# Demo Light for Composing Models

Jan van Eijck

with contributions by Lakshmanan Kuppusamy and Floor Sietsma

January 18, 2011

**Abstract**

Light version of DEMO for composing epistemic models, based on the code for the ESSLLI 2008 course on Dynamic Epistemic Logic (see `http://homepages.cwi.nl/~jve/courses/esslli08/`) extended with vocabulary information [EWS10]. Factual change is also treated. The piece ends with some examples: the muddy children, and hat puzzles, dealing with the interaction of perception and change [Eijar].

# Contents

```
module DemoLight

where
import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,voc)
import ChangeVocab
import ChangePerception
```

# Chapter 1

# Models with Vocabulary

Module declaration. We will use QuickCheck [CH00] for some simple tests.

```
module ModelsVocab where

import List
import Test.QuickCheck
```

Binary relations as lists of ordered pairs:

```
type Rel a = [(a,a)]
```

Test for equality of relations:

```
sameR :: Ord a => Rel a -> Rel a -> Bool
sameR r s = sort (nub r) == sort (nub s)
```

**Operations on relations: converse** Relational converse $R^\smile$ is given by:

$$R^\smile = \{(y, x) \mid (x, y) \in R\}$$

```
cnv :: Rel a -> Rel a
cnv r = [ (y,x) | (x,y) <- r ]
```

**Operations on relations: composition** The relational composition of two relations $R$ and $S$ on a set $A$:

$$R \circ S = \{(x, z) \mid \exists y \in A(xRy \wedge ySz)\}$$

For the implementation, it is useful to declare a new infix operator for relational composition.

```
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
  nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Note that `(@@)` is the prefix version of `@@`.

Testing for Euclideanness

A relation $R$ is euclidian if $\forall xyz((Rxy \wedge Rxz) \rightarrow Ryz)$.

In other words: $R$ is euclidean iff $R^\smile \circ R \subseteq R$.

Use this for a test of Euclideanness:

$$A \subseteq B \ :\equiv \ \forall x \in A : x \in B.$$

```
containedIn :: Eq a => [a] -> [a] -> Bool
containedIn xs ys = all (\ x -> elem x ys) xs
```

```
euclR :: Eq a => Rel a -> Bool
euclR r = (cnv r @@ r) `containedIn` r
```

Test for Seriality

A relation $R$ is *serial* if $\forall x \exists y Rxy$ holds.

Here is a test:

```
serialR :: Eq a => Rel a -> Bool
serialR r =
  all (not.null)
    (map (\ (x,y) -> [ v | (u,v) <- r, y == u]) r)
```

**Testing for S5** An accessibility relation is S5 if it is an equivalence.

```
reflR :: Eq a => [a] -> Rel a -> Bool
reflR xs r =
  [(x,x) | x <- xs] `containedIn` r

symmR :: Eq a => Rel a -> Bool
symmR r = cnv r `containedIn` r

transR :: Eq a => Rel a -> Bool
transR r = (r @@ r) `containedIn` r

isS5 :: Eq a => [a] -> Rel a -> Bool
isS5 xs r = reflR xs r && transR r && symmR r
```

### Testing for KD45

An accessibility relation is KD45 if it is serial, transitive and euclidean.

```
isKD45 :: Eq a => Rel a -> Bool
isKD45 r = transR r && serialR r && euclR r
```

### Representing Epistemic Models: Agents

An infinite number of agents, with names for the first five of them:

```
data Agent = Ag Int deriving (Eq,Ord)

a,alice, b,bob, c,carol, d,dave, e,ernie  :: Agent
a = Ag 0; alice = Ag 0
b = Ag 1; bob   = Ag 1
c = Ag 2; carol = Ag 2
d = Ag 3; dave  = Ag 3
e = Ag 4; ernie = Ag 4

instance Show Agent where
  show (Ag 0) = "a"; show (Ag 1) = "b";
  show (Ag 2) = "c"; show (Ag 3) = "d" ;
  show (Ag 4) = "e";
  show (Ag n) = 'a': show n
```

### Representing Epistemic Models: Basic Propositions

```
data Prp = P Int | Q Int | R Int deriving (Eq,Ord)

instance Show Prp where
  show (P 0) = "p"; show (P i) = "p" ++ show i
  show (Q 0) = "q"; show (Q i) = "q" ++ show i
  show (R 0) = "r"; show (R i) = "r" ++ show i
```

### A Datatype for Epistemic Models

```
data EpistM state = Mo
             [state]
             [Agent]
             [Prp]
             [(state,[Prp])]
             [(Agent,state,state)]
             [state]  deriving (Eq,Show)
```

Example Epistemic Model

```
s5example :: EpistM Integer
s5example =
 Mo [0..3]
    [a,b,c]
    [P 0, Q 0]
    [(0,[]),(1,[P 0]),(2,[Q 0]),(3,[P 0, Q 0])]
    ([ (a,x,x) | x <- [0..3] ] ++
     [ (b,x,x) | x <- [0..3] ] ++
     [ (c,x,y) | x <- [0..3], y <- [0..3] ])
    [1]
```

Extracting domain, vocabulary, relations, and valuation from an epistemic model

```
dom :: EpistM a -> [a]
dom (Mo states _ _ _ _ _) = states

rel :: Agent -> EpistM a -> Rel a
rel ag (Mo states agents _ val rels actual) =
      [ (x,y) | (agent,x,y) <- rels, ag == agent ]

valuation ::  EpistM a -> [(a,[Prp])]
valuation (Mo _ _ _ val  _ _ ) = val

vocab ::  EpistM a -> [Prp]
vocab  (Mo _ _ voc _ _ _) = voc

valStat :: Eq a => a -> EpistM a -> [Prp]
valStat x (Mo states agents voc val rels actual)
   = [y | (state,[y]) <- val, state == x ]

vcbSet :: Eq a => EpistM a -> [Prp]
vcbSet  (Mo states agents voc val rels actual)
       = nub ( [y | (states,[y]) <- val] )

actual :: EpistM a -> [a]
actual (Mo _ _ _ _ _ actual) = actual
```

**From equivalence relations to partitions** Every equivalence relation $R$ on $A$ corresponds to a partition on $A$: the set $\{[a]_R \mid a \in A\}$, where $[a]_R = \{b \in A \mid (a, b) \in R\}$.

```
rel2partition :: Ord a => [a] -> Rel a -> [[a]]
rel2partition [] r = []
rel2partition (x:xs) r =
   xclass :  rel2partition (xs \\ xclass) r
     where
       xclass = x : [ y | y <- xs, elem (x,y) r ]
```

**Displaying S5 Models** The function `rel2partition` can be used to write a display function for S5 models that shows each accessibility relation as a partition, as follows.

```
showS5 :: (Ord a,Show a) => EpistM a -> [String]
showS5 m@(Mo states agents voc val rels actual) =
  show states :
  show voc    :
  show val    :
  map show [ (ag, (rel2partition states) (rel ag m))
                                      | ag <- agents ]
  ++
  [show actual]
```

Here `@` is used to introduce a shorthand or name for a datastructure.

```
displayS5 ::  (Ord a,Show a) => EpistM a -> IO()
displayS5 = putStrLn . unlines . showS5
```

**Blissful Ignorance** Blissful ignorance is the state where you don't know anything, but you know also that there is no reason to worry, for you know that nobody knows anything.

A Kripke model where every agent from agent set $A$ is in blissful ignorance about a (finite) set of propositions $P$, with $|P| = k$, looks as follows:

$$M = (W, V, R) \text{ where}$$

$$\begin{array}{lcl} W & = & \{0, \ldots, 2^k - 1\} \\ V & = & \text{ any surjection in } W \to \mathcal{P}(P) \\ R & = & \{x \stackrel{a}{\to} y \mid x, y \in W, a \in A\}. \end{array}$$

Note that $V$ is in fact a bijection, for $|\mathcal{P}(P)| = 2^k = |W|$.

**Generating Models for Blissful Ignorance**

```
initM :: [Agent] -> [Prp] -> EpistM Integer
initM ags props = (Mo worlds ags props val accs points)
  where
    worlds = [0..(2^k-1)]
    k      = length props
    val    = zip worlds (sortL (powerList props))
    accs   = [ (ag,st1,st2) | ag  <- ags,
                              st1 <- worlds,
                              st2 <- worlds      ]
    points = worlds
```

The model $\epsilon$ for blissful ignorance with an empty vocabulary:

```
epsilon :: [Agent] -> EpistM Integer
epsilon ags = initM ags []
```

powerList, sortL (sort by length)

```
powerList  :: [a] -> [[a]]
powerList  [] = [[]]
powerList  (x:xs) =
  (powerList xs) ++ (map (x:) (powerList xs))

sortL :: Ord a => [[a]] -> [[a]]
sortL  = sortBy
  (\ xs ys -> if length xs < length ys
                then LT
              else if length xs > length ys
                then GT
              else compare xs ys)
```

**General Knowledge** The general knowledge accessibility relation of a set of agents $C$ is given by

$$\bigcup_{c \in C} R_c.$$

```
genK :: Ord state => [(Agent,state,state)]
                      -> [Agent] -> Rel state
genK r ags = [ (x,y) | (ag,x,y) <- r, ag `elem` ags ]
```

### Right Section of a Relation

If $R$ is a binary relation on $A$, and $a \in A$, then $aR$ is the set

$$\{b \in A \mid aRb\}.$$

```
rightS :: Ord a => Rel a -> a -> [a]
rightS r x = (sort.nub) [ z | (y,z) <- r, x == y ]
```

### General Knowledge Alternatives

```
genAlts :: Ord state => [(Agent,state,state)]
        -> [Agent] -> state -> [state]
genAlts r ags = rightS (genK r ags)
```

**Closures of Relations** If $\mathcal{O}$ is a set of properties of relations on a set $A$, then the $\mathcal{O}$ closure of a relation $R$ on $A$ is *the smallest relation $S$ that includes $R$ and that has all the properties in $\mathcal{O}$.*

The closures of relations that we need are the transitive closure and the reflexive transitive closure.

**Reflexive Transitive Closure** Let a set $A$ be given. Let $R$ be a binary relation on $A$. Let $I = \{(x, x) \mid x \in A\}$.

We define $R^n$ for $n \geq 0$, as follows:

- $R^0 = I$.
- $R^{n+1} = R \circ R^n$.

Next, define $R^*$ by means of:

$$R^* = \bigcup_{n \in \mathbb{N}} R^n.$$

**Computing Reflexive Transitive Closure** If $A$ is finite, any $R$ on $A$ is finite as well. In particular, there will be $k$ with $R^{k+1} \subseteq R^0 \cup \cdots \cup R^k$.

Thus, in the finite case reflexive transitive closure can be computed by successively computing $\bigcup_{n \in \{0,..,k\}} R^n$ until $R^{k+1} \subseteq \bigcup_{n \in \{0,..,k\}} R^n$.

In other words: the reflexive transitive closure of a relation $R$ can be computed from $I$ by repeated application of the operation

$$\lambda S.(S \cup (R \circ S)),$$

until the operation reaches a fixpoint. A more efficient computation of the reflexive transitive closure of $R$ is by repeated application of the operation

$$\lambda S.(S \cup (S \circ S)),$$

starting from $I \cup R$, until the operation reaches a fixpoint.

**Least Fixpoint** A fixpoint of an operation $f$ is an $x$ for which $f(x) = x$.

Least fixpoint calculation:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

Computing Reflexive Transitive Closure, at last:

```
rtc :: Ord a => [a] -> Rel a -> Rel a
rtc xs r = lfp (\ s -> (sort.nub) (s ++ (s@@s))) ri
  where ri = nub (r ++ [(x,x) | x <- xs ])
```

Some test properties:

```
prop_RtcRefl ::  Ord a => [a] -> Rel a -> Bool
prop_RtcRefl xs r = reflR xs (rtc xs r)
```

```
prop_RtcTr ::  Ord a => [a] -> Rel a -> Bool
prop_RtcTr xs r = transR (rtc xs r)
```

**Computing Transitive Closure** Same as computing reflexive transitive closure, but starting out from the relation $R$.

```
tc :: Ord a => Rel a -> Rel a
tc r = lfp (\ s -> (sort.nub) (s ++ (s @@ s))) r
```

A test property:

```
prop_TcTr ::  Ord a => Rel a -> Bool
prop_TcTr r = transR (tc r)
```

**Computing Common Knowledge** The common knowledge relation for group of agents $C$ is the relation

$$(\bigcup_{c \in C} R_c)^*.$$

Given that the $R_c$ are represented as a list of triples

`[(Agent,state,state)]`

we can define a function that extracts the common knowledge relation:

```
commonK :: Ord state => [(Agent,state,state)]
           -> [Agent] -> [state] -> Rel state
commonK r ags xs = rtc xs (genK r ags)
```

### Common Knowledge Alternatives

```
commonAlts :: Ord state => [(Agent,state,state)]
        -> [Agent] -> [state] -> state -> [state]
commonAlts r ags xs s = rightS (commonK r ags xs) s
```

### Representing Formulas

```
data Form = Top
          | Prp Prp
          | Neg Form
          | Conj [Form]
          | Disj [Form]
          | K Agent Form
          | CK [Agent] Form
          deriving (Eq,Ord)
```

CK is the operator for common knowledge.

### Example formulas

```
p = Prp (P 0)
q = Prp (Q 0)
```

```
instance Show Form where
  show Top           = "T"
  show (Prp p)      = show p
  show (Neg f)       = '-':(show f)
  show (Conj fs)     = '&': show fs
  show (Disj fs)     = 'v': show fs
  show (K agent f)   = '[':show agent++"]"++show f
  show (CK agents f) = 'C': show agents ++ show f
```

Getting the proposition letters from a formula:

```
getPs :: Form -> [Prp]
getPs Top     = []
getPs (Prp p) = [p]
getPs (Neg f) = getPs f
getPs (Conj fs) = (sort.nub.concat) (map getPs fs)
getPs (Disj fs) = (sort.nub.concat) (map getPs fs)
getPs (K agent f)   = getPs f
getPs (CK agents f) = getPs f
```

### Valuation Lookup

```
apply :: Eq a => [(a,b)] -> a -> b
apply [] _ = error "argument not in list"
apply ((x,z):xs) y | x == y    = z
                   | otherwise = apply xs y
```

This can be used to look up the valuation for a world in a model.

### Maybe types for Booleans and Quantifiers

The following operators implement Strong Kleene evaluation in partial models ([Kle52], Chapter 12, Section 64). The three truth values are: `Nothing` for 'undefined', `Just True` for 'true', and `Just False` for 'false'. The type for these three values is `Maybe Bool`.

```
maybe_Not :: Maybe Bool -> Maybe Bool
maybe_Not Nothing = Nothing
maybe_Not (Just x) = Just (not x)

maybe_And :: [Maybe Bool] -> Maybe Bool
maybe_And [] = Just True
maybe_And (Nothing:xs) = Nothing
maybe_And ((Just True):xs) = maybe_And (xs)
maybe_And ((Just False):xs) = Just False

maybe_Or :: [Maybe Bool] -> Maybe Bool
maybe_Or [] = Just False
maybe_Or ((Just True):xs) = Just True
maybe_Or ((Just False):xs) = maybe_Or (xs)
```

`maybe_And` can be used for the definition of `maybe_All`, and `maybe_Or` can be used for the definition of `maybe_Any`.

```
maybe_All:: (Maybe Bool -> Maybe Bool) -> [Maybe Bool] ->
            Maybe Bool
maybe_All f = (maybe_And). map f

maybe_Any :: (Maybe Bool -> Maybe Bool) -> [Maybe Bool] ->
             Maybe Bool
maybe_Any f = (maybe_Or). map f
```

### Evaluation

We will use this to define partial evaluation, to take into account that a formula may use proposition letters that are not in the vocabulary of a model.

```
isTrueAtMayb :: Ord state =>
     EpistM state -> state -> Form -> Maybe Bool
isTrueAtMayb m w Top = Just True
isTrueAtMayb
 m@(Mo worlds agents voc val acc points) w (Prp p) =
 if notElem p voc then Nothing else
 if elem p (concat [props|(w',props) <- val, w'==w]) then
    (Just True)
 else Just False
isTrueAtMayb m w (Neg f)   = maybe_Not (isTrueAtMayb m w f)
isTrueAtMayb m w (Conj fs) =
        maybe_And (map (isTrueAtMayb m w) fs)
isTrueAtMayb m w (Disj fs) =
        maybe_Or  (map (isTrueAtMayb m w) fs)
```

```
isTrueAtMayb
 m@(Mo worlds agents voc val acc points) w (K ag f) =
 maybe_And (map (flip (isTrueAtMayb m) f)
          (rightS (rel ag m) w))
isTrueAtMayb
 m@(Mo worlds agents voc val acc points) w (CK ags f) =
 maybe_And (map (flip (isTrueAtMayb m) f)
          (commonAlts acc ags worlds w))
```

`isTrue`, for truth in a model, is also three-valued:

```
isTrue :: Ord state => EpistM state -> Form -> Maybe Bool
isTrue m@(Mo worlds agents voc val acc points) f =
  maybe_And [isTrueAtMayb m s f | s <- points ]
```

**Finally: Public Announcement Update**

```
upd_pa :: Ord state =>
          EpistM state -> Form -> EpistM state
upd_pa m@(Mo states agents voc val rels actual) f =
  (Mo states' agents voc val' rels' actual')
   where
   states' = [ s | s <- states, isTrueAtMayb m s f ==
             Just True]
   val'    = [(s,p) | (s,p) <- val,
                      s `elem` states'   ]
   rels'   = [(ag,x,y) | (ag,x,y) <- rels,
                         x `elem` states',
                         y `elem` states' ]
   actual' = [ s | s <- actual, s `elem` states' ]
```

**Examples**

```
m0 = initM [a,b,c] [P 0,Q 0]
```

**Conversion of States to Integers** Convert *any type of state list* to `[0..]`:

```
convert :: Eq state =>
           EpistM state -> EpistM Integer
convert (Mo states agents voc val rels actual) =
 Mo states' agents voc val' rels' actual'
   where
    states' = map f states
    val'    = map (\ (x,y) -> (f x,y)) val
    rels'   = map (\ (x,y,z) -> (x, f y, f z)) rels
    actual' = map f actual
    f       = apply (zip states [0..])
```

**Generated Submodels**

```
gsm :: Ord state => EpistM state ->  EpistM state
gsm (Mo states ags voc val rel points) =
  (Mo states' ags voc val' rel' points)
   where
   states' = closure rel ags points
   val'    = [(s,props) | (s,props)     <- val,
                          elem s states'             ]
   rel'    = [(ag,s,s') | (ag,s,s') <- rel,
                          elem s states',
                          elem s' states'            ]
```

The closure of a state list, given a relation and a list of agents:

```
closure ::  Ord state =>
            [(Agent,state,state)] ->
            [Agent] -> [state] -> [state]
closure rel agents xs = lfp f xs
 where f = \ ys -> (nub.sort) (ys ++ (expand rel agents ys))
```

The expansion of a relation $R$ given a state set $S$ and a set of agents $B$ is given by $\{t \mid s \xrightarrow{b} t \in R, s \in S, b \in B\}$.

```
expand :: Ord state =>
           [(Agent,state,state)] ->
           [Agent] -> [state] -> [state]
expand rel agents ys = (nub . sort . concat)
  [ alternatives rel ag state | ag    <- agents,
                                state <- ys       ]
```

The epistemic alternatives for agent $a$ in state $s$ are the states in $sR_a$ (the states reachable through $R_a$ from $s$):

```
alternatives :: Eq state =>
     [(Agent,state,state)] ->
       Agent -> state -> [state]
alternatives rel ag current =
  [ s' | (a,s,s') <- rel, a == ag, s == current ]
```

Bisimulation: we compute the maximal bisimulation relation on an epistemic model by means of partition refinement.

**Partition Refinement** Given: A Kripke model **M**.

Problem: find the Kripke model that results from replacing each state $s$ in $\mathbf{M}$ by its bisimilarity class $|s|_{\underline{\leftrightarrow}}$.

The problem of finding the smallest Kripke model modulo bisimulation is similar to the problem of minimizing the number of states in a finite automaton [J.E71].

We will use partition refinement, in the spirit of [PT87].

### Partition Refinement Algorithm

- Start out with a partition of the state set where all states with the same valuation are in the same class.
- Given a partition $\Pi$, for each block $b$ in $\Pi$, partition $b$ into sub-blocks such that two states $s, t$ of $b$ are in the same sub-block iff for all agents $a$ it holds that $s$ and $t$ have $\xrightarrow{a}$ transitions to states in the same block of $\Pi$. Update $\Pi$ to $\Pi'$ by replacing each $b$ in $\Pi$ by the newly found set of sub-blocks for $b$.
- Halt as soon as $\Pi = \Pi'$.

```
type State = Integer
```

### Valuation Comparison

```
sameVal :: (Eq a,Eq b) => [(a,b)] -> a -> a -> Bool
sameVal val w1 w2 = apply val w1 == apply val w2
```

**From Equivalence Relations to Partitions** Relations as characteristic functions.

```
cf2part :: (Eq a) =>
             [a] -> (a -> a -> Bool) -> [[a]]
cf2part [] r = []
cf2part (x:xs) r = xblock : cf2part rest r
  where
  (xblock,rest) = (x : filter (r x) xs,
                   filter (not . (r x)) xs)
```

**Initial Partition** We start with the partition based on the relation 'having the same valuation':

```
initPartition :: Eq a => EpistM a -> [[a]]
initPartition (Mo states agents voc val rel actual) =
  cf2part states (\ x y -> sameVal val x y)
```

**The block of an object in a partition** The block of `x` in a partition is the block that has `x` as an element.

```
bl :: Eq a => [[a]] -> a -> [a]
bl part x = head (filter (elem x) part)
```

**Accessible Blocks** For an agent from a given state, given a model and a partition:

```
accBlocks :: Eq a =>
           EpistM a -> [[a]] -> a -> Agent -> [[a]]
accBlocks m@(Mo _ _ _ _ rel _) part s ag =
    nub [ bl part y | (ag',x,y) <- rel,
                      ag' == ag, x == s ]
```

**Having the same accessible blocks under a partition**

```
sameAB :: Ord a =>
          EpistM a -> [[a]] -> a -> a -> Bool
sameAB m@(Mo states ags voc val rel actual) part s t =
    and [ sort (accBlocks m part s ag)
          == sort (accBlocks m part t  ag) | ag <- ags ]
```

**Refinement Step of Partition by Block Splitting** Splitting the blocks `bl` of `p`:

```
refineStep :: Ord a => EpistM a -> [[a]] -> [[a]]
refineStep m p = refineP m p p
  where
  refineP :: Ord a =>
             EpistM a -> [[a]] -> [[a]] -> [[a]]
  refineP m part [] = []
  refineP m part (bl:blocks) =
     newblocks ++ (refineP m part blocks)
       where
         newblocks =
          cf2part bl (\ x y -> sameAB m part x y)
```

**Refining a Partition** The refining process can be implemented as a least fixpoint computation on the operation of taking refinement steps.

```
refine :: Ord a => EpistM a -> [[a]] -> [[a]]
refine m = lfp (refineStep m)
```

Remark: least fixpoint computation is an element of many refinement processes.

It is an example of what is called a *design pattern* in Software Engineering [GHJV95].

### Construction of Minimal Model

```
minimalModel :: Ord a => EpistM a -> EpistM [a]
minimalModel m@(Mo states agents voc val rel actual) =
  (Mo states' agents voc val' rel' actual')
     where
     states'   = refine m (initPartition m)
     f         = bl states'
     val'      = (nub . sort)
        (map (\ (x,y)   -> (f x, y))      val)
     rel'      = (nub . sort)
        (map (\ (x,y,z) -> (x, f y, f z)) rel)
     actual'   = map f actual
```

Example:

```
*ModelsVocab> displayS5 s5example
[0,1,2,3]
[p,q]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0],[1],[2],[3]])
(b,[[0],[1],[2],[3]])
(c,[[0,1,2,3]])
[1]

*ModelsVocab> displayS5 $ minimalModel s5example
[[0],[1],[2],[3]]
[p,q]
[([0],[]),([1],[p]),([2],[q]),([3],[p,q])]
(a,[[[0]],[[1]],[[2]],[[3]]])
(b,[[[0]],[[1]],[[2]],[[3]]])
(c,[[[0],[1],[2],[3]]])
[[1]]
```

**Map to Bisimulation Minimal Model** Map the states to their bisimilarity classes.

Next, convert the bisimilarity classes back into integers:

```
bisim ::  Ord a => EpistM a -> EpistM State
bisim = convert . minimalModel . gsm
```

# Chapter 2

# Action Models, Update

```
module ActionVocab where

import List
import ModelsVocab
import Test.QuickCheck
```

**Definition of Action Models** Datatype for Action Models. No need to specify a vocabulary, for the vocabulary is implicitly given by the list of precondition formulas.

```
data AM state = Am
              [state]
              [Agent]
              [(state,Form)]
              [(Agent,state,state)]
              [state]  deriving (Eq,Show)
```

Extracting the list of preconditions from an action model:

```
preconditions :: AM state -> [Form]
preconditions (Am _ _ pairs _ _) = map snd pairs
```

Extracting the vocabulary from an action model:

```
voc :: AM state -> [Prp]
voc am = (sort.nub.concat) (map getPs (preconditions am))
```

### Functions from agent lists to action models

```
type FAM state = [Agent] -> AM state
```

### Updating with an Action Model

```
up :: (Eq state, Ord state) =>
       EpistM state -> FAM state
                    -> EpistM (state,state)
```

```
up m@(Mo worlds ags voc val rel points) fam =
   Mo worlds' ags' voc val' rel' points'
   where
   Am states ags' pre susp actuals = fam ags
   worlds' = [ (w,s) | w <- worlds, s <- states,
                   isTrueAtMaybb m w (apply pre s) ==
                   Just True]
   val'    = [ ((w,s),props) | (w,props) <- val,
                               s        <- states,
                               elem (w,s) worlds']
   rel'    = [ (ag1,(w1,s1),(w2,s2)) |
                 (ag1,w1,w2) <- rel,
                 (ag2,s1,s2) <- susp,
                  ag1 == ag2,
                  elem (w1,s1) worlds',
                  elem (w2,s2) worlds'   ]
   points' = [ (p,a) | p <- points, a <- actuals,
                       elem (p,a) worlds'         ]
```

**Update and simplify**

```
upd :: (Eq state, Ord state) =>
       EpistM state -> FAM state
                    -> EpistM State
upd m a = bisim (up m a)
```

**Public Announcement Again** Update model consists of a single action, with reflexive arrows for all agents.

Precondition is the formula that expresses the content of the announcement.

```
public :: Form -> FAM State
public form ags = Am [0] ags [(0,form)]
                     [(a,0,0)| a <- ags ] [0]
```

### Composing Two Static Models

Definition of [EWS10].

```
composMod :: (Eq a, Ord a) =>
       EpistM a -> EpistM a
                   -> EpistM (a,a)

composMod m1@(Mo worlds  agents voc  val1 rel1 points)
          m2@(Mo worlds'  agents' voc' val2 rel2 points') =
             (Mo compstat agents compvoc  compval  comprel
                 compoints) where

 compstat = [(x,y) | x <- worlds, y <- worlds', intersect
            (valStat x m1) (vcbSet m2)== intersect
            (valStat y m2) (vcbSet m1)]

 comprel = [(i,(x,y),(r,s))| (i,x,r) <- rel1,
            (j,y,s)  <- rel2, i==j]

 compval = [((x,y), nub ((++) (vcbSet m1) (vcbSet m2)))|
            x <- worlds, y <- worlds']
 compvoc = (sort.nub) ((++) voc voc')
 compoints = [(x,y) | x <- points, y <- points']
```

### Compressing a Parallel Composition of two Models by Bisimulation

```
compos :: (Eq a, Ord a) =>
       EpistM a -> EpistM a -> EpistM State

compos m1@(Mo worlds  agents voc  val1 rel1 points)
       m2@(Mo worlds'  agents' voc' val2 rel2 points') =
            bisim (composMod m1 m2)
```

# Chapter 3

# Adding Factual Change

```
module ChangeVocab where

import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,voc)
```

**Change in the World** Following [BvEK06], we represent changes in the world as substitutions. A substitution maps proposition letters to formulas. Type of a substitition in Haskell:

```
type Subst = [(Prp,Form)]
```

**Action+Change models**

```
data ACM state = Acm
             [state]
             [Agent]
             [(state,(Form,Subst))]
             [(Agent,state,state)]
             [state]  deriving (Eq,Show)
```

Extracting the list of preconditions from an action model:

```
preconditions :: ACM state -> [Form]
preconditions (Acm _ _ pairs _ _) = map (fst.snd) pairs
```

Extracting the vocabulary from an action model:

```
voc :: ACM state -> [Prp]
voc acm = (sort.nub.concat) (map getPs (preconditions acm))
```

**Functions from Agents to A+C models**

```
type FACM state = [Agent] -> ACM state
```

**Getting the precondition and the substitution from an A+C model**

```
prec :: ACM state -> [(state,Form)]
prec (Acm _ _ ps _ _) =
    map (\ (x,(y,_)) -> (x,y)) ps

subst :: ACM state -> [(state,Subst)]
subst (Acm _ _ ps _ _) =
    map (\ (x,(_,y)) -> (x,y)) ps
```

**From Tables to Functions**

```
t2f :: Eq a => [(a,b)] -> a -> b
t2f t = \ x -> maybe undefined id (lookup x t)
```

Substitutions as functions:

```
sub :: Eq a => ACM a -> a -> Prp -> Form
sub am s p = let
     sb = t2f (subst am) s in
  if elem p (map (\ (x,_) -> x) sb) then
     t2f sb p
   else (Prp p)
```

**Changing the World** Valuation at a world in an epistemic model:

```
val :: Eq a => EpistM a -> a -> [Prp]
val m = t2f (valuation m)
```

**New valuation after update with an action model**

```
newVal :: (Eq a, Ord a) =>
  EpistM a -> ACM a -> (a,a) -> [Prp]
newVal m am (w,s) = [ p | p <- allprops, subfct p ]
   where
   allprops = (sort.nub)
    ((val  m w) ++
     (map (\ (x,_) -> x) (t2f (subst am) s)))
   subfct p = isTrueAtMayb m w (sub am s p) == Just True
```

**Updating with an A+C Model**

```
upc :: (Eq state, Ord state) =>
       EpistM state -> FACM state
                    -> EpistM (state,state)
```

```
upc m@(Mo worlds ags vc val rel points) facm =
   Mo worlds' ags' vc' val' rel' points'
   where
    acm@(Acm states ags' ps susp as) = facm ags
    worlds' = [ (w,s) | w <- worlds, s <- states,
                isTrueAtMayb m w (apply (prec acm) s)
                == Just True ]
    vc' = (sort.nub) (vc ++ voc acm)
    val'    = [ ((w,s),newVal m acm (w,s)) |
                                  (w,s) <- worlds' ]
    rel'    = [ (ag1,(w1,s1),(w2,s2)) |
                (ag1,w1,w2) <- rel,
                (ag2,s1,s2) <- susp,
                 ag1 == ag2,
                 elem (w1,s1) worlds',
                 elem (w2,s2) worlds'   ]
    points' = [ (p,a) | p <- points, a <- as,
                        elem (p,a) worlds'    ]
```

**Update and Simplify**

```
upd ::  (Eq state, Ord state) =>
       EpistM state -> FACM state
                    -> EpistM State
upd m a = bisim (upc m a)
```

String a series of updates together:

```
upds :: EpistM State -> [FACM State] -> EpistM State
upds m [] = m
upds m (a:as) = upds (upd m a) as
```

**Public Announcement** See [Pla89, Ger99].

Update model consists of a single action, with reflexive arrows for all agents.

Precondition is the formula that expresses the content of the announcement.

```
public :: Form -> FACM State
public form ags = Acm [0] ags [(0,(form,[]))]
                      [(a,0,0)| a <- ags ] [0]
```

**Example**

```
m0 = upc s5example (public p)
m1 = upd s5example (public p)
```

```
*ChangeVocab> displayS5 s5example
[0,1,2,3]
[p,q]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0],[1],[2],[3]])
(b,[[0],[1],[2],[3]])
(c,[[0,1,2,3]])
[1]

*ChangeVocab> displayS5 m0
[(1,0),(3,0)]
[p,q]
[((1,0),[p]),((3,0),[p,q])]
(a,[[(1,0)],[(3,0)]])
(b,[[(1,0)],[(3,0)]])
(c,[[(1,0),(3,0)]])
[(1,0)]
```

```
*ChangeVocab> displayS5 m1
[0,1]
[p,q]
[(0,[p]),(1,[p,q])]
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
[0]
```

### Public Change

```
pChange :: Subst -> FACM State
pChange subst ags = Acm [0] ags [(0,(Top,subst))]
                      [(a,0,0)| a <- ags ] [0]
```

Example

```
m2 = upc s5example (pChange [(P 0,q)])
m3 = upd s5example (pChange [(P 0,q)])
```

```
*ChangeVocab> displayS5 m2
[(0,0),(1,0),(2,0),(3,0)]
[p,q]
[((0,0),[]),((1,0),[]),((2,0),[p,q]),((3,0),[p,q])]
(a,[[(0,0)],[(1,0)],[(2,0)],[(3,0)]])
(b,[[(0,0)],[(1,0)],[(2,0)],[(3,0)]])
(c,[[(0,0),(1,0),(2,0),(3,0)]])
[(1,0)]
```

```
*ChangeVocab> displayS5 m3
[0,1]
[p,q]
```

```
[(0,[]),(1,[p,q])]
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
[0]
```

**Group Announcement** Computing the update for passing a group announcement to a list of agents: the other agents confuse the action with the action where nothing happens.

In the limit case where the message is passed to all agents, the message is a public announcement.

```
groupM :: [Agent] -> Form -> FACM State
groupM gr form agents =
  if sort gr == sort agents
    then public form agents
    else
      (Acm
       [0,1]
        agents
       [(0,(form,[])),(1,(Top,[]))]
        ([ (a,0,0) | a <- agents ]
         ++ [ (a,0,1) | a <- agents \\ gr ]
         ++ [ (a,1,0) | a <- agents \\ gr ]
         ++ [ (a,1,1) | a <- agents          ])
       [0])
```

**Example**

```
e0 = initM [a,b,c] [P 0,Q 0]
m4 = upc e0 (groupM [a,b] (Neg p))
m5 = upd e0 (groupM [a,b] (Neg p))
```

```
*ChangeVocab> displayS5 e0
[0,1,2,3]
[p,q]
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
(c,[[0,1,2,3]])
[0,1,2,3]


*ChangeVocab> displayS5 m4
[(0,0),(0,1),(1,1),(2,0),(2,1),(3,1)]
[p,q]
[((0,0),[]),((0,1),[]),((1,1),[p]),((2,0),[q]),((2,1),[q]),((3,1),[p,q])]
(a,[[(0,0),(2,0)],[(0,1),(1,1),(2,1),(3,1)]])
(b,[[(0,0),(2,0)],[(0,1),(1,1),(2,1),(3,1)]])
(c,[[(0,0),(0,1),(1,1),(2,0),(2,1),(3,1)]])
[(0,0),(2,0)]


*ChangeVocab> displayS5 m5
[0,1,2,3,4,5]
[p,q]
[(0,[]),(1,[]),(2,[p]),(3,[q]),(4,[q]),(5,[p,q])]
(a,[[0,3],[1,2,4,5]])
(b,[[0,3],[1,2,4,5]])
(c,[[0,1,2,3,4,5]])
[0,3]
```

**Private Messages** Private messages are a special case of group messages:

```
message :: Agent -> Form -> FACM State
message agent = groupM [agent]
```

**Tests** Tests are another special case of group messages:

```
test :: Form -> FACM State
test = groupM []
```

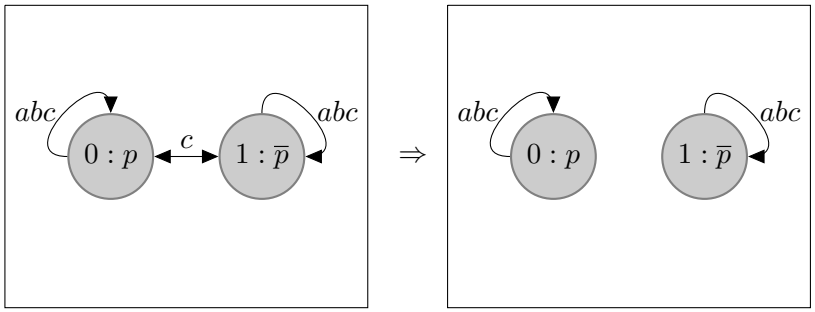
### Communications Whether

- informing everone *whether* $\varphi$,
- informing a group *whether* $\varphi$,
- informing an individual *whether* $\varphi$.

Telling someone whether it rains involves giving her the facts: if it rains you tell her "it rains", if it does not rain you tell her "it does not rain".

In the action model for this there are *two* actual actions. Which one will lead to a new actual world depends on the facts of the matter, and these are determined by the input model ...

**General Form: Group Communication Whether** Informing Everyone Whether $p$.




**Implementation** First the negation of a formula:

```
negation :: Form -> Form
negation (Neg form) = form
negation form       = (Neg form)
```

**Informing a Group Whether $\varphi$**

```
info :: [Agent] -> Form -> FACM State
info group form agents =
   Acm
   [0,1]
   agents
   [(0,(form,[])),(1,(negation form,[]))]
   ([ (a,0,0) | a <- agents ]
     ++ [ (a,1,1) | a <- agents ]
     ++ [ (a,0,1) | a <- others ]
     ++ [ (a,1,0) | a <- others ])
   [0,1]
     where others = agents \\ group
```

**Example**

```
m6 = upc e0 (info [a,b] p)
m7 = upd e0 (info [a,b] p)
```

```
*ChangeVocab> displayS5 m6
[(0,1),(1,0),(2,1),(3,0)]
[p,q]
[((0,1),[]),((1,0),[p]),((2,1),[q]),((3,0),[p,q])]
(a,[[(0,1),(2,1)],[(1,0),(3,0)]])
(b,[[(0,1),(2,1)],[(1,0),(3,0)]])
(c,[[(0,1),(1,0),(2,1),(3,0)]])
[(0,1),(1,0),(2,1),(3,0)]

*ChangeVocab> displayS5 m7
[0,1,2,3]
[p,q]
```

```
[(0,[]),(1,[p]),(2,[q]),(3,[p,q])]
(a,[[0,2],[1,3]])
(b,[[0,2],[1,3]])
(c,[[0,1,2,3]])
[0,1,2,3]
```

# Chapter 4

# Change and Perception

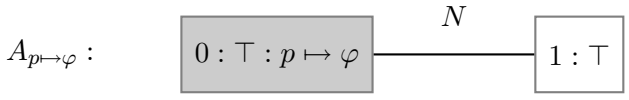
Based on [Eijar].

```
module ChangePerception where

import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,voc)
import ChangeVocab
```

### Unobserved Change

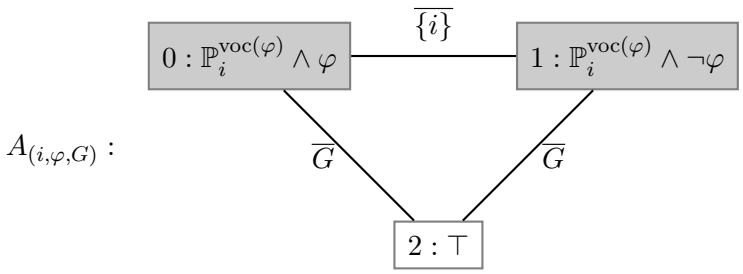
The action model $A_{p:=\varphi}$ for unobserved change $p := \varphi$ looks as follows (note that reflexive arrows are not drawn, and it is assumed that $N$ is the set of all agents):




Implementation:

```
unobserved_change :: Prp -> Form -> FACM State
unobserved_change prp form ags =
  Acm
   [0,1]
   ags
   [(0,(Top,[(prp,form)])), (1,(Top,[]))]
   [(a,s,t)| a <- ags, s <- [0,1], t <- [0,1]]
   [0]
```

The action model $A_{(i,\varphi,G)}$ for a perception by $i$ of $\varphi$, witnessed by $G$, looks as follows (if $Q$ is a list of proposition letters, $\mathbb{P}_i^Q$ expresses that $i$ is able to perceive the letters in $Q$; $\mathrm{voc}(\varphi)$ gives the propositional vocabulary of formula $\varphi$):




Implementation (assuming for simplicity that every agent can perceive every proposition letter):

```
perception :: Agent -> Form -> [Agent] -> FACM State
perception i form group ags =
  Acm [0,1,2]
       ags
      [(0,(form,[])),(1,(Neg form, [])), (2,(Top,[]))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [0]
```

First model:

```
md0 :: EpistM Integer
md0 = Mo [0,1]
         [a,b,c]
         [P 0]
         [(0,[P 0]), (1,[])]
         ([(ag,x,x) | ag <- [a,b,c], x <- [0,1]]
               ++ [(c,0,1),(c,1,0)])
         [0]
```

Second model:

```
md1 = upd md0 (perception b (Prp (P 0)) [b,c])
md1' = upc md0 (perception b (Prp (P 0)) [b,c])
```

Third model:

```
md2 = upd md1 (unobserved_change (P 0) (Neg Top))
md2' = upc md1 (unobserved_change (P 0) (Neg Top))
```

Fourth model:

```
md3 = upd md2 (perception b (Neg p) [b,c])
md3' = upc md2 (perception b (Neg p) [b,c])
```

Fourth model, after perception by all:

```
md3a = upd md2 (perception b (Neg p) [a,b,c])
md3a' = upc md2 (perception b (Neg p) [a,b,c])
```

Different update:

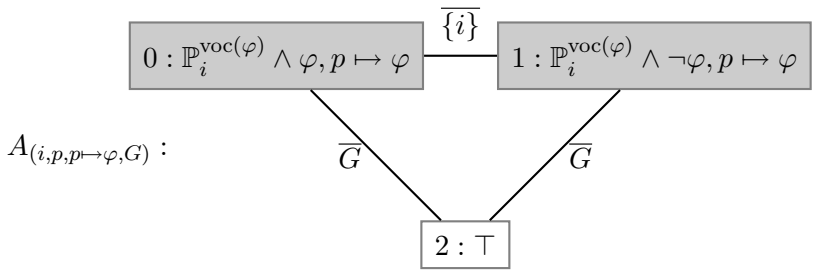
```
md1a = upd md0 (perception c p [a,c])
md1a' = upc md0 (perception c p [a,c])
```

Different initial model:

```
mm0 :: EpistM Integer
mm0 = Mo [0,1]
         [a,b,c]
         [P 0]
         [(0,[P 0]), (1,[])]
         ([(ag,x,x) | ag <- [a,b,c], x <- [0,1]]
            ++ [(a,0,1),(a,1,0)]
            ++ [(c,0,1),(c,1,0)])
            [0]
```

```
mm1 = upd md0 (perception b (Prp (P 0)) [b,c])
mm1' = upc md0 (perception b (Prp (P 0)) [b,c])
```

The action model for perceived change $(i, p, p \mapsto \varphi, G)$ (perception by $i$ of $p$ after a change in $p$ has taken place in the model, with $G$ as witnesses of the act of perception) takes the following shape:




Implementation:

```
perceived_change ::  Agent ->
                       Prp -> Form -> [Agent] -> FACM State
perceived_change i prp form group ags =
  Acm [0,1,2] ags [(0,(Prp prp,[(prp,form)])),
                   (1,(Neg (Prp prp), [(prp,form)])),
                   (2,(Top,[]))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [0]
```

Some updates with this:

```
me1  = upd md0 (perceived_change a (P 0) (Top) [a])
me1' = upc md0 (perceived_change a (P 0) (Top) [a])
```

```
me2  = upd md0 (perceived_change a (P 0) (Neg Top) [a])
me2' = upc md0 (perceived_change a (P 0) (Neg Top) [a])
```

```
me3  = upd md0 (perceived_change a (P 0) (Neg Top) [a,b])
me3' = upc md0 (perceived_change a (P 0) (Neg Top) [a,b])
```

```
me4  = upd md1 (perceived_change a (P 0) (Neg Top) [a,b])
me4' = upc md1 (perceived_change a (P 0) (Neg Top) [a,b])
```

```
ppc ::  Agent -> Prp -> Form -> [Agent] -> FACM State
ppc i prp form group ags =
  Acm [0,1,2] ags [(0,(form,[(prp,form)])),
                   (1,(Neg form, [(prp,form)])),
                   (2,(Top,[]))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [0]
```

```
npc ::  Agent -> Prp -> Form -> [Agent] -> FACM State
npc i prp form group ags =
  Acm [0,1,2] ags [(0,(form,[(prp,form)])),
                   (1,(negation form, [(prp,form)])),
                   (2,(Top,[]))]
      ([(a,s,s)| a <- ags, s <- [0,1,2]]
        ++ [(a,0,1)| a <- ags \\ [i]]
        ++ [(a,1,0)| a <- ags \\ [i]]
        ++ [(a,0,2)| a <- ags \\ group]
        ++ [(a,2,0)| a <- ags \\ group]
        ++ [(a,1,2)| a <- ags \\ group]
        ++ [(a,2,1)| a <- ags \\ group])
      [1]
```

And again:

```
mpc1  = upd md0 (ppc a (P 0) (Top) [a])
mpc1' = upc md0 (ppc a (P 0) (Top) [a])
```

```
mpc2  = upd md0 (npc a (P 0) (Neg Top) [a])
mpc2' = upc md0 (npc a (P 0) (Neg Top) [a])
```

```
mpc3  = upd md0 (npc a (P 0) (Neg Top) [a,b])
mpc3' = upc md0 (npc a (P 0) (Neg Top) [a,b])
```

```
mpc4  = upd md1 (npc a (P 0) (Neg Top) [a,b])
mpc4' = upc md1 (npc a (P 0) (Neg Top) [a,b])
```

# Chapter 5

# The Muddy Children Puzzle

```
module Muddy where

import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,voc)
import ChangeVocab
import ChangePerception
```

Abbreviations for some basic propositions:

```
ma, mb, mc, md :: Form
ma = Prp (P 1) -- this represents Alice is muddy
mb = Prp (P 2) -- this represents Bob is muddy
mc = Prp (P 3) -- this represents Carol is muddy
md = Prp (P 4) -- this represents Dave is muddy
```

Let's model the case where Bob, Carol and Dave are muddy:

```
bcd_dirty = Conj [Neg ma, mb, mc, md]
```

The following series of updates expresses that each child is aware of the state (muddy or not) of the other children:

```
awareness = [info [b,c,d] ma,
             info [a,c,d] mb,
             info [a,b,d] mc,
             info [a,b,c] md ]
```

Formulas for knowing whether one is muddy:

```
aKn =  Disj [K a ma, K a (Neg ma)]
bKn =  Disj [K b mb, K b (Neg mb)]
cKn =  Disj [K c mc, K c (Neg mc)]
dKn =  Disj [K d md, K d (Neg md)]
```

We start with an initial situation where the four agents are blissfully unaware about the muddiness facts, and update with the test expressing that b,c,d are in fact muddy. This gives the following model:

```
mu0 = upd (initM [a,b,c,d] [P 1, P 2, P 3, P 4])
          (test bcd_dirty)
```

Next, add awareness information:

```
mu1 = upds mu0 awareness
```

This gives:

```
*Muddy> displayS5 mu1
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[p1,p2,p3,p4]
[(0,[]),(1,[p1]),(2,[p2]),(3,[p3]),(4,[p4]),(5,[p1,p2]),(6,[p1,p3]),
 (7,[p1,p4]),(8,[p2,p3]),(9,[p2,p4]),(10,[p3,p4]),(11,[p1,p2,p3]),
 (12,[p1,p2,p4]),(13,[p1,p3,p4]),(14,[p2,p3,p4]),(15,[p1,p2,p3,p4])]
(a,[[0,1],[2,5],[3,6],[4,7],[8,11],[9,12],[10,13],[14,15]])
(b,[[0,2],[1,5],[3,8],[4,9],[6,11],[7,12],[10,14],[13,15]])
(c,[[0,3],[1,6],[2,8],[4,10],[5,11],[7,13],[9,14],[12,15]])
(d,[[0,4],[1,7],[2,9],[3,10],[5,12],[6,13],[8,14],[11,15]])
[14]
```

Update with a public announcement of the father that at least one child is muddy.

```
mu2 = upd  mu1 (public (Disj [ma, mb, mc, md]))
```

This gives:

```
*Muddy> displayS5 mu2
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]
[p1,p2,p3,p4]
[(0,[p1]),(1,[p2]),(2,[p3]),(3,[p4]),(4,[p1,p2]),(5,[p1,p3]),(6,[p1,p4]),
 (7,[p2,p3]),(8,[p2,p4]),(9,[p3,p4]),(10,[p1,p2,p3]),(11,[p1,p2,p4]),
 (12,[p1,p3,p4]),(13,[p2,p3,p4]),(14,[p1,p2,p3,p4])]
(a,[[0],[1,4],[2,5],[3,6],[7,10],[8,11],[9,12],[13,14]])
(b,[[0,4],[1],[2,7],[3,8],[5,10],[6,11],[9,13],[12,14]])
(c,[[0,5],[1,7],[2],[3,9],[4,10],[6,12],[8,13],[11,14]])
(d,[[0,6],[1,8],[2,9],[3],[4,11],[5,12],[7,13],[10,14]])
[13]
```

The first round: they all say they don't know their state.

```
mu3 = upd  mu2
       (public (Conj[Neg aKn, Neg bKn, Neg cKn, Neg dKn]))
```

```
\*Muddy> displayS5 mu3
[0,1,2,3,4,5,6,7,8,9,10]
[p1,p2,p3,p4]
[(0,[p1,p2]),(1,[p1,p3]),(2,[p1,p4]),(3,[p2,p3]),(4,[p2,p4]),(5,[p3,p4]),
 (6,[p1,p2,p3]),(7,[p1,p2,p4]),(8,[p1,p3,p4]),(9,[p2,p3,p4]),(10,[p1,p2,p3,p4])]
(a,[[0],[1],[2],[3,6],[4,7],[5,8],[9,10]])
(b,[[0],[1,6],[2,7],[3],[4],[5,9],[8,10]])
(c,[[0,6],[1],[2,8],[3],[4,9],[5],[7,10]])
(d,[[0,7],[1,8],[2],[3,9],[4],[5],[6,10]])
[9]
```

The second round: they still all don't know their state.

```
mu4 = upd  mu3
       (public (Conj[Neg aKn, Neg bKn, Neg cKn, Neg dKn]))
```

```
*Muddy> displayS5 mu4
[0,1,2,3,4]
[p1,p2,p3,p4]
[(0,[p1,p2,p3]),(1,[p1,p2,p4]),(2,[p1,p3,p4]),(3,[p2,p3,p4]),(4,[p1,p2,p3,p4])]
(a,[[0],[1],[2],[3,4]])
(b,[[0],[1],[2,4],[3]])
(c,[[0],[1,4],[2],[3]])
(d,[[0,4],[1],[2],[3]])
[3]
```

Now b, c and d say they know. In the final model all is known to everyone.

```
mu5 = upds mu4 [public (Conj[bKn, cKn, dKn])]
```

```
*Muddy> displayS5 mu5
[0]
[p1,p2,p3,p4]
[(0,[p2,p3,p4])]
(a,[[0]])
```

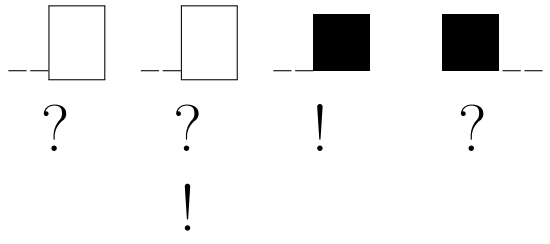
```
(b,[[0]])
(c,[[0]])
(d,[[0]])
[0]
```

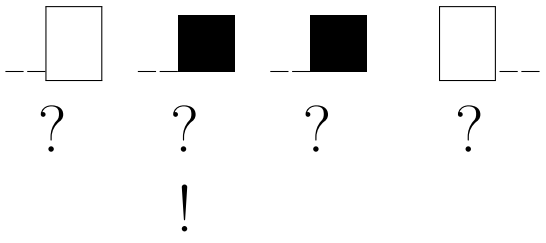
# Chapter 6

# The Wise Men Puzzle; or: The Riddle of the Caps

```
module WiseMen

where
import List
import ModelsVocab hiding (m0)
import ActionVocab hiding (upd,public,preconditions,voc)
import ChangeVocab
import ChangePerception
```






**Analysis with Model Checking**

- Four agents: $a, b, c, d$, occupying positions $1, 2, 3, 4$.
- Four basic propositions $p_1$, $p_2$, $p_3$, $p_4$.
- $p_i$ expresses that the guy at position $i$ is wearing a *white* cap.

Initial model

```
mo0 = initM [a,b,c,d] [P 1, P 2, P 3, P 4]
```

```
p1,p2,p3,p4 :: Form
p1 = Prp (P 1); p2 = Prp (P 2)
p3 = Prp (P 3); p4 = Prp (P 4)

capsInfo :: Form
capsInfo =
  Disj [Conj [f, g, Neg h, Neg j] |
              f <- [p1, p2, p3, p4],
              g <- [p1, p2, p3, p4] \\ [f],
              h <- [p1, p2, p3, p4] \\ [f,g],
              j <- [p1, p2, p3, p4] \\ [f,g,h],
              f < g, h < j                    ]
```

```
mo1 = upd mo0 (public capsInfo)
```

```
*WiseMen> displayS5 mo1
[0,1,2,3,4,5]
[p1,p2,p3,p4]
[(0,[p1,p2]),(1,[p1,p3]),(2,[p1,p4]),
 (3,[p2,p3]),(4,[p2,p4]),(5,[p3,p4])]
(a,[[0,1,2,3,4,5]])
(b,[[0,1,2,3,4,5]])
(c,[[0,1,2,3,4,5]])
(d,[[0,1,2,3,4,5]])
[0,1,2,3,4,5]
```

```
awarenessFirstCap  = info [b,c] p1
awarenessSecondCap = info [c]   p2

mo2  = upd (upd mo1 awarenessFirstCap)
                    awarenessSecondCap
```

```
*WiseMen> displayS5 mo2
[0,1,2,3,4,5]
[p1,p2,p3,p4]
[(0,[p1,p2]),(1,[p1,p3]),(2,[p1,p4]),
 (3,[p2,p3]),(4,[p2,p4]),(5,[p3,p4])]
(a,[[0,1,2,3,4,5]])
(b,[[0,1,2],[3,4,5]])
(c,[[0],[1,2],[3,4],[5]])
(d,[[0,1,2,3,4,5]])
[0,1,2,3,4,5]
```

```
bK =  Disj [K b p2, K b (Neg p2)]
cK =  Disj [K c p3, K c (Neg p3)]

mo3a = upd  mo2 (public cK)
mo3b = upd  mo2 (public (Neg cK))
```

```
*WiseMen> displayS5 mo3a
[0,1]
[p1,p2,p3,p4]
[(0,[p1,p2]),(1,[p3,p4])]
(a,[[0,1]])
(b,[[0],[1]])
(c,[[0],[1]])
(d,[[0,1]])
[0,1]

*WiseMen> displayS5 mo3b
[0,1,2,3]
[p1,p2,p3,p4]
[(0,[p1,p3]),(1,[p1,p4]),(2,[p2,p3]),(3,[p2,p4])]
(a,[[0,1,2,3]])
(b,[[0,1],[2,3]])
(c,[[0,1],[2,3]])
(d,[[0,1,2,3]])
[0,1,2,3]
```

```
impl :: Form -> Form -> Form
impl form1 form2 = Disj [Neg form1, form2]

equiv :: Form -> Form -> Form
equiv form1 form2 =
  Conj [form1 `impl` form2, form2 `impl` form1]
```

```
test1 = isTrue mo3a bK
test2 = isTrue mo3b bK
test3 = isTrue mo3a (K a (equiv p1 p2))
test4 = isTrue mo3b (K a (equiv p1 (Neg p2)))
```

```
*WiseMen> test1
Just True
*WiseMen> test2
Just True
*WiseMen> test3
Just True
*WiseMen> test4
Just True
```

```
mo4a = upd mo3a (public bK)
mo4b = upd mo3b (public bK)
```

```
*WiseMen> displayS5 mo4a
[0,1]
[p1,p2,p3,p4]
[(0,[p1,p2]),(1,[p3,p4])]
(a,[[0,1]])
```

```
(b,[[0],[1]])
(c,[[0],[1]])
(d,[[0,1]])
[0,1]


*WiseMen> displayS5 mo4b
[0,1,2,3]
[p1,p2,p3,p4]
[(0,[p1,p3]),(1,[p1,p4]),(2,[p2,p3]),(3,[p2,p4])]
(a,[[0,1,2,3]])
(b,[[0,1],[2,3]])
(c,[[0,1],[2,3]])
(d,[[0,1,2,3]])
[0,1,2,3]
```

# Bibliography


[BvEK06] J. van Benthem, J. van Eijck, and B. Kooi. Logics of communication and change. *Information and Computation*, 204(11):1620–1662, 2006.

[CH00] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. Of International Conference on Functional Programming (ICFP)*, ACM SIGPLAN, 2000.

[Eijar] Jan van Eijck. Perception and change in update logic. In Jan van Eijck and Rineke Verbrugge, editors, *Games, Actions and Social Software*. Springer, 2011 (to appear).

[EWS10] Jan van Eijck, Yanjing Wang, and Floor Sietsma. Composing models. In Wiebe van der Hoek, editor, *Online Proceedings of LOFT 2010*, `http://loft2010.csc.liv.ac.uk/`, 2010.

[Ger99] J. Gerbrandy. *Bisimulations on Planet Kripke*. PhD thesis, ILLC, Amsterdam, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

[J.E71] J.E.Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*. Academic Press, 1971.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland and P. Noordhoff, 1952.

[Pla89] J. A. Plaza. Logics of public communications. In M. L. Emrich, M. S. Pfeifer, M. Hadzikadic, and Z. W. Ras, editors, *Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems*, pages 201–216, 1989.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.