Relational Dynamic Influence Diagram Language (RDDL): Language Description

Scott Sanner (ssanner@gmail.com)
NICTA and the Australian National University

Abstract

The Relational Dynamic Influence Diagram Language (RDDL) is a uniform language where states, actions, and observations (whether discrete or continuous) are parameterized variables and the evolution of a fully or partially observed (stochastic) process is specified via (stochastic) functions over next state variables conditioned on current state and action variables (n.b., concurrency is allowed). Parameterized variables are simply templates for ground variables that can be obtained when given a particular problem instance defining possible domain objects. Semantically, RDDL is simply a dynamic Bayes net (DBN) [1] (with potentially many intermediate layers) extended with a simple influence diagram (ID) [2] utility node representing immediate reward. An objective function specifies how these immediate rewards should be optimized over time for optimal control. For a ground instance, RDDL is just a factored MDP (or POMDP, if partially observed).

Contents

1 What's wrong with (P)PDDL?

In short, nothing is wrong with (P)PDDL. Every planning domain language serves a purpose to compactly specify a set of planning problems with common characteristics for exploitation by domain-independent (but domain language-specific) planners.

However, it would be unreasonable to assume there is one single compact and correct syntax for specifying all useful planning problems. Thus, RDDL is not intended as a replacement for the PDDL family of languages [3] or PPDDL [4], rather it is intended to model a class of problems that are difficult to model with PPDDL and PDDL. If (P)PDDL suffices for a problem description, then RDDL's expressivity is not needed.

As a motivating example for RDDL, we discuss the *cell transition model* (CTM) of traffic flow [5], which requires the following constructs not jointly expressible in (P)PDDL:

1. Each traffic signal is independently controlled by a concurrently executed action.
2. Cars move independently and stochastically.[1]
3. The full CTM uses integers to model counts of vehicles, real values to model traffic speed and density, and stochastic difference equations to specify transitions.
4. The CTM dynamics are simple, complexity derives from a nonfluent network topology. One would like to plan for *given* nonfluents *independent* of an initial state.
5. One would like to minimize traffic density in a CTM, which requires *summing* over *all* traffic cells (which change with each domain instance).
6. In concurrent domains, action preconditions cannot be checked locally, they must be checked globally, e.g., a joint configuration of two or more traffic signals may be illegal. For this one needs global state-action constraint checks.

Many other domains are difficult to formalize in PPDDL. Multi-elevator control with independent random arrivals, logistics domains with independently moving vehicles and noise, and UAVs with sensors for partially observed state are all important domains that cannot be specified in PPDDL. The obvious solution might simply be to extend PPDDL, as PDDL has been extended numerous times [3]. However, stochastic effects and concurrency are difficult to jointly reconcile in an effects-based language. If we take the approach that concurrent actions that possibly conflict (c.f., *probabilistic mutex* [6]) are disallowed — similar to the way concurrency is handled in PDDL 2.1 [7] — then we end up with a restrictive definition of concurrency that prevents concurrent actions that may only conflict 1% of the time. Instead we opt for *unrestricted concurrency* [8], for which it appears there is no well-defined PDDL-style transition semantics. Rather than add a layer of stochastic conflict resolution to PPDDL, a dynamic Bayes net (DBN) [1] transition formalism offers a simple solution — hence the motivation for RDDL.

[1] While a careful encoding of a *probabilistic* effect under a *forall* effect can encode this in PPDDL, it is not clear there is any way to resolve conflicting stochastic effects (two cars that stochastically move into a traffic cell, where there is only room for one of them).

2 Principles of RDDL

RDDL is influenced by the PDDL family [3], PPDDL [4], stochastic programs [9], influence diagrams [2], the SPUDD [10] and Symbolic Perseus [11, 12] representations for factored MDPs and POMDPs, first-order probabilitic inference (FOPI) – especially *parfactors* [13], and (factored) first-order MDPs and POMDPs [14, 15, 16].

A central design principle of RDDL is that the language should be simple and uniform with its expressive power deriving from composition of simple constructs.

2.1 What RDDL Is

RDDL is based on the following principles:

- *Everything* is a a parameterized variable (fluent or nonfluent)
  - Action fluents
  - State fluents
  - [Optional] Observation fluents (for partially observed domains)
  - [Optional] Intermediate fluents (derived predicates, correlated effects, ...)
  - [Optional] Constant nonfluents (general constants, topology relations, ...)
- Flexible fluent types
  - Binary (predicate) fluents
  - Multi-valued (enumerated) fluents
  - Integer and continuous fluents (numerical fluents from PDDL 2.1 [7])
- The semantics is simply a *ground Dynamic Bayes Net (DBN)*
  - Supports factored state and observations
  - Supports factored actions, hence concurrency (and never conflicts!)
  - Supports intermediate state fluents for multi-layered DBNs
    - Express (stochastic) derived predicates (c.f., PDDL 1.2 [17] and 2.2 [18])
    - Express correlated effects
    - Stratification by levels enforces a well-defined relational multi-layer DBN
  - Naturally supports independent exogenous events
- General expressions in transition and reward functions
  - Logical expressions ($\wedge, |, \sim, =>, <=>$ plus $\exists/\forall$ quantification over variables)

  - Arithmetic expressions ($+, -, *, /$ plus $\sum/\prod$ aggregation over variables)
  - (In)equality comparison expressions ($==, \sim=, <, >, <=, >=$)
  - Conditional expressions (if-then-else, switch)
  - Basic probability distributions (Bernoulli, Discrete, Normal, Poisson, ...)
- Classical Planning as well as General (PO)MDP objectives
  - Arbitrary reward (goals, numerical preferences) (c.f., PDDL 3.0 [19])
  - Finite horizon
  - Discounted or undiscounted
- State/action constraints
  - Encode legal actions (i.e., action preconditions)
  - Assert state invariants (e.g., a package cannot be in two locations)

2.2 What RDDL Isn't (Yet)

Notably, RDDL does not (at this time) support the following language features:

- Continuous time (c.f., PDDL2.1 [7])
- Durative actions / options / semi-(PO)MDPs (c.f., PDDL2.1 [7], also *options* [20])
- Temporal state/action goals or preferences (c.f., PDDL3.0 [19])
- Non-determinism or strict uncertainty (c.f., `oneof` construct in PPDDL [4])
- Game-theoretic constructs (c.f., Game Description Language (GDL) [21])
- Object fluents (c.f., PPDDL3.1/functional STRIPS [22]; enumerated types can substitute when the number of enumerated type values is fixed for all instances)

All features other than continuous time would be straightforward to add to RDDL.

3 RDDL Examples

Before we provide a formal language description, perhaps the best introduction to the language is through a few examples.

3.1 Simple Boolean Propositional Domain

We begin with a simple use of RDDL to encode a non-parameterized DBN with three boolean state variables p, q, r and one boolean action variable a.

dbn_prop.rddl

```
////////////////////////////////////////////////////////////////////////
// A simple propositional 2-slice DBN (variables are not parameterized).
//
// Author: Scott Sanner (ssanner [at] gmail.com)
////////////////////////////////////////////////////////////////////////
domain prop_dbn {

    requirements = { reward-deterministic };

    // Define the state and action variables (not parameterized here)
    pvariables {
        p : { state-fluent,  bool, default = false };
        q : { state-fluent,  bool, default = false };
        r : { state-fluent,  bool, default = false };
        a : { action-fluent, bool, default = false };
    };

    // Define the conditional probability function for each next
    // state variable in terms of previous state and action
    cpfs {
        p' = if (p ^ r) then Bernoulli(.9) else Bernoulli(.3);

        q' = if (q ^ r) then Bernoulli(.9)
                        else if (a) then Bernoulli(.3) else Bernoulli(.8);

        r' = if (~q) then KronDelta(r) else KronDelta(r <=> q);
    };

    // Define the reward function; note that boolean functions are
    // treated as 0/1 integers in arithmetic expressions
    reward = p + q - r;
}

// Define an instance of the above problem specifying an initial
// state and objective to achieve (discount and horizon)
instance inst_dbn {

    domain = prop_dbn;
    init-state {
        p = true;  // could also just say 'p' by itself
        q = false; // default so unnecessary, could also say '~q' by itself
        r;         // same as r = true
    };

    max-nondef-actions = 1; // No concurrency here, so set to 1
    horizon  = 20;
    discount = 0.9;
}
```

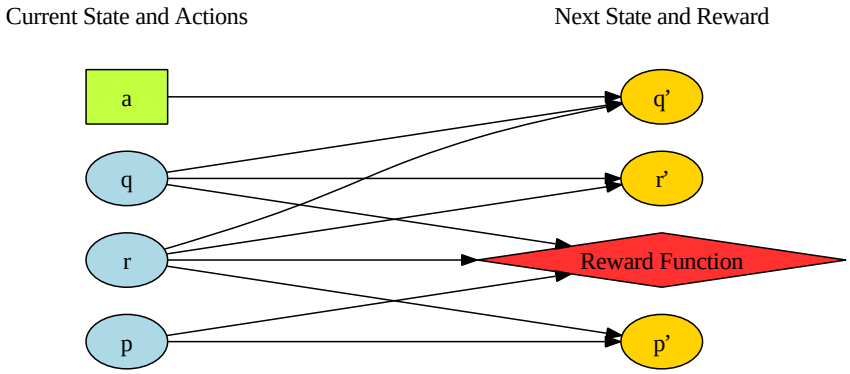

Figure 1: DBN and influence diagram for `dbn_prop.rddl` automatically produced by `rddl.viz.RDDL2Graph` in `rddlsim` Java package [23].

Before getting into details of this domain definition, we note that it can be simply represented by a DBN [1] and influence diagram [2] as provided in Figure 1.

Following is a line-by-line discussion of the domain description:

- All domains need an identifying name (here `prop_dbn`) provided on line 6.
- Domains should list their requirements as done on line 8, see Section 4.1.1 for a listing of possible requirements and their meaning.
- Lines 11–16 define parameterized variables (pvariables), although in this case we do not use parameters so these variables are in fact just the simple boolean propositional variables. `default` is used to specify the most common value of a pvariable, which is useful for minimizing communication in client/server interaction.
- Lines 20–27 list the domain transition function. Next-state variables are shown primed (p', q', r') to differentiate them from current state variables (p, q, r). The definition for p' simply gives the following conditional probability $P(p'|p, r)$:

p	r	p'	$P(p'\|p, q)$
true	*true*	*true*	0.9
true	*true*	*false*	0.1
true	*false*	*true*	0.3
true	*false*	*false*	0.7
false	*true*	*true*	0.3
false	*true*	*false*	0.7
false	*false*	*true*	0.3
false	*false*	*false*	0.7

(1)

Likewise a similar conditional probability can be generated for $P(q'|q, r, a)$; note here that the transition probability is dependent upon the action a. $P(r'|r, q)$ is a conditional expression over a Kronecker delta function. A Kroneckor delta simply places probability 1.0 on it's argument and 0 on all other possible values, so it is

useful whenever a transition is deterministic. Here, if q is false, then r' is assigned the value of r, otherwise r' is assigned the boolean value of the logical expression $r \Leftrightarrow q$. Note that if the argument of a delta function is from a continuous domain rather than a discrete domain, the Dirac delta function `DiracDelta` would be used instead.

- Line 31 lists the reward function, which determines what the agent should optimize at each step of time. Here we note that boolean variables are used in an arithmetic expression; whenever a logical expression is used in such an arithmetic expression, *true* is treated as 1 and *false* as 0.
- Lines 36–48 define an instance of this domain. Typically an instance will define domain objects, but this is not a parameterized domain, so only an initial state, action restrictions, and objective are provided here.
  - `init-state` lists ground fluent atoms and their truth assignment. Default fluent assignments need not be provided, but it is not an error to do so.
  - `max-nondef-actions` is used to specify how many actions in a domain are allowed to use a non-default value – a value larger than 1 would be specified for concurrent domains, but for non-concurrent domains like this one, a value of 1 should be used.
  - The objective evaluated by RDDL is simply the expected (i.e., average) sum of discounted rewards over multiple trials, where here the `discount` factor $\gamma = 0.9$ and `horizon` $h = 20$. At the end of each trial, the RDDL simulator returns the value $V_\pi(s_0)$ for the state-action trajectory encountered during the trial starting from the `init-state` definition of state s_0 and following the client agent's policy $\pi : S \rightarrow A$ which provides an action $a \in A$ for each state $s \in S$ encountered during the trial:

$$V_\pi(s_0) = \sum_{t=0}^{h} \gamma^t \cdot R(s_t, \pi(s_t)). \tag{2}$$

    Here $R(s_t, a_t)$ is the `reward` (sampled if requirement `reward-deterministic` is not specified) in state s_t at time t when action $a_t = \pi(s_t)$ is taken. The state trajectory $(s_0, \ldots, s_h)$ is simply sampled according to the defined `cpfs`.

3.2 Non-parameterized Partially-observed Domain

Before we move on to a true relational parameterized domain example, we first extend the previous `dbn_prop.rddl` with defined enumerated types, intermediate variables, and partial observability.

dbn_types_interm_po.rddl

```
////////////////////////////////////////////////////////////////////////
// A simple DBN (variables are not parameterized) exhibiting use of
// bools, ints, reals, enumerated types, intermediate variables, and
// observation variables.
//
// Author: Scott Sanner (ssanner [at] gmail.com)
////////////////////////////////////////////////////////////////////////
domain prop_dbn2 {

    requirements = {
        reward-deterministic, // Reward is a deterministic function
        integer-valued,       // Uses integer variables
        continuous,           // Uses continuous variables
        multivalued,          // Uses enumerated variables
        intermediate-nodes,   // Uses intermediate nodes
        partially-observed    // Uses observation nodes
    };

    // User-defined types
    types {
        enum_level : {@low, @medium, @high}; // An enumerated type
    };

    pvariables {
        p : { state-fluent,  bool, default = false };
        q : { state-fluent,  bool, default = false };
        r : { state-fluent,  bool, default = false };

        i1 : { interm-fluent, int,        level = 1 };
        i2 : { interm-fluent, enum_level, level = 2 };

        o1 : { observ-fluent, bool };
        o2 : { observ-fluent, real };

        a : { action-fluent, bool, default = false };
    };

    cpfs {

        // Some standard Bernoulli conditional probability tables
        p' = if (p ^ r) then Bernoulli(.9) else Bernoulli(.3);

        q' = if (q ^ r) then Bernoulli(.9)
                        else if (a) then Bernoulli(.3) else Bernoulli(.8);

        // KronDelta is a delta function for a discrete argument
        r' = if (~q) then KronDelta(r) else KronDelta(r <=> q);

```

```
49          // Just set i1 to a count of true state variables
50          i1 = KronDelta(p + q + r);
51
52          // Choose a level with given probabilities that sum to 1
53          i2 = Discrete(enum_level,
54                        @low : if (i1 >= 2) then 0.5 else 0.2,
55                        @medium : if (i1 >= 2) then 0.2 else 0.5,
56                        @high : 0.3
57                      );
58
59          // Note: Bernoulli parameter must be in [0,1]
60          o1 = Bernoulli( (p + q + r)/3.0 );
61
62          // Conditional linear stochastic equation
63          o2 = switch (i2) {
64                  case @low    : i1 + 1.0 + Normal(0.0, i1*i1),
65                  case @medium : i1 + 2.0 + Normal(0.0, i1*i1/2.0),
66                  case @high   : i1 + 3.0 + Normal(0.0, i1*i1/4.0) };
67      };
68
69      // A boolean functions as a 0/1 integer when a numerical value is needed
70      reward = p + q - r + 5*(i2 == @high);
71  }
72
73  instance inst_dbn {
74      domain = prop_dbn2;
75      init-state { p; r; };
76      max-nondef-actions = 1;
77      horizon  = 20;
78      discount = 0.9;
79  }
```

The DBN and influence diagram for this RDDL description is provided in Figure 2.

Here we simply cover the differences between this domain and the previous domain `dbn_prop.rddl`.

- In lines 10–17, we've added a number of requirements since this domain uses integer, continuous, and multivalued (enumerated) pvariables in addition to boolean variables. The domain uses intermediate variables that help determine the next state, but are not part of the state. Also the domain is partially observed, which means that in simulation, the server will determine both state and observations during simulation, but only provide the observations to the client agent for use in its policy decision.
- Lines 20–22 define the possible values for a user-defined enumerated (multivalue) type named `enum_level`.
- Lines 24–36 present additional pvariable definitions for the intermediate and ob-

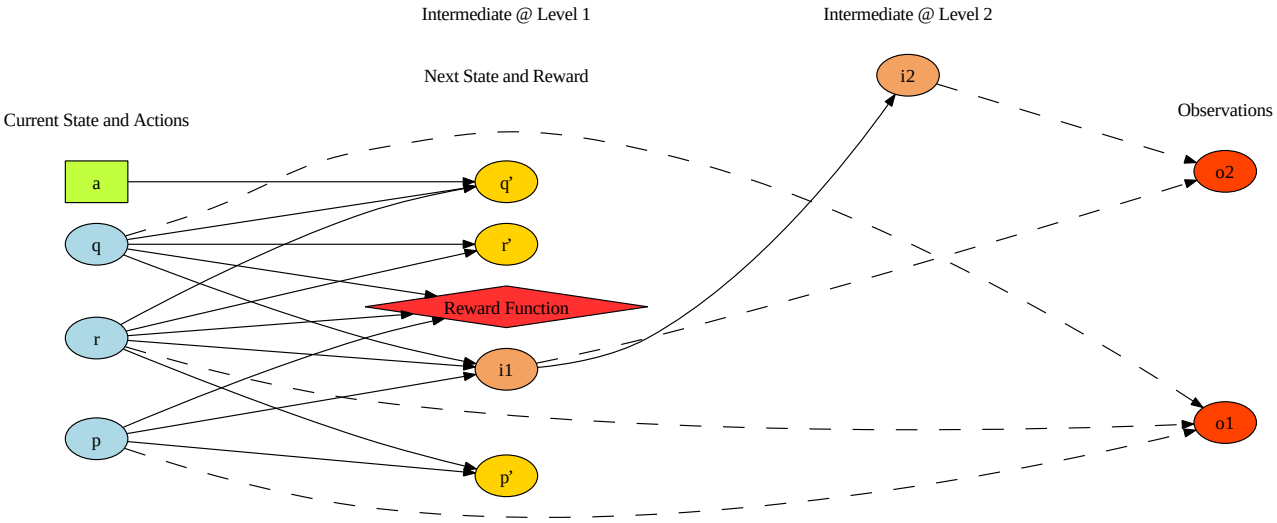

Figure 2: DBN and influence diagram for `dbn_types_interm_po.rddl` automatically produced by `rddl.viz.RDDL2Graph` in `rddlsim` Java package [23].

servation fluents. Again, parameters are not used here, but here we show types can also be `int`, `real`, or any of the user-defined types, in this case `enum_level`. Intermediate fluents must list a level of stratification. Intermediate variables are strictly stratified so that an intermediate variable can only condition on intermediate pvariables of a strictly lower level, or state pvariables. Intermediate and observation pvariables do not specify a default value.

- Lines 40–47 start with cpf definitions that are identical to the previous domain.
- Line 50 shows a simple cpf for an `int` type, where the value of intermediate variable `i1` is simply deterministically set to the sum $p + q + r$ (which takes values in $\{0, 1, 2, 3\}$). For an actually stochastic distribution, a `Poisson` with an appropriate rate parameter could be used in place of this `KronDelta`.
- Lines 53–57 show a useful way to sample a multivalued parameter from a `Discrete` distribution (the k-ary extension of the `Bernoulli` distribution). The first parameter here specifies the variable type being sampled (so that the simulator can perform type-checking). Next, each of the possible values are listed with the probabilities of each value. Note that these values must sum to 1.0 (otherwise the RDDL simulator will complain that the distribution is not well-defined). `i2` conditions on `i1` to determine the distribution and one will note that it sums to 1.0 for all values of `i1`.
- Line 60 is a standard Bernoulli sample where we simply show here that the parameters of any expression or random variable, can themselves be expressions. A Bernoulli parameter must be in $[0, 1]$ and one can verify this expression guarantees

that property; such properties are checked at runtime by the RDDL simulator.

- Lines 63–66 show that RDDL can be easily used to encode (stochastic) difference equations and via composition, more complex constructs like the *conditional* stochastic difference equation shown here, which makes use of a `switch` statement over various enumerated values of intermediate variable `i1`. We point out here that the parameters of distributions, in this case `Normal` with respective μ and σ^2 parameters, can be expressions.
- Line 70 demonstrates that intermediate pvariables can be used in a reward, and also that logical equality `==` can be used with any pvariable.

For a full listing of distributions that can be currently used with RDDL, please see Section 4.1.4.

3.3 Parameterized Domain: Concurrent Interactive Game of Life

Previously we showed non-parameterized RDDL domains that showed off the expressiveness of the language for specifying factored MDPs and POMDPs with potentially hybrid mixes of multivalued, integer, or continuous states and actions.

Already, this non-parameterized version of RDDL makes for quite an expressive language, but it is not always compact when variables and their cpfs must be repeated in a domain.

For example, a traffic domain can be modeled with traffic cells and all cells have essentially the same behavior — traffic flows into a cell from upstream cells when a cell is not at full capacity, and traffic flows out of a cell when the traffic signals permit and the downstream cells are not at capacity. There are simple rules that govern the behavior of a traffic cell and hence it does not make sense to repeatedly copy these rules for `cell-1`, `cell-2`, ..., `cell-n`. Obviously, here we would want to parameterize (i.e., lift) the transition dynamics and this requires parameterizing the RDDL DBN.

In Section 3.4, we provide an external link to the parameterized traffic domain specified in RDDL; however, because traffic is a fairly complex domain, we instead choose to demonstrate the parameterized DBN properties of RDDL in an interactive, stochastic, and potentially concurrent version of John H. Conway's *Game of Life* [24].

In short, the Game of Life specifies simple rules for a cellular automata where the next state properties of a cell depend on its surrounding cells. In the following RDDL description, we parameterize cells by their (x, y) coordinates and specify neighboring cells by a nonfluent boolean pvariable. The cpf transition function dynamics are based on the original rules plus some additional enhancements for stochasticity, resetting a dead row, and agent interaction — an agent can concurrently set a number of cells up to `max-nondef-actions` defined in an instance. We note that this domain explicitly defines the neighbor topology with nonfluents, thus allowing a lifted planner to exploit a fixed topology in its solution.

game_of_life_stoch.rddl

```
////////////////////////////////////////////////////////////////////
// A simple DBN to encode Conway's cellular automata "game of life"
// on a grid with some additional rules.  One gets a reward for
// generating patterns that keep the most cells alive.
//
// Author: Scott Sanner (ssanner [at] gmail.com)
////////////////////////////////////////////////////////////////////
domain game_of_life {

    requirements = { reward-deterministic };

    types {
        x_pos : object;
        y_pos : object;
    };

    pvariables {
        // Probability cell topology non-fluents (unchanging)
        PROB_REGENERATE : { non-fluent, real, default = 0.5 };
        NEIGHBOR(x_pos,y_pos,x_pos,y_pos) : {non-fluent,bool,default=false};

        // State, intermediate and action fluents
        alive(x_pos,y_pos) : { state-fluent, bool, default = false };
        count-neighbors(x_pos,y_pos) : { interm-fluent, int, level = 1 };
        set(x_pos,y_pos) : { action-fluent, bool, default = false };
    };

    cpfs {
        // Conway's game of life rules:
        // 1. Under-population: cell with < 2 live neighbors dies
        // 2. Overcrowding:     cell with > 3 live neighbors dies
        // 3. Survival:         cell with 2 or 3 live neighbors lives
        // 4. Reproduction:     cell with 3 live neighbors becomes live
        //
        // Scott's additional rules for RDDL:
        // 5. Stochastic: above rules hold with PROB_REGENERATE certainty
        // 6. Extra rule: all cells at same x-pos dead => random regeneration
        // 7. Interactivity: agent can concurrently set different cells

        // Store alive-neighbor count for each cell
        count-neighbors(?x,?y) =
            KronDelta(sum_{?x2 : x_pos, ?y2 : y_pos}
                      [NEIGHBOR(?x,?y,?x2,?y2) ^ alive(?x2,?y2)]);

        // Determine whether cell (?x,?y) is alive in next state
        alive'(?x,?y) = if (forall_{?y2 : y_pos} ~alive(?x,?y2))
                            then Bernoulli(PROB_REGENERATE) // Rule 6

```

```
                            else if ([alive(?x,?y)
                                      ^ (count-neighbors(?x,?y) >= 2)
                                      ^ (count-neighbors(?x,?y) <= 3)]
                                     | [~alive(?x,?y)
                                       ^ (count-neighbors(?x,?y) == 3)]
                                     | set(?x,?y))
                            then Bernoulli(PROB_REGENERATE)
                            else Bernoulli(1.0 - PROB_REGENERATE);
    };

    // Reward is number of alive cells
    reward = sum_{?x : x_pos, ?y : y_pos} alive(?x,?y);

    state-action-constraints {
        // Assertion: ensure PROB_REGENERATE is a valid probability
        (PROB_REGENERATE >= 0.0) ^ (PROB_REGENERATE <= 1.0);

        // Precondition: perhaps we should not set a cell if already alive
        forall_{?x : x_pos, ?y : y_pos} alive(?x,?y) => ~set(?x,?y);
    };
}

// Define numerical and topological constants
non-fluents game2x2 {
    domain = game_of_life;
    objects {
        x_pos : {x1,x2};
        y_pos : {y1,y2};
    };
    non-fluents {
        PROB_REGENERATE = 0.9; // Numerical constants are just non-fluents
        NEIGHBOR(x1,y1,x1,y2); NEIGHBOR(x1,y1,x2,y1); NEIGHBOR(x1,y1,x2,y2);
        NEIGHBOR(x1,y2,x1,y1); NEIGHBOR(x1,y2,x2,y1); NEIGHBOR(x1,y2,x2,y2);
        NEIGHBOR(x2,y1,x1,y1); NEIGHBOR(x2,y1,x1,y2); NEIGHBOR(x2,y1,x2,y2);
        NEIGHBOR(x2,y2,x1,y1); NEIGHBOR(x2,y2,x1,y2); NEIGHBOR(x2,y2,x2,y1);
    };
}

instance is1 {
    domain = game_of_life;
    non-fluents = game2x2;
    init-state {
        alive(x1,y1);
        alive(x2,y2);
    };
    max-nondef-actions = 3; // Allow up to 3 cells to be set concurrently
    horizon  = 20;
    discount = 0.9;
}
```

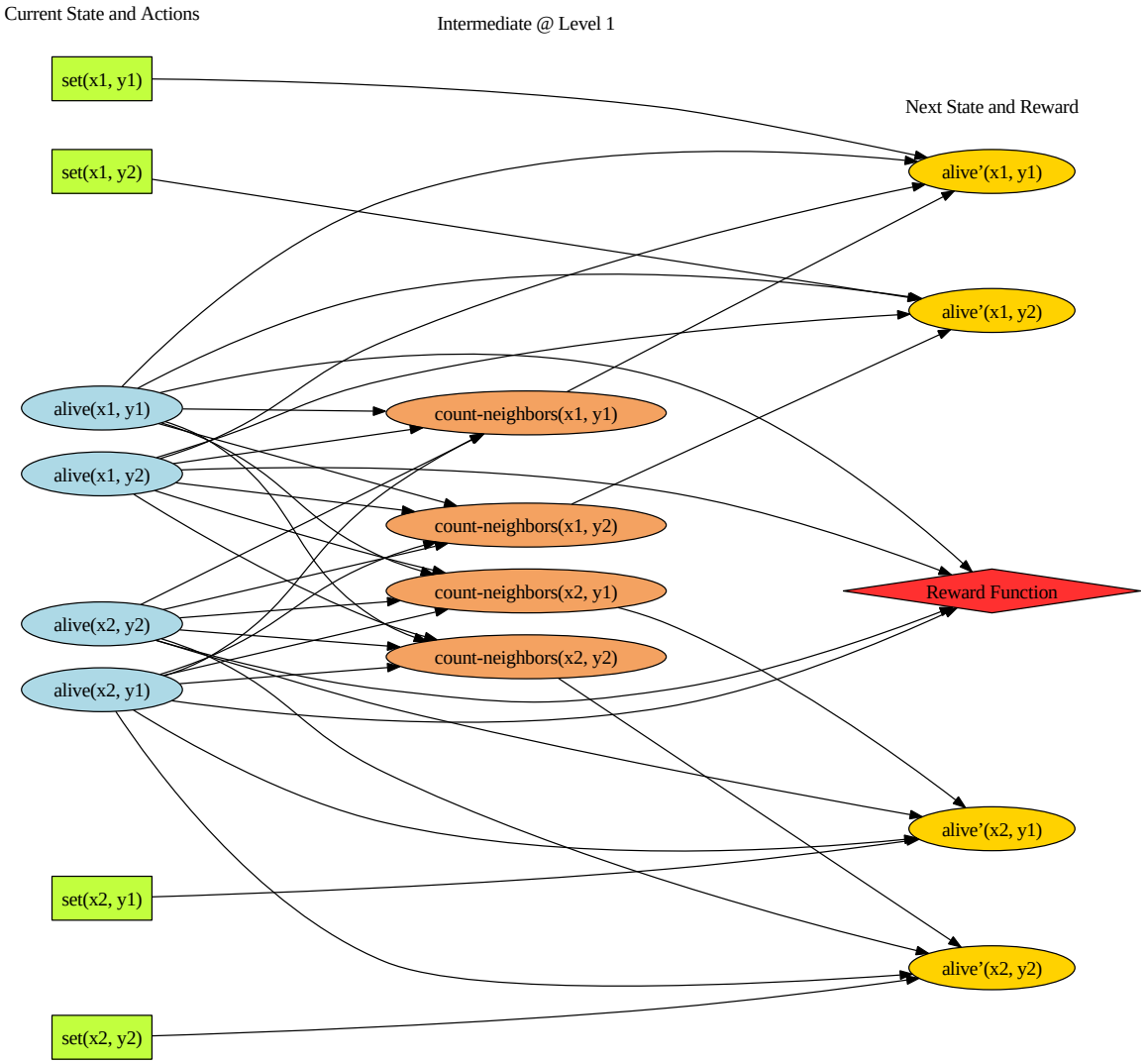

Figure 3: DBN and influence diagram for `game_of_life_stoch.rddl` automatically produced by `rddl.viz.RDDL2Graph` in `rddlsim` Java package [23].

The DBN and influence diagram for this RDDL description and instance `is1` is provided in Figure 3. This diagram is crucial for understanding that the *semantics of RDDL* is simply *a DBN over the ground pvariables of the domain instance*.

Perhaps the most **confusing issue for those familiar with PPDDL** will be the **semantics of parameterized actions in RDDL**. For this we again refer to Figure 3 where we note that there are *four ground action fluents* denoted by green rectangles. We note that *each* of these ground fluents is a *separate variable* taking on a distinct value determined by the user, and if we examine line 54 of the cpf for `alive`, we see that it conditions on *all* of these ground action fluent truth value assignments as needed.

This is in contrast to the PPDDL view of actions where all of the action information is given in the action name and parameters. Here an action is not viewed as a parameterized variable so it does not make sense to say a PPDDL action consists of multiple ground boolean variables (or int, real, or enumerated variables) as is the case in RDDL.

The view of RDDL actions as templates for ground variables directly supports concurrency. If actions are boolean pvariables as for the action pvariable `alive` in the Game of Life domain and *false* is the default value, then taking a single action in domain instance `is1` corresponds to setting any one of $set(x1, y1), set(x1, y2), set(x2, y1), set(x2, y2)$ to be true and the rest to be false. This corresponds to the non-concurrent case where `max-nondef-actions=1` and only one action is executed at time. However, if `max-nondef-actions=3` then up to three of $set(x1, y1), set(x1, y2), set(x2, y1), set(x2, y2)$ can be set to true, thereby allowing up to three concurrent actions. One will note that the cpf semantics for `alive` in the Game of Life domain description still holds in this concurrent case; hence, changing `max-nondef-actions` is all that is needed to control concurrency in RDDL.[2]

Having explained some of the major details of `game_of_life_stoch.rddl`, we proceed to highlight some remaining novel aspects of this domain:

- In lines 12–15, we've defined two user-defined object types for the x and y positions used to parameterize cells in the Game of Life.
- In lines 17–26, we note the definition of pvariables with parameters. Here the parameters listed are just the object types previously defined.
- In lines 19–20, we first note the definition of non-fluent pvariables. This is used for any pvariables that will not change during planning, but which can change between instances. Non fluents can be specified separately from an instance as shown in line 72 and referenced in the instance `is1` on line 89.
- In lines 29–57, we define parameterized cpfs:
  - In lines 41–43, since the count of alive neighbors of a cell is needed multiple times to determine the next state of every cell, we simply compute it for each cell and store it in a temporary intermediate variable. We note here the use of a `sum` over x and y position objects to perform this sum over all possible neighboring cells. As before, logical expressions (here in [...]) are treated as 0/1 values when used in an arithmetic expression (here `sum`).
  - Line 46 implements the rule to determine whether each cell is alive in the next state. Lines 46–47 use a universal quantifier over objects in the `if` condition test to implement Rule 6 in the comments, lines 49–54 implement Conway's standard rules, and lines 55–56 simply make the outcome predicted by Conway's rules stochastic according to the non-fluent `PROB_REGENERATE`.

[2] Of course, if multiple concurrent actions could interfere with each other, this would have to be handled directly in the cpf semantics for any affected pvariables. This is addressed in the Sidewalk domain referenced in Section 3.4.

- Line 60 specifies the deterministic reward, which is simply a sum over alive cells (again, this sum scales with the number of cells in a particular domain instance).
- Lines 62–68 demonstrate `state-action` constraints, which have not been used previously. `state-action` constraints serve the following two purposes:
  - *Logical assertions* on all states that can be reached from any legal initial state. For example, line 64 ensures that the `PROB_REGENERATE` pvariable is a valid probability in $[0, 1]$. Such a constraint could also apply to any (quantified) logical expression over fluents.
  - *Action preconditions* for local and global precondition checks. Because preconditions in concurrent domains must be checked globally — two or more actions may mutually constain each other — we adopt the uniform approach of specifying all action preconditions in the `state-action` constraints section, whether concurrent or not. An example of a simple *local* action precondition is given in line 67.

  Any joint state and action that violate a `state-action` constraint during a trial should cause the RDDL trial simulator to abort in error since there was either an error in the domain description leading to an illegal state, or the agent made an error in the policy and tried to execute an illegal action. *Implicitly, if the agent only executes legal actions, then all possible sampled trajectories should satisfy the state-action constraints.* State-action constraints are crucial for lifted and regression-style planners that plan independently of any initial state (and hence cannot exploit reachability from an initial state to determine legal states).
- Lines 72–85 define a non-fluents section where a cell topology is specified. This particular assignment to non-fluents is referenced in line 89 of the instance definition. The separation of non-fluents from an initial state is intended to support lifted planning that is independent of an initial state, while allowing a planner to exploit specific nonfluent structure common to many problem instances (e.g. a cellular topology for the Game of Life, or a road network in a logistics domain).
- Line 94 specifies that `max-nondef-actions=3`, which is used to allow multiple `set` actions to be executed concurrently in this domain as explained previously. If this domain is intended to support only serial actions then this should be changed to `max-nondef-actions=1`.

3.4 Additional Models

RDDL is a very expressive language, so to give the reader a sense of a few other interesting domains that can be encoded in RDDL, we refer them to the following domains (with external links that are hosted on the `rddlsim` code repository [23]):

- Multi-intersection traffic control: This domain specification uses a simple binary cell transition model (a higher fidelity cell transition model would model velocity

and density as real values and use stochastic difference equation updates). It is a good example of how the topology of a particular problem can be compiled away into the nonfluents.

- Sidewalk: This is a simple domain that illustrates how to handle conflicts in RDDL, in this case, two people walking on a sidewalk and trying to reach opposite ends without colliding. Here, intermediate variables are used to detect a conflict and then the next state variable cpfs condition on this conflict detection in determining the next state.
- System Administration: This is a commonly referenced factored MDP/POMDP domain is used here to demonstrate various expressive abilities of RDDL.

4 RDDL File Structure

A RDDL file may contain three types of top-level declarations: domains, non-fluents, and instances. The following is a minimal description, **we rely on the previous code and listings for examples of each construct listed below**.

4.1 domain block

A domain description consists of a requirements statement, parameter type definitions, variable definitions, transition dynamics, and a reward.

4.1.1 requirements block

- `continuous`: this domain uses real-valued parameterized variables
- `multivalued`: this domain uses enumerated pvariables
- `reward-deterministic`: this domain does not use a stochastic reward
- `intermediate-nodes`: this domain uses intermediate pvariable nodes
- `constrained-state`: this domain uses state constraints
- `partially-observed`: this domain uses observation pvariables so it is treated as a POMDP (not an MDP as is otherwise the case)
- `concurrent`: this domain does not permit multiple non-default actions
- `integer-valued`: this domain does not use integer variables
- `cpf-deterministic`: this domain uses deterministic conditional functions for transitions (it is important to note that RDDL can also be used to model deterministic domains)

4.1.2 types

Allowed types are `object` and *enumerated* types. Enumerated type values must be specified in a set and must be prefixed with an `@` symbol.

4.1.3 pvariables

Allowed pvariable types are `non-fluent`, `state-fluent`, `action-fluent`, `interm-fluent`, and `observ-fluent`. The first three require a default value, and `interm-fluent` requires a stratification level.

Possible pvariable ranges are `bool`, `int`, `real`, *object*, or *enumerated*. The latter two require the user-defined name as the range specification.

4.1.4 cpfs

If the requirement `cpf-deterministic` is specified, then this section should be named `cdf` (conditional deterministic function) in place of `cpf` (conditional probabilistic function). cdfs should not reference any probability distributions; cpfs should also use a probability distribution or a `KronDelta` or `DiracDelta` if the cpf is actually deterministic.

cpfs and cdfs must be specified for all non-fluent, non-action pvariables. cpfs begin with a pvariable name and logical variable specification (variables must begin with `?`) corresponding to the argument types listed in the pvariable declaration. A pvariable name for a next-state fluent must be *primed* with a `'` to differentiate it from any mentions of the current-state value of the pvariable.

cpf expressions are compositional and can consist of the following constructs:

- Constants
  - `true`, `false` (evaluated respectively as 1 or 0 if used in arithmetic expressions)
  - integers (-2,0,1790,...) and reals (-2.0, 0.0001, 3.14159)
  - enumerated values (although these have no boolean or arithmetic evaluation)
- Grouping can use either balanced parens (...) or brackets [...]
- Logical expressions ($\wedge, |, \sim, =>, <=>$ plus $\exists/\forall$ quantification over variables)
  - Negation $\sim$ or any binary logical connective $\wedge, |, \sim, =>, <=>$
  - $\exists/\forall$ quantification over *object types* using `forall` and `exists`
- Arithmetic expressions ($+, -, *, /$) plus $\sum/\prod$ aggregation over variables)
  - Any binary arithmetic expression using $+, -, *, /$

  - $\sum$ and $\prod$ aggregation over *object types* using `sum` and `prod`
- (In)equality comparison expressions ($==, \sim=, <, >, <=, >=$)
  - Equality ($==$) and disequality ($\sim=$) between any identical range pvariables
  - Inequality ($<, >, <=, >=$) between any numerically valued pvariables (real, int, bool) or expressions
- Conditional expressions
  - `if-then-else`: see numerous code examples
  - `switch`: see code example in `dbn_types_interm_po.rddl`, lines 63–66
- Basic probability distributions (note: all parameters can be expressions)
  - `KronDelta`(v): places all probability mass on its discrete argument v, discrete sample is thus deterministic
  - `DiracDelta`(v): places all probability mass on its continuous argument v, continuous sample is thus deterministic
  - `Bernoulli`(p): samples a boolean with probability of *true* given by parameter $p \in [0, 1]$
  - `Discrete`(`var-name`,$\vec{p}$): samples an enumerated value with probability vector $\vec{p}$ ($\sum_i \vec{p}_i = 1$) where $\vec{p}$ is described as in the example of lines 53–57 in `dbn_types_interm_po.rddl`.
  - `Normal`(μ,σ^2): samples a continuous value from a Normal distribution with mean μ and variance σ^2, $\sigma^2 > 0$.
  - `Poisson`(λ): samples an integer value from a Poisson distribution with rate parameter λ per fixed time interval, $\lambda > 0$.
  - (more to come in future)

4.1.5 reward

A `reward` section specifies any arithmetic expression that can be evaluated/sampled to a numerical constant (so no unbound variables) over the current state of any `non-fluent`, `state-fluent`, `action-fluent`, or `interm-fluent` pvariables.

If the `reward-deterministic` requirement is specified, the reward specification should not reference any distributions (e.g., `Bernoulli`).

4.1.6 state-action constraints

A `state-action` constraints section consists of lines containing logical expressions that can be evaluated to true or false (so no unbound variables) over the current state of any `non-fluent`, `state-fluent`, or `action-fluent` pvariables.

Note that intermediate variables *cannot* be referenced in the state-action constraints as this would correspond to checking the (partial) outcome of an action, rather than its preconditions.

A violation of any `state-action` constraint should lead to termination of the current RDDL simulator trial with an error.

4.2 non-fluents block

An `non-fluents` block describes an instantiation of non-fluents, e.g, a fixed cell topology in the Game of Life or a road topology in a logistics or traffic domain, and the object domains that parameterize those non-fluent variables. Only user-defined object domains used as a non-fluent parameter need to be specified in this section. Other object domains can be specified in the `instance` block.

The `non-fluents` block may contain `domain`, `objects`, and `non-fluents` sections.

4.3 instance block

An `instance` block consists of remaining object instantiations not made in an optional non-fluents specification, an initial state, and an objective criterion.

The `instance` block may contain `domain`, `non-fluents`, `objects`, `init-state`, `max-nondef-actions` (for concurrency), `horizon`, and `discount` sections.

See the discussion after `prop_dbn` to understand how RDDL evaluates the objective on any trial.

5 `rddlsim` RDDL Simulator

For now, please refer to the documentation provided in the root directory of the `rddlsim` code repository located at http://code.google.com/p/rddlsim/.

References

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

[2] Ronald A. Howard and James E. Matheson. Influence diagrams. In Ronald A. Howard and James E. Matheson, editors, *Readings on the Principles and Applications of Decision Analysis*. Strategic Decision Group, Menlo Park, CA, 1984.

[3] Malte Helmert. PDDL resources: http://ipc.informatik.uni-freiburg.de/PddlResources, 2009.

[4] Hakan Younes and Michael Littman. PPDDL: The probabilistic planning domain definition language: http://www.cs.cmu.edu/~lorens/papers/ppddl.pdf, 2004.

[5] Carlos Daganzo. The cell transmission model: Network traffic. Institute of transportation studies, research reports, working papers, proceedings, Institute of Transportation Studies, UC Berkeley, 1994.

[6] Avrim Blum and John Langford. Probabilistic planning in the graphplan framework. In *5th European Conference on Planning (ECP)*, pages 319–332, London, UK, 2000.

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

[8] Iain Little and Sylvie Thiebaux. Concurrent probabilistic planning in the graphplan framework. In *ICAPS*, pages 263–273. AAAI, 2006.

[9] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.

[10] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Uncertainty in Artificial Intelligence (UAI-99)*, pages 279–288, Stockholm, 1999.

[11] Pascal Poupart. *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, 2005.

[12] Pascal Poupart. Symbolic perseus code repository, 2005.

[13] David Poole. First-order probabilistic inference. In *IJCAI*, pages 985–991, 2003.

[14] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *IJCAI-01*, pages 690–697, Seattle, 2001.

[15] Scott Sanner and Craig Boutilier. Approximate solution techniques for factored first-order MDPs. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 07)*, 2007.

[16] Scott Sanner and Kristian Kersting. Symbolic dynamic programming for first-order poMDPs. In *In Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, Atlanta, Georgia, July 19-23 2010. AAAI Press.

[17] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – the planning domain definition language – version 1.2. Technical report, Yale Center for Computational Vision and Control, October 1998.

[18] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of IPC-4. Technical report, Albert-Ludwigs-Universitt Freiburg, Institut fr Informatik, January 2004.

[19] Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3. Technical report, Dipartimento di Elettronica per l'Automazione, Universit degli Studi di Brescia, August 2005.

[20] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[21] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game Description Language specification. Technical report, Stanford University Logic Group, March 2008.

[22] Hctor Geffner. Functional strips: A more flexible language for planning and problem solving. In Jack Minker, editor, *Logic-based Artificial Intelligence*, pages 188–209. Kluwer, 2000.

[23] Scott Sanner and Sungwook Yoon. rddlsim RDDL simulator: `http://code.google.com/p/rddlsim/`, 2010.

[24] M. Gardner. Column: Mathematical games. *Scientific American*, October 1970.