Using a Deontic Logic to Model Social Practices

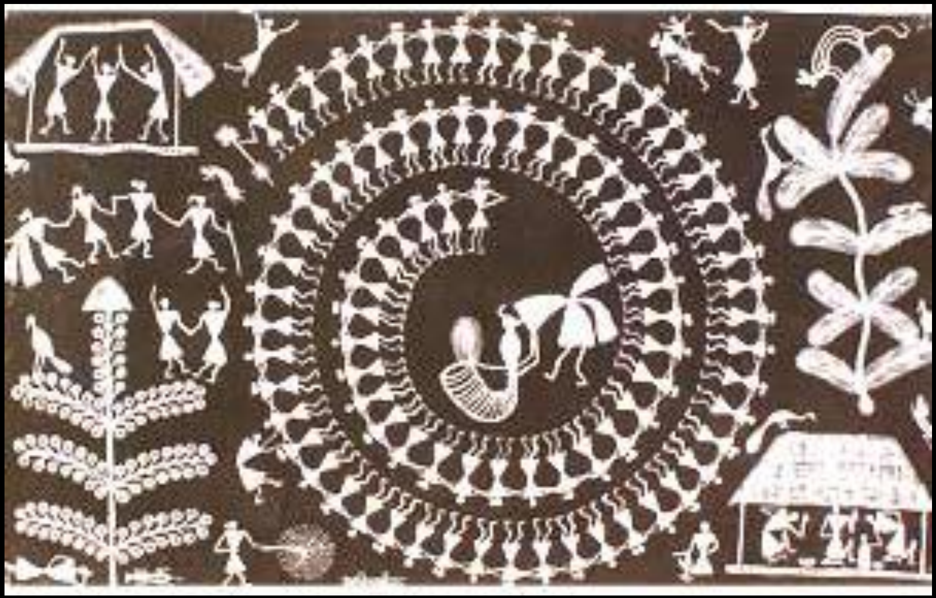
Sim Tribe

You are the leader of a tribe

You can make any society you want

Any society you want

- Worship the Chicken God
- Polygamy for men, monogamy for women
- Eating is taboo



Example 1: Eating is Taboo

- > Demo

Modeling Social Practices

- Game-state is a blackboard of assertions
- “The world is everything that is the case”
- A practice is a set of declarative conditionals

Eating is Taboo

```
rules
      // Eating is taboo in the presence of others!
      x:Agent /\ y:Agent /\ Engine:Test(SameLot, x, y) -> x:CanEat.False

      // Punishment
      x:Agent /\ y:Agent /\ Motor:Animate:x.Eat /\ x:CanEat.False /\ Engine:Test(SameLot, x, y) -> [] y:Act.Punish.x
      [] x:Act.Punish.y -> [] Motor:PointAt:x.y /\ [] Motor:LookAt:x.y.At /\ [] Motor:Animate:x.Punish.y

    // When you should eat
      x:Stomach:Filled.Empty /\ x:NeedsToEat /\ y:Edible /\ y:Health.Alive /\ x:CanEat.True -> [] x:Act.Eat.y

      // What is edible
      x:Apple -> x:Edible
      x:Biscuit -> x:Edible
      x:Sheep -> x:Edible

      // How to eat
      [] x:Act.Eat.y -> [] Motor:Route:x.y /\ [] Motor:LookAt:x.y.At
      [] x:Act.Eat.y /\ Engine:Test(IsNear, x, y)  /\ Engine:Test(IsWatching, x, y)  -> [] Motor:Animate:x.Eat.y

      // Stomach
      x:Agent /\ Motor:Animate:x.Eat.y.Succeeded.t ->  x:Stomach:Filled.Full /\ x:Stomach:FilledWhen.t /\ y:Health.Dead /\ [] Engine:Destroy(y)
    x:Agent /\ x:Stomach:FilledWhen.y /\ Engine:Test(Elapsed, y, T1000) -> x:Stomach:Filled.Empty

      // Tummy is initially empty
      Begin /\ x:Agent -> x:Stomach:FilledWhen.T0

      // Sheep start off alive
      Begin /\ x:Sheep -> x:Health.Alive

      // People are permitted to eat by default
      x:Agent -> x:CanEat.True
end
```

Eating is Taboo

```
x:Stomach:Filled.Empty &&
x:NeedsToEat &&
y:Edible &&
y:Health.Alive &&
x:CanEat.True
->
[] x:Act.Eat.y
```

Eating is Taboo

```
x:Agent &&
y:Agent &&
SameLot:x:y
->
x:CanEat.False
```

Eating is Taboo

```
x:Agent
&& y:Agent
&& Motor:Animate:x.Eat
&& x:CanEat.False
&& SameLot:x:y
->
[] y:Act.Punish.x
```

Eating is Taboo

```
rules
    // Eating is taboo in the presence of others!
    x:Agent /\ y:Agent /\ Engine:Test(SameLot, x, y) -> x:CanEat.False

    // Punishment
    x:Agent /\ y:Agent /\ Motor:Animate:x.Eat /\ x:CanEat.False /\ Engine:Test(SameLot, x, y) -> [] y:Act.Punish.x
    [] x:Act.Punish.y -> [] Motor:PointAt:x.y /\ [] Motor:LookAt:x.y.At /\ [] Motor:Animate:x.Punish.y

  // When you should eat
    x:Stomach:Filled.Empty /\ x:NeedsToEat /\ y:Edible /\ y:Health.Alive /\ x:CanEat.True -> [] x:Act.Eat.y

    // What is edible
    x:Apple -> x:Edible
    x:Biscuit -> x:Edible
    x:Sheep -> x:Edible

    // How to eat
    [] x:Act.Eat.y -> [] Motor:Route:x.y /\ [] Motor:LookAt:x.y.At
    [] x:Act.Eat.y /\ Engine:Test(IsNear, x, y)  /\ Engine:Test(IsWatching, x, y)  -> [] Motor:Animate:x.Eat.y

    // Stomach
    x:Agent /\ Motor:Animate:x.Eat.y.Succeeded.t ->  x:Stomach:Filled.Full /\ x:Stomach:FilledWhen.t /\ y:Health.Dead /\ [] Engine:Destroy(y)
  x:Agent /\ x:Stomach:FilledWhen.y /\ Engine:Test(Elapsed, y, T1000) -> x:Stomach:Filled.Empty

    // Tummy is initially empty
    Begin /\ x:Agent -> x:Stomach:FilledWhen.T0

    // Sheep start off alive
    Begin /\ x:Sheep -> x:Health.Alive

    // People are permitted to eat by default
    x:Agent -> x:CanEat.True
end
```

Example 2: Queuing

- > Demo

Queuing

```
rules
      // People should run away from wolves
      x:Wolf.T /\ y:Participant -> [] Motor:Avoid:y:x /\ [] Motor:LookAt:y.x.At /\ [] Motor:Animate:y.Help.x

      // when the head of the queue has finished licking, the next in line gets his turn
      Queue:Head.x /\ Motor:Animate:x.Lick.z.Succeeded /\ Queue:After:x.y -> Next:Finish:x:y:z

      // When x finishes, and y is after him, y becomes the head of the queue
      Finish:x:y:z -> Queue:Head.y /\ Exiting:x:z.T

      // When x is exiting the queue, he should get out of the way of the target object
      Exiting:x:z.T -> [] Motor:Avoid:x:z

      // When he is out of range, he has finished exiting
      Exiting:x:z.T /\ Engine:Test(IsAwayFrom, x, z) -> Next:x:Participant.F /\ Next:Exiting:x:z.F

      // somebody joining the queue at the front when there is nobody else in it
      x:Participant.T /\ Queue:Contains:x.False /\ Queue:Tail.Null -> Next:AddHead.x

      // somebody joining the end of the queue
      x:Participant.T /\ Queue:Contains:x.False /\ Queue:Contains:y.True /\ y:Participant.T /\ Queue:Tail.y -> Next:AddAfter:y.x

      // We need the AddAfter intermediary as a way of expressing Next(Queue:After:x.y /\ Queue:Tail.y), but we cannot express Next(p /\ q) directly
      AddAfter:x.y -> Queue:After:x.y /\ Queue:Tail.y

      // We need the AddHead intermediary as way of expressing Next(Queue:Head.x /\ Queue:Tail.x) even though we cannot express Next(p /\ q) directly
      AddHead.x -> Queue:Head.x /\ Queue:Tail.x

      // When somebody is at the tail of the queue, there is nobody behind him and they have joined the queue
      x:Participant.T /\ Queue:Tail.x -> Queue:After:x.Null /\ Next:Queue:Contains:x.True

      // the head of the queue is tasked with licking the target
      Queue:Head.x /\ x:Participant.T /\ Queue:Target.y -> [] x:Act.Lick.y

      // a general routing action
      [] x:Act.phi.y -> [] Motor:Route:x.y /\ [] Motor:LookAt:x.y.At
      [] x:Act.phi.y /\ Engine:Test(IsNear, x, y)  /\ Engine:Test(IsWatching, x, y)  -> [] Motor:Animate:x.phi.y


      // If the head of the queue walks off, everyone is surprised
      x:Participant.T /\ y:Participant.T /\ Queue:Head.y /\ Queue:Target.z /\ Engine:Test(NotWatching, y, z) -> [] x:Act.WhereAreYouGoing.y
      [] x:Act.WhereAreYouGoing.y -> [] Motor:LookAt:x.y.At /\ [] Motor:Animate:x.WhereAreYouGoing.y

      // everyone stands behind the person in front of them
      Queue:After:x.y /\ y:Participant.T -> [] Motor:LookAt:y.x.At
      Queue:After:x.y /\ y:Participant.T /\ Engine:Test(DistanceExceeds, x, y, T100) -> [] Motor:Approach:y.x.Behind

      x:Participant.T -> Queue:Contains:x.False // initially, people are not in the queue
      Begin -> Queue:Tail.Null // initially, queue is empty
end
```

Example 3: Status

- > Demo

Communicating Status

```
rules

// Women are dominant over males
x:Sex.Male /\ y:Sex.Female -> y:Rel:x.Dominant

// Taller people are dominant over smaller people
x:Sex.s /\ y:Sex.s /\ x:Height.h1 /\ y:Height.h2 /\ Engine:Test(Greater, h1, h2) -> x:Rel:y.Dominant

// Inferior is the inverse of Dominant
y:Rel:x.Dominant -> x:Rel:y.Inferior

// If superior is looking at inferior, inferior should look down
x:Rel:y.Dominant /\ Engine:Test(IsWatching, x, y) -> [] Motor:LookAt:y.x.Down

// People who are moving are interesting to look at
SameLot:x:y /\ Engine:Test(IsMoving, x) -> [] Motor:LookAt:y:x.At

// People who are superior are interesting to look at
x:Rel:y.Inferior -> [] Motor:LookAt:x.y.At

// Dominant sim should look at the mouse (for testing)
x:Rel:y.Dominant -> [] Motor:LookAt:x.Mouse.At


end
```

Communicating Status

```
x:Rel:y.Dominant &&
IsWatching:x:y
->
[] Motor:LookAt:y.x.Down
```

Example 4: Game

- > Demo

Tic Tac Toe

```
rules
    x:Wolf.T /\ G:Other:y -> [] Motor:Avoid:y:x /\ [] Motor:LookAt:y.x.At /\ [] Motor:Animate:y.Help.x

    Square11.s /\ Square12.s /\ Square13.s /\ G:Symbol:x.s -> Next:G:HasWon:x
    Square21.s /\ Square22.s /\ Square23.s /\ G:Symbol:x.s -> Next:G:HasWon:x
    Square31.s /\ Square32.s /\ Square33.s /\ G:Symbol:x.s -> Next:G:HasWon:x
    Square11.s /\ Square21.s /\ Square31.s /\ G:Symbol:x.s -> Next:G:HasWon:x
    Square12.s /\ Square22.s /\ Square32.s /\ G:Symbol:x.s -> Next:G:HasWon:x
    Square13.s /\ Square23.s /\ Square33.s /\ G:Symbol:x.s -> Next:G:HasWon:x
    Square11.s /\ Square22.s /\ Square33.s /\ G:Symbol:x.s -> Next:G:HasWon:x
    Square13.s /\ Square22.s /\ Square31.s /\ G:Symbol:x.s -> Next:G:HasWon:x

    G:HasWon:x /\ G:Other:x.y -> G:Playing.F /\ [] Motor:Animate:x.Hooray.y /\ [] Motor:Animate:y.Sigh.x /\ [] Motor:LookAt:x.y.At /\ [] Motor:LookAt:y.x.Down

    UserInput.x.t.z /\ G:Move.x /\ G:Symbol:x.t /\ z.Empty -> Motor:Animate:x.t.z.Succeeded
    UserInput.x.t.z /\ G:Symbol:x.u /\ G:Different:t:u /\ G:Other:x.y -> [] Motor:Animate:y.NotYourSymbol.x /\ [] Motor:PointAt:y.x /\ [] Motor:LookAt:y.x.At
    UserInput.x.t.z /\ G:Other:x.y /\ G:Move.y -> [] Motor:Animate:y.ItsNotYourMove.x /\ [] Motor:PointAt:y.x /\ [] Motor:LookAt:y.x.At

    G:Playing.T /\ G:Move.x /\ square.Empty /\ G:Symbol:x.t /\ G:Other:x.y -> [] x:Act.Mark.square.t /\ [] Motor:LookAt:y.x.At
    [] x:Act.Mark.square.t -> [] Motor:Route:x.square /\ [] Motor:LookAt:x.square.At
    [] x:Act.Mark.square.t /\ Engine:Test(IsVeryNear, x, square)  -> [] Motor:Animate:x.t.square
    Motor:Animate:x.t.z.Succeeded /\ G:Symbol:x.t -> Next:G:HasMoved:x:z:t
    G:HasMoved:x:square:t /\ G:Other:x.y -> G:Move.y /\ square.t
end
```

Summary

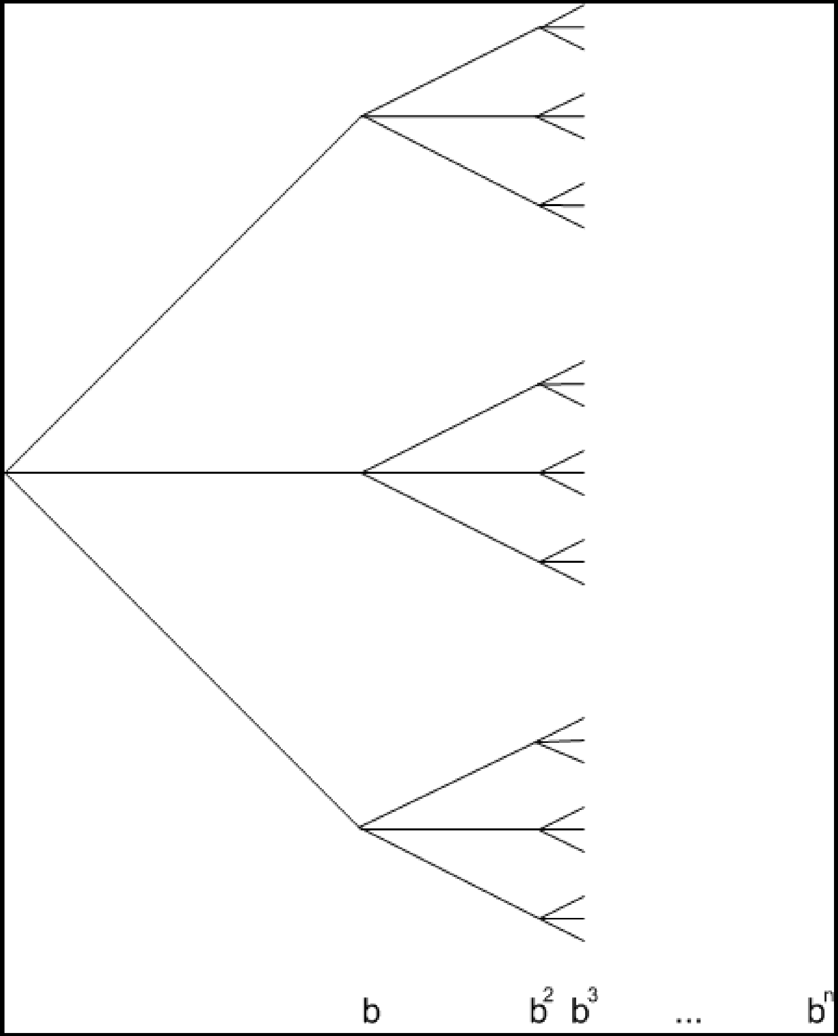
- Game-state is a blackboard of declarative assertions
- Some of these assertions are *deontic*
- A practice is a set of conditionals

Advantages of Representation

- Concise
- Robust
- Each conditional can be learned separately

Very Long Term Planning

Planning is Hard



Polynomial-Time Planning

- Instead of searching ***$O(b^n)$*** nodes, we just search ***$b*n$*** nodes!
- This means we can have very long-term plans involving hundreds of actions!
- > Demo

How it Works

Each interaction has an associated <u>tradeoff</u>

Cook Hamburger:

```
Meat + Stove + Time
->
Hunger + Cooking Skill
```

How it Works

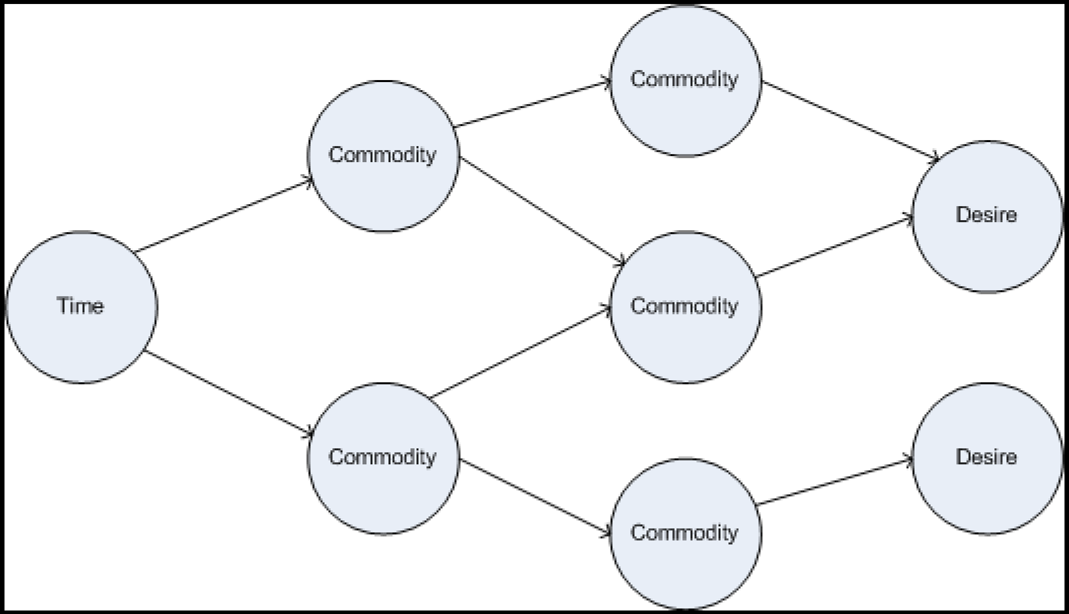
Each interaction has an associated <u>tradeoff</u>

Read Cookery Book:

```
CookeryBook + Time
->
Cooking Skill
```

How it Works

- The tradeoffs determine a commodity graph
- We work through the graph from right to left

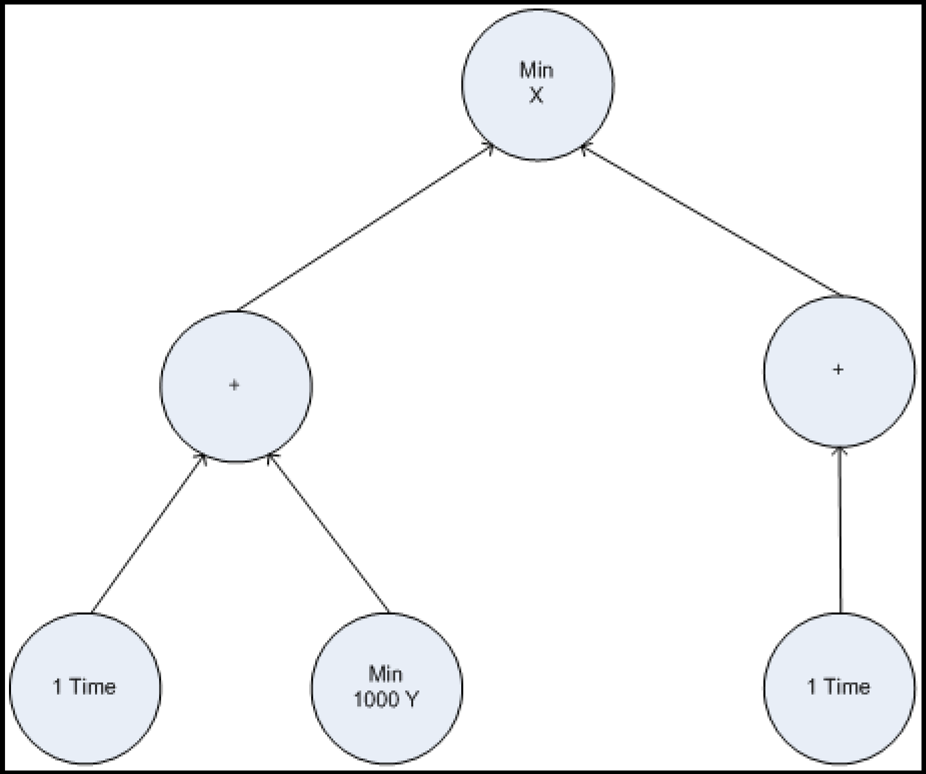


How it Works

- How do we decide how to divvy up the rhs between the lhs?
- We use the labor-value of the commodity
- The labor-value of a commodity is the amount of time it takes to acquire one unit of it
- We calculate the labor-value from the tradeoffs

Calculating Labor Value

- 1 Time + 1000 Y → 1 X
- 100 Time → 1 Y
- 1 Time → 1 X



Summary

- This planning approach is O(b*n)
- It distinguishes between the *use-value* of each commodity and the *labor-value* of each commodity
- It enables very long-term plans, involving hundreds of actions

Thanks

- Tom Barnet-Lamb (tblamb@harvard.edu)
- Peter Ingebretson (pingebretson@maxis.com)
- Thanks to Sims 3 Designers for the 2D Prototype