Goals as Reward-Generating Programs Domain Specific Language

February 20, 2022

1 DSL Docuemntation as of February 20, 2022

1.1 Game Definition

⟨*game*⟩ ::= (define (game ⟨*name*⟩)
(:domain ⟨*name*⟩) # A specification of the environment
(:setup ⟨*setup*⟩) | () # The optional setup conditions
(:constraints ⟨*constraints*⟩) # The preferences that specify gameplay
(:terminal ⟨*terminal*⟩) | () # Optional terminal conditions
(:scoring ⟨*scoring*⟩) # The scoring rules that define how to arrive from preference specifications to a score
)

⟨*name*⟩ ::= /[A-z]+(_[A-z0-9]+)*/ # Any name: a letter, optionally followed by letters, numbers, and underscores

1.2 Setup

PDDL doesn't have any close equivalent of this, but when reading through the games participants specify, they often require some transformation of the room from its initial state before the game can be played. We could treat both as parts of the gameplay, but we thought there's quite a bit to be gained by splitting them – for example, the policies to setup a room are quite different from the policies to play a game (much more static).

The one nuance here came from the (game-conserved ...) and (game-optional ...) elements. It seemed to us that some setup elements should be maintained throughout gameplay (for example, if you place a bin somewhere to throw into, it shouldn't move unless specified otherwise). Other setup elements can, or often must change – for example, if you set the balls on the desk to throw them, you'll have to pick them up off the desk to throw them. These elements provide that context, which could be useful for verifying that agents playing the game don't violate these conditions.

⟨*setup*⟩ ::= (and ⟨*setup*⟩ ⟨*setup*⟩$^+$) # A setup can be expanded to a conjunction, a disjunction, a quantification, or a setup statement (see below).
| (or ⟨*setup*⟩ ⟨*setup*⟩$^+$)
| (not ⟨*setup*⟩)
| (exists (⟨*typed list(variable)*⟩) ⟨*setup*⟩)
| (forall (⟨*typed list(variable)*⟩) ⟨*setup*⟩)
| ⟨*setup-statement*⟩

⟨*setup-statement*⟩ ::= # A setup statement specifies that a predicate is either optional during gameplay or must be preserved during gameplay.
| (game-conserved ⟨*setup-predicate*⟩)
| (game-optional ⟨*setup-predicate*⟩)

⟨*setup-predicate*⟩ ::= # A setup predicate can be a conjunction, disjunction, negation, quantification, function comparison, or a predicate.
| (and ⟨*setup-predidcate*⟩$^+$)
| (or ⟨*setup-predicate*⟩$^+$)
| (not ⟨*setup-predicate*⟩
| (exists (⟨*typed list(variable)*⟩) ⟨*setup-predicate*⟩)
| (forall (⟨*typed list(variable)*⟩) ⟨*setup-predicate*⟩)
| ⟨*f-comp*⟩
| ⟨*predicate*⟩

⟨*f-comp*⟩ ::= # A function comparison: either comparing two function evaluations, or checking that two ore more functions evaluate to the same result.
| (⟨*comp-op*⟩ ⟨*function-eval-or-number*⟩ ⟨*function-eval-or-number*⟩)
| (= ⟨*function-eval-or-number*⟩$^+$)

⟨*comp-op*⟩ ::= ⟨ | ⟨= | = | ⟩ | ⟩= # Any of the comparison operators.

⟨*function-eval-or-number*⟩ ::= ⟨*function-eval*⟩ | ⟨*number*⟩

⟨*function-eval*⟩ ::= (⟨*name*⟩ ⟨*function-term*⟩$^+$) # An evaluation of a function on any number of arguments.

⟨*function-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩ | ⟨*number*⟩ | ⟨*predicate*⟩

⟨*variable-list*⟩ ::= (⟨*variable-type-def*⟩$^+$) # One or more variables definitions, enclosed by parentheses.

⟨*variable-type-def*⟩ ::= ⟨*variable*⟩$^+$ - ⟨*type-def*⟩ # Each variable is defined by a variable (see next) and a type (see after).

⟨*variable*⟩ ::= /\?[a-z][a-z0-9]*/ # a question mark followed by a letter, optionally followed by additional letters or numbers.

⟨*type-def*⟩ ::= ⟨*name*⟩ | ⟨*either-types*⟩ # A veriable type can either be a single name, or a list of type names, as specified by the next rule:

⟨*either-types*⟩ ::= (either ⟨*name*⟩$^+$)

⟨*predicate*⟩ ::= (⟨*name*⟩ ⟨*predicate-term*⟩*)

⟨*predicate-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩

1.3 Gameplay Preferences

The gameplay preferences specify the core of a game's semantics, capturing how a game should be played by specifying temporal constraints over predicates.
PDDL calls their temporal preferences 'constraints', but that's not entirely the right name for us. Maybe we should rename?

⟨*constraints*⟩ ::= ⟨*pref-def*⟩ | (and ⟨*pref-def*⟩$^+$) # One or more preferences.

⟨*pref-def*⟩ ::= ⟨*pref-forall*⟩ | ⟨*preference*⟩ # A preference definitions expands to either a forall quantification (see below) or to a preference.

⟨*pref-forall*⟩ ::= (forall ⟨*variable-list*⟩ ⟨*preference*⟩) # this syntax is used to specify variants of the same preference for different object, which differ in their scoring. These are specified using the ⟨*pref-name-and-types*⟩ syntax element's optional types, see scoring below.

⟨*preference*⟩ ::= (preference ⟨*name*⟩ ⟨*preference-quantifier*⟩) # A preference is defined by a name and a quantifer that includes the preference body.

⟨*preference-quantifier*⟩ ::= # A preference can quantify exsistentially or universally over one or more variables, or none.
| (exists (⟨*variable-list*⟩) ⟨*preference-body*⟩
| (forall (⟨*variable-list*⟩) ⟨*preference-body*⟩)
| ⟨*preference-body*⟩)

⟨*preference-body*⟩ ::= ⟨*then*⟩ | ⟨*at-end*⟩ # A preference body can be expanded to conditions over a sequence of states (⟨*then*⟩) or over the terminal state (⟨*at-end*⟩).

⟨*at-end*⟩ ::= (at-end ⟨*pref-predicate*⟩) # Specifies a prediicate that should hold in the terminal state.

⟨*then*⟩ ::= (then ⟨*seq-func*⟩ ⟨*seq-func*⟩$^+$) # Specifies a series of conditions that should hold over a sequence of states – see below for the specific operators (⟨*seq-func*⟩s), and Section 2 for translation of these definitions to linear temporal logicl (LTL).

⟨*seq-func*⟩ ::= ⟨*once*⟩ | ⟨*once-measure*⟩ | ⟨*hold*⟩ | ⟨*hold-while*⟩ # Four of thse temporal sequence functions currently exist:

⟨*once*⟩ ::= (once ⟨*pref-predicate*⟩) # The predicate specified must hold for a single world state.

⟨*once-measure*⟩ ::= (once ⟨*pref-predicate*⟩ ⟨*function-eval*⟩) # The predicate specified must hold for a single world state, and record the value of the function evaluation, to be used in scoring.

⟨*hold*⟩ ::= (hold ⟨*pref-predicate*⟩) # The predicate specified must hold for every state between the previous temporal operator and the next one.

⟨*hold-while*⟩ ::= (hold-while ⟨*pref-predicate*⟩ ⟨*pref-predicate*⟩$^+$) # The predicate specified must hold for every state between the previous temporal operator and the next one. While it does, at least one state must satisfy each of the predicates specified in the second arumgnet onward.

⟨*pref-predicate*⟩ ::= # A preference predicate can be a conjunction, disjunction, negation, quantification, function comparison, or a predicate.
| (and ⟨*pref-predicate*⟩$^+$)
| (or ⟨*pref-predicate*⟩$^+$)
| (not ⟨*pref-predicate*⟩)
| (exists ⟨*variable-list*⟩ ⟨*pref-predicate*⟩)
| (forall ⟨*variable-list*⟩ ⟨*pref-predicate*⟩)
| ⟨*f-comp*⟩
| ⟨*predicate*⟩

⟨*f-comp*⟩ ::= # A function comparison: either comparing two function evaluations, or checking that two ore more functions evaluate to the same result.
| (⟨*comp-op*⟩ ⟨*function-eval-or-number*⟩ ⟨*function-eval-or-number*⟩)
| (= ⟨*function-eval-or-number*⟩$^+$)

⟨*comp-op*⟩ ::= ⟨ | ⟨= | = | ⟩ | ⟩= # Any of the comparison operators.

⟨*function-eval-or-number*⟩ ::= ⟨*function-eval*⟩ | ⟨*number*⟩

⟨*function-eval*⟩ ::= (⟨*name*⟩ ⟨*function-term*⟩$^+$) # An evaluation of a function on any number of arguments.

⟨*function-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩ | ⟨*number*⟩ | ⟨*predicate*⟩

⟨*variable-list*⟩ ::= (⟨*variable-type-def*⟩$^+$) # One or more variables definitions, enclosed by parentheses.

⟨*variable-type-def*⟩ ::= ⟨*variable*⟩$^+$ - ⟨*type-def*⟩ # Each variable is defined by a variable (see next) and a type (see after).

⟨*variable*⟩ ::= /\?[a-z][a-z0-9]*/ # a question mark followed by a letter, optionally followed by additional letters or numbers.

⟨*type-def*⟩ ::= ⟨*name*⟩ | ⟨*either-types*⟩ # A veriable type can either be a single name, or a list of type names, as specified by the next rule:

⟨*either-types*⟩ ::= (either ⟨*name*⟩$^+$)

⟨*predicate*⟩ ::= (⟨*name*⟩ ⟨*predicate-term*⟩*)

⟨*predicate-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩

1.4 Terminal Conditions

Some participants explicitly specify terminal conditions, but we consider this to be optional.

⟨*terminal*⟩ ::= # The terminal condition is specified by a conjunction, disjunction, negation, or comparson (see below).
| (and ⟨*terminal*⟩$^+$)
| (or ⟨*terminal*⟩+)
| (not ⟨*terminal*⟩)
| ⟨*terminal-comp*⟩

⟨*terminal-comp*⟩ ::= (⟨*comp-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩) # A comparison operator is used to compare two scoring expressions (see next section).

⟨*comp-op*⟩ ::= ⟨ | ⟨= | = | ⟩ | ⟩=

For a full specification of the ⟨scoring-expr⟩ token, see the scoring section below.

1.5 Scoring

Scoring rules specify how to count preferences (count once, once for each unique objects that fulfill the preference, each time a preference is satisfied, etc.), and the arithmetic to combine counted preference statisfactions to get a final score.
PDDL calls their equivalent section (:metric ...), but we renamed because it made more sense to in the context of games.

⟨*scoring*⟩ ::= (maximize ⟨*scoring-expr*⟩)
| (minimize ⟨*scoring-expr*⟩) # The scoring conditions maximize or minimize a scoring expression.

⟨*scoring-expr*⟩ ::= # A scoring expression can be an arithmetic operation over other scoring expressions, a reference to the total time or score, a comparison, or a preference scoring evaluation.
| (⟨*multi-op*⟩ ⟨*scoring-expr*⟩$^+$) # Either addition or multiplication.
| (⟨*binary-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩) # Either division or subtraction.
| (- ⟨*scoring-expr*⟩)
| (total-time)
| (total-score)
| ⟨*scoring-comp*⟩
| ⟨*preference-eval*⟩

⟨*scoring-comp*⟩ ::= # A scoring comparison: either comparing two expressions, or checking that two ore more expressions are equal.
| (⟨*comp-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩)
| (= ⟨*scoring-expr*⟩$^+$)

⟨*preference-eval*⟩ ::= # A preference evaluation applies one of the scoring operators (see below) to a particular preference referenced by name (with optional types).
| ⟨*count-nonoverlapping*⟩
| ⟨*count-once*⟩
| ⟨*count-once-per-objects*⟩
| ⟨*count-nonoverlapping-measure*⟩
| ⟨*count-unique-positions*⟩
| ⟨*count-same-positions*⟩
| ⟨*count-maximal-nonoverlapping*⟩
| ⟨*count-maximal-overlapping*⟩
| ⟨*count-maximal-once-per-objects*⟩
| ⟨*count-maximal-once*⟩
| ⟨*count-once-per-external-objects*⟩

⟨*count-nonoverlapping*⟩ ::= (count-nonoverlapping ⟨*pref-name-and-types*⟩) # Count how many times the preference is satisfied by non-overlapping sequences of states.

⟨*count-once*⟩ ::= (count-once ⟨*pref-name-and-types*⟩) # Count whether or not this preference was satisfied at all.

⟨*count-once-per-objects*⟩ ::= (count-once-per-objects ⟨*pref-name-and-types*⟩) # Count once for each unique combination of objects quantified in the preference that satisfy it.

⟨*count-nonoverlapping-measure*⟩ ::= (count-nonoverlapping-measure ⟨*pref-name-and-types*⟩) # Can only be used in preferences including a ⟨*once-measure*⟩ modal, maps each preference satistifaction to the value of the function evaluation in the ⟨*once-measure*⟩.

⟨*count-unique-positions*⟩ ::= (count-unique-positions ⟨*pref-name-and-types*⟩) # Count how many times the preference was satisfied with quantified objects that remain stationary within each preference satisfcation, and have different positions between different satisfactions.

⟨*count-same-positions*⟩ ::= (count-same-positions ⟨*pref-name-and-types*⟩) # Count how many times the preference was satisfied with quantified objects that remain stationary within each preference satisfcation, and have (approximately) the same position between different satisfactions.

⟨*note*⟩ : # All of the count-maximal-... operators refer to counting only for preferences inside a (forall ...), and count only for the object quantified externally that has the most preference satisfactions to it. If there exist multiple preferences in a single (forall ...) block, score for the single object that satisfies the most over all such preferences.

⟨*count-maximal-nonoverlapping*⟩ ::= (count-maximal-nonoverlapping ⟨*pref-name-and-types*⟩) # For the single externally quantified object with the most satisfcations, count non-overlapping satisfactions of this preference.

⟨*count-maximal-overlapping*⟩ ::= (count-maximal-overlapping ⟨*pref-name-and-types*⟩) # For the single externally quantified object with the most satisfcations, count how many satisfactions of this preference with different objects overlap in their states.

⟨*count-maximal-once-per-objects*⟩ ::= (count-maximal-once-per-objects ⟨*pref-name-and-types*⟩) # For the single externally quantified object with the most satisfcations, count this preference for each set of quantified objects that satisfies it.

⟨*count-maximal-once*⟩ ::= (count-maximal-once ⟨*pref-name-and-types*⟩) # For the externally quantified object with the most satisfcations (across all preferences in the same (forall ...) block), count this preference at most once.

⟨*count-once-per-external-objects*⟩ ::= (count-once-per-external-objects ⟨*pref-name-and-types*⟩) # Similarly to count-once-per-objects, but counting only for each unique object or combination of objects quantified in the (forall ...) block including this preference.

⟨*pref-name-and-types*⟩ ::= ⟨*name*⟩ ⟨*pref-object-type*⟩* # The optional ⟨*pref-object-type*⟩s are used to specify a particular instance of the preference for a given object, see the ⟨*pref-forall*⟩ syntax above.

⟨*pref-object-type*⟩ ::= : ⟨*name*⟩ # The optional type name specification for the above syntax. For example, pref-name:dodgeball would refer to the preference where the first quantified object is a dodgeball.

1.6 Predicates

The predicates are not defined as part of the DSL, but rather we envision them is being specific to a domain and being specified to any model as an input or something to be conditioned on.

The following describes all predicates currently found in our game dataset:

```
(= <arg1> <arg2>) [7 references] ; Are these two objects the same object?
(above <arg1> <arg2>) [5 references] ; Is the first object above the second object?
(adjacent <arg1> <arg2>) [76 references] ; Are the two objects adjacent? [will probably be
    implemented as distance below some threshold]
(adjacent_side <3 or 4 arguments>) [14 references] ; Are the two objects adjacent on the sides
    specified? Specifying a side for the second object is optional, allowing to specify <obj1> <
    side1> <obj2> or <obj1> <side1> <obj2> <side2>
(agent_crouches ) [2 references] ; Is the agent crouching?
```

```
(agent_holds <arg1>) [327 references] ; Is the agent holding the object?
(between <arg1> <arg2> <arg3>) [7 references] ; Is the second object between the first object
    and the third object?
(broken <arg1>) [2 references] ; Is the object broken?
(equal_x_position <arg1> <arg2>) [2 references] ; Are these two objects (approximately) in the
    same x position? (in our environment, x, z are spatial coordinates, y is the height)
(equal_z_position <arg1> <arg2>) [5 references] ; Are these two objects (approximately) in the
    same z position? (in our environment, x, z are spatial coordinates, y is the height)
(faces <arg1> <arg2>) [6 references] ; Is the front of the first object facing the front of the
     second object?
(game_over ) [4 references] ; Is this the last state of gameplay?
(game_start ) [3 references] ; Is this the first state of gameplay?
(in <2 or 3 arguments>) [121 references] ; Is the second argument inside the first argument? [a
     containment check of some sort, for balls in bins, for example]
(in_motion <arg1>) [311 references] ; Is the object in motion?
(is_setup_object <arg1>) [10 references] ; Is this the object of the same type referenced in
    the setup?
(object_orientation <arg1> <arg2>) [15 references] ; Is the first argument, an object, in the
    orientation specified by the second argument? Used to check if an object is upright or
    upside down
(on <arg1> <arg2>) [165 references] ; Is the second object on the first one?
(open <arg1>) [3 references] ; Is the object open? Only valid for objects that can be opened,
    such as drawers.
(opposite <arg1> <arg2>) [4 references] ; So far used only with walls, or sides of the room, to
     specify two walls opposite each other in conjunction with other predicates involving these
    walls
(rug_color_under <arg1> <arg2>) [11 references] ; Is the color of the rug under the object (
    first argument) the color specified by the second argument?
(same_type <arg1> <arg2>) [3 references] ; Are these two objects of the same type?
(toggled_on <arg1>) [4 references] ; Is this object toggled on?
(touch <arg1> <arg2>) [49 references] ; Are these two objects touching?
(type <arg1> <arg2>) [9 references] ; Is the first argument, an object, an instance of the type
     specified by the second argument?
```

1.7 Functions

Functions operate simlarly to predicates, but rather than returning a boolean value, they return a numeric value or a type. Similarly to predicates, they are not parts of the DSL per se, but might vary by environment.

The following describes all functions currently found in our game dataset:

```
(building_size ) [2 references] ; Takes in an argument of type building, and returns how many
    objects comprise the building (as an integer).
(color ) [26 references] ; Take in an argument of type object, and returns the color of the
    object (as a color type object).
(distance ) [114 references] ; Takes in two arguments of type object, and returns the distance
    between the two objects (as a floating point number).
(distance_side ) [5 references] ; Similarly to the adjacent_side predicate, but applied to
    distance. Takes in three or four arguments, either <obj1> <side1> <obj2> or <obj1> <side1> <
    obj2> <side2>, and returns the distance between the first object on the side specified to
    the second object (optionally to its specified side).
(type ) [2 references] ; Takes in an argument of type object, and returns the type of the
    object (as a string).
(x_position ) [4 references] ; Takes in an argument of type object, and returns the x position
    of the object (as a floating point number).
```

1.8 Types

The types are also not defined as part of the DSL, but we envision them as operating similarly to the predicates.

The following describes all types currently found in our game dataset:

```
game_object [33 references] ; Parent type of all objects
agent [87 references] ; The agent
building [21 references] ; Not a real game object, but rather, a way to refer to structures the
    agent builds
----------  Blocks  ----------
block [27 references] ; Parent type of all block types:
bridge_block [11 references]
cube_block [40 references]
blue_cube_block [8 references]
tan_cube_block [1 reference]
yellow_cube_block [8 references]
flat_block [5 references]
pyramid_block [14 references]
blue_pyramid_block [3 references]
red_pyramid_block [2 references]
triangle_block [3 references]
yellow_pyramid_block [2 references]
cylindrical_block [12 references]
tall_cylindrical_block [7 references]
----------  Balls  ----------
ball [40 references] ; Parent type of all ball types:
beachball [23 references]
basketball [18 references]
dodgeball [110 references]
blue_dodgeball [6 references]
red_dodgeball [4 references]
pink_dodgeball [18 references]
golfball [28 references]
green_golfball [2 references]
----------  Colors  ----------
color [6 references] ; Likewise, not a real game object, mostly used to refer to the color of
   the rug under an object
blue [1 reference]
brown [1 reference]
green [5 references]
pink [14 references]
orange [3 references]
purple [4 references]
red [8 references]
tan [1 reference]
white [1 reference]
yellow [14 references]
----------  Other moveable/interactable objects  ----------
alarm_clock [8 references]
book [11 references]
blinds [2 references] ; The blinds on the windows
chair [17 references]
cellphone [6 references]
cd [6 references]
credit_card [1 reference]
curved_wooden_ramp [17 references]
desktop [6 references]
doggie_bed [26 references]
hexagonal_bin [124 references]
key_chain [5 references]
lamp [2 references]
laptop [7 references]
main_light_switch [3 references] ; The main light switch on the wall
mug [3 references]
triangular_ramp [10 references]
green_triangular_ramp [1 reference]
pen [2 references]
```

```
pencil [2 references]
pillow [12 references]
teddy_bear [14 references]
watch [2 references]
----------  Immoveable objects  ----------
bed [48 references]
corner [N/A references] ; Any of the corners of the room
south_west_corner [2 references] ; The corner of the room where the south and west walls meet
door [9 references] ; The door out of the room
desk [40 references]
desk_shelf [2 references] ; The shelves under the desk
drawer [5 references] ; Either drawer in the side table
top_drawer [6 references] ; The top of the two drawers in the nightstand near the bed.
floor [24 references]
rug [37 references]
shelf [10 references]
bottom_shelf [1 reference]
top_shelf [5 references]
side_table [4 references] ; The side table/nightstand next to the bed
sliding_door [2 references] ; The sliding doors on the south wall (big windows)
east_sliding_door [1 reference] ; The eastern of the two sliding doors (the one closer to the
    desk)
wall [17 references] ; Any of the walls in the room
north_wall [1 reference] ; The wall with the door to the room
south_wall [1 reference] ; The wall with the sliding doors
west_wall [2 references] ; The wall the bed is aligned to
----------  Non-object-type predicate arguments  ----------
back [3 references]
front [8 references]
left [2 references]
right [2 references]
sideways [3 references]
upright [10 references]
upside_down [2 references]
front_left_corner [1 reference] ; The front-left corner of a specific object (as determined by
    its front)
```

2 Modal Definitions in Linear Temporal Logic

2.1 Linear Temporal Logic definitions

Linear Temporal Logic (LTL) offers the following operators, and using φ and ψ as the symbols (in our case, predicates). I'm trying to translate from standard logic notation to something that makes sense in our case, where we're operating sequence of states $S_0, S_1, \cdots, S_n$.

- **Next**, $X\psi$: at the next timestep, ψ will be true. If we are at timestep i, then $S_{i+1} \vdash \psi$
- **Finally**, $F\psi$: at some future timestep, ψ will be true. If we are at timestep i, then $\exists j > i : S_j \vdash \psi$
- **Globally**, $G\psi$: from this timestep on, ψ will be true. If we are at timestep i, then $\forall j : j \geq i : S_j \vdash \psi$
- **Until**, $\psi U \varphi$: ψ will be true from the current timestep until a timestep at which φ is true. If we are at timestep i, then $\exists j > i : \forall k : i \leq k < j : S_k \vdash \psi$, and $S_j \vdash \varphi$.
- **Strong release**, $\psi M \varphi$: the same as until, but demanding that both ψ and φ are true simultaneously: If we are at timestep i, then $\exists j > i : \forall k : i \leq k \leq j : S_k \vdash \psi$, and $S_j \vdash \varphi$.

  Aside: there's also a **weak until**, $\psi W \varphi$, which allows for the case where the second is never true, in which case the first must hold for the rest of the sequence. Formally, if we are at timestep i, *if* $\exists j > i : \forall k : i \leq k < j : S_k \vdash \psi$, and $S_j \vdash \varphi$, and otherwise, $\forall k \geq i : S_k \vdash \psi$. Similarly there's **release**, which is the similar variant of strong release. We're leaving those two as an aside since we don't know we'll need them.

2.2 Satisfying a (then ...) operator

Formally, to satisfy a preference using a (then ...) operator, we're looking to find a sub-sequence of $S_0, S_1, \cdots, S_n$ that satisfies the formula we translate to. We translate a (then ...) operator by translating the constituent sequence-functions (once, hold, while-hold)[1] to LTL. Since the translation of each individual sequence function leaves the last operand empty, we append a 'true' ($\top$) as the final operand, since we don't care what happens in the state after the sequence is complete.

(once ψ) := $\psi X \cdots$

(hold ψ) := $\psi U \cdots$

(hold-while ψ α β $\cdots \nu$) := $(\psi M \alpha) X (\psi M \beta) X \cdots X (\psi M \nu) X \psi U \cdots$ where the last $\psi U \cdots$ allows for additional states satisfying ψ until the next modal is satisfied.

For example, a sequence such as the following, which signifies a throw attempt:

```
(then
    (once (agent_holds ?b))
    (hold (and (not (agent_holds ?b)) (in_motion ?b)))
    (once (not (in_motion ?b)))
)
```

Can be translated to LTL using ψ := (agent_holds ?b), φ := (in_motion ?b) as:

$\psi X (\neg\psi \wedge \varphi) U (\neg\varphi) X \top$

Here's another example:

```
(then
    (once (agent_holds ?b))   α
    (hold-while
        (and (not (agent_holds ?b)) (in_motion ?b))  β
        (touch ?b ?r)  γ
    )
    (once  (and (in ?h ?b) (not (in_motion ?b))))  δ
)
```

If we translate each predicate to the letter appearing in blue at the end of the line, this translates to:

$\alpha X (\beta M \gamma) X \beta U \delta X \top$

2.3 Satisfying (at-end ...) operators

Thankfully, the other type of temporal specification we find ourselves using as part of preferences is much simpler to translate. Satisfying an (at-end ...) operator does not require any temporal logic, since the predicate it operates over is evauated at the terminal state of gameplay.

[1] These are the ones we've used so far in the interactive experiment dataset, even if we previously defined other ones, too.