PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects

Håkan L. S. Younes
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
lorens@cs.cmu.edu

Michael L. Littman
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
mlittman@cs.rutgers.edu

April 13, 2004

1 PDDL Basics

PDDL is a language for specifying deterministic planning domains and problems. We describe the basic building blocks of PDDL in this section. In the next section we introduce extensions necessary to express probabilistic and decision-theoretic planning domains and problems.

1.1 Planning Domains

A PDDL planning domain consists of a set T of types, a subtyping relation $ST \subset T \times T$, a set C of global objects (domain *constants*), a set P of predicates, a set F of functions, and a set AS of action schemata. Predicates and functions are used to encode *state variables*.

When defining a domain in PDDL, it is given a unique name that is used when referring to the domain in problem definitions (see Section 1.2). Figure 1 shows the definition of a domain named "test-domain" in PDDL. The statement

```
(:requirements :typing :equality :conditional-effects :fluents)
```

signals to a planner reading the domain definition that support for typing, equality, conditional effects, and fluents are required in order to correctly handle the domain being defined. Requirements are explained in more detail in Appendix A, where a full grammar for PPDDL is provided. Tokens starting with a question mark, for example `?x`, are *variables*, which are not to be confused with *state* variables.

1.1.1 Types

Objects and variables are *terms*, and in PDDL all terms have some type $\tau \in T$. The domain definition in Figure 1 declares two types: `car` and `box`. The constant `goldie` is declared to be of type `car`, while the first parameter of the action schema `load` is declared to be of type `box`.

```
(define (domain test-domain)
   (:requirements :typing :equality :conditional-effects
                  :fluents)
   (:types car box)
   (:constants goldie - car)
   (:predicates (parked ?x - car) (holding ?x - box)
                (in ?x - box ?y - car))
   (:functions (fuel-level ?x - car))
   (:action load
            :parameters (?x - box ?y - car)
            :precondition (and (holding ?x) (parked ?y))
            :effect (and (in ?x ?y)
                         (forall (?z - car)
                            (when (not (= ?z ?y))
                               (not (in ?x ?z))))))
   (:action refuel
            :parameters (?x - car)
            :precondition (< (fuel-level ?x) 10)
            :effect (increase (fuel-level ?x) 1)))
```

Figure 1: Definition of a simple planning domain in PDDL.

All declared types are by default subtypes of the built-in PDDL type `object`. A type is also a subtype of itself (the subtyping relation ST is reflexive), and if τ_1 is a subtype of τ_2 and τ_2 is a subtype of τ_3, then τ_1 is a subtype of τ_3 (ST is transitive). For the example domain we have $T = \{\texttt{object}, \texttt{car}, \texttt{box}\}$ with ST containing the elements $\langle \texttt{object}, \texttt{object} \rangle$, $\langle \texttt{car}, \texttt{object} \rangle$, $\langle \texttt{car}, \texttt{car} \rangle$, $\langle \texttt{box}, \texttt{object} \rangle$, and $\langle \texttt{box}, \texttt{box} \rangle$.

It is possible to specify the supertype of a type in the type declaration of a domain definition. For example, the type declaration

```
(:types car - object saab volvo - car)
```

declares the type `car` as a subtype of `object`, and the types `saab` and `volvo` as subtypes of `car`.

The types `car` and `object` are examples of *simple* types. PDDL also includes support for *union* types $\tau_1 \cup \ldots \cup \tau_n$, with the restriction that each τ_i is a simple type.[1] As an example of the use of union types, consider the declaration

```
(:constants herbie - (either saab volvo))
```

of a domain constant `herbie` of type $\texttt{saab} \cup \texttt{volvo}$. A union type $\tau_1 \cup \ldots \cup \tau_n$ is a subtype of τ if τ_i is a subtype of τ for all $i \in [1, n]$. On the other hand, τ is a subtype of a union type $\tau_1 \cup \ldots \cup \tau_n$ if τ is a subtype of τ_i for *some* $i \in [1, n]$.

[1]The original version of PDDL (Ghallab et al. 1998) allowed unions of union types. Later versions, including PDDL2.1 (Fox & Long 2003) only allow unions of simple types. The restriction is not significant as for any type that is a union of union types, we can find an equivalent type that is a union of simple types by "flattening" the union.

1.1.2 Predicates and Functions

In PDDL, predicates are used to encode boolean state variables, while functions are used to encode numeric state variables. A function declaration in PDDL, such as

```
(:functions (fuel-level ?x - car))
```

in the example domain (Figure 1), is in reality a declaration of a function from PDDL objects to numeric state variables. A predicate is also a function with PDDL objects as domain, but with boolean state variables as range. The type of a function (predicate) parameter restricts the domain of the function (predicate). The function `fuel-level`, for example, only applies to objects of type `car` or a subtype of `car`.

The value of the application `(parked goldie)` is a boolean state variable $parked_{\text{goldie}}$. Say that in addition to the domain constant `goldie`, there is also an object `ups-box` of type `box`. Given this set of objects, the predicates and functions of "test-domain" give rise to a state space made up of the following state variables:

Name	Type
$parked_{\text{goldie}}$	boolean
$holding_{\text{ups-box}}$	boolean
$in_{\text{ups-box,goldie}}$	boolean
$\mathit{fuel\text{-}level}_{\text{goldie}}$	numeric

A function that does not take any arguments represents a single state variable with the same name as the function. It is then possible to refer directly to the state variable in PDDL domain and problem definitions without having to use function application. For example, given a 0-ary function `score`, the formulas `(= (score) 17)` and `(= score 17)` and equivalent.

It is worth noting that a domain definition alone does not, in general, determine the extent of the state space for planning problems linked to the domain, unless all functions and predicates take no arguments. In addition to objects declared as domain constants, objects can also be declared in problem definitions. Only when the complete set of objects for a planning problem is known can the state space be determined.

1.1.3 Actions

Actions in PDDL can be thought of as representing sets of state transitions, with a state being a particular assignment to the set of state variables of a planning problem. An action consists of a precondition, characterizing the set of states that the action is applicable in, and an effect. The effect specifies updates to state variables that occur at the execution of the action.

Basic effects specify updates to individual state variables. For a boolean state variable b, the effect b (or an application yielding the state variable b) simply means that b should be set to true in the next state. To set b to false, the notation `(not` b`)` is used. For a numeric state variable x, the general form for updates is `(`⟨*assign-op*⟩ x ⟨*f-exp*⟩`)`, where ⟨*assign-op*⟩ is one of `assign`, `scale-up`, `scale-down`, `increase`, or `decrease`, and ⟨*f-exp*⟩ is a numeric expression.

Effects can be combined by using conjunction. PDDL also includes support for conditional effects of the form `(when` c e`)`, meaning that the effect e only occurs in states satisfying the

Name	Type	Init
$parked_{\text{goldie}}$	boolean	true
$holding_{\text{ups-box}}$	boolean	true
$holding_{\text{cereal-box}}$	boolean	false
$in_{\text{ups-box,goldie}}$	boolean	false
$in_{\text{cereal-box,goldie}}$	boolean	true
$\mathit{fuel\text{-}level}_{\text{goldie}}$	numeric	7

Table 1: State variables and their initial values for "test-problem" defined in Figure 2.

condition c, and universally quantified effects. Examples of all these kinds of effects are given in Figure 1.

An action schema, in the definition of a domain, declares a function from PDDL objects to actions, much in the same way as functions and predicates are functions from PDDL objects to state variables. The action schema

```
(:action refuel
         :parameters (?x - car)
         :precondition (< (fuel-level ?x) 10)
         :effect (increase (fuel-level ?x) 1))
```

in "test-domain", when applied to the object `goldie`, returns an action $\mathsf{refuel}_{\text{goldie}}$. This action is applicable in states satisfying the condition $\mathit{fuel\text{-}level}_{\text{goldie}} < 10$, and has the effect that $\mathit{fuel\text{-}level}_{\text{goldie}}$ is increased by one.

1.2 Planning Problems

A planning problem consists of a set of state variables V, a set of actions A, an initial state s_0, a goal condition ϕ identifying a set of goal states, and an optimization metric f that is typically a function of numeric state variables evaluated in a goal state. A state is simply an assignment of values to the set of state variables.

In PDDL, a planning problem is always associated with a domain definition, and the definition of a planning problem includes a declaration of a set of problem-specific objects O. The state variables V for the planning problem are obtained from O, C, P, and F as all type-consistent applications of predicates or functions to objects (including domain constants). The set A of actions is obtained similarly as all type-consistent applications of action schemata in AS to objects in $O \cup C$.

Figure 2 shows the definition of a simple planning problem associated with a domain named "test-domain" (defined in Figure 1). The problem definition declares two problem-specific objects, `ups-box` and `cereal-box`, both of type `box`. The set V of state variables for this planning problem is listed in Table 1. The table also shows the value of each state variable in the initial state s_0 of the problem, as specified by `(:init ...)` in the problem definition. Note that boolean state variables not mentioned in the init specification, for example $holding_{\text{cereal-box}}$, are assumed to be false in the initial state (*closed world* assumption). The actions for "test-problem" are $\mathsf{load}_{\text{ups-box,goldie}}$, $\mathsf{load}_{\text{cereal-box,goldie}}$, and $\mathsf{refuel}_{\text{goldie}}$.

```
(define (problem test-problem)
   (:domain test-domain)
   (:objects ups-box cereal-box - box)
   (:init (parked goldie)
          (holding ups-box)
          (in cereal-box goldie)
          (= (fuel-level goldie) 7))
   (:goal (and (in ups-box goldie) (>= (fuel-level goldie) 9))))
```

Figure 2: Definition of a simple planning problem in PDDL associated with the domain named "test-domain" defined in Figure 1. Note that a hyphen ("`-`") can be part of a name for objects, types, etc., but that it is also used in the assignment of types to objects and variables. A hyphen is assumed to be part of a name token, unless it is preceded by white space. For example, `cereal-box` is a single name token, while `cereal-box - box` specifies that `cereal-box` has type `box`.

2 Probabilistic and Decision-Theoretic Extensions

We now describe syntactic extensions to PDDL that allows us to specify Markov decision processes (MDPs). The key extension is support for probabilistic effects. Rewards, an essential part of MDPs, are modeled using an existing language feature: fluents. However, we restrict the use of rewards so that full support for fluents does not become a prerequisite for MDP planning.

2.1 Probabilistic Effects

In order to define probabilistic and decision-theoretic planning problems, we need to add support for probabilistic effects. The syntax for probabilistic effects is

$$\texttt{(probabilistic}\ p_1\ e_1\ \ldots\ p_k\ e_k\texttt{)}$$

meaning that effect e_i occurs with probability p_i. We require that the constraints $p_i \geq 0$ and $\sum_{i=1}^{k} p_i = 1$ are fulfilled: a probabilistic effect declares an exhaustive set of probability-weighted outcomes. However, we allow a probability-effect pair to be left out if the effect is empty. In other words,

$$\texttt{(probabilistic}\ p_1\ e_1\ \ldots\ p_{k-1}\ e_{k-1}\texttt{)}$$

with $\sum_{i=1}^{k-1} p_i \leq 1$ is syntactic sugar for

$$\texttt{(probabilistic}\ p_1\ e_1\ \ldots\ p_{k-1}\ e_{k-1}\ p_k\ \texttt{(and))}$$

with $p_k = 1 - \sum_{i=1}^{k-1} p_i$. For example, the effect `(probabilistic 0.9 (clogged))` means that with probability 0.9 the state variable $clogged$ becomes true in the next state, while with probability 0.1 the state remains unchanged. Outcomes are not required to be mutually exclusive.

Figure 3 shows an encoding in PPDDL of the "Bomb and Toilet" example described by Kushmerick et al. (1995). In this problem, there are two packages, one of which contains a bomb. The bomb can be defused by dunking the package containing the bomb in the toilet. There is a 0.05 probability of the toilet becoming clogged when a package is placed in it. The problem definition

```
(define (domain bomb-and-toilet)
   (:requirements :conditional-effects :probabilistic-effects)
   (:predicates (bomb-in-package ?pkg) (toilet-clogged)
                (bomb-defused))
   (:action dunk-package
            :parameters (?pkg)
            :effect (and (when (bomb-in-package ?pkg)
                               (bomb-defused))
                         (probabilistic 0.05 (toilet-clogged)))))

(define (problem bomb-and-toilet)
   (:domain bomb-and-toilet)
   (:requirements :negative-preconditions)
   (:objects package1 package2)
   (:init (probabilistic 0.5 (bomb-in-package package1)
                         0.5 (bomb-in-package package2)))
   (:goal (and (bomb-defused) (not (toilet-clogged)))))
```

Figure 3: PPDDL encoding of "Bomb and Toilet" example.

Name	Type	Init 1	Init 2
$bomb\text{-}in\text{-}package_{\text{package1}}$	boolean	true	false
$bomb\text{-}in\text{-}package_{\text{package2}}$	boolean	false	true
$toilet\text{-}clogged$	boolean	false	false
$bomb\text{-}defused$	boolean	false	false

Table 2: State variables and their initial values for the "Bomb and Toilet" problem.

in Figure 3 also shows that initial conditions in PPDDL can be probabilistic. In this particular example we define two possible initial states with equal probability (0.5) of being the true initial state. Table 2 lists the state variables for the "Bomb and Toilet" problem and their values in the two possible initial states. Intuitively, we can think of the initial conditions of a PPDDL planning problem as being the effects of an action forced to be scheduled right before time 0. Also, note that the goal of the problem involves negation, which is why the problem definition declares the `:negative-preconditions` requirements flag.

PPDDL allows arbitrary nesting of conditional and probabilistic effects. This is in contrast to popular propositional encodings, such as probabilistic STRIPS operators (PSOs) (Kushmerick et al. 1995) and factored PSOs (Dearden & Boutilier 1997), which do not allow conditional effects nested inside probabilistic effects. While arbitrary nesting does not add to the expressiveness of the language, it can allow for exponentially more compact representations of certain effects given the same set of state variables and actions (Rintanen 2003). However, any PPDDL action can be translated into a *set* of PSOs with at most a polynomial increase in size of the representation. Consequently, it follows from the results of Littman (1997) that PPDDL is representationally equivalent to dynamic Bayesian networks (Dean & Kanazawa 1989), which is another popular representation for MDP planning problems.

```
(define (domain coffee-delivery)
   (:requirements :negative-preconditions
                  :disjunctive-preconditions
                  :conditional-effects :mdp)
   (:predicates (in-office) (raining) (has-umbrella) (is-wet)
                (has-coffee) (user-has-coffee))
   (:action buy-coffee
            :effect (and (when (not (in-office))
                               (probabilistic 0.8 (has-coffee)))
                         (when (user-has-coffee)
                               (increase (reward) 0.8))
                         (when (not (is-wet))
                               (increase (reward) 0.2))))
   ...)
```

Figure 4: Part of PPDDL encoding of "Coffee Delivery" domain.

2.2 Rewards

Markovian rewards, associated with state transitions, can be encoded using fluents. PPDDL reserves the fluent $reward$, accessed as `(reward)` or `reward`, to represent the total accumulated reward since the start of execution. Rewards are associated with state transitions through update rules in action effects. The use of the $reward$ fluent is restricted to action effects of the form `(`⟨*additive-op*⟩ ⟨*reward fluent*⟩ ⟨*f-exp*⟩`)`, where ⟨*additive-op*⟩ is either `increase` or `decrease`, and ⟨*f-exp*⟩ is a numeric expression not involving $reward$. Action preconditions and effect conditions are not allowed to refer to the $reward$ fluent, which means that the accumulated reward does not have to be considered part of the state space. The initial value of $reward$ is zero. These restrictions on the use of the $reward$ fluent allow a planner to handle domains with rewards, without having to implement full support for fluents.

A new requirements flag, `:rewards`, is introduced to signal that support for rewards is required. Domains that require both probabilistic effects and rewards can declare the `:mdp` requirements flag, which implies `:probabilistic-effects` and `:rewards`.

Figure 4 shows part of the PPDDL encoding of a coffee delivery domain described by Dearden & Boutilier (1997). A reward of 0.8 is awarded if the user has coffee when the "buy-coffee" action is executed, and a reward of 0.2 is awarded when "buy-coffee" is executed in a state where $is\text{-}wet$ is false. Note that a total reward of 1.0 can be awarded as a result of executing the "buy-coffee" action if it is executed in a state where both $user\text{-}has\text{-}coffee$ and $\neg is\text{-}wet$ hold.

Action effects with inconsistent transition rewards are not permitted. For example, the effect `(probabilistic 0.5 (increase (reward) 1))` is semantically invalid because it associates a reward of both 1 and 0 to a self-transition.

2.3 Plan Objectives

Regular PDDL goals is used to express goal-type performance objectives. A goal statement `(:goal` ϕ`)` for a probabilistic planning problem encodes the objective that the probability of

achieving ϕ should be maximized, unless an explicit optimization metric is specified for the planning problem. For planning problems instantiated from a domain declaring the `:rewards` requirement, the default plan objective is to maximize the expected reward. A goal statement in the specification of a reward oriented planning problem identifies a set of absorbing states. In addition to transition rewards specified in action effects, it is possible to associate a one-time reward for entering a goal state. This is done using the `(:goal-reward` f`)` construct, where f is a numeric expression.

In general, a statement `(:metric maximize` f`)` in a problem definition means that the expected value of f should be maximized. PPDDL defines `goal-probability` as a special optimization metric that can be used to explicitly specify that the plan objective is to maximize (or minimize) the probability of goal achievement.

3 Formal Semantics

We present a formal semantics for PPDDL planning problems in terms of a mapping to a probabilistic transition system with rewards. A planning problem defines a set of state variables V, possibly containing both Boolean and numeric state variables. An assignment of values to state variables defines a state, and the state space S of the planning problem is the set of states representing all possible assignments of values to variables. In addition to V, a planning problem defines an initial-state distribution $p_0 : S \to [0, 1]$ with $\sum_{s \in S} p_0(s) = 1$ (i.e. p_0 is a probability distribution over states), a formula ϕ over V characterizing a set of goal states $G = \{s \mid s \models \phi\}$, a one-time reward r_G associated with entering a goal state, and a set of actions A instantiated from PPDDL action schemata. For goal-directed planning problems, without explicit rewards, we use $r_G = 1$.

An action $a \in A$ consists of a precondition ϕ_a and an effect e_a. Action a is applicable in a state s if and only if $s \models \phi_a$. It is an error to apply a to a state such that $s \not\models \phi_a$. This is consistent with the semantics of PDDL2.1 (Fox & Long 2003) and permits the modeling of forced chains of actions. Effects are recursively defined as follows (cf. Rintanen 2003):

1. $\top$ is the null-effect, represented in PPDDL by `(and)`.
2. b and $\neg b$ are effects if $b \in V$ is a Boolean state variable.
3. $x \leftarrow f$ is an effect if $x \in V$ is a numeric state variable and f is a real-valued function on numeric state variables.
4. $r \uparrow f$ is an effect if f is a real-valued function on numeric state variables.
5. $e_1 \wedge \ldots \wedge e_n$ is an effect if $e_1, \ldots, e_n$ are effects.
6. $c \rhd e$ is an effect if c is a formula over V and e is an effect.
7. $p_1 e_1 | \ldots | p_n e_n$ is an effect if $e_1, \ldots, e_n$ are effects, $p_i \geq 0$ for all $i \in \{1, \ldots, n\}$, and $\sum_{i=1}^{n} p_i = 1$.

Items 2 through 4 are referred to as *simple effect*. The effect b sets the Boolean state variable b to true in the next state, while $\neg b$ sets b to false in the next state. For $x \leftarrow f$, the value of f in

the current state becomes the value of the numeric state variable x in the next state. Effects on the form $r \uparrow f$ are used to associate reward with transitions as described below.

An action $a = \langle \phi_a, e_a \rangle$ defines a transition probability matrix P_a and a transition reward matrix R_a, with p^a_{ij} being the probability of transitioning to state j when applying a in state i, and r^a_{ij} being the reward associated with the state transition from i to j when caused by a. We can compute P_a and R_a by first translating e_a into an effect on the form $p_1 e_1 | \ldots | p_n e_n$, where each e_i is a deterministic effect. Rintanen (2003) calls this form Unary Nondeterminism Normal Form. Any effect e can be translated into this form by using the following equivalences:

$$\begin{aligned}
e \equiv & 1 e \\
e \wedge (p_1 e_1 | \ldots | p_k e_n) \equiv & p_1 (e \wedge e_1) | \ldots | p_n (e \wedge e_n) \\
c \rhd (p_1 e_1 | \ldots | p_n e_n) \equiv & p_1 (c \rhd e_1) | \ldots | p_n (c \rhd e_n) \\
p_1 (p'_1 e'_1 | \ldots | p'_k e'_k) | p_2 e_2 | \ldots | p_n e_n \equiv & (p_1 p'_1) e_1 | \ldots | (p_1 p'_k) e'_k | p_2 e_2 | \ldots | p_n e_n
\end{aligned}$$

We further rewrite the effect of an action by translating each e_i into an effect on the form $(c_{i1} \rhd e_{i1}) \wedge \ldots \wedge (c_{in_i} \rhd e_{in_i})$, where each e_{ij} is a conjunction of simple effects and the conditions are mutually exclusive and exhaustive (i.e. $c_{ij} \wedge c_{ik} \equiv \bot$ for all $j \neq k$ and $\bigvee_{j=1}^{n_i} c_{ij} \equiv \top$). The following equivalences allow us to perform the desired translation:

$$\begin{aligned}
e \equiv & \top \rhd e \\
c \rhd e \equiv & (c \rhd e) \wedge (\neg c \rhd \top) \\
c \rhd (c' \rhd e) \equiv & (c \wedge c') \rhd e \\
(c_1 \rhd e_1) \wedge (c_2 \rhd e_2) \equiv & ((c_1 \wedge c_2) \rhd (e_1 \wedge e_2)) \wedge ((c_1 \wedge \neg c_2) \rhd e_1) \\
& \wedge ((\neg c_1 \wedge c_2) \rhd e_2) \wedge ((\neg c_1 \wedge \neg c_2) \rhd \top)
\end{aligned}$$

An effect on the form $c \rhd e$, where e is a conjunction of simple effects, defines a set of state transitions. We assume that e is consistent. Actions with inconsistent effects are not valid PPDDL actions, and care should be taken when designing a PPDDL domain to ensure that no instantiations of action schemata can have inconsistent effects. A conjunction of simple effects is inconsistent if it contains both b and $\neg b$, or multiple *non-commutative* updates of a single numeric state variable. Two effects $x \leftarrow f$ and $x \leftarrow f'$ are commutative if $f(s[x = f'(s)]) = f'(s[x = f(s)])$, where $f(s)$ is the value of f evaluated in state s and $s[x = y]$ denotes a state with all state variables having the same value as in state s, except for x which has value y, i.e. numeric effects are commutative if they are insensitive to ordering. Under these assumptions, the following function can be defined:

$$\begin{aligned}
\tau(s, s', \top) \doteq & s' \\
\tau(s, s', b) \doteq & s'[b = \top] \\
\tau(s, s', \neg b) \doteq & s'[b = \bot] \\
\tau(s, s', x \leftarrow f) \doteq & s'[x = f(s)] \\
\tau(s, s', r \uparrow f) \doteq & s' \\
\tau(s, s', e_1 \wedge e_2) \doteq & \tau(s, \tau(s, s', e_1), e_2)
\end{aligned}$$

We can use τ to describe the set of state transitions defined by the effect $c \rhd e$:

$$T(c \rhd e) = \{\langle s, s' \rangle | s \models c \text{ and } s' = \tau(s, s, e)\}.$$

Given this definition of $T(c \rhd e)$, we can compute a transition matrix T_{ij} for each $c_{ij} \rhd e_{ij}$. The element at row s and column s' of T_{ij} is 1 if $\langle s, s' \rangle \in T(c_{ij} \rhd e_{ij})$, and 0 otherwise. Since we have ensured that the conditions c_{ij} are mutually exclusive, we get $P_a = \sum_{i=1}^{n} p_i T_i$ as the transition probability matrix for action a, where $T_i = \sum_{j=1}^{n_i} T_{ij}$. Finally, we need to make all states that satisfy the goal condition ϕ of the problem absorbing. This is accomplished by modifying P_a: for each s such that $s \models \phi$, we set the entry at row s and column s to 1 and the remaining entries on the same row to 0.

The reward associated with a conjunction of simple effects can be defined as follows:

$$\begin{aligned} r(s, \top) &\doteq 0 \\ r(s, b) &\doteq 0 \\ r(s, \neg b) &\doteq 0 \\ r(s, x \leftarrow f) &\doteq 0 \\ r(s, r \uparrow f) &\doteq f(s) \\ r(s, e_1 \wedge e_2) &\doteq r(s, e_1) + e(s, e_2) \end{aligned}$$

The effect $c_{ij} \rhd e_{ij}$ associates reward $r(s, e_{ij})$ with each transition $\langle s, s' \rangle \in T(c_{ij} \rhd e_{ij})$. We define a transition reward matrix R_{ij} for $c_{ij} \rhd e_{ij}$. The element at row s and column s' of R_{ij} is $r(s, e_{ij})$ for $s' = \tau(s, s, e_{ij})$ and 0 if $\langle s, s' \rangle \notin T_{ij}$. We then sum over all $c_{ij} \rhd e_{ij}$ to get a transition reward matrix for e_i: $R_i = \sum_{j=1}^{n_i} R_{ij}$.

The same transition may occur in multiple outcomes of the effect $p_1 e_1 | \ldots | p_n e_n$, and we require the reward for a specific transition to be consistent across outcomes. Let $\bullet$ represent the fact that the reward is undefined for a transition. We define $\tilde{R}_i$ to be R_i with an element at row s and column s' set to $\bullet$ if the element at row s and column s' of T_i is zero (i.e. e_i does not define a transition from s to s'). We define an element-wise matrix operator $\odot$ as follows:

$$\begin{aligned} \bullet \odot x &\doteq x \\ x \odot \bullet &\doteq x \\ x \odot x &\doteq x \\ x \odot y &\doteq \mathit{error} \text{ if } x \neq y \end{aligned}$$

We can now define the transition reward matrix for action a: $R_a = R_G + \bigodot_{i=1}^{n} \tilde{R}_i$. R_G represents the one-time reward associated with goal states. The entry at row s and column s' of R_G is set to r_G if $s \not\models \phi$ and $s' \models \phi$, and 0 otherwise. The transition reward matrix is well-defined if and only if the transition rewards are consistent across all outcomes of an action.

Consider the "Bomb and Toilet" example in Figure 3. This planning problem has four state variables, and thus a state space of size 16.[2] Let b_1 be the state variable $\mathit{bomb\text{-}in\text{-}package}_{\text{package1}}$,

[2]Not all 16 states are reachable for this problem. For example, the bomb is in exactly one of the two packages, so $\mathit{bomb\text{-}in\text{-}package}_{\text{package1}} \equiv \neg \mathit{bomb\text{-}in\text{-}package}_{\text{package2}}$ for all states which means there are at most 8 reachable states.

b_2 the state variable $bomb\text{-}in\text{-}package_{\text{package2}}$, b_3 the state variable $toilet\text{-}clogged$, and b_4 the state variable $bomb\text{-}defused$. The action dunk-package$_{\text{package1}}$ has precondition $\top$ (i.e. is applicable in all states) and effect $(b_1 \rhd b_4) \wedge (0.05b_3|0.95\top)$. We transform this effect, using the equivalences given above, to the effect $0.05((b_1 \rhd (b_3 \wedge b_4)) \wedge (\neg b_1 \rhd b_3))|0.95((b_1 \rhd b_4) \wedge (\neg b_1 \rhd \top))$. We use the following encoding of states:

b_1	b_2	b_3	b_4	state
$\bot$	$\bot$	$\bot$	$\bot$	1
$\bot$	$\bot$	$\bot$	$\top$	2
$\bot$	$\bot$	$\top$	$\bot$	3
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\top$	$\top$	$\top$	$\top$	16

Given this encoding, we get the following transition probability matrix for dunk-package$_{\text{package1}}$:

$$P = \begin{pmatrix}
\frac{19}{20} & 0 & \frac{1}{20} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{19}{20} & 0 & \frac{1}{20} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{19}{20} & 0 & \frac{1}{20} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{19}{20} & 0 & \frac{1}{20} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Note that all states satisfying the goal condition $\neg b_3 \wedge b_4$ for the given planning problem have been made absorbing. The goal states are 2, 6, 10, and 14. We also get the following transition reward

matrix:

$$R = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A BNF Grammar for PPDDL1.0

We provide the full syntax for PPDDL1.0 using an extended BNF notation with the following conventions:

- Each rule is of the form ⟨*non-terminal*⟩ ::= *expansion*.
- Alternative expansions are separated by a vertical bar ("|").
- A syntactic element surrounded by square brackets ("[" and "]") is optional.
- Expansions and optional syntactic elements with a superscripted requirements flag are only available if the requirements flag is specified for the domain or problem currently being defined. For example, [⟨*types-def*⟩]`:typing` in the syntax for domain definitions means that ⟨*types-def*⟩ may only occur in domain definitions that include the `:typing` flag in the requirements declaration.
- An asterisk ("*") following a syntactic element x means that zero or more occurrences of x; a plus ("+") following x means at least one occurrence of x.
- Parameterized non-terminals, for example ⟨*typed list (*x*)*⟩, represent separate rules for each instantiation of the parameter.
- Terminals are written using `typewriter` font.
- The syntax is Lisp-like. In particular this means that case is not significant (e.g. `?x` and `?X` are equivalent), parenthesis are an essential part of the syntax and have no semantic meaning in the extended BNF notation, and any number of whitespace characters (space, newline, tab, etc.) may occur between tokens.

A.1 Domains

The syntax for domain definitions is the same as for PDDL2.1, except that durative actions are not allowed. Declarations of constants, predicates, and functions are allowed in any order with respect to one another, but they must all come after any type declarations and precede any action declarations.

⟨*domain*⟩ ::= `(` `define` `(` `domain` ⟨*name*⟩ `)`
[⟨*require-def*⟩]
[⟨*types-def*⟩]`:typing`
[⟨*constants-def*⟩]
[⟨*predicates-def*⟩]
[⟨*functions-def*⟩]`:fluents`
⟨*structure-def*⟩`*` `)`

⟨*require-def*⟩ ::= `(` `:requirements` ⟨*require-key*⟩`*` `)`

⟨*require-key*⟩ ::= *See Section A.4*

⟨*types-def*⟩ ::= `(` `:types` ⟨*typed list (name)*⟩ `)`

⟨*constants-def*⟩ ::= (:constants ⟨*typed list (name)*⟩)
⟨*predicates-def*⟩ ::= (:predicates ⟨*atomic formula skeleton*⟩*)
⟨*atomic formula skeleton*⟩ ::= (⟨*predicate*⟩ ⟨*typed list (variable)*⟩)
⟨*predicate*⟩ ::= ⟨*name*⟩
⟨*functions-def*⟩ ::= (:functions ⟨*function typed list (function skeleton)*⟩)
⟨*function skeleton*⟩ ::= (⟨*function symbol*⟩ ⟨*typed list (variable)*⟩)
⟨*function symbol*⟩ ::= ⟨*name*⟩
⟨*structure-def*⟩ ::= ⟨*action-def*⟩
⟨*action-def*⟩ ::= *See Section A.2*
⟨*typed list (*x*)*⟩ ::= ⟨x⟩* |:typing ⟨x⟩+ - ⟨*type*⟩ ⟨*typed list (*x*)*⟩
⟨*type*⟩ ::= (either ⟨*primitive type*⟩+) | ⟨*primitive type*⟩
⟨*primitive type*⟩ ::= ⟨*name*⟩
⟨*function typed list (*x*)*⟩ ::= ⟨x⟩* |:typing ⟨x⟩+ - ⟨*function type*⟩ ⟨*function typed list (*x*)*⟩
⟨*function type*⟩ ::= number

A ⟨*name*⟩ is a string of characters starting with an alphabetic character followed by a possibly empty sequence of alphanumeric characters, hyphens ("-"), and underscore characters ("_"). A ⟨*variable*⟩ is a ⟨*name*⟩ immediately preceded by a question mark ("?"). For example, in-office and ball_2 are names, and ?gripper is a variable.

A.2 Actions

Action definitions and goal descriptions have the same syntax as in PDDL2.1.

⟨*action-def*⟩ ::= (:action ⟨*action symbol*⟩
[:parameters (⟨*typed list (variable)*⟩)]
⟨*action-def body*⟩)
⟨*action symbol*⟩ ::= ⟨*name*⟩
⟨*action-def body*⟩ ::= [:precondition ⟨*GD*⟩]
[:effect ⟨*effect*⟩]
⟨*GD*⟩ ::= ⟨*atomic formula (term)*⟩ | (and ⟨*GD*⟩*)
|:equality (= ⟨*term*⟩ ⟨*term*⟩)
|:equality (not(= ⟨*term*⟩ ⟨*term*⟩))
|:negative-preconditions (not ⟨*atomic formula (term)*⟩)
|:disjunctive-preconditions (not ⟨*GD*⟩)
|:disjunctive-preconditions (or ⟨*GD*⟩*)
|:disjunctive-preconditions (imply ⟨*GD*⟩ ⟨*GD*⟩)
|:existential-preconditions (exists (⟨*typed list (variable)*⟩)
⟨*GD*⟩)
|:universal-preconditions (forall (⟨*typed list (variable)*⟩)
⟨*GD*⟩)
|:fluents ⟨*f-comp*⟩
⟨*atomic formula (*x*)*⟩ ::=(⟨*predicate*⟩ ⟨x⟩*) | ⟨*predicate*⟩
⟨*term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩
⟨*f-comp*⟩ ::= (⟨*binary-comp*⟩ ⟨*f-exp*⟩ ⟨*f-exp*⟩)

⟨*binary-comp*⟩ ::= `<` | `<=` | `=` | `>=` | `>`
⟨*f-exp*⟩ ::= ⟨*number*⟩ | ⟨*f-head (term)*⟩
| `(` ⟨*binary-op*⟩ ⟨*f-exp*⟩ ⟨*f-exp*⟩ `)` | `(` `-` ⟨*f-exp*⟩ `)`
⟨*f-head (*x*)*⟩ ::= `(` ⟨*function symbol*⟩ ⟨x⟩`*` `)` | ⟨*function symbol*⟩
⟨*binary-op*⟩ ::= `+` | `-` | `*` | `/`

A ⟨*number*⟩ is a sequence of numeric characters, possibly with a single decimal point ("`.`") at any position in the sequence.

The syntax for effects has been extended to allow for probabilistic effects, which can be arbitrarily interleaved with conditional effects and universal quantification.

⟨*effect*⟩ ::= ⟨*p-effect*⟩ | `(` `and` ⟨*effect*⟩`*` `)`
|`:conditional-effects` `(` `forall` `(` ⟨*typed list (variable)*⟩ `)` ⟨*effect*⟩ `)`
|`:conditional-effects` `(` `when` ⟨*GD*⟩ ⟨*effect*⟩ `)`
|`:probabilistic-effects` `(` `probabilistic` ⟨*prob-effect*⟩$^+$ `)`
⟨*p-effect*⟩ ::= ⟨*atomic formula (term)*⟩ | `(` `not` ⟨*atomic formula (term)*⟩ `)`
|`:fluents` `(` ⟨*assign-op*⟩ ⟨*f-head (term)*⟩ ⟨*f-exp*⟩ `)`
|`:rewards` `(` ⟨*additive-op*⟩ ⟨*reward fluent*⟩ ⟨*f-exp*⟩ `)`
⟨*prob-effect*⟩ ::= ⟨*probability*⟩ ⟨*effect*⟩
⟨*assign-op*⟩ ::= `assign` | `scale-up` | `scale-down` | ⟨*additive-op*⟩
⟨*additive-op*⟩ ::= `increase` | `decrease`
⟨*reward fluent*⟩ ::= `(reward )` | `reward`

A ⟨*probability*⟩ is a ⟨*number*⟩ with a value in the interval $[0, 1]$.

A.3 Problems

The syntax for problem definitions has been extended to allow for the specification of a probability distribution over initial states, and also to permit the association of a one-time reward with entering a goal state. It is otherwise identical to the syntax for PDDL2.1 problem definitions.

⟨*problem*⟩ ::= `(` `define` `(` `problem` ⟨*name*⟩ `)`
`(` `:domain` ⟨*name*⟩ `)`
[⟨*require-def*⟩]
[⟨*objects-def*⟩]
[⟨*init*⟩]
⟨*goal*⟩ `)`
⟨*objects-def*⟩ ::= `(` `:objects` ⟨*typed list (name)*⟩ `)`
⟨*init*⟩ ::= `(` `:init` ⟨*init-el*⟩`*` `)`
⟨*init-el*⟩ ::= ⟨*p-init-el*⟩
|`:probabilistic-effects` `(` `probabilistic` ⟨*prob-init-el*⟩`*` `)`
⟨*p-init-el*⟩ ::= ⟨*atomic formula (name)*⟩ |`:fluents` `(` `=` ⟨*f-head (name)*⟩ ⟨*number*⟩ `)`
⟨*prob-init-el*⟩ ::= ⟨*probability*⟩ ⟨*a-init-el*⟩
⟨*a-init-el*⟩ ::= ⟨*p-init-el*⟩ | `(` `and` ⟨*p-init-el*⟩`*` `)`
⟨*goal*⟩ ::= ⟨*goal-spec*⟩ [⟨*metric-spec*⟩] | ⟨*metric-spec*⟩

⟨*goal-spec*⟩ ::= `( :goal` ⟨*GD*⟩ `)` [`( :goal-reward` ⟨*ground-f-exp*⟩ `)`]:rewards
⟨*metric-spec*⟩ ::= `( :metric` ⟨*optimization*⟩ ⟨*ground-f-exp*⟩ `)`
⟨*optimization*⟩ ::= `minimize` | `maximize`
⟨*ground-f-exp*⟩ ::= ⟨*number*⟩ | ⟨*f-head (name)*⟩
| `(` ⟨*binary-op*⟩ ⟨*ground-f-exp*⟩ ⟨*ground-f-exp*⟩ `)` | `( -` ⟨*ground-f-exp*⟩ `)`
|:probabilistic-effects `goal-probability`
|:rewards ⟨*reward fluent*⟩

A.4 Requirements

Below is a table of all requirements in PPDDL1.0. Some requirements imply others; some are abbreviations for common sets of requirements. If a domain stipulates no requirements, it is assumed to declare a requirement for `:strips`.

Requirement	*Description*
`:strips`	Basic STRIPS-style adds and deletes
`:typing`	Allow type names in declarations of variables
`:equality`	Support = as built-in predicate
`:negative-preconditions`	Allow negated atoms in goal descriptions
`:disjunctive-preconditions`	Allow disjunctive goal descriptions
`:existential-preconditions`	Allow `exists` in goal descriptions
`:universal-preconditions`	Allow `forall` in goal descriptions
`:quantified-preconditions`	= `:existential-preconditions` + `:universal-preconditions`
`:conditional-effects`	Allow `when` and `forall` in action effects
`:probabilistic-effects`	Allow `probabilistic` in action effects
`:rewards`	Allow reward fluent in action effects and optimization metric
`:fluents`	Allow numeric state variables
`:adl`	= `:strips` + `:typing` + `:equality` + `:negative-preconditions` + `:disjunctive-preconditions` + `:quantified-preconditions` + `:conditional-effects`
`:mdp`	= `:probabilistic-effects` + `:rewards`

References

Dean, Thomas and Keiji Kanazawa. 1989. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150.

Dearden, Richard and Craig Boutilier. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1–2):219–283.

Fox, Maria and Derek Long. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

Ghallab, Malik, Adele E. Howe, Craig A. Knoblock, Drew McDermott, Ashwin Ram, Manuela M. Veloso, Daniel S. Weld, and David Wilkins. 1998. PDDL—the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CT.

Kushmerick, Nicholas, Steve Hanks, and Daniel S. Weld. 1995. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1–2):239–286.

Littman, Michael L. 1997. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 748–754, Providence, RI. American Association for Artificial Intelligence, AAAI Press.

Rintanen, Jussi. 2003. Expressive equivalence of formalism for planning with sensing. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, eds., *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, pages 185–194, Trento, Italy. AAAI Press.