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

Håkan L. S. Younes **Michael L. Littman**[1]

October 2004
CMU-CS-04-167

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We desribe a variation of the planning domain definition language, PDDL, that permits the modeling of probabilistic planning problems with rewards. This language, PPDDL1.0, was used as the input language for the probabilistic track of the 4th International Planning Competition. We provide the complete syntax for PPDDL1.0 and give a semantics of PPDDL1.0 planning problems in terms of Markov decision processes.

[1]Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA

Keywords: PDDL, probabilistic planning, Markov decision processes

1 Introduction

A standard domain description language, PDDL (Ghallab et al. 1998; McDermott 2000; Fox and Long 2003), for deterministic planning domains has simplified sharing of domain models and problems in the planning community, and has enabled direct comparisons of different planning systems. As a result, there has been considerable progress in planning research with deterministic domain models since the first International Planning Competition in 1998.

The 4th International Planning Competition included a probabilistic track for the first time in an attempt to create a common platform for the evaluation of probabilistic and decision theoretic planning systems. This document briefly describes the input language, PPDDL1.0, that was used for the probabilistic track. PPDDL1.0 is essentially a syntactic extension of levels 1 and 2 of PDDL2.1 (Fox and Long 2003). Section 2 gives a comprehensive introduction to PDDL so that prior knowledge of PDDL2.1, or any of its predecessors, is not required in order to understand this document. Readers already familiar with PDDL can jump immediately to Section 3, which focuses on new language features specific to PPDDL, including probabilistic effects and rewards. Section 4 provides a semantics for PPDDL1.0 planning problems in terms of *Markov decision processes* (MDPs; Howard 1960; Puterman 1994). The complete syntax for PPDDL1.0 is given in Appendix A.

Note that, unlike PDDL2.1, PPDDL1.0 does not impose a specific structure on plans, except that only a single action can be executed at any point in time. The problem of plan representation has been left entirely to the planning systems, and planning systems may even choose to have no plan representation at all. At the planning competition, planning systems were evaluated using a client-server model where the server sent states to a client (planning system) and the client was expected to respond with an action to execute for each state it received.

PPDDL1.0 is a first step towards a general language for describing probabilistic and decision theoretic planning problems. Directions for future extensions include, but are not limited to, concurrency, observability, and time. Of these, concurrency is the most straightforward to add, requiring only a slightly modified semantics in line with that proposed by Mausam and Weld (2004). Adding a temporal dimension to actions would permit the modeling of semi-Markov decision processes (Howard 1971), or in the presence of concurrency, generalized semi-Markov decision processes (Younes and Simmons 2004). Restrictions in the observability of state features would be necessary to model contingent probabilistic planning problems (Draper et al. 1994), as well as partially observable MDPs (Åström 1965).

2 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.

2.1 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 2.2). Figure 1 shows the definition of a domain named "test-domain" in

```
(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.

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.

2.1.1 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`.

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

2.1.2 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
$\textit{fuel-level}_{\text{goldie}}$	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)` are 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.

2.1.3 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.

[1]The original version of PDDL (Ghallab et al. 1998) allowed unions of union types. Later versions, including PDDL2.1 (Fox and 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.

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 `(`$\langle$*assign-op*$\rangle$ x $\langle$*f-exp*$\rangle$`)`, where $\langle$*assign-op*$\rangle$ is one of `assign`, `scale-up`, `scale-down`, `increase`, or `decrease`, and $\langle$*f-exp*$\rangle$ 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 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. It is considered an error to apply an action in a state where the precondition does not hold (cf. Fox and Long 2003). If we want the $\mathsf{refuel}_{\text{goldie}}$ actions to be universally applicable, but with no state change occurring if $\mathit{fuel\text{-}level}_{\text{goldie}} < 10$ does not hold, then we should use conditional effects:

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

2.2 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 $O \cup C$. The process of obtaining all state variables and actions for a planning problem through the exhaustive application of predicates, functions, and action schemata to objects is referred to as *grounding*.

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 $\mathit{holding}_{\text{cereal-box}}$, 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`.

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
$\textit{fuel-level}_{\text{goldie}}$	numeric	7

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

3 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.

3.1 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. We do, however, allow a probability-effect pair to be left out if the effect is empty. In other words, the effect `(probabilistic` p_1 e_1 $\ldots$ p_l e_l`)` with $\sum_{i=1}^{l} p_i < 1$ is syntactic sugar for `(probabilistic` p_1 e_1 $\ldots$ p_l e_l q `(and))` with $q = 1 - \sum_{i=1}^{l} p_i$. 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). The requirements flag `:probabilistic-effects` signals that probabilistic effects are

```
(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.

used in the domain definition. 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 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 for any given execution. 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 and 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). Any PPDDL action can, however, be translated into a *set* of PSOs with at most a polynomial increase in the size of the representation. Consequently, it follows from the results of Littman (1997) that PPDDL, *after grounding* (i.e. full instantiation of action schemata), is representationally equivalent to dynamic Bayesian networks (Dean and Kanazawa 1989), which is another popular representation for MDP planning problems.

Still, it is worth noting that a single PPDDL action *schema* can represent a large number of actions and a single predicate can represent a large number of state variables, meaning that PPDDL often can represent planning problems more succinctly than other representations. For example, the number of actions that can be represented using m objects and n action schemata with arity c is $m \cdot n^c$, which is not bounded by any polynomial in the size of the original representation $(m + n)$. Grounding is by no means a prerequisite for PPDDL planning, so planners could conceivably take advantage of the more compact representation by working directly with action schemata.

3.2 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 and Boutilier (1997). A reward of 0.8 is awarded if the user has coffee after the "buy-coffee" action has been executed, and a reward of 0.2 is awarded if $is\text{-}wet$ is false after execution of "buy-coffee". Note that a total reward of 1.0 can be awarded as a result of executing the "buy-coffee" action if execution of the action leads to a state where both $user\text{-}has\text{-}coffee$ and $\neg is\text{-}wet$ hold.

3.3 Plan Objectives

Regular PDDL goals are 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-achieved` as a special optimization metric, which can be used to explicitly specify that the plan objective is to maximize (or minimize) the probability of goal achievement. The value of the `goal-achieved` fluent is 1 if a goal state has been visited during execution, and remains 0 so long as a goal state has not been visited. The expected value of `goal-achieved` is therefore equal to the probability of goal achievement.

```
(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.

4 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_G over V characterizing a set of goal states $G = \{s \mid s \models \phi_G\}$, 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 $r_G = 1$.

4.1 Probability and Reward Structure

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 $s \models \neg\phi_G \wedge \phi_a$. It is an error to apply a to a state such that $s \not\models \neg\phi_G \wedge \phi_a$. Goal states are absorbing, so no action may be applied to a state satisfying ϕ_G. The requirement that ϕ_a must hold in order for a to be applicable is consistent with the semantics of PDDL2.1 (Fox and 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. $e_1 \wedge \cdots \wedge e_n$ 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. $p_1e_1|\dots|p_ne_n$ is an effect if $e_1, \dots, e_n$ are effects, $p_i \geq 0$ for all $i \in \{1, \dots, n\}$, and $\sum_{i=1}^{n} p_i = 1$.

Items 2 through 4 are referred to as *simple effects*. 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 of the form $r \uparrow f$ are used to associate rewards with transitions as described below.

An action $a = \langle \phi_a, e_a \rangle$ defines a transition probability matrix P_a and a state reward vector R_a, with $P_a(i,j)$ being the probability of transitioning to state j when applying a in state i, and $R_a(i)$ being the *expected* reward for executing action a in state i. We can compute P_a and R_a by first translating e_a into an effect of the form $p_1e_1|\dots|p_ne_n$, where each e_i is a deterministic effect. The purpose of this translation is to bring nondeterminism to the top so that we can compute P_a as $\sum_{i=1}^{n} p_iT_i$, where T_i is a 0-1 transition matrix for e_i. Rintanen (2003) calls this form Unary Nondeterminism Normal Form (1ND) and shows that any effect e can be translated into this form by using the following equivalences:

$$
\begin{aligned}
e \equiv& 1e \\
e \wedge (p_1e_1|\dots|p_ke_n) \equiv& p_1(e \wedge e_1)|\dots|p_n(e \wedge e_n) \\
c \rhd (p_1e_1|\dots|p_ne_n) \equiv& p_1(c \rhd e_1)|\dots|p_n(c \rhd e_n) \\
p_1(p'_1e'_1|\dots|p'_ke'_k)|p_2e_2|\dots|p_ne_n \equiv& (p_1p'_1)e'_1|\dots|(p_1p'_k)e'_k|p_2e_2|\dots|p_ne_n
\end{aligned}
$$

The translation into 1ND can result in an exponential increase in the size of the effect formula, although this representational explosion can be avoided by splitting a single action into multiple actions that are forced to be executed in sequence. The number of actions that would have to be added is at most linear in the length of the effect formula.

We further rewrite the effect of an action by translating each e_i into an effect of the form $(c_{i1} \rhd e_{i1}) \wedge \dots \wedge (c_{in_i} \rhd e_{in_i})$, 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:

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

This rewrite can also result in an exponential increase in the size of the effect formula, but it will enable us to succinctly define the transition matrix T_i for each deterministic effect e_i.

An effect of 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, which intuitively means that the constituent parts of e can be evaluated in arbitrary order without resulting in a different successor state. 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. That is, numeric effects are commutative if they are insensitive to ordering.

Under these assumptions, we define a function τ that returns a successor state given a state and an effect formula. The definition of τ takes two states: the current state and an intermediate successor state. State changes are accumulated in the second parameter, while the current state is kept unmodified in the first parameter so that real-valued functions can be evaluated in this state. The definition of τ is as follows:

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

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 with effect $p_1 e_1 | \ldots | p_n e_n$, where $T_i = \sum_{j=1}^{n_i} T_{ij}$. To capture the notion of failed preconditions, we introduce an error state $s_\bot$. We set the entry at column $s_\bot$ to 1 for each row s such that $s \not\models \phi_a$ and the remaining entries of these rows are set to zero. For rows s such that $s \models \phi_a$, the entry at column $s_\bot$ is set to zero. Finally, we need to make all states that satisfy the goal condition ϕ_G of the problem absorbing. This is accomplished by modifying P_a: for each s such that $s \models \phi_G$, 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:

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

The effect $c_{ij} \rhd e_{ij}$ associates reward $r(s, e_{ij})$ with each transition $\langle s, s'\rangle \in T(c_{ij} \rhd e_{ij})$, and an additional reward r_G if s' is a goal state (i.e. $s' \models \phi_G$), or no reward if s is a goal state. 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 $r(s, e_{ij})$ if $\langle s, s'\rangle \in T(c_{ij} \rhd e_{ij})$ and $s \not\models \phi_G$ and $s' \not\models \phi_G$, $r(s, e_{ij}) + r_G$ if $\langle s, s'\rangle \in T(c_{ij} \rhd e_{ij})$ and $s \not\models \phi_G$ and $s' \models \phi_G$, and 0 otherwise. We sum over all $c_{ij} \rhd e_{ij}$ to get a transition reward matrix for e_i: $R_i = \sum_{j=1}^{n_i} R_{ij}$. To obtain the expected transition reward for an action with effect $p_1 e_1 | \ldots | p_n e_n$, we compute the matrix $R = \sum_{i=1}^{n} p_i R_i$. The elements of the state reward vector R_a can then be computed as follows: $R_a(i) = \sum_{j=1}^{|S|} R(i, j)$.

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 $bomb\text{-}in\text{-}package_{\text{package1}}$, 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 $\textsf{dunk-package}_{\text{package1}}$ 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 $\textsf{dunk-package}_{\text{package1}}$:

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

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 state reward vector:

$$R_a = (0, 0, 0, 0, 0, 0, 0, 0, \frac{19}{20}, 0, 0, 0, \frac{19}{20}, 0, 0, 0)$$

4.2 Optimality Criteria

We have shown how to construct an MDP from the PPDDL encoding of a planning problem. The plan objective is to maximizes the expected reward for the MDP. This can be interpreted as expected *discounted* reward or expected *total* reward depending on the situation. For process oriented planning problems (e.g. the "Coffee Delivery" problem), the former is typically what we want, while the latter often is the interpretation

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

chosen for goal oriented problems (e.g. the "Bomb and Toilet" problem). PPDDL can be used to encode both kind of planning problems, but does not include any facility for enforcing a specific interpretation.

For the probabilistic track of the 4th International Planning Competition, we used expected total reward as the optimality criterion. This requires some care in the design of planning problems in order to ensure that the expected total reward is bounded for the optimal policy. The following restrictions were made for problems used at the planning competition:

1. Each problem had a goal statement, identifying a set of absorbing goal states.
2. A positive reward was associated with transitioning into a goal state.
3. A negative reward (cost) was associated with each action.
4. A "done" action was available in all states, which could be used to end further accumulation of reward.

These conditions ensure that an MDP model of a planning problem is a *positive bounded model* Puterman (1994, pp. 284). The only positive reward is for transitioning into a goal state. Since goal states are absorbing (i.e. they have no outgoing transitions), the maximum value for any state is bounded by the goal reward. Furthermore, the "done" action ensures that there is an action available in each state which guarantees a non-negative future reward.

5 Conversion to Dynamic Bayesian Network

In this section, we focus on PPDDL problems without numeric state variables as such problems are guaranteed to have a finite state space. For a problem with n Boolean state variables, the transition probability matrix for each action may contain up to 2^{2n} entries. We showed how to compute these entries in the previous section, but the computation relied on transformations of action effects that could result in an exponential increase in the representation size compared to the original PPDDL encoding. We will now show how to compute a *factored* representation of the transition probability matrix for a PPDDL action without more than a polynomial increase in size. The transition probability matrix will be represented using a *dynamic Bayesian network* (DBN; Dean and Kanazawa 1989), whose structure can be exploited by algorithms for decision theoretic planning (see, e.g., work by Boutilier et al. 1995; Hoey et al. 1999; Boutilier et al. 1999; Guestrin et al. 2003).

A Bayesian network is a directed graph. Each node of the graph represents a state variable, and a directed edge from one node to another represents a causal dependence. With each node is associated a conditional probability table (CPT). The CPT for state variable X's node represents the probability distribution over possible values for X conditioned on the values of state variables whose nodes are parents of X's node. A Bayesian network is a factored representation of the joint probability distribution over the variables represented in the network.

A DBN is a Bayesian network with a specific structure aimed at capturing temporal dependence. For each state variable X, we create a duplicate state variable X', with X representing the situation at the present time and X' representing the situation one time step into the future. A directed edge from a present-time state variable X to a future-time state variable Y' encodes a temporal dependence. There are no edges between two present-time state variables, or from a future-time to a present-time state variable (the present does not depend on the future). We can, however, have an edge between two future-time state variables. Such edges, called *synchronic* edges, are used to represent correlated effects. A DBN is a factored representation of the joint probability distribution over present-time and future-time state variables, which is also the transition probability matrix for a discrete-time Markov process.

We now show how to generate a DBN representing the transition probability matrix for a PPDDL action. To avoid representational blowup, we introduce a multi-valued auxiliary variable for each probabilistic effect of an action effect. These auxiliary variables are used to indicate which of the possible outcomes of a probabilistic effect that occurs, and this allows us to correlate all the effects of a specific outcome. The auxiliary variable associated with a probabilistic effect with n outcomes can take on n different values. A PPDDL effect e of size $|e|$ can consist of at most $O(|e|)$ distinct probabilistic effects. Hence, the number of auxiliary variables required to encode the transition probability matrix for an action with effect e will be at most $O(|e|)$. Only future-time versions of the auxiliary variables are necessary. For a PPDDL problem with m Boolean state variables, we need on the order of $2m + \max_{a \in A} |e_a|$ nodes in the DBNs representing transition probability matrices for actions.

We provide a compositional approach for generating a DBN that represents the transition probability matrix for a PPDDL action with precondition ϕ_a and effect e_a. We assume that the effect is consistent, i.e. that b and $\neg b$ do not occur in the same outcome with overlapping conditions. The DBN for an empty effect $\top$ simply consists of $2m$ nodes, with each present-time node X connected to its future-time counterpart X'. The CPT for X' has the non-zero entries $\Pr[X' = \top \mid X = \top] = 1$ and $\Pr[X' = \bot \mid X = \bot] = 1$. The same holds for a reward effect $r \uparrow k$, which does not change the value of state variables.

Next, consider the simple effects b and $\neg b$. Let X_b be the state variable associated with the PPDDL atom b. For these effects, we eliminate the edge from X_b to X'_b. The CPT for X'_b has the entry $\Pr[X'_b = \top] = 1$ for effect b and $\Pr[X'_b = \bot] = 1$ for effect $\neg b$.

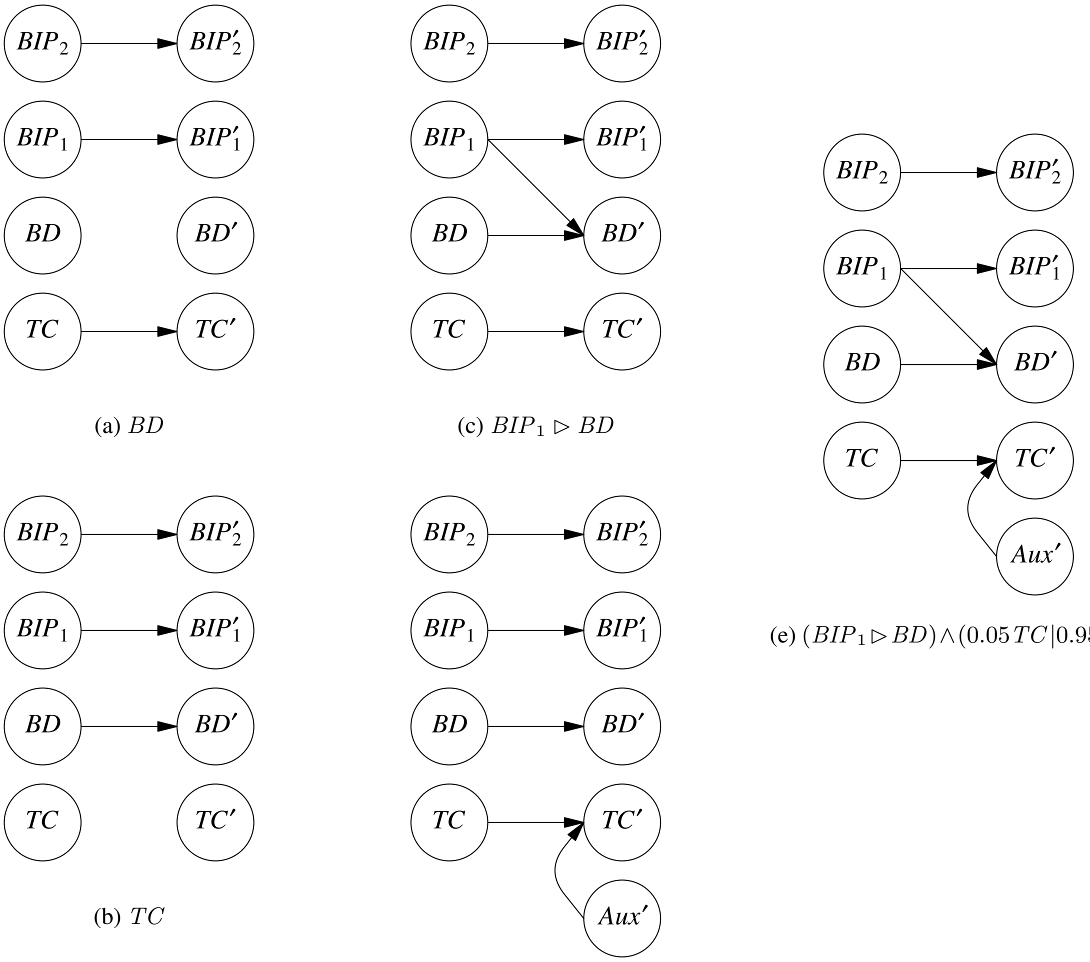
For conditional effects, $c \rhd e$, we take the DBN for e and add edges between the present-time state variables mentioned in the formula c and the future-time state variables in the DBN for e. Entries in the CPT for a state variable X' that correspond to settings of the present-time state variables that satisfy c remain unchanged. The other entries are set to 1 if X is true and 0 otherwise (the value of X does not change if the effect condition is not satisfied).

The DBN for an effect conjunction $e_1 \wedge \cdots \wedge e_n$ is constructed from the DBNs for the n effect conjuncts. The value for $\Pr[X' = \top \mid \mathbf{X}]$ in the DBN for the conjunction is set to the maximum of $\Pr[X' = \top \mid \mathbf{X}]$ over the DBNs for the conjuncts.

Finally, to construct a DBN for a probabilistic effect $p_1 e_1 | \ldots | p_n e_n$, we introduce an auxiliary variable Y' that is used to indicate which one of the n outcomes occurred. The node for Y' does not have any parents, and the entries of the CPT are $\Pr[Y' = i] = p_i$. Given a DBN for e_i, we add a synchronic edge from Y' to all state variables X. The value of $\Pr[X' = \top \mid \mathbf{X}, Y' = j]$ is set to $\Pr[X' = \top \mid \mathbf{X}]$ if $j = i$ and 0 otherwise. We do this for all n outcomes, which results in n DBNs. These DBNs are combined in the same way as for conjunctive effects, and the result is the DBN for the probabilistic effect.

The process of constructing a DBN from a PPDDL encoding of an action is illustrated in Figure 5 for the "Bomb and Toilet" example. Note that the CPTs for state variables only contain 0 and 1 entries. The probabilities of different outcomes are encoded in the CPTs for auxiliary variables.

The effects in the "Bomb and Toilet" example are fairly simple. Figure 6(a) shows the PPDDL encoding for an action in the "Coffee Delivery" domain. The effect of this action has both nested probabilistic effects and correlated effects (the first outcome of the first probabilistic effect is an example of the latter). The structure of the DBN for a this more complex example is shown in Figure 6(b). There are three auxiliary variables because the action effect contains three probabilistic effects. The node labeled HC' (the future-time version of the state variable *has-coffee*) has five parents, including all three auxiliary variables. Consequently, the CPT for this node will have $2^5 = 32$ rows.



(a) BD

(b) TC

(c) $BIP_1 \rhd BD$

(d) $0.05\,TC|0.95\top$

(e) $(BIP_1 \rhd BD) \wedge (0.05\,TC|0.95\top)$

BIP_1	BIP'_1: $\top$	BIP'_1: $\bot$
$\top$	1	0
$\bot$	0	1

BIP_2	BIP'_2: $\top$	BIP'_2: $\bot$
$\top$	1	0
$\bot$	0	1

BIP_1	BD	BD': $\top$	BD': $\bot$
$\top$	$\top$	1	0
$\top$	$\bot$	1	0
$\bot$	$\top$	1	0
$\bot$	$\bot$	0	1

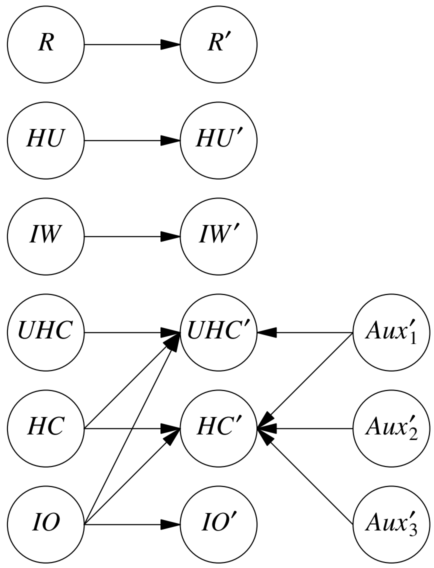
TC	Aux'	TC': $\top$	TC': $\bot$
$\top$	1	1	0
$\top$	2	1	0
$\bot$	1	1	0
$\bot$	2	0	1

Aux': 1	Aux': 2
0.05	0.95

Figure 5: Construction of a DBN for the dunk-package$_{\text{package1}}$ action of the bomb and toiled example. The final DBN is shown in (e) and its CPTs are listed at the bottom of the figure.

```
(:action deliver-coffee
        :effect (and (when (and (in-office) (has-coffee))
                        (probabilistic 0.8 (and (user-has-coffee)
                                               (not (has-coffee))
                                               (increase (reward) 0.8))
                                       0.2 (and (probabilistic 0.5 (not (has-coffee)))
                                               (when (user-has-coffee)
                                                  (increase (reward) 0.8)))))
                 (when (and (not (in-office)) (has-coffee))
                    (and (probabilistic 0.8 (not (has-coffee)))
                         (when (user-has-coffee)
                            (increase (reward) 0.8))))
                 (when (and (not (has-coffee)) (user-has-coffee))
                    (increase (reward) 0.8))
                 (when (not (is-wet))
                    (increase (reward) 0.2))))
```

(a)



(b)

Figure 6: PPDDL encoding (a) and DBN (b) for the “deliver-coffee” action of the “Coffee Delivery” domain.

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*⟩]$^{\texttt{:typing}}$ 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 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*⟩]$^{\texttt{:typing}}$
		[⟨*constants-def*⟩]
		[⟨*predicates-def*⟩]
		[⟨*functions-def*⟩]$^{\texttt{:fluents}}$
		⟨*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*⟩* $\mid^{\texttt{:typing}}$ ⟨*x*⟩$^{+}$ `-` ⟨*type*⟩ ⟨*typed list* (*x*)⟩
⟨*type*⟩	::=	`( either` ⟨*primitive type*⟩$^{+}$ `)` $\mid$ ⟨*primitive type*⟩
⟨*primitive type*⟩	::=	⟨*name*⟩
⟨*function typed list* (*x*)⟩	::=	⟨*x*⟩* $\mid^{\texttt{:typing}}$ ⟨*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*)⟩ $\mid$ `( and` ⟨*GD*⟩* `)` $\mid^{\texttt{:equality}}$ `( =` ⟨*term*⟩ ⟨*term*⟩ `)` $\mid^{\texttt{:equality}}$ `( not ( =` ⟨*term*⟩ ⟨*term*⟩ `) )` $\mid^{\texttt{:negative-preconditions}}$ `( not` ⟨*atomic formula* (*term*)⟩ `)` $\mid^{\texttt{:disjunctive-preconditions}}$ `( not` ⟨*GD*⟩ `)` $\mid^{\texttt{:disjunctive-preconditions}}$ `( or` ⟨*GD*⟩* `)` $\mid^{\texttt{:disjunctive-preconditions}}$ `( imply` ⟨*GD*⟩ ⟨*GD*⟩ `)` $\mid^{\texttt{:existential-preconditions}}$ `( exists (` ⟨*typed list* (*variable*)⟩ `)` ⟨*GD*⟩ `)` $\mid^{\texttt{:universal-preconditions}}$ `( forall (` ⟨*typed list* (*variable*)⟩ `)` ⟨*GD*⟩ `)` $\mid^{\texttt{:fluents}}$ ⟨*f-comp*⟩
⟨*atomic formula* (*x*)⟩	::=	`(` ⟨*predicate*⟩ ⟨*x*⟩* `)` $\mid$ ⟨*predicate*⟩
⟨*term*⟩	::=	⟨*name*⟩ $\mid$ ⟨*variable*⟩
⟨*f-comp*⟩	::=	`(` ⟨*binary-comp*⟩ ⟨*f-exp*⟩ ⟨*f-exp*⟩ `)`
⟨*binary-comp*⟩	::=	$<$ $\mid$ $<=$ $\mid$ $=$ $\mid$ $>=$ $\mid$ $>$
⟨*f-exp*⟩	::=	⟨*number*⟩ $\mid$ ⟨*f-head* (*term*)⟩ $\mid$ `(` ⟨*binary-op*⟩ ⟨*f-exp*⟩ ⟨*f-exp*⟩ `)` $\mid$ `( -` ⟨*f-exp*⟩ `)`
⟨*f-head* (*x*)⟩	::=	`(` ⟨*function symbol*⟩ ⟨*x*⟩* `)` $\mid$ ⟨*function symbol*⟩
⟨*binary-op*⟩	::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/`

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

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

$$
\begin{array}{lll}
\langle \textit{effect} \rangle & ::= & \langle \textit{p-effect} \rangle \mid \texttt{(and } \langle \textit{effect} \rangle^* \texttt{)} \\
 & & \mid^{\texttt{:conditional-effects}} \texttt{(forall (} \langle \textit{typed list } (\textit{variable}) \rangle \texttt{) } \langle \textit{effect} \rangle \texttt{)} \\
 & & \mid^{\texttt{:conditional-effects}} \texttt{(when } \langle \textit{GD} \rangle\ \langle \textit{effect} \rangle \texttt{)} \\
 & & \mid^{\texttt{:probabilistic-effects}} \texttt{(probabilistic } \langle \textit{prob-effect} \rangle^{+} \texttt{)} \\
\langle \textit{p-effect} \rangle & ::= & \langle \textit{atomic formula (term)} \rangle \mid \texttt{(not } \langle \textit{atomic formula } (\textit{term}) \rangle \texttt{)} \\
 & & \mid^{\texttt{:fluents}} \texttt{(} \langle \textit{assign-op} \rangle\ \langle \textit{f-head } (\textit{term}) \rangle\ \langle \textit{f-exp} \rangle \texttt{)} \\
 & & \mid^{\texttt{:rewards}} \texttt{(} \langle \textit{additive-op} \rangle\ \langle \textit{reward fluent} \rangle\ \langle \textit{f-exp} \rangle \texttt{)} \\
\langle \textit{prob-effect} \rangle & ::= & \langle \textit{probability} \rangle\ \langle \textit{effect} \rangle \\
\langle \textit{assign-op} \rangle & ::= & \texttt{assign} \mid \texttt{scale-up} \mid \texttt{scale-down} \mid \langle \textit{additive-op} \rangle \\
\langle \textit{additive-op} \rangle & ::= & \texttt{increase} \mid \texttt{decrease} \\
\langle \textit{reward fluent} \rangle & ::= & \texttt{(reward)} \mid \texttt{reward}
\end{array}
$$

A $\langle \textit{probability} \rangle$ is a $\langle \textit{number} \rangle$ 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.

$$
\begin{array}{lll}
\langle \textit{problem} \rangle & ::= & \texttt{(define (problem } \langle \textit{name} \rangle \texttt{)} \\
 & & \qquad \texttt{(:domain } \langle \textit{name} \rangle \texttt{)} \\
 & & \qquad [\langle \textit{require-def} \rangle] \\
 & & \qquad [\langle \textit{objects-def} \rangle] \\
 & & \qquad [\langle \textit{init} \rangle] \\
 & & \qquad \langle \textit{goal} \rangle \texttt{)} \\
\langle \textit{objects-def} \rangle & ::= & \texttt{(:objects } \langle \textit{typed list } (\textit{name}) \rangle \texttt{)} \\
\langle \textit{init} \rangle & ::= & \texttt{(:init } \langle \textit{init-el} \rangle^* \texttt{)} \\
\langle \textit{init-el} \rangle & ::= & \langle \textit{p-init-el} \rangle \\
 & & \mid^{\texttt{:probabilistic-effects}} \texttt{(probabilistic } \langle \textit{prob-init-el} \rangle^* \texttt{)} \\
\langle \textit{p-init-el} \rangle & ::= & \langle \textit{atomic formula } (\textit{name}) \rangle \mid^{\texttt{:fluents}} \texttt{(= } \langle \textit{f-head } (\textit{name}) \rangle\ \langle \textit{number} \rangle \texttt{)} \\
\langle \textit{prob-init-el} \rangle & ::= & \langle \textit{probability} \rangle\ \langle \textit{a-init-el} \rangle \\
\langle \textit{a-init-el} \rangle & ::= & \langle \textit{p-init-el} \rangle \mid \texttt{(and } \langle \textit{p-init-el} \rangle^* \texttt{)} \\
\langle \textit{goal} \rangle & ::= & \langle \textit{goal-spec} \rangle\ [\langle \textit{metric-spec} \rangle] \mid \langle \textit{metric-spec} \rangle \\
\langle \textit{goal-spec} \rangle & ::= & \texttt{(:goal } \langle \textit{GD} \rangle \texttt{) } [\texttt{(:goal-reward } \langle \textit{ground-f-exp} \rangle \texttt{)}]^{\texttt{:rewards}} \\
\langle \textit{metric-spec} \rangle & ::= & \texttt{(:metric } \langle \textit{optimization} \rangle\ \langle \textit{ground-f-exp} \rangle \texttt{)} \\
\langle \textit{optimization} \rangle & ::= & \texttt{minimize} \mid \texttt{maximize} \\
\langle \textit{ground-f-exp} \rangle & ::= & \langle \textit{number} \rangle \mid \langle \textit{f-head } (\textit{name}) \rangle \\
 & & \mid \texttt{(} \langle \textit{binary-op} \rangle\ \langle \textit{ground-f-exp} \rangle\ \langle \textit{ground-f-exp} \rangle \texttt{)} \\
 & & \mid \texttt{(- } \langle \textit{ground-f-exp} \rangle \texttt{)} \\
 & & \mid \texttt{(total-time)} \mid \texttt{total-time} \\
 & & \mid \texttt{(goal-achieved)} \mid \texttt{goal-achieved} \\
 & & \mid^{\texttt{:rewards}} \langle \textit{reward fluent} \rangle
\end{array}
$$

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

Åström, K. J. 1965. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications* 10, no. 1: 174–205.

Boutilier, Craig, Thomas Dean, and Steve Hanks. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11: 1–94.

Boutilier, Craig, Richard Dearden, and Moisés Goldszmidt. 1995. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, edited by Chris S. Mellish, 1104–1111, Montreal, Canada. Morgan Kaufmann Publishers.

Dean, Thomas and Keiji Kanazawa. 1989. A model for reasoning about persistence and causation. *Computational Intelligence* 5, no. 3: 142–150.

Dearden, Richard and Craig Boutilier. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89, no. 1–2: 219–283.

Draper, Denise, Steve Hanks, and Daniel S. Weld. 1994. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, edited by Kristian Hammond, 31–36, Chicago, IL. AAAI Press.

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.

Guestrin, Carlos, Daphne Koller, Ronald Parr, and Shobha Venkataraman. 2003. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research* 19: 399–468.

Hoey, Jesse, Robert St-Aubin, Alan Hu, and Craig Boutilier. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, edited by Kathryn B. Laskey and Henri Prade, 279–288, Stockholm, Sweden. Morgan Kaufmann Publishers.

Howard, Ronald A. 1960. *Dynamic Programming and Markov Processes*. New York, NY: John Wiley & Sons.

———. 1971. *Dynamic Probabilistic Systems*, vol. II: Semi-Markov and Decision Processes. New York, NY: John Wiley & Sons.

Kushmerick, Nicholas, Steve Hanks, and Daniel S. Weld. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76, no. 1–2: 239–286.

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

Mausam and Daniel S. Weld. 2004. Solving concurrent Markov decision processes. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, 716–722, San Jose, CA. AAAI Press.

McDermott, Drew. 2000. The 1998 AI planning systems competition. *AI Magazine* 21, no. 2: 35–55.

Puterman, Martin L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York, NY: John Wiley & Sons.

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

Younes, Håkan L. S. and Reid G. Simmons. 2004. Solving generalized semi-Markov decision processes using continuous phase-type distributions. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, 742–747, San Jose, CA. AAAI Press.