5.1 DURATIVE ACTIONS

Until now, we have presumed actions to be instantaneous occurrences that transition the world from one state to another. In its most basic form, temporal planning does not remove the instantaneous transition of one state to another (a topic left for hybrid planning in Chapter 6), however the conditions on actions and the manner in which they affect the world is generalised to address a span of time rather than an instantaneous occurrence. From the modelling perspective, we distinguish between a classical instantaneous action and a durative action by using `(:durative-action ...)` to describe the latter in the domain description.

The use of durative actions is indicated in a planning domain by adding `:durative-actions` to the list of `:requirements`. Further, if we wish to optimise for the total time a plan takes to achieve its goal, the following metric is added to the problem description:

```
(:metric minimize (total-time))
```

Note that the function term `(total-time)` is implicitly defined in temporal planning domains; it does not need to be declared in the `:functions` section, and it cannot be used anywhere else in the domain or problem.

A useful feature for defining the duration of actions is to condition it on the parameters of the action. This is done by defining (usually static) functions which are used to write expressions that evaluate to the duration, in the same way that we have seen expressions for action costs in Chapter 2 and general numeric expressions in Chapter 4. As an example, this is how we would define a function for the driving time between two locations in the domain description:

```
(:functions (drive_time ?l1 ?l2 - location))
```

To make use of this, the initial state section of the problem definition must include assertions such as `(= (drive_time Ottawa Montreal) 120)`. Note that the specific units used for the constants and durations are up to the domain designer's interpretation (in the previous example, this would be 120 minutes). That said, every duration specified is assumed to be in the same time unit, and may be any non-negative real number. Consider the following example of a durative action that includes all of the new components introduced for specifying the more complex action type:

```
(:durative-action drive
 :parameters (?c - car ?l1 ?l2 - loc)
 :duration (= ?duration (drive_time ?l1 ?l2))
 :condition (and (at start (car_at ?c ?l1))
                 (over all (road_open ?l1 ?l2))
                 (at end (free_space_at ?l2)))
 :effect (and (at start (not (car_at ?c ?l1)))
```

```
            (at start (free_space_at ?l1))
            (at end (car_at ?c ?l2))
            (at end (not (free_space_at ?l2))))
)
```

The first thing we notice in this example is the duration section. The syntax is `(= ?duration <expression>)`, where *<expression>* can be any numeric expression, built from constants, function terms and arithmetic operators, as we have seen in the preceding chapters. Here, the duration of the drive action is the driving time between the two locations that instantiate the durative action.

Next, we have the `:condition` section of the drive action (note the difference from a traditional action's `:precondition` section). The types of conditions fall into three main categories: (1) conditions that must hold when the action begins (specified as `(at start ...)`); (2) conditions that must hold during the entire action execution (specified as `(over all ...)`); and (3) conditions that must hold when the action completes (specified as `(at end ...)`).

We can see that in the `drive` action, the car must start at location `?l1`. For the duration of the action, the road must be open as well—this may model a domain where roads can be closed for maintenance. Note that the road should also be open when the action begins, but it is more appropriately assigned to an `(over all ...)` condition, as we do not want a road to be closed off while cars are currently driving. Finally, the last condition stipulates that there is space at the destination location: for demonstrative purposes, we assume that each location can only hold a single car at a time. Note that the car can begin to drive when the destination location does not have space, as long as space becomes available prior to the car arriving.

Next, we have the effects of the action. As mentioned earlier, effects of a durative action are instantaneous and appear either at the start or end of the action (using the same syntax as the `:condition` section). At the start of the `drive` action, we both delete the fact that the car is at the source location and add the fact that there is now space there. Conversely, at the end of the action, we add the new location of the car and remove the fact that states there is space at the destination.

The `drive` action is depicted in Figure 5.1. The two instantaneous conditions appear before the start and end of the action (indicated by the vertical bars), and the `over all` condition is shown inside the action itself (as it must hold for the entire duration). The effects (both positive and negative) appear on the right side of the start and end action indicators.

Generalising to the temporal setting allows us to model the concurrent execution of overlapping actions. As a larger example, let's consider a temporal version of the elevator domain presented in Section 3.1.1. Example 13 below shows the beginning of a temporal elevator domain. We will show how to define the domain's actions later in this section.

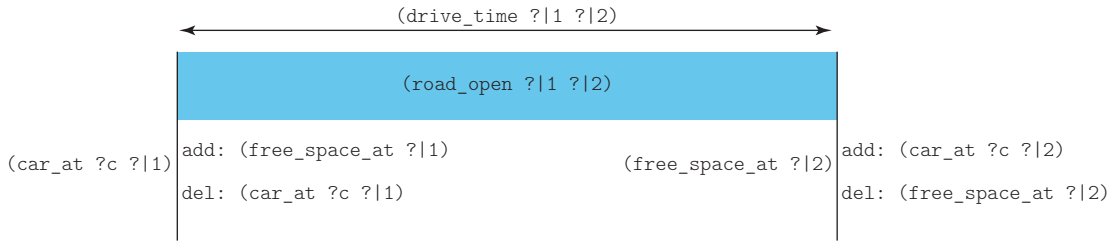

Figure 5.1: Visual representation of the drive action.

```
(define (domain temporal-elevators)
  (:requirements :typing :fluents :durative-actions)

  (: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)
   )

  (:functions
   (person_speed ?person - passenger)
   (elevator_speed ?lift - elevator)
   (floor_distance ?f1 ?f2 - num)
   )

  ;; Actions...
  )
```

PDDL Example 13: Elevator domain using durative actions.

We now have functions defined to represent the individual passenger's speed, elevator speed, and distance between floors. We assume that the units for the numeric constants are consistent. E.g., meters/sec for speed, seconds for time, and meters for distance. Now consider the elevator movement actions:

```
(:durative-action move-up
 :parameters (?lift - elevator ?cur ?nxt - num)
 :duration (= ?duration (/ (floor_distance ?cur ?nxt)
                           (elevator_speed ?lift)))
 :condition (and (at start (lift-at ?lift ?cur))
                 (over all (next ?cur ?nxt)))
 :effect (and (at start (not (lift-at ?lift ?cur)))
              (at end (lift-at ?lift ?nxt)))
 )

(:durative-action move-down
 :parameters (?lift - elevator ?cur ?nxt - num)
 :duration (= ?duration (/ (floor_distance ?cur ?nxt)
                           (elevator_speed ?lift)))
 :condition (and (at start (lift-at ?lift ?cur))
                 (over all (next ?nxt ?cur)))
 :effect (and (at start (not (lift-at ?lift ?cur)))
              (at end (lift-at ?lift ?nxt)))
 )
```

Notice the combination of values that are used to derive the duration of elevator movement: the distance divided by the speed. The `over all` condition is an invariant, as we never change the value of the `next` predicate in this domain. The other conditions and effects stipulate that the elevator changes location, but notice that *during* a move action, the elevator will not have any `lift-at` fact true; this rules out its use for boarding or disembarking passengers during this time.

Next, we have the actions for boarding and disembarking the passengers:

```
(:durative-action board
 :parameters (?per - passenger ?flr - num ?lift - elevator)
 :duration (= ?duration (person_speed ?per))
 :condition (and (over all (lift-at ?lift ?flr))
                 (at start (passenger-at ?per ?flr)))
 :effect (and (at start (not (passenger-at ?per ?flr)))
              (at end (boarded ?per ?lift)))
 )

(:durative-action leave
 :parameters (?per - passenger ?flr - num ?lift - elevator)
 :duration (= ?duration (person_speed ?per))
 :condition (and (over all (lift-at ?lift ?flr))
```

```
                    (at start (boarded ?per ?lift)))
   :effect (and (at end (passenger-at ?per ?flr))
                (at start (not (boarded ?per ?lift))))
   )
```

The passengers themselves each have a speed at which they can enter the elevator, and we require that the elevator remain at the floor during this entire time. Similar to the lift location, all `passenger-at` facts for a particular passenger will be false during the execution of these actions. Finally, we have the problem definition extended to the temporal setting:

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

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

  (:init
   ;; Same fluents as the classical planning example
   (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)

   ;; Define how fast each of the passengers move (in seconds)
   (= (person_speed p1) 2)
   (= (person_speed p2) 3)
   (= (person_speed p3) 2)

   ;; Define the speed of the elevators (in meters / second)
   (= (elevator_speed e1) 2)
   (= (elevator_speed e2) 3)

   ;; Define the distance between the floors (in meters)
   (= (floor_distance n1 n2) 3)
   (= (floor_distance n2 n3) 4)
   (= (floor_distance n3 n4) 4)
```

```
 (= (floor_distance n4 n5) 3)
 (= (floor_distance n5 n4) 3)
 (= (floor_distance n4 n3) 4)
 (= (floor_distance n3 n2) 4)
 (= (floor_distance n2 n1) 3)
 )

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

(:metric minimize (total-time))
)
```

PDDL Example 14: Elevator problem using durative actions.

The only differences in the temporal setting are the specification of the static function terms (used to define action duration) and the specification that we would like to minimize the total execution time taken of the plan. Figure 5.2 shows an example solution to the above problem (computed using an off-the-shelf temporal planner, Optic [Benton et al., 2012]). The domain can be found and interactively solved at `editor.planning.domains/pddl-book/elevator`. The textual representation of the plan is given in Figure 5.3.

The individual colours represent different action types in the domain, and the x-axis corresponds to time. Note that many of the actions occur simultaneously: for example, boarding of elevator #2 begins while elevator #1 is still traveling, and both elevators move concurrently for a time.

Required Concurrency

In some cases, a temporal planning problem can be solved by modelling and solving it as a purely classical problem, ignoring time, and then scheduling the actions in the resulting plan to satisfy duration and precendence constraints. For example, closer inspection of the schedule in Figure 5.3 shows that each action simply starts (almost) as early as possible, and ends after the required duration. (For example, `(board p1 n2 e1)` starts as soon as `(move-up e1 n1 n2)` ends; it could not start earlier since the preceding action adds `(lift-at e1 n2)` which is required `over all` by the boarding action.) This method of temporal planning fails, however, if there are actions whose execution *must* overlap for a plan to be valid. This is known as *required concurrency*.

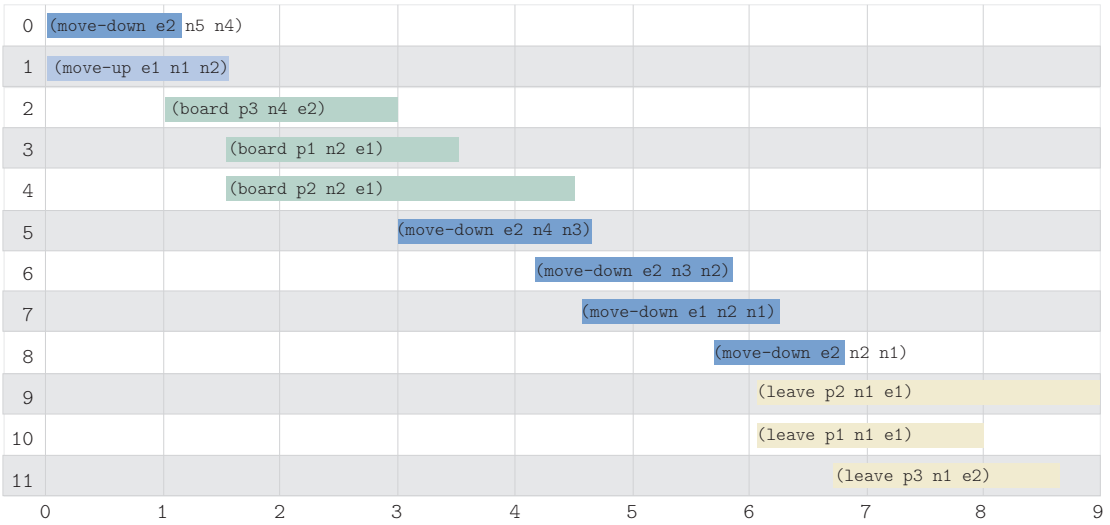

Figure 5.2: A visual representation of the plan for the temporal elevator problem.

```
0.000: (move-down e2 n5 n4)  [1.000]
0.000: (move-up e1 n1 n2)  [1.500]
1.000: (board p3 n4 e2)  [2.000]
1.500: (board p1 n2 e1)  [2.000]
1.501: (board p2 n2 e1)  [3.000]
3.000: (move-down e2 n4 n3)  [1.333]
4.334: (move-down e2 n3 n2)  [1.333]
4.501: (move-down e1 n2 n1)  [1.500]
5.668: (move-down e2 n2 n1)  [1.000]
6.001: (leave p2 n1 e1)  [3.000]
6.001: (leave p1 n1 e1)  [2.000]
6.668: (leave p3 n1 e2)  [2.000]
```

Figure 5.3: A plan for the temporal elevator problem (Example 14).

As an example, let us model the decision to open the elevator doors at a floor with an explicit action:

```
(:durative-action open-door
 :parameters (?lift - elevator)
 :duration (= ?duration (door_speed ?lift))
 :condition (at start (not (open ?lift)))
```

```
:effect (and (at start (open ?lift))
             (at end (not (open ?lift))))
)
```

We then modify the `board` and `leave` actions to require that `(over all (open ?lift))` holds. There is now required concurrency between the action that holds the elevator door open, and the movement of passengers: the `open-door` action essentially envelopes the passengers' movement actions. Only passengers who move fast enough to complete their boarding or leaving action within the duration of the `open-door` action can now enter or leave the elevator, i.e., `(person_speed ?per)` must be less than `(door_speed ?lift)`.

Example: Forcing Action Non-Overlap

In the preceding example, actions `board` and `leave` are forced to overlap with `open-door`. This is achieved through a predicate that is added only during the duration of the enveloping action and required by the enveloped actions. A complementary example is to force actions to *not* overlap through the use of a *unary resource*. Consider the following modification of the `open-door` action:

```
(:durative-action open-door
 :parameters (?lift - elevator ?flr - num)
 :duration (= ?duration (door_speed ?lift))
 :condition (and (at start (not (open ?lift)))
                 (at start (can-stop-at ?flr))
                 (overall (lift-at ?lift ?flr)))
 :effect (and (at start (open ?lift))
              (at start (not (can-stop-at ?flr)))
              (at end (not (open ?lift)))
              (at end (can-stop-at ?flr)))
)
```

Here, the predicate `can-stop-at` restricts the domain such that only one elevator can open its door at a particular floor at any time. We achieve this by requiring the fact at the start of the action, and deleting it only for the duration of the action, i.e., deleting it at the start of the action and adding it again at the end. Note that this resource is associated with only the floor, and not the elevator as well.

Example: Restricting Plan Duration

As a final encoding trick for this section, we will introduce one further action to the elevators domain: `execute`. PDDL has no special-purpose mechanism for specifying a maximum plan duration, but we can get around this by creating an auxiliary action that envelopes every other action in the plan. The process is two-fold: (1) we add a predicate `(enabled)` to the domain and

make it an `over all` condition for every action; and (2) we add the predicate `(can-execute)` to the domain and to the initial state, and add the following action to the domain (making the maximum duration 10 units of time):

```
(:durative-action execute
 :parameters ()
 :duration (= ?duration 10)
 :condition (at start (can-execute))
 :effect (and (at start (enabled))
              (at start (not (can-execute)))
              (at end (not (enabled))))
)
```

Notice that because `(can-execute)` is both required and deleted at the start of `execute`, and not added by any action in the domain, the `execute` action can only occur once. Further, since `(enabled)` will only hold during that execution, and since every action has this fact as an overall condition, every action (aside from `execute`) must occur entirely within the duration of this new action. Changing the duration will change the maximum amount of time that is given for all other actions to occur.

Now that we have concluded a description of the basic temporal planning syntax and usage, we will move to describing a commonly used extension for modelling predictable events: timed initial literals.

5.2 PLANNING WITH PREDICTABLE EVENTS

Now that we have made the shift to a setting where actions must be orchestrated temporally in order to produce a solution, a natural question is how we may specify temporal restrictions on the state of the world. In the temporal fragment of PDDL, this can be done through what is known as *timed initial literals* (TILs): changes to the state of the world that occur a set time after the start of the plan.

TILs are specified as part of the initial state of a temporal problem, and may be either positive (adding a fact) or negative (deleting a fact). The syntax of TILs is:

```
(at <time> <literal>)
```

where *<time>* is a numeric constant (integer or decimal) and *<literal>* is a positive or negative ground literal. The status of the symbol `at` is somewhat ambiguous. Due to its special role in writing TILs, it could be considered a reserved word, at least in the temporal fragment of PDDL. However, there are also many examples of domain models that use `at` as a predicate, and many planners accept this dual use. The use of TILs in a problem is indicated with the keyword `:timed-initial-literals` in the `:requirements` section.

The introduction of TILs does not strictly increase the expressivity of temporal planning problems that can be defined, as one can use modelling tricks to capture the same behaviour, e.g.,