenger ?floor - num)
(in-maintenance)
```

The first represents the requested floor for a passenger (i.e., their destination), and the second will hold only when the elevator is in maintenance mode. The initial state for the problem is augmented to include the passenger requests:

```
  (requested p1 n1)
  (requested p2 n1)
  (requested p3 n1)
```

We further change the problem's goal for the elevator to be in maintenance mode:

```
(:goal (in-maintenance))
```

Finally, we add the action that ensures the elevator enters maintenance mode only when all of the requests have been processed:

```
(:action enter-maintenance-mode
    :parameters (?lift - elevator)
    :precondition (forall (?person - passenger)
                    (and (not (boarded ?person ?lift))
                         (forall (?floor - num)
                           (imply (requested ?person ?floor)
                                  (passenger-at ?person ?floor)))))
    :effect (in-maintenance)
)
```

The example also shows the final logical operator that can be used for constructing conditions in the ADL variant of PDDL: `imply`. The condition `(imply` ϕ_1 ϕ_2`)` holds when either (1) ϕ_2 holds, or (2) ϕ_1 does not hold. Note that `imply` should not be confused with `when`: the former is a logical operator that can only be used in formulas, such as in action preconditions, goals, or, indeed, the condition part of a conditional effect; the latter can only be used in the effects part of an action, where it defines a conditional effect. The two are *not* interchangeable.

In this section we have seen a range of syntactic additions to the base PDDL language. Through the careful combination of quantified conditions and effects, complex behaviour can be modelled with these additional operators far more succinctly.

3.2 AXIOMS

Another important extension of PDDL is state dependent *axioms*, which can be used to derive the truth value of some predicates. So far only actions could affect the truth value of a predicate, but now we are going to distinguish between the *standard predicates* and *derived predicates* whose truth value in a state is inferred through axioms only. Derived predicates can then be used as preconditions, goals, or in the context of action effects only, as actions cannot change their truth value directly.

Axioms can be used to elegantly model complex concepts such as graph reachability, network flow, transitivity closure, and relational properties among objects. As a result, axioms typically simplify action schemas, making the preconditions more readable and avoiding effects that are just logical consequences of the standard predicates. Axioms do not increase the language expressiveness in terms of what can be modeled, but decrease the domain description size and plan length, as compiling away axioms can result in an exponential growth in the number of actions [Thiébaux et al., 2005].

The syntax of an axiom is

```
(:derived <predicate> <condition>)
```

The interpretation of this form is "when *<condition>* holds in the current state, *<predicate>* should hold true in the state". We assume negation as failure to know when the derived predicate is false; that is, when *<predicate>* cannot be derived through any axiom, then *<predicate>* is false.

Axioms can nest rules recursively by using the same *<predicate>* in the *<condition>* formula. For our running example, we will define a rule to derive the static relationship describing when one floor is above/below another. That is, we make the transitive closure of the `(next ?n1 - num ?n2 - num)` predicate. We need to first add a predicate to our domain definition:

```
  (above ?blw - num ?abv - num)
```

The first parameter of the predicate represents the floor number which is below the floor in the second parameter. The axiom rule to derive the truth of `above` is added to the domain in the same section where actions are defined:

```
(:derived (above ?blw ?abv - num)
    (or (next ?blw ?abv)
        (exists (?z - num)
            (and (next ?blw ?z)
                 (above ?z ?abv))))
)
```

This axiom shows that conditions can be composed using the recently introduced ADL features. The derived predicate `above` appears in the head of the rule, and a disjunctive condition with an existential quantifier recursively defines the relationship between pairs of floors. The axiom implies floor `?abv` is above if a predicate `next` explicitly states this relationship, or there is another floor `?z` which is right next to `?blw` and for which the `above` derived predicate also holds with respect to `?abv`.

As all conditions mentioned in our example are either static predicates or the same derived predicate, then derived predicate `above` is also static, and defines an invariant which is true in all reachable states from the initial state. This is not the case of all derived predicates as they can be defined in terms of predicates whose truth value is changed by actions. Nevertheless, there are certain restrictions on how axioms can be specified, because we want to make sure the truth value of derived predicates are algorithmically easy to derive, do not incur in significant extra computation, and are uniquely defined. The first rule is that no action can change the truth value of a derived predicate. The second rule states that no derived predicate appears *negated* in the conditions. This rule makes sure that no negative interactions appear from the application of axioms, and a unique fixed point is reachable. Both rules are respected in the axiom we introduced for the elevators domain.

We can now use the derived predicate above to allow elevator movement not only to consecutive floors, but to any floor above or below. We further show that with existential quantifiers we can restrict an elevator to only move to a floor if there is a passenger waiting there, or there is a passenger in the elevator that wants to disembark.

```
(:action move-up
    :parameters (?lift - elevator ?cur ?nxt - num)
    :precondition (and (lift-at ?lift ?cur) (above ?cur ?nxt)
                       (exists (?person - passenger)
                               (or (passenger-at ?person ?nxt)
                                   (and (requested ?person ?nxt)
                                        (boarded ?person ?lift)))))
    :effect (and (not (lift-at ?lift ?cur))
                 (lift-at ?lift ?nxt))
)
```

PDDL Example 11: Revised `move-up` action for the elevators domain using axioms.

Action `move-down` has the same action schema but the derived predicate parameters are reversed: `(above ?nxt ?cur)`. Action `load` is modeled as in our previous example, boarding all passengers in current floor using a universal quantifier. Action `unload` can be extended further to make sure only passengers who requested the floor disembark.

```
(:action unload
        :parameters (?floor - num ?lift - elevator)
        :precondition (and (lift-at ?lift ?floor))
        :effect (and (forall (?person - passenger)
                          (when (and (boarded ?person ?lift)
                                     (requested ?person ?floor))
                                (and (passenger-at ?person ?floor)
                                     (not (boarded ?person ?lift))
```

```
                                   (not (requested ?person ?floor)))))
)
```

PDDL Example 12: Domain definition for an elevator: extending the unload action.

We finally change the goal to compute a plan where one elevator ends up in the first floor, the other in the top floor, and all passenger requests are served.

```
(:goal (and
        (lift-at e1 n1) (lift-at e2 n5)
        (forall (?person - passenger ?floor - num)
                (not (requested ?person ?floor)))
        ))
```

Axioms represent a powerful tool for domain designers to compactly specify key properties of the domain. The planner support for axioms is limited compared to the vanilla classical planning syntax, however for domains with properties such as actions conditioned on the transitive closure of key predicates, axiom capable planners can be far superior compared to the alternative of rewriting the domain [Miura and Fukunaga, 2017, Thiébaux et al., 2005].

3.3 PREFERENCES AND PLAN QUALITY

So far, we have defined planning as the problem of finding a path that maps the initial state into a goal state that achieves a set of goals. We have seen that the goal is typically a conjunction of predicates in STRIPS, but can be expressed using universal and existential quantifiers using the subset of ADL extensions to the language. In this subsection we are going to introduce the notion of soft goals to define some properties that we would like to achieve, but are not compulsory. In many real-world problems not all goals are achievable and hence instead of not computing a plan at all, we rather try to achieve as many goals as possible. This situation arises when goals are mutually exclusive, or they make the problem computationally harder. For example, the well known Traveling Salesman Problem becomes computationally easy if we make the requirement to visit a city at most once as a preference instead of a requirement. Furthermore, not all preferences need to have the same weight, so we will introduce the syntax to specify the quality of a plan in terms of the preferences or soft goals that have been achieved. If preferences are used, we have to include the requirement flag `:preferences` in the domain definition.

The syntax of a preference is

```
(preference [<name>] <body>)
```

The *<name>* is optional. The interpretation of preferences depends on their context as they can be defined as action preconditions or as goals, but not as conditions of conditional effects. If they are defined as a precondition, their interpretation is of the form "when the action is applied, if formula *<body>* does not hold, it will incur in a penalty in the overall plan quality". Unless specified, the penalty for each occurrence of an unsatisfied preference is a cost of 1 (we show how to specify non-unit costs below as part of the goal metric). If a preference is specified within the `(:goal ...)` scope, then the truth value of *<body>* the violation penalty is incurred if the formula does not hold in the goal state. Further, nesting of preferences is not allowed and preferences can only appear inside a conjunction or a universal quantifier. The expression in *<body>* can be formulated using conjunctions, disjunctions, universal, and existential quantifiers. For example, the preference

```
(preference allRequestsServed (forall (?person - passenger ?floor - num)
                                       (not (requested ?person ?floor))))
```

is satisfied if all passengers requests are served. If one request is not served it will incur in the same penalty as if no requests are served at all. If we reverse the nesting and write the preference inside the universal quantifier,

```
(forall (?person - passenger ?floor - num)
        (preference requestServed (not (requested ?person ?floor))))
```

then each unserved passenger will incur in a penalty. This statement will create a preference with the name `requestServed` for every floor a passenger can request. Therefore, note that the interpretation changes depending on the nesting of the universal quantifier. Remember that universal quantifiers are read as conjunction of over all legal instantiations of the quantification.

Heterogeneous penalties can be specified using the syntax

```
  (is-violated <name>)
```

where *<name>* is the identifier used in one or more preferences. The interpretation of `is-violated` is as an integer value representing the number of preferences not satisfied with the identifier *<name>*. It can be understood as a counter. For example, `(is-violated allRequestsServed)` can be 0 or 1, while `(is-violated requestServed)` can have any value from 0 to n, where n is the maximum number of possible requests. These counters can be combined through standard arithmetic operators to define the quality of a plan. For example, this is how we would specify a linear weighted combination of the preferences introduced so far:

```
(:metric minimize (+  (* 2 (is-violated requestServed))
                      (* 100 (is-violated AllRequestsServed))))
```

A plan that reaches a goal state satisfying all preferences would have a penalty of 0 according to the expression above. If just one request is not served the penalty is 102, two unserved requests would have a penalty of 104, and so on.

As a full example, let us extend the basic elevator domain from before (Example 9) to express the preference of not filling the elevators with more than five passengers. For doing so we need to create a new predicate to keep track of the number of passengers per lift `(current-load ?lift - elevator ?n1 - num)`. The initial state specifies that the lifts are initially empty. In our 2-elevators example, the predicates `(current-load e1 n0) (current-load e2 n0)` are included in the `(:init ...)` section. The `board` action increases `current-load` every time a passenger boards a lift, and `leave` decreases the counter when a passenger leaves. Whenever someone boards a lift, the preconditions capture the preference of having zero to five passengers at most.

```
(:action board
    :parameters (?person - passenger ?floor - num ?lift - elevator
                 ?cur_load - num ?nxt_load - num )
    :precondition (and (lift-at ?lift ?floor)
                       (passenger-at ?person ?floor)
                       (next ?cur_load ?nxt_load)
                       (current-load ?lift ?cur_load)
                       (preference maxLoad (or
                          (current-load ?lift n0)
                          (current-load ?lift n1)
                          (...)
                          (current-load ?lift n4))))
    :effect (and (not (passenger-at ?person ?floor))
                 (not (current-load ?lift ?cur_load))
                 (boarded ?person ?lift)
                 (current-load ?lift ?nxt_load)))

(:action leave
    :parameters (?person - passenger ?floor - num ?lift - elevator
                 ?cur_load - num ?nxt_load - num)
    :precondition (and (lift-at ?lift ?floor)
                       (boarded ?person ?lift)
                       (next ?nxt_load ?cur_load)
                       (current-load ?lift ?cur_load))
    :effect (and (passenger-at ?person ?floor)
                 (current-load ?lift ?nxt_load)
                 (not (boarded ?person ?lift))
                 (not (current-load ?lift ?cur_load))))
```

We can now use the same goal definition as before, in Example 10, and the best plan would not load an elevator with more than five passengers. To realize the effect of the preference we need to add more passengers to the problem as the original definition had only three passengers.

3.4 STATE TRAJECTORY CONSTRAINTS

While we have seen how to encode preferences over goals and action preconditions, also known as soft goals/preconditions (in contrast to hard goals/preconditions that must be satisfied by any valid plan), we now introduce how to specify both hard and soft constraints on entire state trajectories. State trajectory constraints typically express conditions that are not state dependent, but rather depend on the path taken to reach a state [Gerevini et al., 2009]. As such, hard constraints encode properties valid plans must satisfy, while soft constraints encode preferences used to define plan quality.

Constraints are expressed through temporal modal operators, akin to operators in linear temporal logic (LTL) [Pnueli, 1977], over first-order formulae using domain predicates. Constraint formulae cannot express occurrences of actions as they refer to state properties only, which appear as predicates[1]. The five basic modal operators are:

1. `(always <condition>)`
2. `(sometime <condition>)`
3. `(at-most-once <condition>)`
4. `(at-end <condition>)`
5. `(within <num> <condition>)`

While the interpretation of LTL formulae is different over finite and infinite plans[2], the interpretation of PDDL modal operators is specified only over finite plans. The first modal operator `always` enforces a condition to be true over all states in the plan, `sometime` enforces that the formula is satisfied by one or more states along the plan, very similar to the notion of "at-least-once". The third operator states that the formulae *becomes* true at most once along the plan. That is, a plan can satisfy this condition by never making the formula true, or by making sure that once it becomes true, it can be falsified but not made true again. The fourth modal operator is equivalent to the standard goal conditions we have seen so far, which states that a formulae must hold in the final state of a plan. If a goal formula has no modal operator, we assume that it takes the `at-end` modality by default. The last modal operator states that the formula is satisfied before *<num>* steps of the plan have been executed from the initial state.

Arbitrary nesting of temporal modalities is not allowed, however, three custom nestings have been introduced to capture commonly used constraints:

6. `(sometime-after <condition1> <condition2>)`

[1]This restriction is not limiting, as new fluents that represent the occurrence of an action can be introduced, and added to the state only when an action occurs (i.e., as an add effect).

[2]An infinite plan can be created if a loop is added to the plan and all actions in the loop remain always applicable [Patrizi et al., 2011].

7. (sometime-before *<condition1>* *<condition2>*)

8. (always-within *<num>* *<condition1>* *<condition2>*)

The first states that if first condition is true in some state, then future trajectories have to satisfy (sometime *<condition2>*). The second states that before *<condition1>* becomes true (sometime *<condition2>*) must be satisfied. The third states that anytime *<condition1>* is satisfied, (within *<num>* *<condition2>*) has to hold, taking as reference the state that satisfied the first condition instead of the initial state to count the number of steps.

The final group of state trajectories are those that extend always modality over intervals:

9. (hold-after *<num>* *<condition>*)

10. (hold-during *<num1>* *<num2>* *<condition>*)

The first states that (always *<condition>*) must be satisfied from the *<num>*th step in the state trajectory onwards. The second states that (always *<condition>*) must hold between the *<num1>* and *<num2>* steps, including *<num1>*.

In order to make a state trajectory constraint soft, we just have to use the same preference syntax introduced previously. The :constraints flag has to be added to the domain file and all constraints have to be declared in the (:constraints ...) scope, which can be written right after the (:goal ...) declaration. For example, the same preference we introduced into the board action to prefer plans that do not load more than five passengers per lift, can now be expressed with the following constraint:

```
(:constraints
 (forall (?lift - elevator)
         (preference maxLoad (always (not current-load ?lift n6)))))
```

Standard goals which have to hold true in the final state of a valid plan can be included in the (:goal ...) specification or added with the (at-end ...) constraint. The goal from the previous example where all passengers had to be moved to floor 1 can be added to the (:constraints ...) section as shown below:

```
(:constraints
 (and
  (forall (?lift - elevator)
          (preference maxLoad (always (not current-load ?lift n6))))
  (at-end (passenger-at p1 n1))
  (at-end (passenger-at p1 n2))
  (at-end (passenger-at p1 n3))))
```

We can express rich behavior once we can make use of these modal operators. For instance, we can specify that each passenger can be boarded at most once, to avoid multi-elevator trips:

```
(:constraints
 (forall (?person - passenger ?lift - elevator)
         (at-most-once (boarded ?person ?lift))))
```

We can specify that an elevator should not visit a floor with a boarded passenger more than once, to avoid unnecessary time in the elevator:

```
(:constraints
 (forall (?lift - elevator ?floor - num ?person - passenger)
         (at-most-once (and  (boarded ?person ?lift)
                             (lift-at ?lift ?floor)))))
```

We can even specify that at most one elevator is at the same level at any point in the plan:

```
(:constraints
 (forall (?lift1 ?lift2 - elevator ?floor - num)
         (always (imply (and (lift-at ?lift1 ?floor)
                             (lift-at ?lift2 ?floor))
                        (= ?lift1 ?lift2)))))
```

The range of temporally extended goals allowed for this fragment of PDDL is limited to only the ten we have listed in this section. Nonetheless, combined with the expressive power of the conditions themselves for a single state, the technique represents a powerful tool in the arsenal of a domain modeler to express non-Markovian preferences and constraints.

3.5 EXPRESSIVENESS AND COMPLEXITY

In Section 2.5, we discussed the complexity of the STRIPS subset of PDDL for the computational problems of plan existence, generation, optimal generation, and validation. The first three problems are PSPACE-hard in the ground version and EXPSPACE-hard in parameterized STRIPS, while the problem of validation is linear in the size of the plan. These results hold for the extensions described in this chapter, namely conditional effects, universal and existential quantification, derived predicates, preferences, and finite state trajectory constraints, as none of these features increase the computational complexity of the problems being solved.

However, Nebel [2000] proposed a different way to study the expressiveness of the extensions of PDDL that we have introduced in this chapter, via *compilations* of each extension of the language into STRIPS. A compilation is a problem reformulation that preserves properties such as plan existence, and optimal plan cost. Nebel suggested that if the size of the reformulated problem is at most polynomial in the size of the original problem and the length of valid plans does not increase by more than a constant factor, then the extension does not add expressive power to the language, but rather is "syntactic sugar" allowing modellers to more elegantly capture certain properties of the problem, e.g., allowing negative preconditions and compiling them away with newly introduced fluents to re-achieve the STRIPS property. If every possible compilation of a feature is exponential in problem size, or induces plans lengths in the compilation larger than a constant increase, then the feature adds expressiveness. In the remainder of this section, we will briefly describe compilations of each of the language features introduced in this chapter into the STRIPS subset, to clarify whether they add to the expressiveness of PDDL.

Universal quantifiers can be replaced by a conjunction over the objects of the quantified parameter, and existential quantifiers can be replaced by a disjunction. For example:

```
(forall (?lift - elevator ?floor - num) (lift-at ?lift ?floor))
;Can be compiled into:
(and  (lift-at e1 n1) (lift-at e1 n2) ... (lift-at e1 nx)
      (...)
      (lift-at ex n1) (lift-at ex n2) ... (lift-at ex nx))

(exists (?lift - elevator ?floor - num) (lift-at ?lift ?floor))
;Can be compiled into:
(or   (lift-at e1 n1) (lift-at e1 n2) ... (lift-at e1 nx)
      (...)
      (lift-at ex n1) (lift-at ex n2) ... (lift-at ex nx))
```

These compilations are linear and do not increase the length of a plan. Thus, quantifiers can be considered "syntactic sugar".

A disjunctive action precondition can be compiled away by creating a copy of the action for each disjunct, whose precondition is that disjunct. In any state where the original, disjunctive, precondition holds, at least one of the disjuncts is true and thus at least one of the copies of the action applicable. This, however, only works if the precondition formula is in disjunctive normal form (DNF), meaning it is a disjunction of conjunctions of literals. Any Boolean formula can be rewritten as an equivalent formula in DNF, but doing so may increase the size of the formula exponentially. In fact, preconditions that are general logical formulae cannot be compiled away without either a potential exponential increase in problem size or a super-linear increase in plan length [Nebel, 2000].

Conditional effects can also be compiled away. Two compilation techniques are known: one increases the size of the problem exponentially, but keeps the length of valid plans constant; the other increases the size of the problem only polynomially, but the length of plans more than linearly. It has been shown that there is no compilation that achieves both [Nebel, 2000]. Therefore, conditional effects augment the expressiveness of PDDL in terms of the conciseness of the problem representation. Here, we will only illustrate the first compilation, which is simpler. An action with $C(a)$ conditional effects is replaced by $2^{|C(a)|}$ STRIPS actions, one for every subset of conditional effects, each of whose precondition ensures that it is applicable only in states where exactly that subset of conditional effects would occur. For example, the action

```
(:action unload
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor))
    :effect (and (when (boarded p1 ?lift)
                       (and (passenger-at p1 ?floor)
                            (not (boarded p1 ?lift))))
                 (when (boarded p2 ?lift)
                       (and (passenger-at p2 ?floor)
                            (not (boarded p2 ?lift))))))
```

can be compiled into the following actions:

```
(:action unload-p1
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor)
                       (boarded p1 ?lift)
                       (not (boarded p2 ?lift)))
    :effect (and (passenger-at p1 ?floor)
                 (not (boarded p1 ?lift))))

(:action unload-p2
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor)
                       (not (boarded p1 ?lift))
                       (boarded p2 ?lift))
    :effect (and (passenger-at p2 ?floor)
                 (not (boarded p2 ?lift))))

(:action unload-p1-p2
    :parameters (?floor - num ?lift - elevator)
    :precondition (and (lift-at ?lift ?floor)
                       (boarded p1 ?lift)
                       (boarded p2 ?lift))
    :effect (and (passenger-at p1 ?floor)
                 (passenger-at p2 ?floor)
                 (not (boarded p1 ?lift))
                 (not (boarded p2 ?lift))))
```

Note that the action corresponding to the case when none of the effect conditions is true has been omitted, since it has no effect. In general, this may not be the case since an action may have a mix of conditional and unconditional effects.

The compilation above assumes that the action's effects is a conjunction of "simple" conditional and unconditional effects. However, nested conditional effects can be rewritten as a flat conjunction of conditional effects, and hence are compilable into STRIPS in the same ways. For example:

```
(when <condition1> (and <effect1>
                        (when <condition2> <effect2>)))
```

can be rewritten into the equivalent conjunction

```
(and
  (when <condition1> <effect1>)
  (when (and  <condition1>
              <condition2>) <effect2>))
```

Also note that the increase of model size due to the compilation of conditional effects shown above is only exponential in the number of such effects per action. If, for example, every action has a small number of conditional effects, which does not grow with problem size, this compilation can still be practical. An example of a situation in which it is not is when the domain has quantified conditional effects; although the quantifier can be removed by grounding, this give rises to a number of grounded conditional effects that grow with the number of objects (of the right type) in the problem.

Axioms add expressive power to PDDL as they also reduce the size of the problem. It has been shown that axioms can be compiled away, but that this increases either the problem or plan size by an amount that is exponential in the maximum fixed point depth needed to determine the truth value of a derived predicate [Thiébaux et al., 2005].

Preferences do not increase the expressive power of PDDL, because they can be compiled into STRIPS with action costs efficiently, incurring a linear increase of the size of the problem [Keyder and Geffner, 2009].

For the types of planning problems we have seen so far (where all actions are instantaneous), the state trajectory constraints can be compiled away with only a polynomial increase in the problem size and constant increase in plan length [Gerevini et al., 2009]. Thus, for sequential finite plans, state trajectory constraints do not add a level of expressiveness. When we shift to temporal planning (a fragment of PDDL discussed in Chapter 5), actions may occur concurrently and the compilation increases the plan length linearly, thus causing trajectory constraints to add expressive power to the formalism.