DEMO — A Demo of Epistemic Modelling*

Jan van Eijck

Centrum voor Wiskunde en Informatica
Kruislaan 413
1098 SJ Amsterdam, The Netherlands

Utrecht Instituut voor Linguïstiek / Onderzoeksinstituut voor Taal en Spraak
Universiteit Utrecht
P.O. Box 85253
3508 AG Utrecht, The Netherlands

Netherlands Institute for Advanced Study in the Humanities and Social Sciences
Meijboomlaan 1
2242 PR Wassenaar, The Netherlands

`jve@cwi.nl`

Abstract

This paper introduces and documents *DEMO*, a Dynamic Epistemic Modelling tool. *DEMO* allows modelling epistemic updates, graphical display of update results, graphical display of action models, formula evaluation in epistemic models, translation of dynamic epistemic formulas to PDL formulas. Also, *DEMO* implements the reduction of dynamic epistemic logic to PDL. The paper is an exemplar of tool building for epistemic update logic. It contains the essential code of an implementation of *DEMO* in Haskell, in Knuth's 'literate programming' style.

1 Introduction

In this introduction we shall demonstrate how *DEMO*, which is short for *Dynamic Epistemic MOdelling*,[1] can be used to check semantic intuitions about what goes on in epistemic update situations.[2] For didactic purposes,

* The author is grateful to the Netherlands Institute for Advanced Studies (NIAS) for providing the opportunity to complete this paper as Fellow-in-Residence. This report and the tool that it describes were prompted by a series of questions voiced by Johan van Benthem in his talk at the annual meeting of the Dutch Association for Theoretical Computer Science, in Utrecht, on March 5, 2004. Thanks to Johan van Benthem, Hans van Ditmarsch, Barteld Kooi and Ji Ruan for valuable feedback and inspiring discussion. Two anonymous referees made suggestions for improvement, which are herewith gracefully acknowledged.

[1] Or short for *DEMO of Epistemic MOdelling*, for those who prefer co-recursive acronyms.

[2] The program source code is available from `http://www.cwi.nl/~jve/demo/`.

Johan van Benthem, Dov Gabbay, Benedikt Löwe (eds.). *Interactive Logic Proceedings of the 7th Augustus de Morgan Workshop, London.* Texts in Logic and Games 1, Amsterdam University Press 2007, pp. 305–363.

the initial examples have been kept extremely simple. Although the situation of message passing about just two basic propositions with just three epistemic agents already reveals many subtleties, the reader should bear in mind that *DEMO* is capable of modelling much more complex situations.

In a situation where you and I know nothing about a particular aspect of the state of the world (about whether p and q hold, say), our state of knowledge is modelled by a Kripke model where the worlds are the four different possibilities for the truth of p and q ($\varnothing$, p, q, pq), your epistemic accessibility relation $\sim_a$ is the total relation on these four possibilities, and mine $\sim_b$ is the total relation on these four possibilities as well. There is also c, who like the two of us, is completely ignorant about p and q. This initial model is generated by *DEMO* as follows.

```
DEMO> showM (initE [P 0,Q 0] [a,b,c])
==> [0,1,2,3]
[0,1,2,3]
(0,[])(1,[p])(2,[q])(3,[p,q])
(a,[[0,1,2,3]])
(b,[[0,1,2,3]])
(c,[[0,1,2,3]])
```

Here `initE` generates an initial epistemic model, and `showM` shows that model in an appropriate form, in this case in the partition format that is made possible by the fact that the epistemic relations are all equivalences.

As an example of a different kind of representation, let us look at the picture that can be generated with *dot* [Ga$_0$Ko$_5$No$_0$06] from the file produced by the *DEMO* command `writeP "filename" (initE [P 0,Q 0])`, as represented in Figure 1.

This is a model where none of the three agents a, b or c can distinguish between the four possibilities about p and q. *DEMO* shows the partitions generated by the accessibility relations $\sim_a, \sim_b, \sim_c$. Since these three relations are total, the three partitions each consist of a single block. Call this model `e0`.

Now suppose a wants to know whether p is the case. She asks whether p and receives a truthful answer from somebody who is in a position to know. This answer is conveyed to a in a message. b and c have heard a's question, and so are aware of the fact that an answer may have reached a. b and c have seen *that* an answer was delivered, but they don't know which answer. This is not a secret communication, for b and c know that a has inquired about p. The situation now changes as follows:

```
DEMO> showM (upd e0 (message a p))
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
```

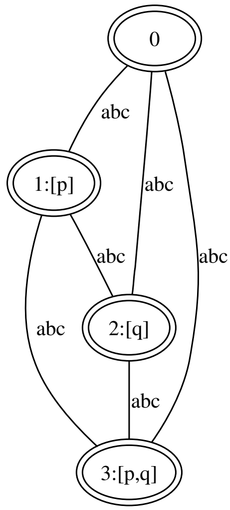

FIGURE 1.

```
(a,[[0,2,3,5],[1,4]])
(b,[[0,1,2,3,4,5]])
(c,[[0,1,2,3,4,5]])
```

Note that `upd` is a function for updating an epistemic model with (a representation of) a communicative action. In this case, the result is again a model where the three accessibility relations are equivalences, but one in which a has restricted her range of possibilities to $1, 4$ (these are worlds where p is the case), while for b and c all possibilities are still open. Note that this epistemic model has two ‘actual worlds’: this means that there are two possibilities that are compatible with ‘how things really are’. In graphical display format these ‘actual worlds’ show up as double ovals, as seen in Figure 2.

DEMO also allows us to display the action models corresponding to the epistemic updates. For the present example (we have to indicate that we want the action model for the case where $\{a, b, c\}$ is the set of relevant agents):

```
showM ((message a p) [a,b,c])
==> [0]
[0,1]
(0,p)(1,
T)
(a,[[0],[1]])
```

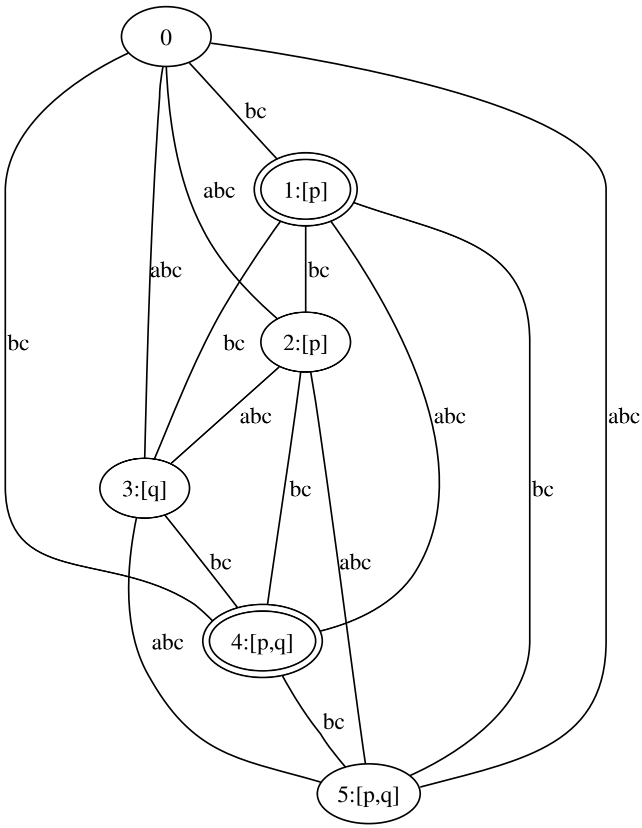

Figure 2.

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

Notice that in the result of updating the initial situation with this message, some subtle things have changed for b and c as well. Before the arrival of the message, $\Box_b(\neg\Box_a p \wedge \neg\Box_a\neg p)$ was true, for b knew that a did not know about p. But now b has heard a's question about p, and is aware of the fact that an answer has reached a. So in the new situation b knows that a knows about p. In other words, $\Box_b(\Box_a p \vee \Box_a\neg p)$ has become true. On the other hand it is still the case that b knows that a knows nothing about q: $\Box_b\neg\Box_a q$ is still true in the new situation. The situation for c is similar to that for b. These things can be checked in *DEMO* as follows:

```
DEMO> isTrue (upd e0 (message a p)) (K b (Neg (K a q)))
True
DEMO> isTrue (upd e0 (message a p)) (K b (Neg (K a p)))
False
```

If you receive the same message about p twice, the second time the message gets delivered has no further effect. Note the use of `upds` for a sequence of updates.

```
DEMO> showM (upds e0 [message a p, message a p])
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
(a,[[0,2,3,5],[1,4]])
(b,[[0,1,2,3,4,5]])
(c,[[0,1,2,3,4,5]])
```

Now suppose that the second action is a message informing b about p:

```
DEMO> showM (upds e0 [message a p, message b p])
==> [1,6]
[0,1,2,3,4,5,6,7,8,9]
(0,[])(1,[p])(2,[p])(3,[p])(4,[p])
(5,[q])(6,[p,q])(7,[p,q])(8,[p,q])(9,[p,q])

(a,[[0,3,4,5,8,9],[1,2,6,7]])
(b,[[0,2,4,5,7,9],[1,3,6,8]])
(c,[[0,1,2,3,4,5,6,7,8,9]])
```

The graphical representation of this model is slightly more difficult to fathom at a glance. See Figure 3. In this model a and b both know about p, but they do not know about each other's knowledge about p. c still knows nothing, and both a and b know that c knows nothing. Both $\Box_a\Box_b p$ and $\Box_b\Box_a p$ are false in this model. $\Box_a\neg\Box_b p$ and $\Box_b\neg\Box_a p$ are false as well, but $\Box_a\neg\Box_c p$ and $\Box_b\neg\Box_c p$ are true.

```
DEMO> isTrue (upds e0 [message a p, message b p]) (K a (K b p))
False
DEMO> isTrue (upds e0 [message a p, message b p]) (K b (K a p))
False
DEMO> isTrue (upds e0 [message a p, message b p]) (K b (Neg (K b p)))
False
DEMO> isTrue (upds e0 [message a p, message b p]) (K b (Neg (K c p)))
True
```

The order in which a and b are informed does not matter:

```
DEMO> showM (upds e0 [message b p, message a p])
==> [1,6]
[0,1,2,3,4,5,6,7,8,9]
```

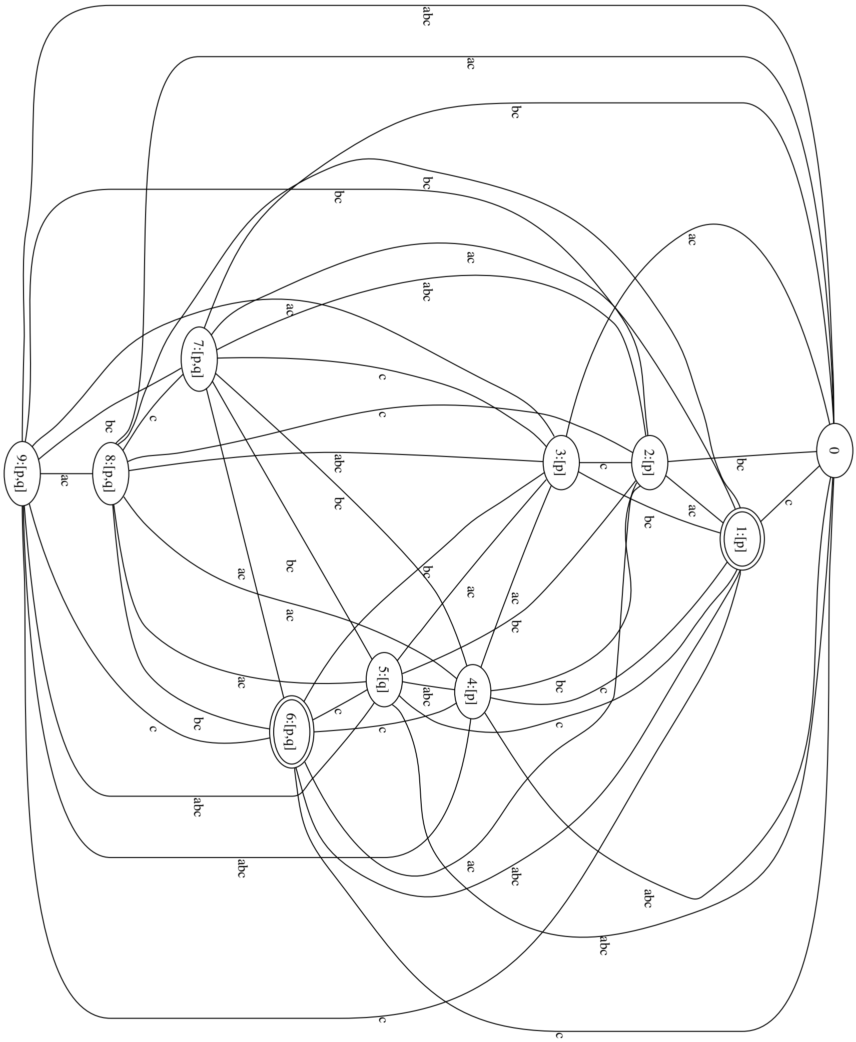

FIGURE 3. Situation after second message

```
(0,[])(1,[p])(2,[p])(3,[p])(4,[p])
(5,[q])(6,[p,q])(7,[p,q])(8,[p,q])(9,[p,q])

(a,[[0,2,4,5,7,9],[1,3,6,8]])
(b,[[0,3,4,5,8,9],[1,2,6,7]])
(c,[[0,1,2,3,4,5,6,7,8,9]])
```

Modulo renaming this is the same as the earlier result. The example shows that the epistemic effects of distributed message passing are quite different from those of a public announcement or a group message.

```
DEMO> showM (upd e0 (public p))
==> [0,1]
[0,1]
(0,[p])(1,[p,q])
(a,[[0,1]])
(b,[[0,1]])
(c,[[0,1]])
```

The result of the public announcement that p is that a, b and c are informed that p and about each other's knowledge about p.

DEMO allows to compare the action models for public announcement and individual message passing:

```
DEMO> showM ((public p) [a,b,c])
==> [0]
[0]
(0,p)
(a,[[0]])
(b,[[0]])
(c,[[0]])

DEMO> showM ((cmp [message a p, message b p, message c p]) [a,b,c])
==> [0]
[0,1,2,3,4,5,6,7]
(0,p)(1,p)(2,p)(3,p)(4,p)
(5,p)(6,p)(7,T)
(a,[[0,1,2,3],[4,5,6,7]])
(b,[[0,1,4,5],[2,3,6,7]])
(c,[[0,2,4,6],[1,3,5,7]])
```

Here `cmp` gives the sequential composition of a list of communicative actions. This involves, among other things, computation of the appropriate preconditions for the combined action model.

More subtly, the situation is also different from a situation where a, b receive the same message that p, with a being aware of the fact that b receives the message and vice versa. Such group messages create common knowledge.

```
DEMO> showM (groupM [a,b] p [a,b,c])
```

```
==> [0]
[0,1]
(0,p)(1,T)
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
```

The difference with the case of the two separate messages is that now a and b are aware of each other's knowledge that p:

```
DEMO> isTrue (upd e0 (groupM [a,b] p)) (K a (K b p))
True
DEMO> isTrue (upd e0 (groupM [a,b] p)) (K b (K a p))
True
```

In fact, this awareness goes on, for arbitrary nestings of $\Box_a$ and $\Box_b$, which is what common knowledge means. Common knowledge can be checked directly, as follows:

```
DEMO> isTrue (upd e0 (groupM [a,b] p)) (CK [a,b] p)
True
```

It is also easily checked in *DEMO* that in the case of the separate messages no common knowledge is achieved.

Next, look at the case where two separate messages reach a and b, one informing a that p and the other informing b that $\neg q$:

```
DEMO> showM (upds e0 [message a p, message b (Neg q)])
==> [2]
[0,1,2,3,4,5,6,7,8]
(0,[])(1,[])(2,[p])(3,[p])(4,[p])
(5,[p])(6,[q])(7,[p,q])(8,[p,q])
(a,[[0,1,4,5,6,8],[2,3,7]])
(b,[[0,2,4],[1,3,5,6,7,8]])
(c,[[0,1,2,3,4,5,6,7,8]])
```

Again the order in which these messages are delivered is immaterial for the end result, as you should expect:

```
DEMO> showM (upds e0 [message b (Neg q), message a p])
==> [2]
[0,1,2,3,4,5,6,7,8]
(0,[])(1,[])(2,[p])(3,[p])(4,[p])
(5,[p])(6,[q])(7,[p,q])(8,[p,q])
(a,[[0,1,3,5,6,8],[2,4,7]])
(b,[[0,2,3],[1,4,5,6,7,8]])
(c,[[0,1,2,3,4,5,6,7,8]])
```

Modulo a renaming of worlds, this is the same as the previous result.

The logic of public announcements and private messages is related to the logic of knowledge, with [$\mathrm{Hi}_1$62] as the pioneer publication. This logic satisfies the following postulates:

- knowledge distribution $\Box_a(\varphi \Rightarrow \psi) \Rightarrow (\Box_a\varphi \Rightarrow \Box_a\psi)$ (if a knows that φ implies ψ, and she knows φ, then she also knows ψ),
- positive introspection $\Box_a\varphi \Rightarrow \Box_a\Box_a\varphi$ (if a knows φ, then a knows that she knows φ),
- negative introspection $\neg\Box_a\varphi \Rightarrow \Box_a\neg\Box_a\varphi$ (if a does not know φ, then she knows that she does not know),
- truthfulness $\Box_a\varphi \Rightarrow \varphi$ (if a knows φ then φ is true).

As is well known, the first of these is valid on all Kripke frames, the second is valid on precisely the transitive Kripke frames, the third is valid on precisely the euclidean Kripke frames (a relation R is euclidean if it satisfies $\forall x\forall y\forall z((xRy \wedge xRz) \Rightarrow yRz))$, and the fourth is valid on precisely the reflexive Kripke frames. A frame satisfies transitivity, euclideanness and reflexivity iff it is an equivalence relation, hence the logic of knowledge is the logic of the so-called S5 Kripke frames: the Kripke frames with an equivalence $\sim_a$ as epistemic accessibility relation. Multi-agent epistemic logic extends this to multi-S5, with an equivalence $\sim_b$ for every $b \in B$, where b is the set of epistemic agents.

Now suppose that instead of open messages, we use *secret* messages. If a secret message is passed to a, b and c are not even aware that any communication is going on. This is the result when a receives a secret message that p in the initial situation:

```
DEMO> showM (upd e0 (secret [a] p))
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
(a,[([],[0,2,3,5]),([],[1,4])])
(b,[([1,4],[0,2,3,5])])
(c,[([1,4],[0,2,3,5])])
```

This is not an S5 model anymore. The accessibility for a is still an equivalence, but the accessibility for b is lacking the property of reflexivity. The worlds $1, 4$ that make up a's conceptual space (for these are the worlds accessible for a from the actual worlds $1, 4$) are precisely the worlds where the b and c arrows are *not* reflexive. b enters his conceptual space from the vantage points 1 and 4, but b does not see these vantage points itself. Similarly for c. In the *DEMO* representation, the list `([1,4],[0,2,3,5])` gives the entry points `[1,4]` into conceptual space `[0,2,3,5]`.

The secret message has no effect on what b and c believe about the facts of the world, but it has effected b's and c's beliefs about the beliefs of a in a disastrous way. These beliefs have become inaccurate. For instance, b

now believes that a does *not* know that p, but he is mistaken! The formula $\Box_b \neg \Box_a p$ is true in the actual worlds, but $\neg \Box_a p$ is false in the actual worlds, for a *does* know that p, because of the secret message. Here is what *DEMO* says about the situation (`isTrue` evaluates a formula in all of the actual worlds of an epistemic model):

```
DEMO> isTrue (upd e0 (secret [a] p)) (K b (Neg (K a p)))
True
DEMO> isTrue (upd e0 (secret [a] p)) (Neg (K a p))
False
```

This example illustrates a regress from the world of knowledge to the world of consistent belief: the result of the update with a secret propositional message does not satisfy the postulate of truthfulness anymore.

The logic of consistent belief satisfies the following postulates:

- knowledge distribution $\Box_a(\varphi \Rightarrow \psi) \Rightarrow (\Box_a \varphi \Rightarrow \Box_a \psi)$,
- positive introspection $\Box_a \varphi \Rightarrow \Box_a \Box_a \varphi$,
- negative introspection $\neg \Box_a \varphi \Rightarrow \Box_a \neg \Box_a \varphi$,
- consistency $\Box_a \varphi \Rightarrow \Diamond_a \varphi$ (if a believes that φ then there is a world where φ is true, i.e., φ is consistent).

Consistent belief is like knowledge, except for the fact that it replaces the postulate of truthfulness $\Box_a \varphi \Rightarrow \varphi$ by the weaker postulate of consistency.

Since the postulate of consistency determines the serial Kripke frames (a relation R is serial if $\forall x \exists y\, xRy$), the principles of consistent belief determine the Kripke frames that are transitive, euclidean and serial, the so-called KD45 frames.

In the conceptual world of secrecy, inconsistent beliefs are not far away. Suppose that a, after having received a secret message informing her about p, sends a message to b to the effect that $\Box_a p$. The trouble is that this is *inconsistent* with what b believes.

```
DEMO> showM (upds e0 [secret [a] p, message b (K a p)])
==> [1,5]
[0,1,2,3,4,5,6,7]
(0,[])(1,[p])(2,[p])(3,[p])(4,[q])
(5,[p,q])(6,[p,q])(7,[p,q])
(a,([],[([],[0,3,4,7]),([],[1,2,5,6])]))
(b,([1,5],[([2,6],[0,3,4,7])]))
(c,([],[([1,2,5,6],[0,3,4,7])]))
```

This is not a KD45 model anymore, for it lacks the property of seriality for b's belief relation. b's belief contains two isolated worlds $1, 5$. Since 1 is the actual world, this means that b's belief state has become inconsistent: from now on, b will believe *anything*.

So we have arrived at a still weaker logic. The logic of possibly inconsistent belief satisfies the following postulates:

- knowledge distribution $\Box_a(\varphi \Rightarrow \psi) \Rightarrow (\Box_a\varphi \Rightarrow \Box_a\psi)$,
- positive introspection $\Box_a\varphi \Rightarrow \Box_a\Box_a\varphi$,
- negative introspection $\neg\Box_a\varphi \Rightarrow \Box_a\neg\Box_a\varphi$.

This is the logic of K45 frames: frames that are transitive and euclidean.

In [vE$_1$04a] some results and a list of questions are given about the possible deterioration of knowledge and belief caused by different kind of message passing. E.g., the result of updating an S5 model with a public announcement or a non-secret message, if defined, is again S5. The result of updating an S5 model with a secret message to some of the agents, if defined, need not even be KD45. One can prove that the result is KD45 iff the model we start out with satisfies certain epistemic conditions. The update result always is K45. Such observations illustrate why S5, KD45 and K45 are ubiquitous in epistemic modelling. See [BldRVe$_1$01, Go$_0$02] for general background on modal logic, and [Ch$_3$80, Fa+95] for specific background on these systems.

If this introduction has convinced the reader that the logic of public announcements, private messages and secret communications is rich and subtle enough to justify the building of the conceptual modelling tools to be presented in the rest of the report, then it has served its purpose.

In the rest of the report, we first fix a formal version of epistemic update logic as an implementation goal. After that, we are ready for the implementation.

Further information on various aspects of dynamic epistemic logic is provided in [Ba$_4$02, Ba$_4$Mo$_3$So$_1$99, vB01b, vB06, vD00, Fa+95, Ge$_2$99a, Ko$_4$03].

2 Design

DEMO is written in a high level functional programming language Haskell [Jo$_2$03]. Haskell is a non-strict, purely-functional programming language named after Haskell B. Curry. The design is modular. Operations on lists and characters are taken from the standard Haskell `List` and `Char` modules. The following modules are part of *DEMO*:

Models The module that defines general models over a number of agents. In the present implementation these are A through E. It turns out that more than five agents are seldom needed in epistemic modelling.

General models have variables for their states and their state adornments. By letting the state adornments be valuations we get *Kripke models*, by letting them be formulas we get *update models*.

MinBis The module for minimizing models under bisimulation by means of partition refinement.

Display The module for displaying models in various formats. Not discussed in this paper.

ActEpist The module that specializes general models to action models and epistemic models. Formulas may contain action models as operators. Action models contain formulas. The definition of formulas is therefore also part of this module.

DPLL Implementation of Davis, Putnam, Logemann, Loveland (DPLL) theorem proving [$\mathrm{Da_1Lo_0Lo_4}$62, $\mathrm{Da_1Pu}$60] for propositional logic. The implementation uses discrimination trees or *tries*, following [$\mathrm{Zh_0St_5}$00]. This is used for formula simplification. Not discussed in this paper.

Semantics Implementation of the key semantic notions of epistemic update logic. It handles the mapping from communicative actions to action models.

DEMO Main module.

3 Main module

```
module DEMO
    (
     module List,
     module Char,
     module Models,
     module Display,
     module MinBis,
     module ActEpist,
     module DPLL,
     module Semantics
    )
    where

import List import Char import Models import Display import MinBis
import ActEpist import DPLL import Semantics
```

The first version of *DEMO* was written in March 2004. This version was extended in May 2004 with an implementation of automata and a translation function from epistemic update logic to Automata PDL. In September 2004, I discovered a direct reduction of epistemic update logic to PDL [$\mathrm{vE_1}$04b]. This motivated a switch to a PDL-like language, with extra modalities for action update and automata update. I decided to leave in the automata for the time being, for nostalgic reasons.

In Summer 2005, several example modules with *DEMO* programs for epistemic puzzles (some of them contributed by Ji Ruan) and for checking of security protocols (with contributions by Simona Orzan) were added, and the program was rewritten in a modular fashion.

In Spring 2006, automata update was removed, and in Autumn 2006 the code was refactored for the present report:

```
version :: String
version = "DEMO 1.06, Autumn 2006"
```

4 Definitions

4.1 Models and updates

In this section we formalize the version of dynamic epistemic logic that we are going to implement.

Let p range over a set of basic propositions P and let a range over a set of agents Ag. Then the language of PDL over P, Ag is given by:

$$\begin{array}{lcl} \varphi & ::= & \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid [\pi]\varphi \\ \pi & ::= & a \mid ?\varphi \mid \pi_1;\pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^* \end{array}$$

Employ the usual abbreviations: $\bot$ is shorthand for $\neg\top$, $\varphi_1 \vee \varphi_2$ is shorthand for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ is shorthand for $\neg(\varphi_1 \wedge \varphi_2)$, $\varphi_1 \leftrightarrow \varphi_2$ is shorthand for $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$, and $\langle\pi\rangle\varphi$ is shorthand for $\neg[\pi]\neg\varphi$. Also, if $B \subseteq Ag$ and B is finite, use B as shorthand for $b_1 \cup b_2 \cup \cdots$. Under this convention, formulas for expressing general knowledge $E_B\varphi$ take the shape $[B]\varphi$, while formulas for expressing common knowledge $C_B\varphi$ appear as $[B^*]\varphi$, i.e., $[B]\varphi$ expresses that it is general knowledge among agents B that φ, and $[B^*]\varphi$ expresses that it is common knowledge among agents B that φ. In the special case where $B = \varnothing$, B turns out equivalent to $?\bot$, the program that always fails.

The semantics of PDL over P, Ag is given relative to labelled transition systems $\mathbf{M} = (W, V, R)$, where W is a set of worlds (or states), $V : W \rightarrow \mathcal{P}(P)$ is a valuation function, and $R = \{\xrightarrow{a} \subseteq W \times W \mid a \in Ag\}$ is a set of labelled transitions, i.e., binary relations on W, one for each label a. In what follows, we shall take the labelled transitions for a to represent the epistemic alternatives of an agent a.

The formulae of PDL are interpreted as subsets of $W_{\mathbf{M}}$ (the state set of $\mathbf{M}$), the actions of PDL as binary relations on $W_{\mathbf{M}}$, as follows:

$$\begin{array}{rcl} [\![\top]\!]^{\mathbf{M}} & = & W_{\mathbf{M}} \\ [\![p]\!]^{\mathbf{M}} & = & \{w \in W_{\mathbf{M}} \mid p \in V_{\mathbf{M}}(w)\} \end{array}$$

$$\begin{array}{rcl}
[\![\neg\varphi]\!]^{\mathbf{M}} & = & W_{\mathbf{M}} - [\![\varphi]\!]^{\mathbf{M}} \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathbf{M}} & = & [\![\varphi_1]\!]^{\mathbf{M}} \cap [\![\varphi_2]\!]^{\mathbf{M}} \\
[\![[\pi]\varphi]\!]^{\mathbf{M}} & = & \{w \in W_{\mathbf{M}} \mid \forall v(\text{ if } (w,v) \in [\![\pi]\!]^{\mathbf{M}} \text{ then } v \in [\![\varphi]\!]^{\mathbf{M}})\}
\end{array}$$

$$\begin{array}{rcl}
[\![a]\!]^{\mathbf{M}} & = & \xrightarrow{a}_{\mathbf{M}} \\
[\![?\varphi]\!]^{\mathbf{M}} & = & \{(w,w) \in W_{\mathbf{M}} \times W_{\mathbf{M}} \mid w \in [\![\varphi]\!]^{\mathbf{M}}\} \\
[\![\pi_1;\pi_2]\!]^{\mathbf{M}} & = & [\![\pi_1]\!]^{\mathbf{M}} \circ [\![\pi_2]\!]^{\mathbf{M}} \\
[\![\pi_1 \cup \pi_2]\!]^{\mathbf{M}} & = & [\![\pi_1]\!]^{\mathbf{M}} \cup [\![\pi_2]\!]^{\mathbf{M}} \\
[\![\pi^*]\!]^{\mathbf{M}} & = & ([\![\pi]\!]^{\mathbf{M}})^*
\end{array}$$

If $w \in W_{\mathbf{M}}$ then we use $\mathbf{M} \models_w \varphi$ for $w \in [\![\varphi]\!]^{\mathbf{M}}$. The paper [$\mathrm{Ba_4Mo_3So_1}$03] proposes to model epistemic actions as epistemic models, with valuations replaced by preconditions. See also: [vB01b, vB06, vD00, $\mathrm{vE_1}$04b, Fa+95, $\mathrm{Ge_2}$99a, $\mathrm{Ko_4}$03, $\mathrm{Ru_0}$04].

Action models for a given language $\mathcal{L}$. Let a set of agents Ag and an epistemic language $\mathcal{L}$ be given. An action model for $\mathcal{L}$ is a triple $A = ([s_0,\ldots,s_{n-1}],\mathrm{pre},T)$ where $[s_0,\ldots,s_{n-1}]$ is a finite list of action states, $\mathrm{pre} : \{s_0,\ldots,s_{n-1}\} \to \mathcal{L}$ assigns a precondition to each action state, and $T : Ag \to \mathcal{P}(\{s_0,\ldots,s_{n-1}\}^2)$ assigns an accessibility relation $\xrightarrow{a}$ to each agent $a \in Ag$.

A pair $\mathbf{A} = (A,s)$ with $s \in \{s_0,\ldots,s_{n-1}\}$ is a pointed action model, where s is the action that actually takes place.

The list ordering of the action states in an action model will play an important role in the definition of the program transformations associated with the action models.

In the definition of action models, $\mathcal{L}$ can be any language that can be interpreted in PDL models. Actions can be executed in PDL models by means of the following product construction:

Action Update. Let a PDL model $\mathbf{M} = (W,V,R)$, a world $w \in W$, and a pointed action model (A,s), with $A = ([s_0,\ldots,s_{n-1}],\mathrm{pre},T)$, be given. Suppose $w \in [\![\mathrm{pre}(s)]\!]^{\mathbf{M}}$. Then the result of executing (A,s) in $(\mathbf{M},w)$ is the model $(\mathbf{M} \otimes A,(w,s))$, with $\mathbf{M} \otimes A = (W',V',R')$, where

$$\begin{array}{rcl}
W' & = & \{(w,s) \mid s \in \{s_0,\ldots,s_{n-1}\}, w \in [\![\mathrm{pre}(s)]\!]^{\mathbf{M}}\} \\
V'(w,s) & = & V(w) \\
R'(a) & = & \{((w,s),(w',s')) \mid (w,w') \in R(a), (s,s') \in T(a)\}.
\end{array}$$

In case there is a set of actual worlds and a set of actual actions, the definition is similar: those world/action pairs survive where the world satisfies the preconditions of the action. See below.

The language of PDL^{DEL} (update PDL) is given by extending the PDL language with update constructions $[A, s]\varphi$, where (A, s) is a pointed action model. The interpretation of $[A, s]\varphi$ in $\mathbf{M}$ is given by:

$$[\![[A, s]\varphi]\!]^{\mathbf{M}} = \{w \in W_{\mathbf{M}} \mid \text{if } \mathbf{M} \models_w \text{pre}(s) \text{ then } (w, s) \in [\![\varphi]\!]^{\mathbf{M} \otimes A}\}.$$

Using $\langle A, s\rangle\varphi$ as shorthand for $\neg[A, s]\neg\varphi$, we see that the interpretation for $\langle A, s\rangle\varphi$ turns out as:

$$[\![\langle A, s\rangle\varphi]\!]^{\mathbf{M}} = \{w \in W_{\mathbf{M}} \mid \mathbf{M} \models_w \text{pre}(s) \text{ and } (w, s) \in [\![\varphi]\!]^{\mathbf{M} \otimes A}\}.$$

Updating with multiple pointed update actions is also possible. A multiple pointed action is a pair (A, S), with A an action model, and S a subset of the state set of A. Extend the language with updates $[A, S]\varphi$, and interpret this as follows:

$$[\![[A, S]\varphi]\!]^{\mathbf{M}} = \{w \in W_{\mathbf{M}} \mid \forall s \in S(\text{if } \mathbf{M} \models_w \text{pre}(s) \\ \text{then } \mathbf{M} \otimes A \models_{(w,s)} \varphi)\}.$$

In [vE$_1$04b] it is shown how dynamic epistemic logic can be reduced to PDL by program transformation. Each action model $\mathbf{A}$ has associated program transformers $T_{ij}^{\mathbf{A}}$ for all states s_i, s_j in the action model, such that the following hold:

Lemma 4.1 (Program Transformation, Van Eijck [vE$_1$04b]). Assume A has n states $s_0, \ldots, s_{n-1}$. Then:

$$\mathbf{M} \models_w [A, s_i][\pi]\varphi \text{ iff } \mathbf{M} \models_w \bigwedge_{j=0}^{n-1} [T_{ij}^{A}(\pi)][A, s_j]\varphi.$$

This lemma allows a reduction of dynamic epistemic logic to PDL, a reduction that we shall implement in the code below.

4.2 Operations on action models

Sequential Composition. If $(\mathbf{A}, S)$ and $(\mathbf{B}, T)$ are multiple pointed action models, their sequential composition $(\mathbf{A}, S) \odot (\mathbf{B}, T)$ is given by:

$$(\mathbf{A}, S) \odot (\mathbf{B}, T) := ((W, \text{pre}, R), S \times T),$$

where

- $W = W_{\mathbf{A}} \times W_{\mathbf{B}}$,
- $\text{pre}(s, t) = \text{pre}(s) \wedge \langle \mathbf{A}, S\rangle \text{pre}(t)$,
- R is given by: $(s, t) \xrightarrow{a} (s', t') \in R$ iff $s \xrightarrow{a} s' \in R_{\mathbf{A}}$ and $t \xrightarrow{a} t' \in R_{\mathbf{B}}$.

The unit element for this operation is the action model

$$\mathbf{1} = ((\{0\}, 0 \mapsto \top, \{0 \xrightarrow{a} 0 \mid a \in Ag\}), \{0\}).$$

Updating an arbitrary epistemic model $\mathbf{M}$ with $\mathbf{1}$ changes nothing.

Non-deterministic Sum. The non-deterministic sum $\oplus$ of multiple pointed action models $(\mathbf{A}, S)$ and $(\mathbf{B}, T)$ is the action model $(\mathbf{A}, S) \oplus (\mathbf{B}, T)$ is given by:

$$(\mathbf{A}, S) \oplus (\mathbf{B}, T) := ((W, \text{pre}, R), S \uplus T),$$

where $\uplus$ denotes disjoint union, and where

- $W = W_{\mathbf{A}} \uplus W_{\mathbf{B}}$,
- $\text{pre} = \text{pre}_{\mathbf{A}} \uplus \text{pre}_{\mathbf{B}}$,
- $R = R_{\mathbf{A}} \uplus R_{\mathbf{B}}$.

The unit element for this operation is called $\mathbf{0}$: the multiple pointed action model given by $((\varnothing, \varnothing, \varnothing), \varnothing)$.

4.3 Logics for communication

Here are some specific action models that can be used to define various languages of communication.

In order to model a **public announcement of** φ, we use the action model $(\mathbf{S}, \{0\})$ with

$$S_{\mathbf{S}} = \{0\}, p_{\mathbf{S}} = 0 \mapsto \varphi, R_{\mathbf{S}} = \{0 \xrightarrow{a} 0 \mid a \in A\}.$$

If we wish to model an **individual message to** b **that** φ, we consider the action model $(\mathbf{S}, \{0\})$ with $S_{\mathbf{S}} = \{0, 1\}$, $p_{\mathbf{S}} = 0 \mapsto \varphi, 1 \mapsto \top$, and $R_{\mathbf{S}} = \{0 \xrightarrow{b} 0, 1 \xrightarrow{b} 1\} \cup \{0 \sim_a 1 \mid a \in A - \{b\}\}$; similarly, for a **group message to** B **that** φ, we use the action model $(\mathbf{S}, \{0\})$ with

$$S_{\mathbf{S}} = \{0, 1\}, p_{\mathbf{S}} = 0 \mapsto \varphi, 1 \mapsto \top, R_{\mathbf{S}} = \{0 \sim_a 1 \mid a \in A - B\}.$$

A **secret individual communication to** b **that** φ is modelled by $(\mathbf{S}, \{0\})$ with

$$\begin{aligned} S_{\mathbf{S}} &= \{0, 1\}, \\ p_{\mathbf{S}} &= 0 \mapsto \varphi, 1 \mapsto \top, \\ R_{\mathbf{S}} &= \{0 \xrightarrow{b} 0\} \cup \{0 \xrightarrow{a} 1 \mid a \in A - \{b\}\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\}, \end{aligned}$$

and a **secret group communication to** B **that** φ by $(\mathbf{S}, \{0\})$ with

$$\begin{aligned} S_{\mathbf{S}} &= \{0, 1\}, \\ p_{\mathbf{S}} &= 0 \mapsto \varphi, 1 \mapsto \top, \\ R_{\mathbf{S}} &= \{0 \xrightarrow{b} 0 \mid b \in B\} \cup \{0 \xrightarrow{a} 1 \mid a \in A - B\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\}. \end{aligned}$$

We model a **test of** φ by the action model $(\mathbf{S}, \{0\})$ with

$$S_{\mathbf{S}} = \{0,1\}, p_{\mathbf{S}} = 0 \mapsto \varphi, 1 \mapsto \top, R_{\mathbf{S}} = \{0 \xrightarrow{a} 1 \mid a \in A\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\},$$

an **individual revelation to** b **of a choice from** $\{\varphi_1, \ldots, \varphi_n\}$ by the action model $(\mathbf{S}, \{1, \ldots, n\})$ with

$$\begin{aligned} S_{\mathbf{S}} &= \{1, \ldots, n\}, \\ p_{\mathbf{S}} &= 1 \mapsto \varphi_1, \ldots, n \mapsto \varphi_n, \\ R_{\mathbf{S}} &= \{s \xrightarrow{b} s \mid s \in S_{\mathbf{S}}\} \cup \{s \xrightarrow{a} s' \mid s, s' \in S_{\mathbf{S}}, a \in A - \{b\}\}, \end{aligned}$$

and a **group revelation to** B **of a choice from** $\{\varphi_1, \ldots, \varphi_n\}$ by the action model $(\mathbf{S}, \{1, \ldots, n\})$ with

$$\begin{aligned} S_{\mathbf{S}} &= \{1, \ldots, n\}, \\ p_{\mathbf{S}} &= 1 \mapsto \varphi_1, \ldots, n \mapsto \varphi_n, \\ R_{\mathbf{S}} &= \{s \xrightarrow{b} s \mid s \in S_{\mathbf{S}}, b \in B\} \cup \{s \xrightarrow{a} s' \mid s, s' \in S_{\mathbf{S}}, a \in A - B\}. \end{aligned}$$

Finally, **transparent informedness of** B **about** φ is represented by the action model $(\mathbf{S}, \{0,1\})$ with $S_{\mathbf{S}} = \{0,1\}$, $p_{\mathbf{S}} = 0 \mapsto \varphi, 1 \mapsto \neg\varphi$, $R_{\mathbf{S}} = \{0 \xrightarrow{a} 0 \mid a \in A\} \cup \{0 \xrightarrow{a} 1 \mid a \in A - B\} \cup \{1 \xrightarrow{a} 0 \mid a \in A - B\} \cup \{1 \xrightarrow{a} 1 \mid a \in A\}$. Transparent informedness of B about φ is the special case of a group revelation of B of a choice from $\{\varphi, \neg\varphi\}$. Note that all but the revelation action models and the transparent informedness action models are single pointed (their sets of actual states are singletons).

On the syntactic side, we now define the corresponding languages. The language for the logic of group announcements is defined by:

$$\begin{aligned} \varphi &::= \top \mid p \mid \neg\varphi \mid \bigwedge[\varphi_1, \ldots, \varphi_n] \mid \bigvee[\varphi_1, \ldots, \varphi_n] \mid \Box_a\varphi \\ &\quad \mid E_B\varphi \mid C_B\varphi \mid [\pi]\varphi \end{aligned}$$

$$\pi ::= \mathbf{1} \mid \mathbf{0} \mid \text{public } B\ \varphi \mid \odot[\pi_1, \ldots, \pi_n] \mid \oplus[\pi_1, \ldots, \pi_n]$$

We use the semantics of **1**, **0**, **public** $B\ \varphi$, and the operations on multiple pointed action models from Section 4.2. For the logic of tests and group announcements, we allow tests $?\varphi$ as basic programs and add the appropriate semantics. For the logic of individual messages, the basic actions are messages to individual agents. In order to give it a semantics, we start out from the semantics of **message** $a\ \varphi$. Finally, the logic of tests, group announcements, and group revelations is as above, but now also allowing revelations from alternatives. For the semantics, we use the semantics of **reveal** $B\ \{\varphi_1, \ldots, \varphi_n\}$.

5 Kripke models

```
module Models where

import List
```

5.1 Agents

```
data Agent = A | B | C | D | E deriving (Eq,Ord,Enum,Bounded)
```

Give the agents appropriate names:

```
a, alice, b, bob, c, carol, d, dave, e, ernie :: Agent
a = A; alice = A
b = B; bob   = B
c = C; carol = C
d = D; dave  = D
e = E; ernie = E
```

Make agents showable in an appropriate way:

```
instance Show Agent where
  show A = "a"; show B = "b"; show C = "c"; show D = "d" ; show E = "e"
```

5.2 Model datatype

It will prove useful to generalize over states. We first define general models, and then specialize to action models and epistemic models. In the following definition, `state` and `formula` are variables over types. We assume that each model carries a list of distinguished states.

```
data Model state formula = Mo
                           [state]
                           [(state,formula)]
                           [Agent]
                           [(Agent,state,state)]
                           [state]
                           deriving (Eq,Ord,Show)
```

Decomposing a pointed model into a list of single-pointed models:

```
decompose ::  Model state formula -> [Model state formula]
decompose (Mo states pre agents rel points) =
  [ Mo states pre agents rel [point] | point <- points ]
```

It is useful to be able to map the precondition table to a function. Here is a general tool for that. Note that the resulting function is partial; if the function argument does not occur in the table, the value is undefined.

```
table2fct :: Eq a => [(a,b)] -> a -> b
table2fct t = \ x -> maybe undefined id (lookup x t)
```

Another useful utility is a function that creates a partition out of an equivalence relation:

```
rel2part :: (Eq a) => [a] -> (a -> a -> Bool) -> [[a]]
rel2part [] r = []
rel2part (x:xs) r = xblock : rel2part rest r
  where
  (xblock,rest) = partition (\ y -> r x y) (x:xs)
```

The *domain* of a model is its list of states:

```
domain :: Model state formula -> [state]
domain (Mo states _ _ _ _) = states
```

The *eval* of a model is its list of state/formula pairs:

```
eval :: Model state formula -> [(state,formula)]
eval (Mo _ pre _ _ _) = pre
```

The *agentList* of a model is its list of agents:

```
agentList :: Model state formula -> [Agent]
agentList (Mo _ _ ags _ _) = ags
```

The *access* of a model is its labelled transition component:

```
access :: Model state formula -> [(Agent,state,state)]
access (Mo _ _ _ rel _) = rel
```

The distinguished points of a model:

```
points :: Model state formula -> [state]
points (Mo _ _ _ _ pnts) = pnts
```

When we are looking at models, we are only interested in generated submodels, with as their domain the distinguished state(s) plus everything that is reachable by an accessibility path.

```
gsm :: Ord state => Model state formula -> Model state formula
gsm (Mo states pre ags rel points) = (Mo states' pre' ags rel' points)
   where
   states' = closure rel ags points
   pre'    = [(s,f)     | (s,f)     <- pre,
                          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
  | xs' == xs = xs
  | otherwise = closure rel agents xs'
     where
     xs' = (nub . sort) (xs ++ (expand rel agents xs))
```

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

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

6 Model minimization under bisimulation

```
module MinBis where

import List
import Models
```

6.1 Partition refinement

Any Kripke model can be simplified by replacing each state s by its bisimulation class $[s]$. 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 [$\text{Ho}_4 71$]. We will use partition refinement, in the spirit of [$\text{Pa}_1\text{Ta}_0 87$]. Here is the algorithm:

- Start out with a partition of the state set where all states with the same precondition function are in the same class. The equality relation to be used to evaluate the precondition function is given as a parameter to the algorithm.

- Given a partition Π, for each block b in Π, 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 Π. Update Π to Π' by replacing each b in Π by the newly found set of sub-blocks for b.

- Halt as soon as $\Pi = \Pi'$.

Looking up and checking of two formulas against a given equivalence relation:

```
lookupFs :: (Eq a,Eq b) =>
      a -> a -> [(a,b)] -> (b -> b -> Bool) -> Bool
lookupFs i j table r = case lookup i table of
  Nothing -> lookup j table == Nothing
  Just f1 -> case lookup j table of
      Nothing -> False
      Just f2 -> r f1 f2
```

The following computes the initial partition, using a particular relation for equivalence of formulas:

```
initPartition :: (Eq a, Eq b) => Model a b -> (b -> b -> Bool) -> [[a]]
initPartition (Mo states pre ags rel points) r =
  rel2part states (\ x y -> lookupFs x y pre r)
```

Refining a partition:

```
refinePartition :: (Eq a, Eq b) =>
                     Model a b -> [[a]] -> [[a]]
refinePartition m p = refineP m p p
  where
  refineP :: (Eq a, Eq b) => Model a b -> [[a]] -> [[a]] -> [[a]]
  refineP m part [] = []
  refineP m part (block:blocks) =
     newblocks ++ (refineP m part blocks)
       where
         newblocks =
           rel2part block (\ x y -> sameAccBlocks m part x y)
```

The following is a function that checks whether two states have the same accessible blocks under a partition:

```
sameAccBlocks :: (Eq a, Eq b) =>
         Model a b -> [[a]] -> a -> a -> Bool
sameAccBlocks m@(Mo states pre ags rel points) part s t =
    and [ accBlocks m part s ag == accBlocks m part t  ag |
                                                 ag <- ags ]
```

The accessible blocks for an agent from a given state, given a model and a partition can be determined by `accBlocks`:

```
accBlocks :: (Eq a, Eq b) =>
               Model a b -> [[a]] -> a -> Agent -> [[a]]
accBlocks m@(Mo states pre ags rel points) part s ag =
    nub [ bl part y | (ag',x,y) <- rel, ag' == ag, x == s ]
```

The block of an object in a partition:

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

Initializing and refining a partition:

```
initRefine :: (Eq a, Eq b) =>
                Model a b -> (b -> b -> Bool) -> [[a]]
initRefine m r = refine m (initPartition m r)
```

The refining process:

```
refine :: (Eq a, Eq b) => Model a b -> [[a]] -> [[a]]
refine m part = if rpart == part
                       then part
                       else refine m rpart
  where rpart = refinePartition m part
```

6.2 Minimization

We now use this to construct the minimal model. Notice the dependence on relational parameter `r`.

```
minimalModel :: (Eq a, Ord a, Eq b, Ord b) =>
                 (b -> b -> Bool) -> Model a b -> Model [a] b
minimalModel r m@(Mo states pre ags rel points) =
  (Mo states' pre' ags rel' points')
     where
     partition = initRefine m r
     states'   = partition
     f         = bl partition
     rel'      = (nub.sort) (map (\ (x,y,z) -> (x, f y, f z)) rel)
     pre'      = (nub.sort) (map (\ (x,y)   -> (f x, y))      pre)
     points'   =  map f points
```

Converting `a`'s into integers, using their position in a given list of `a`'s.

```
convert :: (Eq a, Show a) => [a] -> a  -> Integer
convert = convrt 0
  where
  convrt :: (Eq a, Show a) => Integer -> [a] -> a -> Integer
  convrt n []     x = error (show x ++ " not in list")
  convrt n (y:ys) x | x == y    = n
                    | otherwise = convrt (n+1) ys x
```

Converting an object of type `Model a b` into an object of type `Model Integer b`:

```
conv ::  (Eq a, Show a) =>
           Model a b -> Model Integer b
conv  (Mo worlds val ags acc points) =
      (Mo (map f worlds)
          (map (\ (x,y)   -> (f x, y)) val)
           ags
          (map (\ (x,y,z) -> (x, f y, f z)) acc))
          (map f points)
  where f = convert worlds
```

Use this to rename the blocks into integers:

```
bisim ::  (Eq a, Ord a, Show a, Eq b, Ord b) =>
            (b -> b -> Bool) -> Model a b -> Model Integer b
bisim r = conv . (minimalModel r)
```

7 Formulas, action models and epistemic models

```
module ActEpist where

import List
import Models
import MinBis
import DPLL
```

Module `List` is a standard Haskell module. Module `Models` is described in Chapter 5, and Module `MinBis` in Chapter 6. Module `DPLL` refers to an implementation of Davis, Putnam, Logemann, Loveland (DPLL) theorem proving (not included in this document, but available at `http://www.cwi.nl/~jve/demo`).

7.1 Formulas

Basic propositions:

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

Show these in the standard way, in lower case, with index 0 omitted.

```
instance Show Prop 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
```

Formulas, according to the definition:

$$\varphi \ ::= \ \top \mid p \mid \neg\varphi \mid \bigwedge[\varphi_1,\ldots,\varphi_n] \mid \bigvee[\varphi_1,\ldots,\varphi_n] \mid [\pi]\varphi \mid \ [\mathbf{A}]\varphi$$

$$\pi \ ::= \ a \mid B \mid ?\varphi \mid \bigcirc[\pi_1,\ldots,\pi_n] \mid \bigcup[\pi_1,\ldots,\pi_n] \mid \pi^*$$

Here, p ranges over basic propositions, a ranges over agents, B ranges over non-empty sets of agents, and $\mathbf{A}$ is a multiple pointed action model (see below) $\bigcirc$ denotes sequential composition of a list of programs. We will often write $\bigcirc[\pi_1,\pi_2]$ as $\pi_1;\pi_2$, and $\bigcup[\pi_1,\pi_2]$ as $\pi_1 \cup \pi_2$.

Note that general knowledge among agents B that φ is expressed in this language as $[B]\varphi$, and common knowledge among agents B that φ as $[B^*]\varphi$. Thus, $[B]\varphi$ can be viewed as shorthand for $[\bigcup_{b\in B} b]\varphi$. In case $B = \varnothing$, $[B]\varphi$ turns out to be equivalent to $[?\bot]\varphi$.

For convenience, we have also left in the more traditional way of expressing individual knowledge $\Box_a\varphi$, general knowledge $E_B\varphi$ and common knowledge $C_B\varphi$.

```
data Form = Top
          | Prop Prop
          | Neg  Form
          | Conj [Form]
```

```
                | Disj [Form]
                | Pr Program Form
                | K Agent Form
                | EK [Agent] Form
                | CK [Agent] Form
                | Up AM Form
                deriving (Eq,Ord)

data Program = Ag Agent
             | Ags [Agent]
             | Test Form
             | Conc [Program]
             | Sum  [Program]
             | Star Program
             deriving (Eq,Ord)
```

Some useful abbreviations:

```
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]

xor :: Form -> Form -> Form
xor x y = Disj [ Conj [x, Neg y], Conj [Neg x, y]]
```

The negation of a formula:

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

Show formulas in the standard way:

```
instance Show Form where
  show Top = "T" ; show (Prop p) = show p; show (Neg f) = '-':(show f);
  show (Conj fs)     = '&': show fs
  show (Disj fs)     = 'v': show fs
  show (Pr p f)      = '[': show p ++ "]" ++ show f
  show (K agent f)   = '[': show agent ++ "]" ++ show f
  show (EK agents f) = 'E': show agents ++ show f
  show (CK agents f) = 'C': show agents ++ show f
  show (Up pam f)    = 'A': show (points pam) ++ show f
```

Show programs in a standard way:

```
instance Show Program where
  show (Ag a)       = show a
  show (Ags as)     = show as
  show (Test f)     = '?': show f
  show (Conc ps)    = 'C': show ps
  show (Sum ps)     = 'U': show ps
  show (Star p)     = '(': show p ++ ")*"
```

Programs can get very unwieldy very quickly. As is well known, there is no normalisation procedure for regular expressions. Still, here are some rewriting steps for simplification of programs:

$$
\begin{array}{lcrlcr}
\varnothing & \rightarrow & ?\bot & ?\varphi_1 \cup ?\varphi_2 & \rightarrow & ?(\varphi_1 \vee \varphi_2) \\
?\bot \cup \pi & \rightarrow & \pi & \pi \cup ?\bot & \rightarrow & \pi \\
\bigcup[] & \rightarrow & ?\bot & \bigcup[\pi] & \rightarrow & \pi \\
?\varphi_1 ; ?\varphi_2 & \rightarrow & ?(\varphi_1 \wedge \varphi_2) & ?\top ; \pi & \rightarrow & \pi \\
\pi ; ?\top & \rightarrow & \pi & ?\bot ; \pi & \rightarrow & ?\bot \\
\pi ; ?\bot & \rightarrow & ?\bot & \bigcirc[] & \rightarrow & ?\top \\
\bigcirc[\pi] & \rightarrow & \pi & (?\varphi)^* & \rightarrow & ?\top \\
(?\varphi \cup \pi)^* & \rightarrow & \pi^* & (\pi \cup ?\varphi)^* & \rightarrow & \pi^* \\
\pi^{**} & \rightarrow & \pi^*, & & &
\end{array}
$$

and the $k + m + n$-ary rewriting steps

$$
\bigcup[\pi_1, \ldots, \pi_k, \bigcup[\pi_{k+1}, \ldots, \pi_{k+m}], \pi_{k+m+1}, \ldots, \pi_{k+m+n}] \quad \rightarrow \quad \bigcup[\pi_1, \ldots, \pi_{k+m+n}]
$$

and

$$
\bigcirc[\pi_1, \ldots, \pi_k, \bigcirc[\pi_{k+1}, \ldots, \pi_{k+m}], \pi_{k+m+1}, \ldots, \pi_{k+m+n}] \quad \rightarrow \quad \bigcirc[\pi_1, \ldots, \pi_{k+m+n}].
$$

Simplifying unions by splitting up in test part, accessibility part and rest:

```
splitU :: [Program] -> ([Form],[Agent],[Program])
splitU [] = ([],[],[])
splitU (Test f: ps) = (f:fs,ags,prs)
                       where (fs,ags,prs) = splitU ps
splitU (Ag x: ps) = (fs,union [x] ags,prs)
                       where (fs,ags,prs) = splitU ps
splitU (Ags xs: ps) = (fs,union xs ags,prs)
                       where (fs,ags,prs) = splitU ps
splitU (Sum ps: ps') = splitU (union ps ps')
splitU (p:ps) = (fs,ags,p:prs)
                       where (fs,ags,prs) = splitU ps
```

Simplifying compositions:

```
comprC :: [Program] -> [Program]
comprC [] = []
comprC (Test Top: ps)         = comprC ps
comprC (Test (Neg Top):ps)    = [Test (Neg Top)]
comprC (Test f: Test f': rest) = comprC (Test (canonF (Conj [f,f'])):
                                    rest)
comprC (Conc ps : ps')        = comprC (ps ++ ps')
comprC (p:ps)                 = let ps' = comprC ps
                                 in if ps' == [Test (Neg Top)]
                                    then [Test (Neg Top)] else p: ps'
```

Use this in the code for program simplification:

```
simpl :: Program -> Program
simpl (Ag x)                      = Ag x
simpl (Ags [])                    = Test (Neg Top)
simpl (Ags [x])                   = Ag x
simpl (Ags xs)                    = Ags xs
simpl (Test f)                    = Test (canonF f)
```

Simplifying unions:

```
simpl (Sum prs) =
  let (fs,xs,rest) = splitU (map simpl prs)
      f            = canonF (Disj fs)
  in
    if xs == [] && rest == [] then Test f
    else if xs == [] && f == Neg Top && length rest == 1
      then (head rest)
    else if xs == [] && f == Neg Top then Sum rest
    else if xs == []
      then Sum (Test f: rest)
    else if length xs == 1  && f == Neg Top
      then Sum (Ag (head xs): rest)
    else if length xs == 1 then Sum (Test f: Ag (head xs): rest)
    else if f == Neg Top then Sum (Ags xs: rest)
    else Sum (Test f: Ags xs: rest)
```

Simplifying sequential compositions:

```
simpl (Conc prs) =
    let prs' = comprC (map simpl prs)
    in
      if prs'== []                   then Test Top
      else if length prs' == 1       then head prs'
      else if head prs' == Test Top then Conc (tail prs')
      else                                Conc prs'
```

Simplifying stars:

```
simpl (Star pr) = case simpl pr of
    Test f              -> Test Top
    Sum [Test f, pr']   -> Star pr'
    Sum (Test f: prs') -> Star (Sum prs')
    Star pr'            -> Star pr'
    pr'                 -> Star pr'
```

Property of being a purely propositional formula:

```
pureProp ::  Form -> Bool
pureProp Top       = True
pureProp (Prop _)  = True
pureProp (Neg f)   = pureProp f
pureProp (Conj fs) = and (map pureProp fs)
pureProp (Disj fs) = and (map pureProp fs)
pureProp  _        = False
```

Some example formulas and formula-forming operators:

```
bot, p0, p, p1, p2, p3, p4, p5, p6 :: Form
bot = Neg Top
p0 = Prop (P 0); p = p0; p1  = Prop (P 1); p2 = Prop (P 2)
p3 = Prop (P 3); p4 = Prop (P 4); p5 = Prop (P 5); p6 = Prop (P 6)

q0, q, q1, q2, q3, q4, q5, q6 :: Form
q0 = Prop (Q 0); q = q0; q1 = Prop (Q 1); q2 = Prop (Q 2);
q3 = Prop (Q 3); q4 = Prop (Q 4); q5 = Prop (Q 5); q6 = Prop (Q 6)

r0, r, r1, r2, r3, r4, r5, r6:: Form
r0 = Prop (R 0); r = r0; r1 = Prop (R 1); r2 = Prop (R 2)
r3 = Prop (R 3); r4 = Prop (R 4); r5 = Prop (R 5); r6 = Prop (R 6)

u   = Up ::  AM -> Form -> Form

nkap = Neg (K a p)
nkanp = Neg (K a (Neg p))
nka_p = Conj [nkap,nkanp]
```

7.2 Reducing formulas to canonical form

For computing bisimulations, it is useful to have some notion of equivalence (however crude) for the logical language. For this, we reduce formulas to a canonical form. We will derive canonical forms that are unique up to propositional equivalence, employing a propositional reasoning engine. This is still rather crude, for any modal formula will be treated as a propositional literal. The DPLL (Davis, Putnam, Logemann, Loveland) engine expects clauses represented as lists of integers, so we first have to translate to this format. This translation should start with computing a mapping from positive literals to integers. For the non-propositional operators we use a little bootstrapping, by putting the formula inside the operator in canonical form, using the function `canonF` to be defined below. Also, since the non-propositional operators all behave as Box modalities, we can reduce $\Box\top$ to $\top$:

```
mapping :: Form -> [(Form,Integer)]
mapping f = zip lits [1..k]
  where
  lits = (sort . nub . collect) f
  k    = toInteger (length lits)
  collect :: Form -> [Form]
  collect Top         = []
  collect (Prop p)    = [Prop p]
  collect (Neg f)     = collect f
  collect (Conj fs)   = concat (map collect fs)
  collect (Disj fs)   = concat (map collect fs)
  collect (Pr pr f)   = if canonF f == Top
                          then [] else [Pr pr (canonF f)]
  collect (K ag f)    = if canonF f == Top
```

```
                              then [] else [K ag (canonF f)]
        collect (EK ags f)  = if canonF f == Top
                              then [] else [EK ags (canonF f)]
        collect (CK ags f)  = if canonF f == Top
                              then [] else [CK ags (canonF f)]
        collect (Up pam f)  = if canonF f == Top
                              then [] else [Up pam (canonF f)]
```

The following code corresponds to putting in clausal form, given a mapping for the literals, and using bootstrapping for formulas in the scope of a non-propositional operator. Note that $\Box\top$ is reduced to $\top$, and $\neg\Box\top$ to $\bot$.

```
cf :: (Form -> Integer) -> Form ->
[[Integer]]
cf g (Top)            = []
cf g (Prop p)         = [[g (Prop p)]]
cf g (Pr pr f)        = if canonF f == Top then []
                        else [[g (Pr pr (canonF f))]]
cf g (K ag f)         = if canonF f == Top then []
                        else [[g (K ag (canonF f))]]
cf g (EK ags f)       = if canonF f == Top then []
                        else [[g (EK ags (canonF f))]]
cf g (CK ags f)       = if canonF f == Top then []
                        else [[g (CK ags (canonF f))]]
cf g (Up am f)        = if canonF f == Top then []
                        else [[g (Up am (canonF f))]]
cf g (Conj fs)        = concat (map (cf g) fs)
cf g (Disj fs)        = deMorgan (map (cf g) fs)
```

Negated formulas:

```
cf g (Neg Top)        = [[]]
cf g (Neg (Prop p))   = [[- g (Prop p)]]
cf g (Neg (Pr pr f))  = if canonF f == Top then [[]]
                        else [[- g (Pr pr (canonF f))]]
cf g (Neg (K ag f))   = if canonF f == Top then [[]]
                        else [[- g (K ag (canonF f))]]
cf g (Neg (EK ags f)) = if canonF f == Top then [[]]
                        else [[- g (EK ags (canonF f))]]
cf g (Neg (CK ags f)) = if canonF f == Top then [[]]
                        else [[- g (CK ags (canonF f))]]
cf g (Neg (Up am f))  = if canonF f == Top then [[]]
                        else [[- g (Up am (canonF f))]]
cf g (Neg (Conj fs))  = deMorgan (map (\ f -> cf g (Neg f)) fs)
cf g (Neg (Disj fs))  = concat   (map (\ f -> cf g (Neg f)) fs)
cf g (Neg (Neg f))    = cf g f
```

In order to explain the function `deMorgan`, we recall De Morgan's disjunction distribution which is the logical equivalence of the following expressions:

$$\varphi \vee (\psi_1 \wedge \cdots \wedge \psi_n) \leftrightarrow (\varphi \vee \psi_1) \wedge \cdots \wedge (\varphi \vee \psi_n).$$

Now the following is the code for De Morgan's disjunction distribution (for the case of a disjunction of a list of clause sets):

```
deMorgan :: [[[Integer]]] -> [[Integer]]
deMorgan [] = [[]]
deMorgan [cls] = cls
deMorgan (cls:clss) = deMorg cls (deMorgan clss)
  where
  deMorg :: [[Integer]] -> [[Integer]] -> [[Integer]]
  deMorg cls1 cls2 = (nub . concat) [ deM cl cls2 | cl <- cls1 ]
  deM :: [Integer] -> [[Integer]]  -> [[Integer]]
  deM cl cls = map (fuseLists cl) cls
```

Function `fuseLists` keeps the literals in the clauses ordered.

```
fuseLists :: [Integer] -> [Integer] -> [Integer]
fuseLists [] ys = ys
fuseLists xs [] = xs
fuseLists (x:xs) (y:ys) | abs x < abs y  = x:(fuseLists xs (y:ys))
                        | abs x == abs y = if x == y
                                           then x:(fuseLists xs ys)
                                           else if x > y
                                             then x:y:(fuseLists xs ys)
                                             else y:x:(fuseLists xs ys)
                        | abs x > abs y  = y:(fuseLists (x:xs) ys)
```

Given a mapping for the positive literals, the satisfying valuations of a formula can be collected from the output of the DPLL process. Here `dp` is the function imported from the module `DPLL`.

```
satVals :: [(Form,Integer)] -> Form -> [[Integer]]
satVals t f = (map fst . dp) (cf (table2fct t) f)
```

Two formulas are propositionally equivalent if they have the same sets of satisfying valuations, computed on the basis of a literal mapping for their conjunction:

```
propEquiv :: Form -> Form -> Bool
propEquiv f1 f2 = satVals g f1 == satVals g f2
  where g = mapping (Conj [f1,f2])
```

A formula is a (propositional) contradiction if it is propositionally equivalent to `Neg Top`, or equivalently, to `Disj []`:

```
contrad :: Form -> Bool
contrad f = propEquiv f (Disj [])
```

A formula is (propositionally) consistent if it is not a propositional contradiction:

```
consistent :: Form -> Bool
consistent = not . contrad
```

Use the set of satisfying valuations to derive a canonical form:

```
canonF :: Form -> Form
canonF f = if (contrad (Neg f))
             then Top
             else if fs == []
             then Neg Top
             else if length fs == 1
             then head fs
             else Disj fs
  where g   = mapping f
        nss = satVals g f
        g'  = \ i -> head [ form | (form,j) <- g, i == j ]
        h   = \ i -> if i < 0 then Neg (g' (abs i)) else g' i
        h'  = \ xs -> map h xs
        k   = \ xs -> if xs == []
                         then Top
                         else if length xs == 1
                                 then head xs
                                 else Conj xs
        fs  = map k (map h' nss)
```

This gives:

```
ActEpist> canonF p
p
ActEpist> canonF (Conj [p,Top])
p
ActEpist> canonF (Conj [p,q,Neg r])
&[p,q,-r]
ActEpist> canonF (Neg (Disj [p,(Neg p)]))
-T
ActEpist> canonF (Disj [p,q,Neg r])
v[p,&[-p,q],&[-p,-q,-r]]
ActEpist> canonF (K a (Disj [p,q,Neg r]))
[a]v[p,&[-p,q],&[-p,-q,-r]]
ActEpist> canonF (Conj  [p, Conj [q,Neg r]])
&[p,q,-r]
ActEpist> canonF (Conj  [p, Disj [q,Neg (K a (Disj []))]])
v[&[p,q],&[p,-q,-[a]-T]]
ActEpist> canonF (Conj  [p, Disj [q,Neg (K a (Conj []))]])
&[p,q]
```

7.3 Action models and epistemic models

Action models and epistemic models are built from states. We assume states are represented by integers:

```
type State = Integer
```

Epistemic models are models where the states are of type `State`, and the precondition function assigns lists of basic propositions (this specializes the precondition function to a valuation).

```
type EM = Model State [Prop]
```

Find the valuation of an epistemic model:

```
valuation  :: EM -> [(State,[Prop])]
valuation = eval
```

Action models are models where the states are of type `State`, and the precondition function assigns objects of type `Form`. The only difference between an action model and a static model is in the fact that action models have a precondition function that assigns a formula instead of a set of basic propositions.

```
type AM = Model State Form
```

The preconditions of an action model:

```
preconditions :: AM -> [Form]
preconditions (Mo states pre ags acc points) =
   map (table2fct pre) points
```

Sometimes we need a single precondition:

```
precondition :: AM -> Form
precondition am = canonF (Conj (preconditions am))
```

The zero action model **0**:

```
zero :: [Agent] -> AM
zero ags = (Mo [] [] ags [] [])
```

The purpose of action models is to define relations on the class of all static models. States with precondition $\perp$ can be pruned from an action model. For this we define a specialized version of the `gsm` function:

```
gsmAM :: AM -> AM
gsmAM (Mo states pre ags acc points) =
  let
    points' = [ p | p <- points, consistent (table2fct pre p) ]
    states' = [ s | s <- states, consistent (table2fct pre s) ]
    pre'    = filter (\ (x,_) -> elem x states') pre
    f       = \ (_,s,t) -> elem s states' && elem t states'
    acc'    = filter f acc
  in
  if points' == []
     then zero ags
     else gsm (Mo states' pre' ags acc' points')
```

7.4 Program transformation

For every action model A with states $s_0, \dots, s_{n-1}$ we define a set of n^2 program transformers $T^A_{i,j}$ $(0 \le i < n, 0 \le j < n)$, as follows [vE$_1$04b]:

$$
\begin{aligned}
T^A_{ij}(a) &= \begin{cases} ?\text{pre}(s_i); a & \text{if } s_i \xrightarrow{a} s_j, \\ ?\bot & \text{otherwise} \end{cases} \\
T^A_{ij}(?\varphi) &= \begin{cases} ?(\text{pre}(s_i) \wedge [A, s_i]\varphi) & \text{if } i = j, \\ ?\bot & \text{otherwise} \end{cases} \\
T^A_{ij}(\pi_1; \pi_2) &= \bigcup_{k=0}^{n-1} (T^A_{ik}(\pi_1); T^A_{kj}(\pi_2)) \\
T^A_{ij}(\pi_1 \cup \pi_2) &= T^A_{ij}(\pi_1) \cup T^A_{ij}(\pi_2) \\
T^A_{ij}(\pi^*) &= K^A_{ijn}(\pi)
\end{aligned}
$$

where $K^A_{ijk}(\pi)$ is a (transformed) program for all the π^* paths from s_i to s_j that can be traced through A while avoiding a pass through intermediate states s_k and higher. Thus, $K^A_{ijn}(\pi)$ is a program for all the π^* paths from s_i to s_j that can be traced through A, period.

$K^A_{ijk}(\pi)$ is defined by recursing on k, as follows:

$$
K^A_{ij0}(\pi) = \begin{cases} ?\top \cup T^A_{ij}(\pi) & \text{if } i = j, \\ T^A_{ij}(\pi) & \text{otherwise} \end{cases}
$$

$$
K^A_{ij(k+1)}(\pi) = \begin{cases} (K^A_{kkk}(\pi))^* & \text{if } i = k = j, \\ (K^A_{kkk}(\pi))^*; K^A_{kjk}(\pi) & \text{if } i = k \neq j, \\ K^A_{ikk}(\pi); (K^A_{kkk}(\pi))^* & \text{if } i \neq k = j, \\ K^A_{ijk}(\pi) \cup (K^A_{ikk}(\pi); (K^A_{kkk}(\pi))^*; K^A_{kjk}(\pi)) & \text{otherwise.} \end{cases}
$$

Lemma 7.1 (Kleene Path)**.** Suppose $(w, w') \in [\![T^A_{ij}(\pi)]\!]^{\mathbf{M}}$ iff there is a π path from (w, s_i) to (w', s_j) in $\mathbf{M} \otimes A$. Then $(w, w') \in [\![K^A_{ijn}(\pi)]\!]^{\mathbf{M}}$ iff there is a π^* path from (w, s_i) to (w', s_j) in $\mathbf{M} \otimes A$.

The Kleene path lemma is the key ingredient in the proof of the following program transformation lemma.

Lemma 7.2 (Program Transformation)**.** Assume A has n states $s_0, \dots, s_{n-1}$. Then:

$$
\mathbf{M} \models_w [A, s_i][\pi]\varphi \text{ iff } \mathbf{M} \models_w \bigwedge_{j=0}^{n-1} [T^A_{ij}(\pi)][A, s_j]\varphi.
$$

The implementation of the program transformation functions is given here:

```
transf :: AM -> Integer -> Integer -> Program -> Program
transf am@(Mo states pre allAgs acc points) i j (Ag ag) =
   let
     f = table2fct pre i
   in
   if elem (ag,i,j) acc && f == Top          then Ag ag
   else if elem (ag,i,j) acc && f /= Neg Top then Conc [Test f, Ag ag]
   else Test (Neg Top)
transf am@(Mo states pre allAgs acc points) i j (Ags ags) =
   let ags' = nub [ a | (a,k,m) <- acc, elem a ags, k == i, m == j ]
       ags1 = intersect ags ags'
       f    = table2fct pre i
   in
     if ags1 == [] || f == Neg Top        then Test (Neg Top)
     else if f == Top && length ags1 == 1 then Ag (head ags1)
     else if f == Top                     then Ags ags1
     else Conc [Test f, Ags ags1]
transf am@(Mo states pre allAgs acc points) i j (Test f) =
   let
     g = table2fct pre i
   in
   if i == j
      then Test (Conj [g,(Up am f)])
      else Test (Neg Top)
transf am@(Mo states pre allAgs acc points) i j (Conc [])  =
  transf am i j (Test Top)
transf am@(Mo states pre allAgs acc points) i j (Conc [p]) =
  transf am i j p
transf am@(Mo states pre allAgs acc points) i j (Conc (p:ps)) =
  Sum [ Conc [transf am i k p, transf am k j (Conc ps)] | k <- [0..n] ]
    where n = toInteger (length states - 1)
transf am@(Mo states pre allAgs acc points) i j (Sum [])  =
  transf am i j (Test (Neg Top))
transf am@(Mo states pre allAgs acc points) i j (Sum [p]) =
  transf am i j p
transf am@(Mo states pre allAgs acc points) i j (Sum ps) =
  Sum [ transf am i j p | p <- ps ]
transf am@(Mo states pre allAgs acc points) i j (Star p) =
  kleene am i j n p
    where n = toInteger (length states)
```

The following is the implementation of $K_{ijk}^{\mathbf{A}}$:

```
kleene ::  AM -> Integer -> Integer -> Integer -> Program -> Program
kleene am i j 0 pr =
  if i == j
    then Sum [Test Top, transf am i j pr]
    else transf am i j pr
kleene am i j k pr
  | i == j && j == pred k = Star (kleene am i i i pr)
  | i == pred k           =
    Conc [Star (kleene am i i i pr), kleene am i j i pr]
```

```
        | j == pred k          =
          Conc [kleene am i j j pr, Star (kleene am j j j pr)]
        | otherwise            =
            Sum [kleene am i j k' pr,
                 Conc [kleene am i k' k' pr,
                       Star (kleene am k' k' k' pr), kleene am k' j k' pr]]
            where k' = pred k
```

Transformation plus simplification:

```
tfm ::  AM -> Integer -> Integer -> Program -> Program
tfm am i j pr = simpl (transf am i j pr)
```

The program transformations can be used to translate Update PDL to PDL, as follows:

$$
\begin{array}{rclrcl}
t(\top) & = & \top & t(p) & = & p \\
t(\neg\varphi) & = & \neg t(\varphi) & t(\varphi_1 \wedge \varphi_2) & = & t(\varphi_1) \wedge t(\varphi_2) \\
t([\pi]\varphi) & = & [r(\pi)]t(\varphi) & t([A,s]\top) & = & \top
\end{array}
$$

$$
\begin{array}{rcl}
t([A,s]p) & = & t(\mathrm{pre}(s)) \to p \\
t([A,s]\neg\varphi) & = & t(\mathrm{pre}(s)) \to \neg t([A,s]\varphi) \\
t([A,s](\varphi_1 \wedge \varphi_2)) & = & t([A,s]\varphi_1) \wedge t([A,s]\varphi_2) \\
t([A,s_i][\pi]\varphi) & = & \bigwedge_{j=0}^{n-1} [T_{ij}^A(r(\pi))] t([A,s_j]\varphi) \\
t([A,s][A',s']\varphi) & = & t([A,s]t([A',s']\varphi)) \\
t([A,S]\varphi) & = & \bigwedge_{s \in S} t[A,s]\varphi)
\end{array}
$$

$$
\begin{array}{rclrcl}
r(a) & = & a & r(B) & = & B \\
r(?\varphi) & = & ?t(\varphi) & r(\pi_1;\pi_2) & = & r(\pi_1);r(\pi_2) \\
r(\pi_1 \cup \pi_2) & = & r(\pi_1) \cup r(\pi_2) & r(\pi^*) & = & (r(\pi))^*.
\end{array}
$$

The correctness of this translation follows from direct semantic inspection, using the program transformation lemma for the translation of formulas of type $[A, s_i][\pi]\varphi$.

The crucial clauses in this translation procedure are those for formulas of the forms $[A, S]\varphi$ and $[A, s]\varphi$, and more in particular the one for formulas of the form $[A, s][\pi]\varphi$. It makes sense to give separate functions for the steps that pull the update model through program π given formula φ.

```
step0, step1 :: AM -> Program -> Form -> Form
step0 am@(Mo states pre allAgs acc []) pr f = Top
step0 am@(Mo states pre allAgs acc [i]) pr f = step1 am pr f
step0 am@(Mo states pre allAgs acc  is) pr f =
  Conj [ step1 (Mo states pre allAgs acc [i]) pr f | i <- is ]
step1 am@(Mo states pre allAgs acc [i]) pr f =
   Conj [ Pr (transf am i j (rpr pr))
               (Up (Mo states pre allAgs acc [j]) f) | j <- states ]
```

Perform a single step, and put in canonical form:

```
step :: AM -> Program -> Form -> Form
step am pr f = canonF (step0 am pr f)

t :: Form -> Form
t Top = Top
t (Prop p) = Prop p
t (Neg f) = Neg (t f)
t (Conj fs) = Conj (map t fs)
t (Disj fs) = Disj (map t fs)
t (Pr pr f) = Pr (rpr pr) (t f)
t (K x f) = Pr (Ag x) (t f)
t (EK xs f)  = Pr (Ags xs) (t f)
t (CK xs f)  = Pr (Star (Ags xs)) (t f)
```

Translations of formulas starting with an action model update:

```
t (Up am@(Mo states pre allAgs acc [i]) f) = t' am f
t (Up am@(Mo states pre allAgs acc  is) f)  =
   Conj [ t' (Mo states pre allAgs acc [i]) f | i <- is ]
```

Translations of formulas starting with a single pointed action model update are performed by `t'`:

```
t' :: AM -> Form -> Form
t' am Top            = Top
t' am (Prop p)       = impl (precondition am) (Prop p)
t' am (Neg f)        =  Neg (t' am f)
t' am (Conj fs)      = Conj (map (t' am) fs)
t' am (Disj fs)      = Disj (map (t' am) fs)
t' am (K x f)        = t' am (Pr (Ag x) f)
t' am (EK xs f)      = t' am (Pr (Ags xs) f)
t' am (CK xs f)      = t' am (Pr (Star (Ags xs)) f)
t' am (Up am'f)      = t' am (t (Up am' f))
```

The crucial case is an update action having scope over a program. We may assume that the update action is single pointed.

```
t' am@(Mo states pre allAgs acc [i]) (Pr pr f) =
   Conj [ Pr (transf am i j (rpr pr))
              (t' (Mo states pre allAgs acc [j]) f) | j <- states ]
t' am@(Mo states pre allAgs acc is) (Pr pr f) =
   error "action model not single pointed"
```

Translations for programs:

```
rpr :: Program -> Program
rpr (Ag x)       = Ag x
rpr (Ags xs)     = Ags xs
rpr (Test f)     = Test (t f)
rpr (Conc ps)    = Conc (map rpr ps)
rpr (Sum  ps)    = Sum  (map rpr ps)
rpr (Star p)     = Star (rpr p)
```

Translating and putting in canonical form:

```
tr :: Form -> Form
tr = canonF . t
```

Some example translations:

```
ActEpist> tr (Up (public p) (Pr (Star (Ags [b,c])) p))
T
ActEpist> tr (Up (public (Disj [p,q])) (Pr (Star (Ags [b,c])) p))
[(U[?T,C[?v[p,q],[b,c]]])*]v[p,&[-p,-q]]
ActEpist> tr (Up (groupM [a,b] p) (Pr (Star (Ags [b,c])) p))
[C[C[(U[?T,C[?p,[b,c]]])*,C[?p,[c]]],(U[U[?T,[b,c]],
        C[c,(U[?T,C[?p,[b,c]]])*,C[?p,[c]]]])*]]p
ActEpist> tr (Up (secret [a,b] p) (Pr (Star (Ags [b,c])) p))
[C[C[(U[?T,C[?p,[b]]])*,C[?p,[c]]],(U[U[?T,[b,c]],
        C[?-T,(U[?T,C[?p,[b]]])*,C[?p,[c]]]])*]]p
```

8 Semantics

```
module Semantics
where

import List
import Char
import Models
import Display
import MinBis
import ActEpist
import DPLL
```

8.1 Semantics implementation

The group alternatives of group of agents a are the states that are reachable through $\bigcup_{a \in A} R_a$.

```
groupAlts :: [(Agent,State,State)] -> [Agent] -> State -> [State]
groupAlts rel agents current =
  (nub . sort . concat) [ alternatives rel a current | a <- agents ]
```

The common knowledge alternatives of group of agents a are the states that are reachable through a finite number of R_a links, for $a \in A$.

```
commonAlts :: [(Agent,State,State)] -> [Agent] -> State -> [State]
commonAlts rel agents current =
  closure rel agents (groupAlts rel agents current)
```

The model update function takes a static model and and action model and returns an object of type `Model (State,State) [Prop]`. The up function takes an epistemic model and an action model and returns an epistemic model. Its states are the `(State,State)` pairs that result from the cartesian product construction described in [$\mathrm{Ba_4Mo_3So_1}$99]. Note that the update function uses the truth definition (given below as `isTrueAt`).

We will set up matters in such way that updates with action models get their list of agents from the epistemic model that gets updated. For this, we define:

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

up :: EM -> FAM -> Model (State,State) [Prop]
up m@(Mo worlds val ags acc points) fam =
   Mo worlds' val' ags acc' points'
   where
   am@(Mo states pre _ susp actuals) = fam ags
   worlds' = [ (w,s) | w <- worlds, s <- states,
                       formula <- maybe [] (\ x -> [x]) (lookup s pre),
                       isTrueAt w m formula                         ]
   val'    = [ ((w,s),props) | (w,props) <- val,
                                s        <- states,
                                elem (w,s) worlds'                  ]
   acc'    = [ (ag1,(w1,s1),(w2,s2)) | (ag1,w1,w2) <- acc,
                                       (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'                           ]
```

An action model is tiny if its action list is empty or a singleton list:

```
tiny :: FAM -> Bool
tiny fam = length actions <= 1
  where actions = domain (fam [minBound..maxBound])
```

The appropriate notion of equivalence for the base case of the bisimulation for epistemic models is "having the same valuation".

```
sameVal :: [Prop] -> [Prop] -> Bool
sameVal ps qs = (nub . sort) ps ==  (nub . sort) qs
```

Bisimulation minimal version of generated submodel of update result for epistemic model and pointed action models:

```
upd :: EM -> FAM -> EM
upd sm fam = if tiny fam then conv (up sm fam)
             else bisim (sameVal) (up sm fam)
```

Non-deterministic update with a list of pointed action models:

```
upds  :: EM -> [FAM] -> EM
upds = foldl upd
```

At last we have all ingredients for the truth definition.

```
isTrueAt :: State -> EM -> Form -> Bool
isTrueAt w m Top = True
isTrueAt w m@(Mo worlds val ags acc pts) (Prop p) =
  elem p (concat [ props | (w',props) <- val, w'==w ])
isTrueAt w m (Neg f)   = not (isTrueAt w m f)
isTrueAt w m (Conj fs) = and (map (isTrueAt w m) fs)
isTrueAt w m (Disj fs) = or  (map (isTrueAt w m) fs)
```

The clauses for individual knowledge, general knowledge and common knowledge use the functions `alternatives`, `groupAlts` and `commonAlts` to compute the relevant accessible worlds:

```
isTrueAt w m@(Mo worlds val ags acc pts) (K ag f) =
  and (map (flip ((flip isTrueAt) m) f) (alternatives acc ag w))
isTrueAt w m@(Mo worlds val ags acc pts) (EK agents f) =
  and (map (flip ((flip isTrueAt) m) f) (groupAlts acc agents w))
isTrueAt w m@(Mo worlds val ags acc pts) (CK agents f) =
  and (map (flip ((flip isTrueAt) m) f) (commonAlts acc agents w))
```

In the clause for $[\mathbf{M}]\varphi$, the result of updating the static model M with action model $\mathbf{M}$ may be undefined, but in this case the precondition $P(s_0)$ of the designated state s_0 of $\mathbf{M}$ will fail in the designated world w_0 of M. By making the clause for $[\mathbf{M}]\varphi$ check for $M \models_{w_0} P(s_0)$, truth can be defined as a total function.

```
isTrueAt w m@(Mo worlds val ags rel pts) (Up am f) =
 and [ isTrue  m' f |
       m' <- decompose (upd (Mo worlds val ags rel [w]) (\ ags -> am))]
```

Checking for truth in *all* the designated points of an epistemic model:

```
isTrue :: EM -> Form -> Bool
isTrue (Mo worlds val ags rel pts) form =
   and [ isTrueAt w (Mo worlds val ags rel pts) form | w <- pts ]
```

8.2 Tools for constructing epistemic models

The following function constructs an initial epistemic model where the agents are completely ignorant about their situation, as described by a list of basic propositions. The input is a list of basic propositions used for constructing the valuations.

```
initE :: [Prop] -> [Agent] -> EM
initE allProps ags = (Mo worlds val ags accs points)
  where
    worlds = [0..(2^k - 1)]
    k      = length allProps
    val    = zip worlds (sortL (powerList allProps))
    accs   = [ (ag,st1,st2) | ag  <- ags,
                              st1 <- worlds,
                              st2 <- worlds      ]
    points = worlds
```

This uses the following utilities:

```
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)
```

Some initial models:

```
e00 :: EM
e00 = initE [P 0] [a,b]

e0 :: EM
e0 = initE [P 0,Q 0] [a,b,c]
```

8.3 From communicative actions to action models

Computing the update for a public announcement:

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

Public announcements are S5 models:

```
DEMO> showM (public p [a,b,c])
==> [0]
[0]
(0,p)
(a,[[0]])
(b,[[0]])
(c,[[0]])
```

Computing the update for passing a group announcement to a list of agents: the other agents may or may not be aware of what is going on. In the limit case where the message is passed to all agents, the message is a public announcement.

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

Group announcements are S5 models:

```
Semantics> showM (groupM [a,b] p [a,b,c,d,e])
=> [0]
[0,1]
(0,p)(1,T)
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0,1]])
(d,[[0,1]])
(e,[[0,1]])
```

Computing the update for an individual message to b that φ:

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

Another special case of a group message is a test. Tests are updates that messages to the empty group:

```
test :: Form -> FAM
test = groupM []
```

Computing the update for passing a *secret* message to a list of agents: the other agents remain unaware of the fact that something goes on. In the limit case where the secret is divulged to all agents, the secret becomes a public update.

```
secret :: [Agent] -> Form -> FAM
secret agents form all_agents =
  if sort agents == sort all_agents
    then public form agents
    else
      (Mo
         [0,1]
         [(0,form),(1,Top)]
          all_agents
         ([ (a,0,0) | a <- agents ]
           ++ [ (a,0,1) | a <- all_agents \\ agents ]
           ++ [ (a,1,1) | a <- all_agents           ])
         [0])
```

Secret messages are KD45 models:

```
DEMO> showM (secret [a,b] p [a,b,c])
==> [0]
[0,1]
(0,p)(1,T)
(a,[([],[0]),([],[1])])
(b,[([],[0]),([],[1])])
(c,[([0],[1])])
```

Here is a multiple pointed action model for the communicative action of revealing one of a number of alternatives to a list of agents, in such a way that it is common knowledge that one of the alternatives gets revealed (in [$Ba_4Mo_3So_1$03] this is called *common knowledge of alternatives*).

```
reveal :: [Agent] -> [Form] -> FAM
reveal ags forms all_agents =
  (Mo
     states
     (zip states forms)
     all_agents
     ([ (ag,s,s) | s <- states, ag <- ags ]
       ++
      [ (ag,s,s') | s <- states, s' <- states, ag <- others ])
     states)
    where states = map fst (zip [0..] forms)
          others = all_agents \\ ags
```

Here is an action model for the communication that reveals to a one of p_1, q_1, r_1.

```
Semantics> showM (reveal [a] [p1,q1,r1] [a,b])
==> [0,1,2]
[0,1,2]
(0,p1)(1,q1)(2,r1)
(a,[[0],[1],[2]])
(b,[[0,1,2]])
```

A group of agents B gets (transparently) informed about a formula φ if B get to know φ when φ is true, and B get to know the negation of φ otherwise. Transparency means that all other agents are aware of the fact that B get informed about φ, i.e., the other agents learn that $(\varphi \rightarrow C_B\varphi) \wedge (\neg\varphi \rightarrow C_B\neg\varphi)$. This action model can be defined in terms of `reveal`, as follows:

```
info :: [Agent] -> Form -> FAM
info agents form =
  reveal agents [form, negation form]
```

An example application:

```
Semantics> showM (upd e0 (info [a,b] q))
==> [0,1,2,3]
[0,1,2,3]
(0,[])(1,[p])(2,[q])(3,[p,q])
(a,[[0,1],[2,3]])
(b,[[0,1],[2,3]])
(c,[[0,1,2,3]])

Semantics> isTrue (upd e0 (info [a,b] q)) (CK [a,b] q)
False
Semantics> isTrue (upd e0 (groupM [a,b] q)) (CK [a,b] q)
True
```

Slightly different is informing a set of agents about what is actually the case with respect to formula φ:

```
infm :: EM -> [Agent] -> Form -> FAM
infm m ags f = if isTrue m f
                  then groupM ags f
                  else if isTrue m (Neg f)
                         then groupM ags (Neg f)
                         else one
```

And the corresponding thing for public announcement:

```
publ :: EM -> Form -> FAM
publ m f = if isTrue m f
              then public f
              else if isTrue m (Neg f)
                     then public (Neg f)
                     else one
```

8.4 Operations on action models

The trivial update action model is a special case of public announcement. Call this the `one` action model, for it behaves as 1 for the operation $\otimes$ of action model composition.

```
one :: FAM
one = public Top
```

Composition $\otimes$ of multiple pointed action models.

```
cmpP :: FAM -> FAM -> [Agent] -> Model (State,State) Form
cmpP fam1 fam2 ags =
 (Mo nstates npre ags nsusp npoints)
   where m@(Mo states pre _ susp ss) = fam1 ags
         (Mo states' pre' _ susp' ss') = fam2 ags
         npoints = [ (s,s') | s <- ss, s' <- ss' ]
         nstates = [ (s,s') | s <- states, s' <- states' ]
         npre    = [ ((s,s'), g) | (s,f)     <- pre,
                                   (s',f')   <- pre',
                                    g        <- [computePre m f f']  ]
         nsusp   = [ (ag,(s1,s1'),(s2,s2')) | (ag,s1,s2)    <- susp,
                                              (ag',s1',s2') <- susp',
                                               ag == ag'             ]
```

The utility function for this can be described as follows: compute the new precondition of a state pair. If the preconditions of the two states are purely propositional, we know that the updates at the states commute and that their combined precondition is the conjunction of the two preconditions, provided this conjunction is not a contradiction. If one of the states has a precondition that is not purely propositional, we have to take the epistemic effect of the update into account in the new precondition.

```
computePre  :: AM -> Form -> Form -> Form
computePre m g g'  | pureProp conj = conj
                   | otherwise     = Conj [ g, Neg (Up m (Neg g')) ]
  where conj    = canonF (Conj [g,g'])
```

Compose pairs of multiple pointed action models, and reduce the result to its simplest possible form under action emulation.

```
cmpFAM :: FAM -> FAM -> FAM
-- cmpFAM fam fam' ags = aePmod (cmpP fam fam' ags)
cmpFAM fam fam' ags = conv (cmpP fam fam' ags)
```

Use `one` as unit for composing lists of FAMs:

```
cmp :: [FAM] -> FAM
cmp = foldl cmpFAM one
```

Here is the result of composing two messages:

```
Semantics> showM (cmp [groupM [a,b] p, groupM [b,c] q] [a,b,c])
==> [0]
[0,1,2,3]
(0,&[p,q])(1,p)(2,q)(3,T)
(a,[[0,1],[2,3]])
(b,[[0],[1],[2],[3]])
(c,[[0,2],[1,3]])
```

This gives the resulting action model. Here is the result of composing the messages in the reverse order. The two action models are bisimilar under the renaming $1 \mapsto 2, 2 \mapsto 1$.

```
==> [0]
[0,1,2,3]
(0,&[p,q])(1,q)(2,p)(3,T)
(a,[[0,2],[1,3]])
(b,[[0],[1],[2],[3]])
(c,[[0,1],[2,3]])
```

The following is an illustration of an observation from [$\mathrm{vE_1}$04a]:

```
m2  = initE [P 0,Q 0] [a,b,c]
psi = Disj[Neg(K b p),q]

Semantics>  showM (upds m2 [message a psi, message b p])
==> [1,4]
[0,1,2,3,4,5]
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q])
(5,[p,q])
(a,[[0,1,2,3,4,5]])
(b,[[0,2,3,5],[1,4]])
(c,[[0,1,2,3,4,5]])
```

```
Semantics> showM (upds m2 [message b p, message a psi])
==> [7]
[0,1,2,3,4,5,6,7,8,9,10]
(0,[])(1,[])(2,[p])(3,[p])(4,[p])
(5,[q])(6,[q])(7,[p,q])(8,[p,q])(9,[p,q])
(10,[p,q])
(a,[[0,3,5,7,9],[1,2,4,6,8,10]])
(b,[[0,1,3,4,5,6,9,10],[2,7,8]])
(c,[[0,1,2,3,4,5,6,7,8,9,10]])
```

Power of action models:

```
pow :: Int -> FAM -> FAM
pow n fam = cmp (take n (repeat fam))
```

Non-deterministic sum $\oplus$ of multiple-pointed action models:

```
ndSum' :: FAM -> FAM -> FAM
ndSum' fam1 fam2 ags = (Mo states val ags acc ss)
  where
       (Mo states1 val1 _ acc1 ss1) = fam1 ags
       (Mo states2 val2 _ acc2 ss2) = fam2 ags
       f   = \ x -> toInteger (length states1) + x
       states2' = map f states2
       val2'    = map (\ (x,y)   -> (f x, y)) val2
       acc2'    = map (\ (x,y,z) -> (x, f y, f z)) acc2
       ss       = ss1 ++ map f ss2
       states   = states1 ++ states2'
       val      = val1 ++ val2'
       acc      = acc1 ++ acc2'
```

Example action models:

```
am0 = ndSum' (test p) (test (Neg p)) [a,b,c]

am1 = ndSum' (test p) (ndSum' (test q) (test r)) [a,b,c]
```

Examples of minimization for action emulation:

```
Semantics>  showM am0
==> [0,2]
[0,1,2,3]
(0,p)(1,T)(2,-p)(3,T)
(a,[([0],[1]),([2],[3])])
(b,[([0],[1]),([2],[3])])
(c,[([0],[1]),([2],[3])])

Semantics> showM (aePmod am0)
==> [0]
[0]
(0,T)
(a,[[0]])
(b,[[0]])
(c,[[0]])
```

```
Semantics>  showM am1
==> [0,2,4]
[0,1,2,3,4,5]
(0,p)(1,T)(2,q)(3,T)(4,r)
(5,T)
(a,[([0],[1]),([2],[3]),([4],[5])])
(b,[([0],[1]),([2],[3]),([4],[5])])
(c,[([0],[1]),([2],[3]),([4],[5])])

Semantics> showM (aePmod am1)
==> [0]
[0,1]
(0,v[p,&[-p,q],&[-p,-q,r]])(1,T)
(a,[([0],[1])])
(b,[([0],[1])])
(c,[([0],[1])])
```

Non-deterministic sum $\oplus$ of multiple-pointed action models, reduced for action emulation:

```
ndSum :: FAM -> FAM -> FAM
ndSum fam1 fam2 ags = (ndSum' fam1 fam2) ags
```

Notice the difference with the definition of alternative composition of Kripke models for processes given in [Ho$_3$98, Ch 4]. The `zero` action model is the 0 for the $\oplus$ operation, so it can be used as the base case in the following list version of the $\oplus$ operation:

```
ndS :: [FAM] -> FAM
ndS = foldl ndSum zero
```

Performing a test whether φ and announcing the result:

```
testAnnounce :: Form -> FAM
testAnnounce form = ndS [ cmp [ test form, public form ],
                          cmp [ test (negation form),
                                public (negation form)] ]
```

`testAnnounce form` is equivalent to `info all_agents form`:

```
Semantics> showM (testAnnounce p [a,b,c])
==> [0,1]
[0,1]
(0,p)(1,-p)
(a,[[0],[1]])
(b,[[0],[1]])
(c,[[0],[1]])

Semantics> showM (info [a,b,c] p [a,b,c])
==> [0,1]
[0,1]
(0,p)(1,-p)
```

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

The function `testAnnounce` gives the special case of revelations where the alternatives are a formula and its negation, and where the result is publicly announced.
Note that *DEMO* correctly computes the result of the sequence and the sum of two contradictory propositional tests:

```
Semantics> showM (cmp [test p, test (Neg p)] [a,b,c])
==> []
[]

(a,[])
(b,[])
(c,[])

Semantics> showM (ndS [test p, test (Neg p)] [a,b,c])
==> [0]
[0]
(0,T)
(a,[[0]])
(b,[[0]])
(c,[[0]])
```

9 Examples

9.1 The riddle of the caps

Picture a situation[3] of four people a, b, c, d standing in line, with a, b, c looking to the left, and d looking to the right. a can see no-one else; b can see a; c can see a and b, and d can see no-one else. They are all wearing caps, and they cannot see their own cap. If it is common knowledge that there are two white and two black caps, then in the situation depicted in Figure 4, c knows what colour cap she is wearing.

If c now announces that she knows the colour of her cap (without revealing the colour), b can infer from this that he is wearing a white cap, for b can reason as follows: "c knows her colour, so she must see two caps of the same colour. The cap I can see is white, so my own cap must be white as well." In this situation b draws a conclusion from the fact that c knows her colour.

In the situation depicted in Figure 5, b can draw a conclusion from the fact that c does not know her colour.

In this case c announces that she does not know her colour, and b can infer from this that he is wearing a black cap, for b can reason as follows:

[3] See [vE_1Or05].

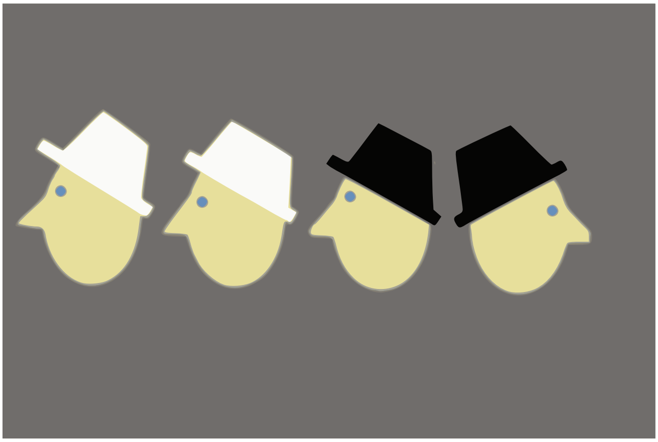

FIGURE 4.

"*c* does not know her colour, so she must see two caps of different colours in front of her. The cap I can see is white, so my own cap must be black."

To account for this kind of reasoning, we use model checking for epistemic updating, as follows. Proposition p_i expresses the fact that the i-th cap, counting from the left, is white. Thus, the facts of our first example situation are given by $p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4$, and those of our second example by $p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4$.

Here is the *DEMO* code for this example (details to be explained below):

```
module Caps where

import DEMO

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                    ]

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

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

mo0  = upd (initE [P 1, P 2, P 3, P 4] [a,b,c,d]) (test capsInfo)
mo1  = upd mo0 (public capsInfo)
mo2  = upds mo1 [awarenessFirstCap, awarenessSecondCap]
```

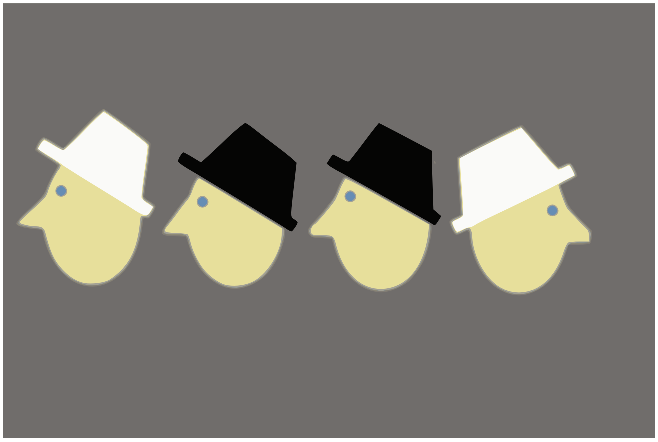

FIGURE 5.

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

An initial situation with four agents a, b, c, d and four propositions p_1, p_2, p_3, p_4, with exactly two of these true, where no-one knows anything about the truth of the propositions, and everyone is aware of the ignorance of the others, is modelled like this:

```
Caps> showM mo0
==> [5,6,7,8,9,10]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
(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,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
(b,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
(c,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
(d,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
```

The first line indicates that worlds $5, 6, 7, 8, 9, 10$ are compatible with the facts of the matter (the facts being that there are two white and two black caps). E.g., 5 is the world where a and b are wearing the white caps. The second line lists all the possible worlds; there are 2^4 of them, since every world has a different valuation. The third through sixth lines give the valuations of worlds. The last four lines represent the accessibility relations for the agents. All accessibilities are total relations, and they are represented here as the corresponding partitions on the set of worlds. Thus, the ignorance of the agents is reflected in the fact that for all of them all worlds are equivalent: none of the agents can tell any of them apart.

The information that two of the caps are white and two are black is expressed by the formula

$$(p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4) \vee (p_1 \wedge p_3 \wedge \neg p_2 \wedge \neg p_4) \vee (p_1 \wedge p_4 \wedge \neg p_2 \wedge \neg p_3)$$
$$\vee (p_2 \wedge p_3 \wedge \neg p_1 \wedge \neg p_4) \vee (p_2 \wedge p_4 \wedge \neg p_1 \wedge \neg p_3) \vee (p_3 \wedge p_4 \wedge \neg p_1 \wedge \neg p_2).$$

A public announcement with this information has the following effect:

```
Caps> showM (upd mo0 (public capsInfo))
==> [0,1,2,3,4,5]
[0,1,2,3,4,5]
(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]])
```

Let this model be called `mo1`. The representation above gives the partitions for all the agents, showing that nobody knows anything. A perhaps more familiar representation for this multi-agent Kripke model is given in Figure 6. In this picture, all worlds are connected for all agents, all worlds are compatible with the facts of the matter (indicated by the double ovals).

Next, we model the fact that (everyone is aware that) b can see the first cap and that c can see the first and the second cap, as follows:

```
Caps> showM (upds mo1 [info [b,c] p1, info [c] p2])
==> [0,1,2,3,4,5]
[0,1,2,3,4,5]
(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]])
```

Notice that this model reveals that in case a, b wear caps of the same colour (situations 0 and 5), c knows the colour of all the caps, and in case a, b wear caps of different colours, she does not (she confuses the cases $1, 2$ and the cases $3, 4$). Figure 7 gives a picture representation.

Let this model be called `mo2`. Knowledge of c about her situation is expressed by the epistemic formula $K_c p_3 \vee K_c \neg p_3$, ignorance of c about her situation by the negation of this formula. Knowledge of b about his situation is expresed by $K_b p_2 \vee K_b \neg p_2$. Let `bK`, `cK` express that b, c know about their situation. Then updating with public announcement of `cK` and with public announcement of the negation of this have different effects:

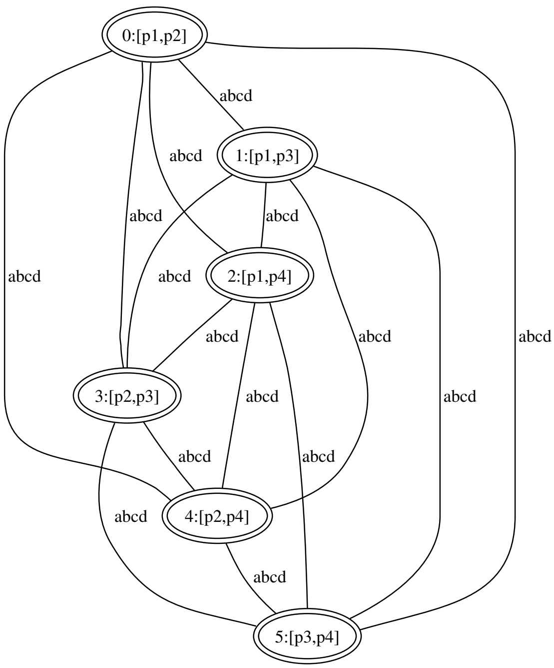

FIGURE 6. Caps situation where nobody knows anything about p_1, p_2, p_3, p_4.

```
Caps> showM (upd mo2 (public cK))
==> [0,1]
[0,1]
(0,[p1,p2])(1,[p3,p4])
(a,[[0,1]])
(b,[[0],[1]])
(c,[[0],[1]])
(d,[[0,1]])

Caps> showM (upd mo2 (public (Neg cK)))
==> [0,1,2,3]
[0,1,2,3]
(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]])
```

In both results, b knows about his situation, though:

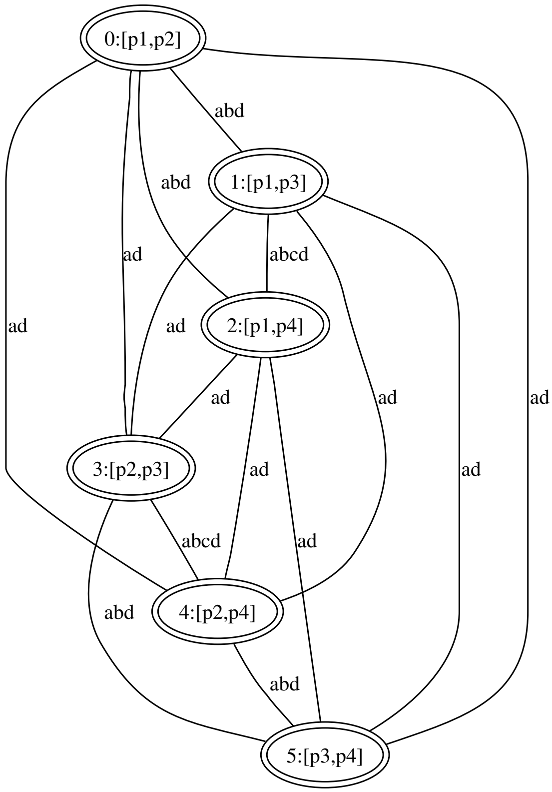

FIGURE 7. Caps situation after updating with awareness of what b and c can see.

```
Caps> isTrue (upd mo2 (public cK)) bK
True
Caps> isTrue (upd mo2 (public (Neg cK))) bK
True
```

9.2 Muddy children

For this example we need four agents a, b, c, d. Four children a, b, c, d are sitting in a circle. They have been playing outside, and they may or may not have mud on their foreheads. Their father announces: "At least one child is muddy!" Suppose in the actual situation, both c and d are muddy.

a	b	c	d
○	○	●	●

Then at first, nobody knows whether he is muddy or not. After public announcement of these facts, $c(d)$ can reason as follows. "Suppose I am clean. Then $d(c)$ would have known in the first round that she was dirty. But she didn't. So I am muddy." After c, d announce that they know their state, $a(b)$ can reason as follows: "Suppose I am dirty. Then c and d would not have known in the second round that they were dirty. But they knew. So I am clean." Note that the reasoning involves awareness about *perception*.

In the actual situation where b, c, d are dirty, we get:

a	b	c	d
○	●	●	●
?	?	?	?
?	?	?	?
?	!	!	!
!	!	!	!

Reasoning of b: "Suppose I am clean. Then c and d would have known in the second round that they are dirty. But they didn't know. So I am dirty. Similarly for c and d." Reasoning of a: "Suppose I am dirty. Then b, c and d would not have known their situation in the third round. But they did know. So I am clean." And so on ... [Fa+95].

Here is the *DEMO* implementation of the second case of this example, with b, c, d dirty.

```
module Muddy where

import DEMO

bcd_dirty = Conj [Neg p1, p2, p3, p4]

awareness = [info [b,c,d] p1,
             info [a,c,d] p2,
             info [a,b,d] p3,
             info [a,b,c] p4 ]

aK =  Disj [K a p1, K a (Neg p1)]
bK =  Disj [K b p2, K b (Neg p2)]
cK =  Disj [K c p3, K c (Neg p3)]
dK =  Disj [K d p4, K d (Neg p4)]

mu0 = upd (initE [P 1, P 2, P 3, P 4] [a,b,c,d]) (test bcd_dirty)
mu1 = upds mu0 awareness
mu2 = upd  mu1 (public (Disj [p1, p2, p3, p4]))
mu3 = upd  mu2 (public (Conj[Neg aK, Neg bK, Neg cK, Neg dK]))
mu4 = upd  mu3 (public (Conj[Neg aK, Neg bK, Neg cK, Neg dK]))
mu5 = upds mu4 [public (Conj[bK, cK, dK])]
```

The initial situation, where nobody knows anything, and they are all aware of the common ignorance (say, all children have their eyes closed, and they all know this) looks like this:

```
Muddy> showM mu0
==> [14]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
(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,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
(b,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
(c,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
(d,[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]])
```

The awareness of the children about the mud on the foreheads of the others is expressed in terms of update models.
Here is the update model that expresses that b, c, d can see whether a is muddy or not:

```
Muddy> showM (info [b,c,d] p1)
==> [0,1]
[0,1]
(0,p1)(1,-p1)
(a,[[0,1]])
(b,[[0],[1]])
(c,[[0],[1]])
(d,[[0],[1]])
```

Let `awareness` be the list of update models expressing what happens when they all open their eyes and see the foreheads of the others. Then updating with this has the following result:

```
Muddy> showM (upds mu0 awareness)
==> [14]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
(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]])
```

Call the result `mu1`. An update of `mu1` with the public announcement that at least one child is muddy gives:

```
Muddy> showM (upd mu1 (public (Disj [p1, p2, p3, p4])))
==> [13]
```

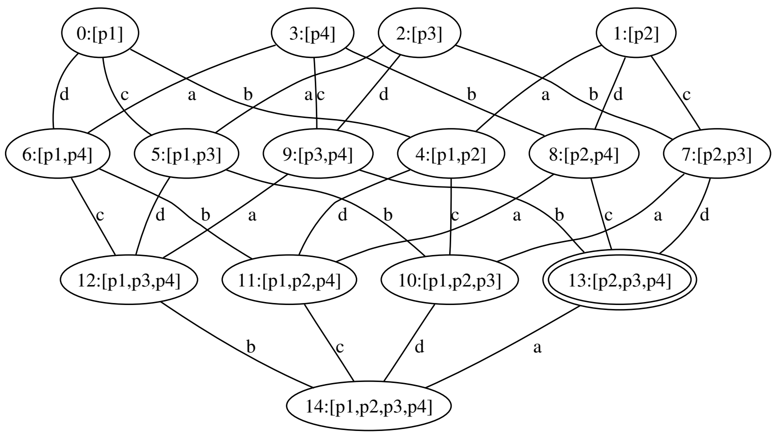

Figure 8.

```
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]
(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]])
```

Figure 8 represents this situation where the double oval indicates the actual world). Call this model `mu2`, and use `aK`, `bK`,`cK`, `dK` for the formulas expressing that a, b, c, d know whether they are muddy (see the code above). Then we get:

```
Muddy> showM (upd mu2 (public (Conj[Neg aK, Neg bK, Neg cK,
                                                   Neg dK])))
==> [9]
[0,1,2,3,4,5,6,7,8,9,10]
(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]])
```

This situation is represented in Figure 9. We call this model `mu3`, and update again with the same public announcement of general ignorance:

```
Muddy> showM (upd mu3 (public (Conj[Neg aK, Neg bK, Neg cK,
                                                      Neg dK])))
==> [3]
[0,1,2,3,4]
(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]])
```

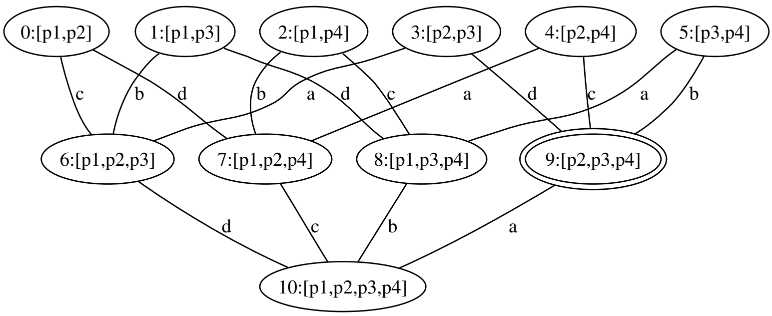

FIGURE 9.

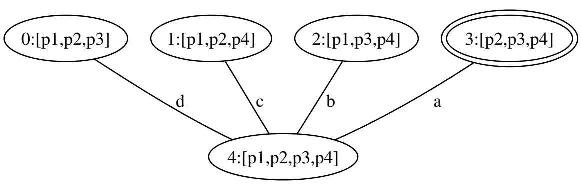

FIGURE 10.

Finally, this situation is represented in Figure 10, and the model is called mu4. In this model, b, c, d know about their situation:

```
Muddy> isTrue mu4 (Conj [bK, cK, dK])
True
```

Updating with the public announcement of this information determines everything:

```
Muddy> showM (upd mu4 (public (Conj[bK, cK, dK])))
==> [0]
[0]
(0,[p2,p3,p4])
(a,[[0]])
(b,[[0]])
(c,[[0]])
(d,[[0]])
```

10 Conclusion and further work

DEMO was used for solving Hans Freudenthal's Sum and Product puzzle by means of epistemic modelling in [vDRu$_0$Ve$_2$05]. There are many variations of this. See the *DEMO* documentation at `http://www.cwi.nl/~jve/demo/` for descriptions and for *DEMO* solutions. *DEMO* is also good at modelling the kind of card problems described in [vD03], such as the Russian card problem. A *DEMO* solution to this was published in [vD+06]. *DEMO* was used for checking a version of the Dining Cryptographers protocol [Ch$_2$88], in [vE$_1$Or05]. All of these examples are part of the *DEMO* documentation.

The next step is to employ *DEMO* for more realistic examples, such as checking security properties of communication protocols. To develop *DEMO* into a tool for blackbox cryptographic analysis — where the cryptographic primitives such as one-way functions, nonces, public and private key encryption are taken as given. For this, a propositional base language is not sufficient. We should be able to express that an agent A generates a nonce n_A, and that no-one else knows the value of the nonce, without falling victim to a combinatorial explosion. If nonces are 10-digit numbers then not knowing a particular nonce means being confused between 10^{10} different worlds. Clearly, it does not make sense to represent all of these in an implementation. What could be done, however, is represent epistemic models as triples (W, R, V), where V now assigns a non-contradictory proposition to each world. Then uncertainty about the value of n_A, where the actual value is N, can be represented by means of two worlds, one where $n_a = N$ and one where $n_a \neq N$. This could be done with basic propositions of the form $e = M$ and $e \neq M$, where e ranges over cryptographic expressions, and M ranges over 'big numerals'. Implementing these ideas, and putting *DEMO* to the test of analysing real-life examples is planned as future work.

References

[Ba$_4$02] A. Baltag. A logic for suspicious players: epistemic action and belief-updates in games. *Bulletin of Economic Research* 54(1):1–45, 2002.

[Ba$_4$Mo$_3$So$_1$99] A. Baltag, L. Moss & S. Solecki. The logic of public announcements, common knowledge, and private suspicions. SEN-R9922, CWI, Amsterdam, 1999.

[Ba$_4$Mo$_3$So$_1$03] A. Baltag, L. Moss & S. Solecki. The logic of public announcements, common knowledge, and private suspicions. Dept of Cognitive Science, Indiana University and Dept of Computing, Oxford University, 2003.

[BldRVe$_1$01] P. Blackburn, M. de Rijke & Y. Venema. *Modal Logic*, *Cambridge Tracts in Theoretical Computer Science* 53. Cambridge University Press, 2001.

[Ch$_1$Ko$_0$Po$_0$06] Z. Chatzidakis, P. Koepke & W. Pohlers, eds. *Logic Colloquium '02*, *Lecture Notes in Logic* 27. Association for Symbolic Logic, 2006.

[Ch$_2$88] D. Chaum. The dining cryptographers problem: unconditional sender and receiver untraceability. *Journal of Cryptology* 1(1):65–75, 1988.

[Ch$_3$80] B. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[Da$_1$Lo$_0$Lo$_4$62] M. Davis, G. Logemann & D. Loveland. A machine program for theorem proving. *Communications of the ACM* 5(7):394–397, 1962.

[Da$_1$Pu60] M. Davis & H. Putnam. A computing procedure for quantification theory. *Journal of the ACM* 7(3):201–215, 1960.

[Fa+95] R. Fagin, J. Halpern, Y. Moses & M. Vardi. *Reasoning about Knowledge*. The MIT Press, 1995.

[Ga$_0$Ko$_5$No$_0$06] E. Gansner, E. Koutsofios & S. North. *Drawing graphs with* dot. Graphviz project, 2006. URL `http://www.graphviz.org`.

[Ge$_2$99a] J. Gerbrandy. *Bisimulations on planet Kripke*. Ph.D. thesis, Universiteit van Amsterdam, 1999. *ILLC Publications* DS-1999-01.

[Go$_0$02] R. Goldblatt. *Logics of Time and Computation, Second Edition, Revised and Expanded*, *CSLI Lecture Notes* 7. CSLI, Stanford, 1992.

[$\text{Hi}_1$62] J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions.* Cornell University Press, 1962.

[$\text{Ho}_3$98] M. Hollenberg. *Logic and Bisimulation.* Ph.D. thesis, Utrecht University, 1998.

[$\text{Ho}_4$71] J.E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In [$\text{Ko}_1\text{Pa}_8$71, pp. 189–196].

[$\text{Jo}_2$03] S.P. Jones. *Haskell 98. Language and Libraries. The Revised Report.* Cambridge University Press, 2003.

[$\text{Ko}_1\text{Pa}_8$71] Z. Kohavi & A. Paz, eds. *Theory of Machines and Computations.* Academic Press, 1971.

[$\text{Ko}_4$03] B. Kooi. *Knowledge, Chance and Change.* Ph.D. thesis, University of Groningen, 2003. *ILLC Publications* DS-2003-01.

[Mo_3Gi_1dR99] L.S. Moss, J. Ginzburg & M. de Rijke, eds. *Logic, Language and Information*, 2. CSLI Publications, 1999.

[$\text{Pa}_1\text{Ta}_0$87] R. Paige & R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing* 16(6):973–989, 1987.

[$\text{Ru}_0$04] J. Ruan. *Exploring the update universe.* Master's thesis, Universiteit van Amsterdam, 2004. *ILLC Publications* MoL-2004-08.

[vB01b] J. van Benthem. Language, logic, and communication. In [vB+01, pp. 7–25].

[vB06] J. van Benthem. 'One is a Lonely Number': On the logic of communication. In [$\text{Ch}_1\text{Ko}_0\text{Po}_0$06, pp. 96–129].

[vB+01] J. van Benthem, P. Dekker, J. van Eijck, M. de Rijke & Y. Venema, eds. *Logic in Action.* ILLC, 2001.

[vD00] H. van Ditmarsch. *Knowledge Games.* Ph.D. thesis, University of Groningen, 2000. *ILLC Publications* DS-2000-06.

[vD03] H. van Ditmarsch. The Russian cards problem. *Studia Logica* 75(1):31–62, 2003.

[$\text{vDRu}_0\text{Ve}_2$05] H. van Ditmarsch, J. Ruan & R. Verbrugge. Model checking sum and product. In [$\text{Zh}_1\text{Ja}_3$05, pp. 790–795].

[vD+06] H. van Ditmarsch, W. van der Hoek, R. van der Meyden & J. Ruan. Model checking Russian cards. *Electronic Notes in Theoretical Computer Science* 149(2):105–123, 2006.

[$\text{vE}_0$05] M. van Eekelen, ed. *6th Symposium on Trends in Functional Programming, TFP 2005: Proceedings.* Institute of Cybernetics, Tallinn, 2005.

[$\text{vE}_1$04a] J. van Eijck. Communicative actions, 2004. Preprint.

[$\text{vE}_1$04b] J. van Eijck. Reducing dynamic epistemic logic to PDL by program transformation. SEN-E0423, CWI, Amsterdam, December 2004.

[vE_1Or05] J. van Eijck & S. Orzan. Modelling the epistemics of communication with functional programming. In [$\text{vE}_0$05, pp. 44–59].

[$\text{Zh}_0\text{St}_5$00] H. Zhang & M.E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning* 24(1–2):277–296, 2000.

[$\text{Zh}_1\text{Ja}_3$05] S. Zhang & R. Jarvis, eds. *AI 2005: Advances in Artificial Intelligence, 18th Australian Joint Conference on Artificial Intelligence, Sydney, Australia, December 5–9, 2005, Proceedings*, *Lecture Notes in Computer Science* 3809. Springer, 2005.