The Young Modeler's Handbook

David V. Pynadath

September 5, 2013

I never satisfy myself until I can make a mechanical model of a thing.
If I can make a mechanical model I can understand it.

LORD KELVIN

1 Worlds

O brave new world,
That has such people in't.

THE TEMPEST, ACT 5, SCENE 1

This is a world.

```
from psychsim.world import World

world = World()
```

A world has people.

```
from psychsim.agent import Agent

rufus = Agent(Rufus)
world.addAgent(rufus)
```

And groups of people.

```
free = Agent('Freedonia')
world.addAgent(free)
```

And killer robots.

```
world.addAgent(Agent('Klaatu'))
```

These are *agents*, because they have names.

```
for name in world.agents.keys():
   print world.agents[name]
```

But names are just string labels, so they're not a good reason to make agents special.

1.1 State

The *state* of a simulation captures the dynamic process by which it evolves. As in a *factored MDP*, we divide an overall state vector, $\vec{S}$, into a set of orthogonal features, $S_0 \times S_1 \times \cdots \times S_n$. Each feature captures a distinct aspect of the simulation state.

```
from psychsim.pwl import *
s = KeyedVector({'S_0': 0.3, 'S_1': 0.7})
s['S_n'] = 0.4
for key in s.keys():
   print key,s[key]
```

Notice that PsychSim allows you to refer to each feature by a meaningful *key*, as in Python's dictionary keys. Keys are treated internally as unstructured strings, but you may find it useful to make use of the the following types of structured keys.

1.1.1 Unary Keys

It can be useful to describe a state feature as being local to an individual agent. Doing so does *not* limit the dependency or influence of the state feature in any way. However, it can be useful to define state features as local to an agent to make the system output more readable. For example, the following defines a state feature "troops" for the previously defined agent "Freedonia".

```
world.defineState(free.name,'troops')
```

For state features which are not local to any agent, a null agent name (i.e., `None`) indicates that the state feature will pertain to the world as a whole. Again, from the system's perspective, this makes no difference, but it can be useful to distinguish global and local state in presentation.

```
world.defineState(None,'treaty)
```

Thus, one reason for creating an agent is to group a set of such state features under a common name, as opposed to leaving them as part of the global world state.

1.1.2 Binary Keys

There can also be state that pertains to the *relationship* between two agents.

```
world.defineRelation(free.name,sylv.name,'trusts')
```

The order in which the agents appear in this definition *does* matter, as reversing the order will generate a reference to a different element of the state vector.

2 Actions

The most common reason for creating an agent is because there is an entity that can take *actions* that change the state of the world. If an entity has a deterministic effect on the world, you can define a single action for it. However, agents typically have multiple actions, and it is the decision among them that is the focus of the simulation.

2.1 Atomic Actions

The `verb` of an individual action is a required field when defining the action.

```
freeReject = free.addAction({'verb': 'reject'})
```

The action created will also have a `subject` field, representing the agent who is performing this action. The `subject` field is automatically filled in with the name of the agent. A third optional field, `object`, can represent the target of the specific action.

```
freeBattle = free.addAction({'verb': 'attack','object': sylv.name})
```

You are free to define any other fields as well to contain other parameterizations of the actions.

```
free.addAction({'verb': 'offer','object': sylv.name,'amount': 50})
```

We will describe the use of these fields in Section 3.2.

2.2 Action Sets

Sometimes an agent can make a decision that simultaneously combines multiple actions into a single choice.

```
freeRejectAndAttack = free.addAction(set([{'verb': 'attack','object
                                          {'verb': 'reject'}]))
```

For the purposes of the agent's decision problem, this option is equivalent to a single atomic action (e.g., `reject-and-attack`). However, as we will see in Section 3.2, separate atomic actions can sometimes simplify the definition of the effects of such a combined action.

When inspecting an agent's action choice, each option is an `ActionSet` instance, supporting all standard Python `set` operations.

```
for action in free.actions:
   print len(action)
freeRejectAndAttack = freeReject | freeBattle
```

As we will see in Section 3.1, not all of the agent's choices may be legal under all circumstances. Rather than inspecting the `actions` attribute itself, we typically examine the context-specific set of action choices instead.

```
for vector in world.state.domain():
   print free.getActions(vector)
```

3 Variables

By default, a state feature is assumed to be real-valued, in $[-1, 1]$. However, both the "defineState" and "defineRelation" methods take optional arguments to modify the valid ranges of the feature. The range of possible values does not affect the execution of the simulation, as simulation normalizes the values back to the $[-1, 1]$ range during decision-making.

```
world.defineState(free.name,'troops',lo=0,hi=50000)
```

It is also possible to specify that a state feature take on only integral values with the optional "domain" argument.

```
world.defineState(free.name,'troops',int,lo=0,hi=50000)
```

One can also define a boolean state feature using the same argument, in which case the optional "lo" and "hi" arguments are obviously moot.

```
world.defineState(None,'treaty,bool)
```

It is also possible to define an enumerated list of possible state features.

```
phases = ['offer','respond','rejection','end','paused',
          'engagement']
world.defineState(None,'phase',list,phases)
```

The domains also work within "defineRelation" calls as well. In other words, relationships can take on the same sets of values as unary state features.

3.1 Legality

```
tree = makeTree({'if': equalRow(stateKey(None,'phase'),'offer'),
                 True: True,
                 False: False})
free.setLegal(action,tree)
```

3.2 Dynamics

3.3 Termination

Termination conditions specify when scenario execution should reach an absorbing end state (e.g., when a final goal is reached, when time has expired). A termination condition is a PWL function (Section 7) with boolean leaves.

```
world.addTermination(makeTree({'if': trueRow(stateKey(None,'treaty'
                                  True: True, False: False}))
```

This condition specifies that the simulation ends if a "treaty" is reached. Multiple conditions can be specified, with termination occurring if any condition is true.

4 Reward

```
goalFTroops = maximizeFeature(stateKey(free.name,'troops'))
free.setReward(goalFTroops,1.)
goalFTerritory = maximizeFeature(stateKey(free.name,'territory'
free.setReward(goalFTerritory,1.)
```

5 Models

A *model* in the PsychSim context is a potential configuration of an agent that may apply in certain worlds or decision-making contexts. All agents have a "True" model that represents their real configuration, which forms the basis of all of their decisions during execution.

It also possible to specify alternate models that represent perturbations of this true model, either to represent the dynamics of the agent's configuration or to represent the perceptions other agents have of it.

```
free.addModel('friend')
```

5.1 Model Attribute: `static`

6 Observations

7 PWL functions

8 Executing a Scenario

```
world.step()
```

9 Algorithms

9.1 Decision Making

9.2 Belief Update

World has $\Pr(M_0)$

World generates an observation ω
But M_0 cannot generate ω

$$\begin{aligned} \Pr(M_1 = m) &= \Pr(M_1 = m | M_0, \omega) && (1) \\ &= \frac{\Pr(M_1 = m, \omega | M_0)}{\Pr(\omega | M_0)} && (2) \\ &\propto && (3) \end{aligned}$$

And done.