Game Creation DSL

Guy Davidson

January 4, 2022

1 DSL Docuemntation as of January 4, 2022

Color-coding

- **Undefined terms (red)**: a term that appears somewhere that I forgot to provide a definition for
- **Unused terms (gray)**: a term that appears in the definitions I documented but does not appear in any games
- **New terms (teal)**: a term that appears for the first time in the newest batch of games I translated

1.1 Game Definition

⟨*game*⟩ ::= (define (game ⟨*name*⟩)
(:domain ⟨*name*⟩)
(:setup ⟨*setup*⟩)
(:constraints ⟨*constraints*⟩)
(:terminal ⟨*terminal*⟩)
(:scoring ⟨*scoring*⟩)
)

⟨*name*⟩ ::= /[A-z]+(_[A-z0-9]+)*/ `#` 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 I 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 me 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*⟩$^+$)
| (or ⟨*setup*⟩ ⟨*setup*⟩$^+$)
| (not ⟨*setup*⟩)
| (exists (⟨*typed list(variable)*⟩) ⟨*setup*⟩)
| (forall (⟨*typed list(variable)*⟩) ⟨*setup*⟩)
| ⟨*setup-statement*⟩

⟨*setup-statement*⟩ ::= (game-conserved ⟨*setup-predicate*⟩)
| (game-optional ⟨*setup-predicate*⟩)

⟨*setup-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*⟩ ::= (⟨*comp-op*⟩ ⟨*function-eval-or-number*⟩ ⟨*function-eval-or-number*⟩)
| (= ⟨*function-eval-or-number*⟩$^+$)

⟨*comp-op*⟩ ::= < | <= | = | > | >=

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

⟨*function-eval*⟩ ::= (⟨*name*⟩ ⟨*function-term*⟩$^+$)

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

⟨*variable-list*⟩ ::= (⟨*variable-type-def*⟩$^+$)

⟨*variable-type-def*⟩ ::= ⟨*variable*⟩$^+$ - ⟨*type-def*⟩

⟨*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*⟩

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

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

⟨*predicate-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩ | ⟨*predicate*⟩ # In at least one case, I wanted to have a predicate act on other predicates, but that doesn't really make sense. See the discussion of the (side ...) predicate below.

1.3 Preferences

PDDL calls their temporal preferences 'constraints', but that's not entirely the right name for us. Maybe we should rename?

Any syntax elements that are defined (because at some point a game needed them) but are currently unused (in the interactive games) will appear in gray .

⟨*constraints*⟩ ::= ⟨*pref-def*⟩ | (and ⟨*pref-def*⟩$^+$)

⟨*pref-def*⟩ ::= ⟨*pref-forall*⟩ | ⟨*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*⟩)

⟨*preference-quantifier*⟩ ::= (exists (⟨*variable-list*⟩) ⟨*preference-body*⟩
| (forall (⟨*variable-list*⟩) ⟨*preference-body*⟩)
| ⟨*preference-body*⟩)

⟨*preference-body*⟩ ::= ⟨*then*⟩ | ⟨*at-end*⟩ | ⟨*always*⟩

⟨*at-end*⟩ ::= (at-end ⟨*pref-predicate*⟩)

⟨*always*⟩ ::= (always ⟨*pref-predicate*⟩)

⟨*then*⟩ ::= (then ⟨*seq-func*⟩ ⟨*seq-func*⟩$^+$)

⟨*seq-func*⟩ ::= ⟨*once*⟩ | ⟨*once-measure*⟩ | ⟨*hold*⟩ | ⟨*hold-while*⟩ | ⟨*hold-for*⟩ | ⟨*hold-to-end*⟩
| ⟨*forall-seq*⟩

⟨*once*⟩ ::= (once ⟨*pref-predicate*⟩)

⟨*once-measure*⟩ ::= (once ⟨*pref-predicate*⟩ ⟨*f-exp*⟩)

⟨*hold*⟩ ::= (hold ⟨*pref-predicate*⟩)

⟨*hold-while*⟩ ::= (hold-while ⟨*pref-predicate*⟩ ⟨*pref-predicate*⟩$^+$)

⟨*hold-for*⟩ ::= (hold-for ⟨*number*⟩ ⟨*pref-predicate*⟩)

⟨*hold-to-end*⟩ ::= (hold-to-end ⟨*pref-predicate*⟩)

⟨*forall-seq*⟩ ::= (forall-sequence (⟨*variable-list*⟩) ⟨*then*⟩)

⟨*pref-predicate*⟩ ::= ⟨*pref_predicate_and*⟩
| ⟨*pref-predicate-or*⟩
| ⟨*pref-predicate-not*⟩
| ⟨*pref-predicate-exists*⟩
| ⟨*pref-predicate-forall*⟩
| ⟨*predicate*⟩ ⟨*f-comp*⟩

⟨*pref-predicate-and*⟩ ::= (and ⟨*pref-predicate*⟩$^+$)

⟨*pref-predicate-or*⟩ ::= (or ⟨*pref-predicate*⟩$^+$)

⟨*pref-predicate-not*⟩ ::= (not ⟨*pref-predicate*⟩)

⟨*pref-predicate-exists*⟩ ::= (exists ⟨*variable-list*⟩ ⟨*pref-predicate*⟩)

⟨*pref-predicate-forall*⟩ ::= (forall ⟨*variable-list*⟩ ⟨*pref-predicate*⟩)

⟨*f-comp*⟩ ::= (⟨*comp-op*⟩ ⟨*function-eval-or-number*⟩ ⟨*function-eval-or-number*⟩)
| (= ⟨*function-eval-or-number*⟩$^+$)

⟨*comp-op*⟩ ::= ⟨ | ⟨= | = | ⟩ | ⟩=

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

⟨*function-eval*⟩ ::= (⟨*name*⟩ ⟨*function-term*⟩$^+$)

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

⟨*variable-list*⟩ ::= (⟨*variable-type-def*⟩$^+$)

⟨*variable-type-def*⟩ ::= ⟨*variable*⟩$^+$ - ⟨*type-def*⟩

⟨*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*⟩

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

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

⟨*predicate-term*⟩ ::= ⟨*name*⟩ | ⟨*variable*⟩ | ⟨*predicate*⟩ # In at least one case, I wanted to have a predicate act on other predicates, but that doesn't really make sense. See the discussion of the (side ...) predicate below.

1.4 Terminal Conditions

There's always assumed to be a time limit after which the game is over if nothing else, but some participants specified other terminal conditions.

⟨*terminal*⟩ ::= (and ⟨*terminal*⟩$^+$)
| (or ⟨*terminal*⟩+)
| (not ⟨*terminal*⟩)
| ⟨*terminal-comp*⟩

⟨*terminal-comp*⟩ ::= (⟨*comp-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩)

⟨*comp-op*⟩ ::= < | <= | = | > | >=

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

1.5 Scoring

PDDL calls their equivalent section (:metric ...), but I renamed because it made more sense to me.
Any syntax elements that are defined (because at some point a game needed them) but are currently unused (in the interactive games) will appear in gray .

⟨*scoring*⟩ ::= (maximize ⟨*scoring-expr*⟩)
| (minimize ⟨*scoring-expr*⟩)

⟨*scoring-expr*⟩ ::= (⟨*multi-op*⟩ ⟨*scoring-expr*⟩$^+$)
| (⟨*binary-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩)
| (- ⟨*scoring-expr*⟩)
| (total-time)
| (total-score)
| ⟨*scoring-comp*⟩
| ⟨*preference-eval*⟩

⟨*scoring-comp*⟩ ::= (⟨*comp-op*⟩ ⟨*scoring-expr*⟩ ⟨*scoring-expr*⟩)
| (= ⟨*scoring-expr*⟩$^+$)

⟨*preference-eval*⟩ ::= ⟨*count-nonoverlapping*⟩
| ⟨*count-once*⟩
| ⟨*count-once-per-objects*⟩
| ⟨*count-longest*⟩
| ⟨*count-shortest*⟩
| ⟨*count-increasing-measure*⟩
| ⟨*count-unique-positions*⟩

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

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

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

⟨*count-longest*⟩ ::= (count-longest ⟨*name*⟩) # count the longest (by number of states) satisfication of this preference

⟨*count-shortest*⟩ ::= (count-shortest ⟨*name*⟩) # count the shortest satisfication of this preference

⟨*count-total*⟩ ::= (count-total ⟨*name*⟩) # count how many states in total satisfy this preference

⟨*count-increasing-measure*⟩ ::= (count-increasing-measure ⟨*name*⟩) # currently unused, will clarify definition if it surfaces again

⟨*count-unique-positions*⟩ ::= (count-unique-positions ⟨*name*⟩) # 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 ⟨*name*⟩) # 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 ⟨*name*⟩) # For the externally quantified object with the most satisfcations, count non-overlapping satisfactions of this preference

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

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

⟨*pref-object-type*⟩ ::= : ⟨*name*⟩

1.6 Predicates

The predicates are not defined as part of the DSL, but rather I 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 the interactive experiment games. Any predicates I forgot to provide a description for will appear in red .

```
(= <arg1> <arg2>) [1 reference] ; Are these two objects the same object?
(above <arg1> <arg2>) [1 reference] ; Is the first object above the second object?
(adjacent <arg1> <arg2>) [43 references] ; Are the two objects adjacent? [will probably be
    implemented as distance below some threshold]
(agent_crouches ) [2 references] ; Is the agent crouching?
(agent_holds <arg1>) [137 references] ; Is the agent holding the object?
(broken <arg1>) [1 reference] ; Is the object broken?
(equal_z_position <arg1> <arg2>) [3 references] ; Are these two objects (approximately) in the same z position?
    (in Unity x, z are spatial coordinates, y is the height)
(faces <arg1> <arg2>) [4 references] ; Is the front of the first object facing the front of the second object?
(in <ambiguous arguments>) [51 references] ; Is the second argument inside the first argument?
    [a containment check of some sort, for balls in bins, for example]
(in_building <arg1>) [1 reference] ; Is the object part of a building?  I dislike this predicate,
    which I previously used as a crutch, and I am trying to find alternatives around it
(in_motion <arg1>) [135 references] ; Is the object in motion?
(object_orientation <arg1> <arg2>) [2 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>) [77 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>) [5 references] ; Is the color of the rug under the object (first argument) the
    color specified by the second argument?
(side <arg1> <arg2>) [8 references] ;  This is not truly a predicate, and requires a more tight solution.
    I so far used it as a crutch to specify that two particular sides of objects are adjacent, for example
    (adjacent (side ?h front) (side ?c back)).  But that makes (side <object> <side-def>) a function returning an
    object, not a predicate, where <side-def> is front, back, etc..  Maybe it should be something like
    (adjacent-side <object1> <side-def1> <object2> <side-def2>)?
(toggled_on <arg1>) [3 references] ; Is this object toggled on?
(touch <arg1> <arg2>) [28 references] ; Are these two objects touching?
(type <arg1> <arg2>) [8 references] ; Is the first argument, an object, an instance of the type
     specified by the second argument?
```

1.7 Types

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

The following describes all types currently found in the interactive experiment games. Any types I forgot to provide a description for will appear in red .

```
game_object [14 references]
agent [42 references]
building [10 references]
----------  Blocks  ----------
block [10 references]
bridge_block [4 references]
cube_block [16 references]
flat_block [1 reference]
pyramid_block [5 references]
cylindrical_block [3 references]
tall_cylindrical_block [1 reference]
----------  Balls  ----------
ball [18 references]
beachball [13 references]
basketball [11 references]
dodgeball [61 references]
blue_dodgeball [6 references]
pink_dodgeball [18 references]
golfball [14 references]
green_golfball [2 references]
----------  Colors  ----------
color [4 references]
green [4 references]
pink [10 references]
orange [3 references]
purple [4 references]
red [8 references]
white [1 reference]
yellow [8 references]
----------  Other moveable/interactable objects  ----------
alarm_clock [4 references]
book [5 references]
blinds [2 references]
chair [5 references]
cellphone [5 references]
cd [3 references]
credit_card [1 reference]
curved_wooden_ramp [10 references]
desktop [4 references]
doggie_bed [9 references]
hexagonal_bin [54 references]
key_chain [4 references]
```

```
lamp [1 reference]
laptop [4 references]
main_light_switch [2 references]
mug [3 references]
triangular_ramp [4 references]
green_triangular_ramp [1 reference]
pillow [4 references]
teddy_bear [6 references]
watch [1 reference]
----------  Immoveable objects  ----------
bed [21 references]
door [5 references]
desk [18 references]
drawer [4 references]
top_drawer [6 references]
floor [9 references]
rug [18 references]
shelf [4 references]
side_table [3 references]
sliding_door [2 references]
wall [13 references]
south_wall [1 reference]
west_wall [2 references]
----------  Non-object-type predicate arguments  ----------
back [3 references]
front [3 references]
left [1 reference]
right [1 reference]
upright [1 reference]
upside_down [1 reference]
```

2 Modal Definitions in LTL

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. I'm leaving those two as an aside since I don't know we'll need them.

Formally, to satisfy a (then ...) preference, 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) := $\psi U \cdots$

[1] These are the ones I've needed so far with the interactive games – if we find ourselves needing additional ones, I'll translate those, too.

(hold-while $\psi\ \alpha\ \beta \cdots \nu$) := $(\psi M \alpha) X (\psi M \beta) X \cdots X (\psi M \nu) X \psi U \cdots$ 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:

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

I'll attempt to check slightly more formally at some point, but I don't think we end up with many structures that are more complex than this. The predicate end up being rather more complex, but that doesn't matter to the LTL translation.

3 Modal Definitions

- These definitions attempt to offer precision on how the (then ...) operator works. It receives a series of sequence-functions (once, hold, etc.), each of which is parameterized by one or more predicate conditions.
- For the inner sequence-functions, I used the parentheses notation to mean "evaluated at these timesteps" – does this notation make sense? Should I also use it for the entire then-expression?
- I've only provided here the for the ones currently used in the interactive experiment.

(then $\langle SF_1 \rangle\ \langle SF_2 \rangle \cdots \langle SF_n \rangle$) := $\exists t_0 \leq t_1 < t_2 < \cdots < t_n$ such that $SF_1(t_0, t_1) \wedge SF_2(t_1, t_2) \wedge \cdots \wedge SF_n(t_{n-1}, t_n) =$ true, that is, each seq-func evaluated at these timesteps evaluates to true.

(once $\langle C \rangle)(t_{i-1}, t_i) := t_i = t_{i-1} + 1, S[t_i] \vdash C$, that is, the condition C holds at the next timestep from the previous assigned timestep.

(hold $\langle C \rangle)(t_{i-1}, t_i) := \forall t : t_{i-1} < t \leq t_i, S[t] \vdash C$, that is, the condition holds for all timesteps starting immediately after the previous timestep and until the current timestep.

(hold-while $\langle C \rangle\ \langle C_a \rangle \cdots \langle C_m \rangle)(t_{i-1}, t_i) := \forall t : t_{i-1} < t \leq t_i, S[t] \vdash C$ and $\exists t_a, \cdots, t_m : t_{i-1} < t_a < \cdots < t_m < t_i$ such that $S[t_a] \vdash C_a, \cdots, S[t_m] \vdash C_m$, that is, the same as hold happens, and while this condition C holds, there exist non-overlapping states in sequence where each of the additional conditions provided hold for at least a single state.

4 Open Questions

- Do we want to define syntax to quantify streaks? Some participants will use language like "every three successful scores in a row get you a point". An alternative to defining syntax or sequences would be to define the preference to count three successful attempts in a row, but that might be more awkward?
- How do we want to work with type hierarchy, such as block or ball being the super-types for all blocks or balls – is it an implicit (either ...) over all of the sub-types? Or do we want to provide the hierarchy in some way to the model, perhaps as part of the enumeration of all valid types in a given environment/scene?
- (I'm sure there are more open questions – will add later)