EVOLUÇÃO DO CONHECIMENTO

Isaque Maçalam Saab Lima

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Mario Roberto Folhadela Benevides

Rio de Janeiro
Setembro de 2013

EVOLUÇÃO DO CONHECIMENTO

Isaque Maçalam Saab Lima

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

____
Prof. Mario Roberto Folhadela Benevides, Ph.D.

____
Prof. Gerson Zaverucha, Ph.D.

____
Prof. Luis Menasché Schechter, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2013

Lima, Isaque Maçalam Saab

Evolução do Conhecimento/Isaque Maçalam Saab Lima. – Rio de Janeiro: UFRJ/COPPE, 2013.

XI, 100 p.: il.; 29, 7cm.

Orientador: Mario Roberto Folhadela Benevides

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 54 – 55.

1. Lógica Epistêmica. 2. Lógica Epistêmica Dinâmica. 3. Lógica Epistêmica Dinâmica com Atribuição. 4. Modelo de Ação. 5. DEMO. I. Benevides, Mario Roberto Folhadela. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Ao meu avô Moussa Nassar
Jreig Abi Saab.

Agradecimentos

Agradeço a Deus, por continuar me abençoando e ter me permitido concluir esse mestrado.

Agradeço ao prof. Mario Folhadela Benevides, pela sua orientação, apoio, incentivo e paciência durante todo o período do meu mestrado e por sempre ter acreditado e confiado em mim.

Agradeço em especial aos meus pais, minha irmã e minha avó pelo constante apoio e por acreditarem sempre no meu potencial.

Agradeço a todo corpo docente da COPPE pelas excelentes aulas que vieram a acrescentar muito em minha vida acadêmica e pessoal.

Agradeço a todos os funcionários da COPPE que contribuíram de forma direta ou indireta para conclusão desse trabalho.

Agradeço a StoneAge por toda a paciência e flexibilidade que ela me proporcionou ao longo de todo o mestrado.

Agradeço, por último mas não menos importante, a todos os meus familiares e amigos por sempre me incentivarem e entenderem a minha ausência em alguns momentos.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

EVOLUÇÃO DO CONHECIMENTO

Isaque Maçalam Saab Lima

Setembro/2013

Orientador: Mario Roberto Folhadela Benevides

Programa: Engenharia de Sistemas e Computação

O objetivo dessa dissertação é desenvolver um *framework* para ser utilizado em lógicas epistêmicas dinâmicas com atribuições (DELWA). A diferença desse trabalho para outros existentes na área, como VAN DITMARSCH *et al.* [1], é a utilização de modelos de ação, da lógica epistêmica dinâmica, para realizar as atribuições booleanas às proposições, ao invés de criar novos mecanismos para realizar as atribuições. Estendemos o conceito de modelo de ação, criando a propriedade de pós-condição em cada estado do modelo, tornando possível atribuir valores booleanos para as proposições. Durante a pesquisa dessa dissertação implementamos também algumas novas funcionalidades no DEMO (verificador de modelos epistêmicos feito em Haskell) para representar modelos de ação com atribuições. Primeiro, discutiremos os conceitos sobre lógica proposicional dinâmica, lógica epistêmica, lógica epistêmica dinâmica, lógica epistêmica dinâmica com atribuição, apresentando uma abordagem diferente da proposta nessa dissertação, e do DEMO. Apresentaremos, por fim, o modelo proposto, algumas aplicações e descreveremos as novas funcionalidades implementadas no DEMO.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

EVOLUTION OF KNOWLEDGE

Isaque Maçalam Saab Lima

September/2013

Advisor: Mario Roberto Folhadela Benevides

Department: Systems Engineering and Computer

The goal of this work is to develop a framework to be used in dynamic epistemic logic with assignment (DELWA). The diference between this work to others in this area, like VAN DITMARSCH *et al.* [1], is the use of action models, from dynamic epistemic logic, to make booleans assignments to the propositions, rather than create a new mechanism to make assignments. We extend the concept of action model creating the property post-condition of each state of the model, making possible to assign boolean values to propositions. During the research we also implemented new features in DEMO (epistemic model checker coded in Haskell) to represent action models with assignments. First, we discuss the concepts of epistemic logic, dynamic epistemic logic, dynamic epistemic logic with assignment, presenting a different approach of the proposed in this dissertation and DEMO. Then we introduce the proposed model, some applications and describe the new features implemented in DEMO.

Sumário

Lista de Figuras x

1 Introdução 1

2 Revisão Bibliográfica 3

2.1 Lógica Epistêmica 3

2.1.1 Sintaxe e Semântica 6

2.1.2 Sistemas Axiomáticos 7

2.2 Lógica Epistêmica Dinâmica 8

2.2.1 Ações Públicas 8

2.2.2 Ações Privadas (Modelos de Ação) 10

2.3 Lógica Dinâmica Proposicional 16

2.3.1 Sintaxe e Semântica 16

2.4 Lógica Epistêmica Dinâmica com Atribuições 17

2.4.1 Atribuições públicas 18

2.4.2 Atribuições atômicas 20

2.5 Verificador de Modelos Epistêmicos Dinâmicos 23

2.5.1 DEMO 24

3 Modelo Proposto 31

3.1 Modelo de Ação com Atribuição 31

3.1.1 Sintaxe e Semântica 34

3.2 Extensão do DEMO 45

4 Aplicações do modelo proposto 47

4.1 Jogo das crianças sujas 47

4.2 Carta no baú 48

4.3 Ações Possíveis 50

5 Conclusões 52

Referências Bibliográficas 54

A Código **56**

A.1 DEMO . . . 56

A.1.1 DEMO.hs . . . 56

A.1.2 SemanticsPA.hs . . . 57

A.1.3 Semantics.hs . . . 59

A.1.4 ActEpist.hs . . . 66

A.1.5 Display.hs . . . 81

A.1.6 DPLL.hs . . . 89

A.1.7 MinAE.hs . . . 94

A.1.8 MinBis.hs . . . 95

A.1.9 Models.hs . . . 98

Lista de Figuras

2.1 Possíveis mundos [Jogo das 3 cartas] 5
2.2 Crianças Sujas - Modelo inicial 9
2.3 Crianças Sujas - pelo menos uma criança suja 9
2.4 Crianças Sujas - pelo menos duas crianças sujas 10
2.5 Crianças Sujas - estado final 10
2.6 Jogo das 3 cartas - Depois da atualização 11
2.7 Modelo de ação 13
2.8 Modelo Epistêmico x Modelo de Ação 14
2.9 Estados gerados pelo produto cartesiano 14
2.10 Estados eliminados pelo produto cartesiano limitado pelas pré-condições 15
2.11 Modelo resultante da ação de Anne mostrar sua carta para Bill 15
2.12 DELWA - Crianças Sujas - Modelo inicial 19
2.13 DELWA - Crianças Sujas - pelo menos uma criança suja 19
2.14 DELWA - Crianças Sujas - pai joga um balde de água em Anne . . . 19
2.15 DELWA - Crianças Sujas - "Alguém sabe o estado real do sistema?" . 20
2.16 Russian Cards 21
2.17 Grafo que representa o modelo em0 26
2.18 Grafo que representa o modelo em1 27
2.19 DEMO - Muddy Children 29
2.20 DEMO - Russian Cards 30

3.1 Russian Cards 32
3.2 Modelo de ação sem atribuição - Russian Cards 32
3.3 Modelo de ação com atribuição - Russian Cards 33
3.4 Modelo Epistêmico antes de aplicar as pós-condições 33
3.5 Modelo Epistêmico depois de aplicar as pós-condições 34
3.6 Composição de modelos de ação 36
3.7 Atualização do modelo epistêmico M1 com os modelos de ação A1 e A2 36
3.8 Atualização do modelo epistêmico M1 com os modelos de ação A2 e A1 37
3.9 Composição dos modelos de ação A1 e A2 antes da eliminação dos estados incompatíveis 37

3.10 Composição dos modelos de ação A1 e A2 após a eliminação dos estados incompatíveis . . . 38
3.11 Composição de modelos de ação A1 e A2 . . . 39
3.12 Atualização do modelo epistêmico M1 com o modelo de ação A3 . . . 39

4.1 Crianças sujas . . . 47
4.2 Atualização: Pai joga um balde em Anne . . . 47
4.3 Crianças sujas, após a atualização . . . 48
4.4 Jogo carta no baú . . . 49
4.5 Ação de abrir/fechar o baú . . . 49
4.6 Ação de Anne espiar o baú . . . 49
4.7 Ação de Bill espiar o baú . . . 50
4.8 Modelo de ação de ligar/desligar a luz . . . 50
4.9 Modelo de ação de ligar a luz . . . 50
4.10 Modelo de ação de desligar a luz . . . 50

Capítulo 1

Introdução

Nessa dissertação iremos estudar sobre lógicas epistêmicas, que são lógicas modais acrescidas dos operadores de conhecimento e crença. Podemos remontar o estudo do conhecimento aos tempos dos filósofos gregos, em que os mesmos se questionavam: “O que conhecemos?”, “O que pode ser conhecido?”, “O que significa dizer que alguém sabe alguma coisa?”, porém a formalização da lógica epistêmica, como conhecemos, começou com HINTIKKA [2], que acrescentou à lógica modal de von Wright, a qual já tinha uma representação sintática para o conhecimento, a noção semântica de conhecimento e crença.

Segundo VAN DITMARSCH *et al.* [3], a representação semântica dada por Hintikka, marcou não só o fim de uma era de inúmeras tentativas dos filósofos, de representar formalmente a noção semântica de conhecimento e crença, como também estabeleceu o início de um período de desenvolvimento na lógica epistêmica, sendo pesquisada por diversas áreas como: filosofia, economia, inteligência artificial, teoria dos jogos e ciência da computação.

Embora a formalização da lógica epistêmica date de 1962 (HINTIKKA [2]) e o conceito de atribuição seja uma operação primitiva das linguagens de programação, a ideia de se juntar esses dois conceitos, para alterar valores das proposições na lógica epistêmica, é bem mais recente, sendo encontrada em VAN DITMARSCH *et al.* [1], KOOI [4] e J. VAN BENTHEN e KOOI [5].

Nesse trabalho, temos como objetivo reforçar a ligação entre tais conceitos, apresentando uma nova abordagem para realizar operações de atribuição em proposições da lógica epistêmica dinâmica. Essa abordagem se diferencia das encontradas em [1], [4] e [5], pois não altera a dinâmica das atualizações para realizar as atribuições, visto que faz-se uso do modelo de ação, da lógica epistêmica dinâmica, para realizá-las, o que não ocorre nos trabalhos acima citados.

Estendemos o conceito de modelo de ação para realizar operações de atribuição, através da adição de uma propriedade de pós-condição na estrutura do modelo de ação. Dessa forma, podemos utilizar esse novo modelo de ação na lógica epistêmica

com atribuição. Explicaremos melhor essa ideia nos capítulos seguintes.

Essa dissertação está dividida da seguinte forma:

- Capítulo 2: O capítulo introduz os conceitos de lógica epistêmica, lógica epistêmica dinâmica e lógica epistêmica dinâmica com atribuições, necessários para o entendimento completo desso presente trabalho. Discutiremos também a verificação de modelos epistêmicos, apresentando o verificador de modelos epistêmicos chamado DEMO (Dynamic Epistemic MOdeling).

- Capítulo 3: Nesse capítulo, são apresentadas as principais contribuições dessa dissertação. Descrevemos o novo modelo de ação para tratar das atribuições booleanas e também mostramos as novas funcionalidades implementadas no DEMO para que o mesmo possa ser usado para verificarmos modelos da lógica epistêmica dinâmica com atribuições.

- Capítulo 4: São expostos alguns exemplos de utilização desse novo modelo, assim como sua utilização com as novas funcionalidades do DEMO.

- Capítulo 5: Nesse capítulo, apresentamos as conclusões do trabalho de pesquisa dessa dissertação e discutimos os possíveis trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Neste capítulo, descreveremos alguns conceitos básicos sobre lógicas epistêmicas e o verificador de modelos epistêmicos (DEMO) que são necessários para um melhor entendimento dessa dissertação. Tais conceitos foram extraídos dos trabalhos de FAGIN *et al.* [6], VAN DITMARSCH *et al.* [3], DELGADO [7], H. VAN DITMARSCH e DE LIMA [8], VAN DITMARSCH *et al.* [1], VAN EIJCK [9], H. VAN DITMARSCH e RUAN [10], B. RENNE e YAP [11], SIETSMA e VAN EIJCK [12], KOOI [13], SACK [14], H. VAN DITMARSCH e VAN DER HOEK [15].

2.1 Lógica Epistêmica

Lógica epistêmica é a lógica modal utilizada para raciocinar sobre conhecimento e crença. Segundo FAGIN *et al.* [6], epistemologia, estudo do conhecimento, tem uma longa tradição na filosofia, estando presente desde os primeiros filósofos gregos, em que os mesmo se questionavam: "O que conhecemos?", "O que pode ser conhecido?", "O que significa dizer que alguém sabe alguma coisa?". A formalização da lógica epistêmica começou com HINTIKKA [2], sendo pesquisada por diferentes áreas como filosofia [16] , economia [17], inteligência artificial [18], teoria dos jogos [17] e ciência da computação [6], que passaram a se interessar por este tema e a aplicá-lo em situações de multi-agentes.

Um agente, em um sistema multi-agentes, não pode levar em conta apenas os fatos que são verdadeiros, mas também o conhecimento que os outros agentes têm sobre os fatos. Uma maneira simples de pensar sobre isso é analisar uma situação de barganha. Um exemplo, muito comum em praias do nordeste, surge da seguinte pergunta: "quanto custa o côco gelado?". O vendedor sabe o valor de custo do côco, mas esse não é o único fator que ele leva em consideração na hora de dizer o preço, ele também leva em consideração o que ele acha que o comprador sabe sobre o preço do côco. O comprador, por sua vez, tem uma ideia de quanto deve custar o côco, mas também considera o que ele acha que o vendedor sabe que ele sabe sobre

o preço do côco, e assim por diante. O preço final do côco é dado a partir do que os dois agentes sabem sobre o sistema.

A lógica epistêmica procura representar o que o agente considera possível diante das informações que ele possui. Como mostrado no exemplo anterior, as informações de um agente podem conter informações sobre as informações dos outros agentes. Esse raciocínio tende a ficar um pouco complicado e, segundo FAGIN *et al.* [6], a maioria das pessoas tende a perder a linha de raciocínio em sentenças do tipo "Dean não sabe se Nixon sabe que o Dean sabe que Nixon sabe que McCord roubou o escritório do O'Brien em Watergate". Porém esse é exatamente o tipo de raciocínio que se precisa ter quando se está analisando o conhecimento em um sistema multi-agentes.

Geralmente, o agente não tem o conhecimento total do sistema e devido a essa falta de conhecimento não tem como afirmar qual o estado real do sistema. Em vez de um único estado tem-se um conjunto de possíveis estados, também conhecidos como possíveis mundos, dentre os quais está o estado real. O conhecimento do agente é representado pelas arestas que ligam os possíveis mundos, quanto menor o número de arestas mais conhecimento o agente tem e mais certeza ele tem sobre o real estado do sistema.

Uma definição importante, em um sistema multi-agentes, é a de conhecimento comum. Dizemos que ϕ é de conhecimento comum para o grupo G se todos os agentes do grupo G conhecem ϕ e todos os agentes sabem que todos os agentes conhecem ϕ e assim por diante. Por exemplo, em um sistema de transito é desejável que todos os motoristas saibam que o sinal vermelho significa pare e o sinal verde significa "siga". Vamos supor que todos os motoristas conheçam (conhecimento de todos) e obedeçam a essas regras. Um motorista vai se sentir seguro? Não, pois ele não sabe que todos os motoristas conhecem e obedecem as regras e por isso pode achar que existe algum motorista que não as conhece e irá avançar o sinal vermelho. Logo, para um motorista se sentir seguro, é necessário que essas regras sejam de conhecimento comum, ou seja, todo mundo sabe que todo mundo sabe as regras e assim por diante.

Outra definição importante, em um sistema multi-agentes, é a de conhecimento distribuído. Um grupo tem conhecimento distribuído de um fato se o conhecimento desse fato pode ser deduzido juntando o conhecimento de cada membro do grupo. Por exemplo, Alice sabe que Bob gosta da Carol ou da Susan e Charlie sabe que o Bob não gosta da Carol. Alice e Charlie têm conhecimento distribuído do fato de Bob gostar da Susan, porém sozinhos Alice e Charlie não sabem de quem Bob gosta. Parafraseando John McCarthy, conhecimento comum pode ser visto como aquilo que "qualquer idiota" ("any fool") sabe e conhecimento distribuído pode ser visto como o que o "homem sábio" ("wise man") saberia. Temos também o

conhecimento individual, que é o conhecimento relativo de um agente sobre o fato ϕ e o conhecimento de todos, que ocorre quando todos os agentes de um grupo G têm conhecimento de ϕ.

Resumo dos tipos de conhecimento:

- Conhecimento individual - É o conhecimento relativo de um agente sobre um fato φ.
- Conhecimento comum (C_G) - Todos os agentes do grupo G tem conhecimento de φ e todos sabem que todos tem conhecimento de φ ...
- Conhecimento distribuído(D_G) - Ocorre quando ao unir o conhecimento individual de todos agentes do grupo G, podemos deduzir φ.
- Conhecimento de todos(E_G) - Ocorre quando todos os agentes de um grupo G tem conhecimento de φ.

Exemplo 1 *(***Jogo das 3 cartas**, *VAN DITMARSCH* et al. *[3].) Seja um jogo com 3 jogadores (A,B e C) e 3 cartas(0,1 e 2), onde as cartas são distribuídas pelos jogadores. Assume-se que cada jogador pode ver apenas a sua carta e que todos têm a informação que cada jogador tem somente uma carta. Utilizamos os símbolos* $0_x, 1_x, 2_x$ *onde* $x \in \{A, B, C\}$ *para dizer "o jogador x tem a carta 0, 1 ou 2". Cada estado é nomeado pelas cartas que cada jogador tem naquele estado, por exemplo,* 012 *é o estado que o jogador* **A** *tem a carta* **0***, o jogador* **B** *tem a carta* **1** *e o jogador* **C** *tem a carta* **2***.* [1]*. As arestas ligam os estados que o jogador não consegue diferenciar.* [2]*.*

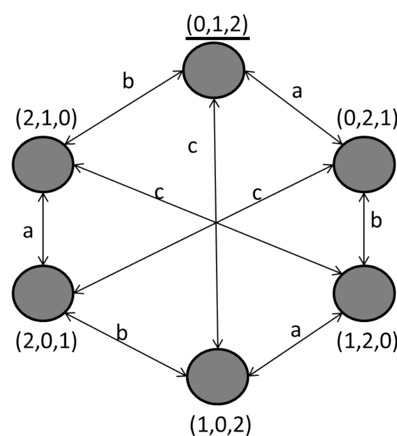

Figura 2.1: Possíveis mundos [Jogo das 3 cartas]

[1]O estado sublinhado é o estado real do sistema.

[2]Omitimos os loops reflexivos da figura.

Nesse exemplo, nenhum jogador sabe qual é o estado real (estado sublinhado de preto) do sistema pois eles possuem apenas a informação da sua própria carta, ou seja, ele não consegue diferenciar os estados onde ele tem a mesma carta. O jogador A não consegue diferenciar o estado 012 do estado 021 pois em ambos os estados ele tem a carta 0 e, como ele não sabe a carta dos outros jogadores, ele não consegue diferenciar os dois estados.

Andando pelas arestas do grafo podemos retirar várias informações. Considere o raciocínio do jogador A quando ele tem a carta 0, nesse caso ele considera que o jogador B pode ter a carta 1 ou a carta 2. Da mesma forma, se o jogador C tem a carta 2 ele pode pensar que o jogador B tem a carta 0 ou a carta 1. Podemos também extrair informações mais complexas do tipo: O jogador A considera que se o jogador B tiver a carta 1, e que se o jogador B pensa que o jogador A tem a carta 2, então o jogador B considera possível que o jogador A pense que o jogador B tem a carta 0. Podemos gerar várias sentenças desse tipo, de informação da informação da informação, só percorrendo as arestas do grafo.

2.1.1 Sintaxe e Semântica

Apresentaremos nessa seção a linguagem, seu modelo semântico e o sistema axiomático para representar uma lógica epistêmica em um sistema multi-agentes.

Definição 1 *A linguagem de um sistema epistêmico consiste em um conjunto contável* Φ *de símbolos proposicionais, um conjunto finito* $\mathcal{A}$ *de agentes, os conectivos booleanos* $\neg$ *e* $\wedge$ *e o operador modal* K_a *para cada agente* a*. As fórmulas são definidas abaixo:*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid K_a\varphi$$

onde $p \in \Phi$*,* $a \in \mathcal{A}$ *e* $K_a\varphi$ *indica que o agente* a *sabe* φ*, para* $a = 1, 2, 3, ..., m$

Utilizamos o modelo proposto por KRIPKE [19] para representar sistemas epistêmicos, pois sua estrutura contém noções de mundos, mundos acessíveis e valoração desses mundos.

Definição 2 *A estrutura de Kripke (Kripke frame) é uma tupla* $\mathcal{F} = (S, R_a)$ *onde*

- S *é um conjunto não vazio de estados;*
- R_a *é uma relação binária em* S*, para cada agente* $a \in \mathcal{A}$*;*

Definição 3 *O modelo de Kripke (modelo epistêmico) é um par* $\mathcal{M} = (\mathcal{F}, \mathbf{V})$*, onde* $\mathcal{F}$ *é a estrutura de Kripke e* $\mathbf{V}$ *é a função de valoração* $\mathbf{V} : \Phi \to 2^S$*, que associa valores verdade às primitivas de* Φ *em cada estado de* S*. Chamamos* $(\mathcal{M}, s)$ *de estado epistêmico.*

Na maioria das aplicações multi-agentes de lógica epistêmica, as relações R_a são relações de equivalência. Sempre que for este o caso, utilizaremos o simbolo $\sim_a$ para cada agente a.

Definição 4 *Dado um modelo epistêmico $\mathcal{M} = \langle S, \sim_a, V \rangle$, a noção de satisfação $\mathcal{M}, s \models \varphi$ é definida a seguir:*

$$
\begin{array}{lll}
\mathcal{M}, s \models p & sse & s \in V(p) \\
\mathcal{M}, s \models \neg\phi & sse & \mathcal{M}, s \not\models \phi \\
\mathcal{M}, s \models \phi \wedge \psi & sse & \mathcal{M}, s \models \phi \text{ e } \mathcal{M}, s \models \psi \\
\mathcal{M}, s \models K_a\phi & sse & \text{para todo } s' \in S : s \sim_a s' \text{ implica } \mathcal{M}, s' \models \phi
\end{array}
$$

$\mathcal{M}$ satisfaz ϕ se existe algum mundo $s \in S$ tal que $(\mathcal{M}, s) \models \phi$. Dizemos que ϕ é satisfatível se existe algum modelo que o satisfaça, caso contrário, dizemos que ϕ é insatisfatível. Uma fórmula ϕ é válida em um frame $\mathcal{F}$ se é verdadeira em todos os modelos sobre $\mathcal{F}$ (para todo $\mathcal{M}$ e $s, (\mathcal{M}, s) \models \phi$).

Exemplo 2 *(***Modelo de Kripke***) Continuando o exemplo anterior, segue o modelo de Kripke que representa o estado epistêmico de cada agente.*

$Hexa1 = \langle S, \sim, V \rangle$:

- $S = \{012, 021, 102, 120, 201, 210\}$
- $\sim_a = \{(012, 012), (012, 021), (021, 021), \dots\}$
- $V(0_a) = \{012, 021\}$, $V(1_a) = \{102, 120\}$, ...

2.1.2 Sistemas Axiomáticos

Axiomas

1. Todas as tautologias proposicionais,
2. $K_a(\varphi \to \psi) \to (K_a\varphi \to K_a\psi)$, para todo φ e $\psi \in \mathcal{M}$ e $i = 1, 2, ..., m$,
3. $K_a\varphi \to \varphi$,
4. $K_a\varphi \to K_aK_a\varphi$ (+introspecção),
5. $\neg K_a\varphi \to K_a\neg K_a\varphi$(-introspecção),

Os axiomas 3, 4 e 5 só são válidos se a relação R_i for respectivamente reflexiva,transitiva e simétrica.

Regras

1. Modus Ponens: de φ e $\varphi \to \psi$ deriva ψ ,
2. Generalização do Conhecimento: de $\vdash \varphi$ deriva $K_a\varphi$.

2.2 Lógica Epistêmica Dinâmica

Pensando no jogo das cartas, apresentado anteriormente, o que aconteceria se o jogador A mostrasse a sua carta para os outros jogadores? Como representaríamos isso na lógica epistêmica? O fato do jogador A mostrar sua carta faz com que o modelo do sistema seja atualizado, ou seja, os jogadores agora têm mais uma informação e podem diminuir sua incerteza sobre o jogo.

Na lógica epistêmica, para representar essa nova informação, teríamos que remodelar o sistema para contemplar essa informação, pois ela não tem nenhum método para atualizar modelos epistêmicos. Nesse ponto que entra a lógica epistêmica dinâmica, pois embora a lógica epistêmica seja robusta para tratar de conhecimento, ela não trata as mudanças na informação do agente.

A lógica epistêmica dinâmica é uma extensão da lógica epistêmica que lida com as mudanças nas informações dos agentes. Nela, temos a noção de atualização, onde novas informações podem ser agregadas ao modelo, mudando assim a incerteza dos agentes nos mundos.

Em lógica epistêmica dinâmica temos 2 tipos de ações:

- Ações Públicas: Todos os agente percebem o resultado da ação. Ex. Broadcast de uma mensagem;
- Ações Privadas: Apenas o grupo de agentes envolvidos percebem o resultado da ação. Ex. Mensagem de um agente para outro.

2.2.1 Ações Públicas

As ações publicas podem ser vistas como um caso específico de ações privadas, onde o grupo de agente é composto por todos os agentes do sistema.

Linguagem

A linguagem dessa lógica consiste em um conjunto contável Φ de símbolos proposicionais, um conjunto finto $\mathcal{A}$ de agentes, os conectivos booleanos $\neg$ e $\wedge$, o operador K_a para cada agente a e o operador $[\varphi]\psi$. As fórmulas são definidas:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid K_a\varphi \mid [\varphi]\psi$$

onde, $[\varphi]\psi$ significa: "depois do anúncio de φ, ψ é verdadeiro".

A consequência de um anuncio público $[\varphi]\psi$ é a eliminação de todos os estados, do modelo, onde ψ não é verdadeiro.

Definição 5 *A noção de satisfação de $\mathcal{M}, s \models [\varphi]\psi$ é definida a seguir:*

- $\mathcal{M}, s \models [\varphi]\psi$ *sse* ($\mathcal{M}, s \models \varphi$ *implica em* $\mathcal{M}, s \models \psi$)

Vamos exemplificar essas mudanças na informação com o jogo das crianças sujas.

Exemplo 3 (Jogo das crianças sujas, Muddy Children) *Nesse jogo, temos 3 crianças (A,B,C) e 2 dessas crianças estão com a testa suja. É de conhecimento comum que cada criança só consegue ver a testa das outras crianças, ou seja, não consegue ver a sua própria testa. Representamos os estados do modelo epistêmico desse jogo pelo rótulo xyz onde* $x,y,z \in \{0,1\}$ *e 0 indica que a criança está limpa e 1 indica que a criança está suja. Ex. o estado 110 representa que a criança A e a criança B estão com a testa suja e que a criança C está com a testa limpa. O modelo inicial do jogo é :*

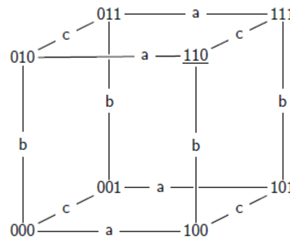

Figura 2.2: Crianças Sujas - Modelo inicial

O pai das crianças faz a seguinte afirmação(anuncio público): "Existe pelo menos uma criança suja.". Nesse momento, o estado onde todas as crianças estão limpas pode ser eliminado.

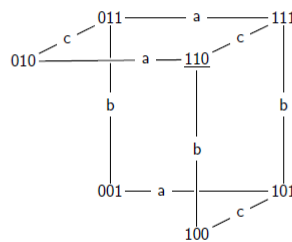

Figura 2.3: Crianças Sujas - pelo menos uma criança suja

Agora o pai pergunta: "Alguém já sabe se está limpo ou sujo?". Como nenhuma criança se pronuncia vira de conhecimento comum que todas veem pelo menos uma criança com a testa suja. Se todas veem pelo menos uma criança com a testa suja se torna conhecimento comum que temos no minimo 2 crianças com a testa suja. Logo podemos eliminar todos os estados onde temos apenas uma criança com a testa suja.

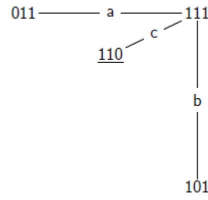

Figura 2.4: Crianças Sujas - pelo menos duas crianças sujas

O pai novamente pergunta: "Alguém já sabe se está limpo ou sujo?". Agora as crianças A e B (que tem a testa suja) já sabem o estado real do sistema, pois elas sabem que tem pelo menos 2 crianças sujas e elas só veem uma criança suja, logo deduzem que elas estão sujas. Quando elas dizem para o pai que sabem se a testa delas está suja ou não a criança C também fica sabendo se sua testa está suja ou não, pois para as outras crianças já saberem o resultado a sua testa tem que estar limpa.

110

Figura 2.5: Crianças Sujas - estado final

2.2.2 Ações Privadas (Modelos de Ação)

A inserção de uma nova informação a um agente é chamada de atualização *("update")* e é representada por um Modelo de Ação (Action Model). Por exemplo, um agente aprende que a proposição ψ é verdadeira. Uma atualização com essa sentença significa remover as arestas do agente aos mundos onde ψ não é verdadeiro. Caso um estado não seja mais considerado um dos estados possíveis por nenhum agente, ele é eliminado. É importante ressaltar que em sistemas multi-agentes, diferentes agentes podem ter diferentes acessos as novas informações.

Uma consideração importante a ser feita na lógica epistêmica dinâmica é que as consequências das ações não são esquecidas pelos agentes, ou seja, se alguma ação afirmou que ϕ é verdadeiro, ϕ nunca poderá ser falso no futuro, pois todos os estados que continham $\neg\phi$ deixaram de existir após a ação.

Exemplo 4 (Atualização do Modelo) *Considerando o exemplo do jogo das cartas, seja a ação de atualização a seguinte sentença: "Jogador A mostra a carta para o jogador B.".*

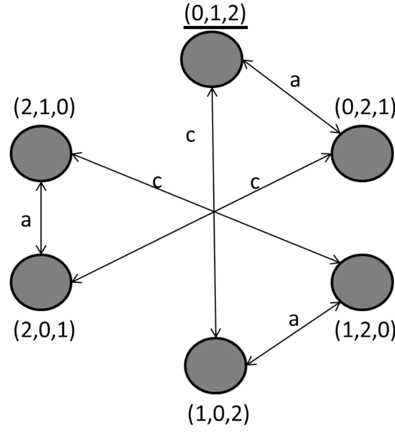

Figura 2.6: Jogo das 3 cartas - Depois da atualização

Resultados da atualização:

- *O jogador B sabe a carta do jogador A;*
- *O jogador C não sabe a carta do jogador A;*
- *O jogador C sabe que o jogador B sabe a carta do jogador A;*
- *O jogador A sabe que o jogador C sabe que o jogador B sabe a sua carta;*

Sintaxe e Semântica

A descrição das ações epistêmicas é feita através de uma estrutura, que se assemelha com o modelo de Kripke, chamada modelo de ação, onde cada ação tem uma pré-condição que precisa ser satisfeita para a ação ser realizada.

Definição 6 *Um modelo de ação* M *é uma estrutura* $\langle \mathsf{S}, \sim, \mathsf{pre} \rangle$

- S *é um domínio finito de pontos de ações ou eventos,*
- $\sim_a$ *é a relação de equivalência em* S*,*
- $\mathsf{pre} : \mathsf{S} \mapsto \mathcal{L}$ *é a função de precondição que atribui uma precondição para cada* $\mathsf{j} \in \mathsf{S}$*. Onde* $\mathcal{L}$ *é a linguagem descrita na próxima definição.*

Definição 7 *A linguagem do modelo de ação consiste em um conjunto contável* Φ *de símbolos proposicionais, um conjunto finito* $\mathcal{A}$ *de agentes, os conectivos booleanos* $\neg$ *e* $\wedge$*, o operador* K_a *para cada agente* a *e o operador* $[\alpha]$ *. As fórmulas são definidas como segue:*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid K_a\varphi \mid [\mathsf{M}, \mathsf{j}]\varphi \mid [\alpha]\varphi$$

$$\alpha ::= (\alpha \cup \alpha) \mid ((\mathsf{M}, \mathsf{j}); (\mathsf{M}', \mathsf{j}))$$

onde φ é igual ao BNF(Backus–Naur Form)[3] *do modelo epistêmico acrescido do operador* $[\alpha]$, $p \in \Phi$, $a \in \mathcal{A}$ *e* (M, j) *é um modelo de ação enraizado*[4].

Para aplicarmos um modelo de ação em um modelo epistêmico, realizamos um produto cartesiano restrito dos seus domínios. É restrito pois só podemos realizar os produtos nos estados onde as pré-condições da ação são verdadeiras.

Definição 8 *Dado um estado epistêmico* $(\mathcal{M}, s)$ *com* $\mathcal{M} = \langle S, \sim_a, V\rangle$ *e um modelo de ação* (M, j) *com* $\mathsf{M} = \langle \mathsf{S}, \sim, \mathsf{pre}\rangle$, *o resultado da execução* (M, j) *em* $(\mathcal{M}, s)$ *é* $(\mathcal{M} \otimes \mathsf{M}, (s, \mathsf{j}))$ *onde* $\mathcal{M} \otimes \mathsf{M} = \langle S', \sim', V'\rangle$ *tal que:*

1. $S' = \{(s, \mathsf{j})$ *tal que* $s \in S, \mathsf{j} \in \mathsf{S},$ *e* $\mathcal{M}, s \models \mathsf{pre}(\mathsf{j})\}$,

2. $(s, \mathsf{j}) \sim'_a (t, \mathsf{k})$ *sse* $(s \sim_a t$ *e* $\mathsf{j} \sim_a \mathsf{k})$,

3. $(s, \mathsf{j}) \in V'(p)$ *sse* $s \in V(p)$.

Se um agente consegue diferenciar duas ações, por consequência ele consegue diferenciar os estados resultantes dessas ações. Dois estados são não diferenciáveis para um agente, se e somente se, esses dois estados são o resultado de duas ações, que o agente não consegue diferenciar, em dois estados que não eram diferenciáveis.

Suponha que temos n modelos de ação para serem aplicados em sequência em um modelo epistêmico. Podemos aplicar o primeiro modelo de ação ao modelo epistêmico inicial M, que resultará em um modelo epistêmico M2, aplicamos o segundo modelo de ação em M2, que resultará em um modelo epistêmico M3, onde será aplicado o terceiro modelo de ação e assim por diante. Em vez de ir aplicando cada modelo de ação aos modelos epistêmicos resultantes podemos fazer a composição dos modelos de ação e aplicar apenas a composição dos modelos de ação ao modelo epistêmico inicial.

Definição 9 (Composição de modelos de ação)

Dado os modelos de ação (M, j) *com* $\mathsf{M} = \langle \mathsf{S}, \sim, \mathsf{pre}\rangle$ *e* $(\mathsf{M}', \mathsf{j}')$ *com* $\mathsf{M}' = \langle \mathsf{S}', \sim', \mathsf{pre}'\rangle$, *a composição deles é o modelo de ação* $(\mathsf{M}; \mathsf{M}', (\mathsf{j}, \mathsf{j}'))$ *com* $\mathsf{M}; \mathsf{M}' = \langle \mathsf{S}'', \sim'', \mathsf{pre}''\rangle$*:*

- $\mathsf{S}'' = \{(\mathsf{j}, \mathsf{j}')$ *tal que* $\mathsf{j} \in \mathsf{S}, \mathsf{j}' \in \mathsf{S}'\}$,

- $(\mathsf{j}, \mathsf{j}') \sim''_a (\mathsf{k}, \mathsf{k}')$ *sse* $(\mathsf{j} \sim_a \mathsf{k}$ *e* $\mathsf{j}' \sim_a \mathsf{k}')$,

- $\mathsf{pre}''(\mathsf{j}, \mathsf{j}') = \langle(\mathsf{M}, \mathsf{j})\rangle \mathsf{pre}'(\mathsf{j}')$.

[3] A forma de Backus-Naur é uma meta-sintaxe usada para expressar gramáticas livres de contexto, isto é, um modo formal de descrever linguagens formais.

[4] Um modelo enraizado com raiz jé um modelo com um elemento distinguido $\mathsf{j} \in \mathsf{S}$.

Como os valores das proposições dos estados não são alterados pelo modelo de ação, a ordem com que os modelos de ação são executados não altera o resultado final, pois todas as pré-condições que eram satisfeitas antes de cada atualização continuam sendo satisfeitas depois das atualização. Logo, também não importa a ordem que é realizada a composição dos modelos de ação pois o modelo final será igual.

Definição 10 *Dado um estado epistêmico* $(\mathcal{M}, s)$ *com* $\mathcal{M} = \langle S, \sim_a, V\rangle$ *e um ponto de ação* (M, j) *com* $\mathsf{M} = \langle \mathsf{S}, \sim, \mathsf{pre}\rangle$*, a noção de satisfação* $\mathcal{M}, s \models \varphi$ *é definida a seguir:*

$$
\begin{array}{lll}
\mathcal{M}, s \models p & sse & s \in V(p) \\
\mathcal{M}, s \models \neg\phi & sse & \mathcal{M}, s \not\models \phi \\
\mathcal{M}, s \models \phi \wedge \psi & sse & \mathcal{M}, s \models \phi \text{ e } \mathcal{M}, s \models \psi \\
\mathcal{M}, s \models K_a\phi & sse & \text{para todo } s' \in S : s \sim_a s' \text{ implica } \mathcal{M}, s' \models \phi \\
\mathcal{M}, s \models [\mathsf{M}, \mathsf{j}]\phi & sse & \mathcal{M}, s \models \mathsf{pre}(\mathsf{j}) \text{ implica } \mathcal{M} \otimes \mathsf{M}, (s, \mathsf{j}) \models \phi
\end{array}
$$

Exemplo 5 (Modelo de Ação)

Representação gráfica do modelo de ação do jogador A mostrar sua carta para o jogador B. Podemos observar que no modelo de ação apenas C não consegue diferenciar as ações.[5].

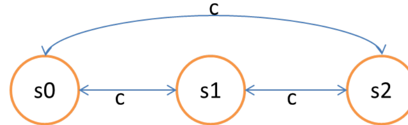

Figura 2.7: Modelo de ação

Pré-condições:

- *Estado s0: o jogador A ter a carta 0.*
- *Estado s1: o jogador A ter a carta 1.*
- *Estado s2: o jogador A ter a carta 2.*

Escrevendo formalmente esse modelo de ação:

- M *= [s0, s1 ,s2],*
- $\sim$ *= ([c, s0, s1] , [c, s0, s2] , [c, s1, s2]),*

[5]Omitimos os loops reflexivos da figura, pois todos os agentes não diferenciam um estado dele mesmo.

- pre = *([s0, 0a] , [s1, 1a] , [s2, 2a]).*

Exemplo 6 (Produto modelo de ação)

Utilizando o exemplo anterior, vamos agora mostrar passo a passo como é realizado o produto do modelo epistêmico com o modelo de ação.

Temos os seguintes modelo epistêmico e modelo de ação para o jogo das 3 cartas onde o jogador A mostra sua carta para o jogador B.

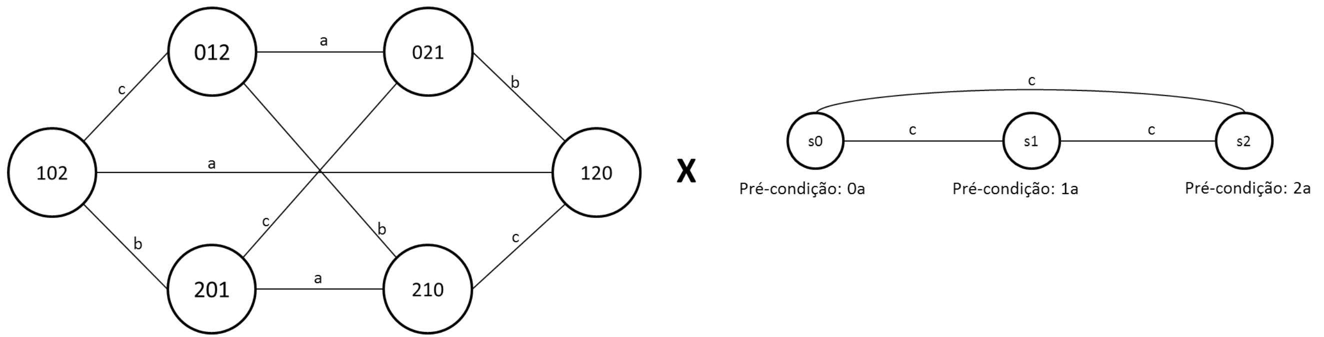

Figura 2.8: Modelo Epistêmico x Modelo de Ação

Inicialmente vamos esquecer das pré-condições e das arestas e focar nos estados. Realizando o produto cartesianos temos os seguintes estados resultantes:

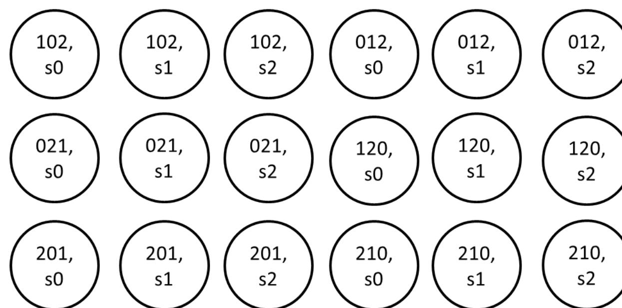

Figura 2.9: Estados gerados pelo produto cartesiano

Como dito na definição 8, o produto cartesiano é limitado, pois só podemos aplicar a estados que atendam as pré-condições, logo temos que eliminar todos os estados que não atendem as pré-condições.

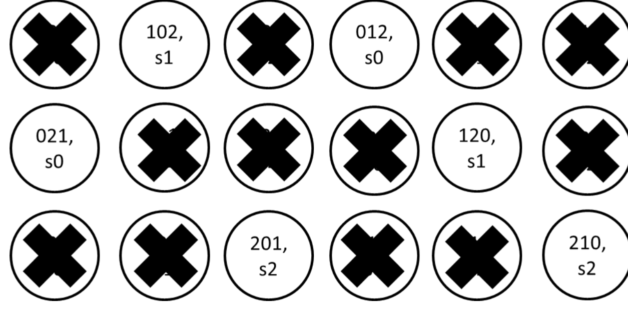

Figura 2.10: Estados eliminados pelo produto cartesiano limitado pelas pré-condições

Agora vamos colocar as arestas, lembrando que na definição 8, as arestas resultantes do produto são definidas por: $(s, \mathsf{j}) \sim'_a (t, \mathsf{k})$ *sse* $(s \sim_a t$ *e* $\mathsf{j} \sim_a \mathsf{k})$. *Logo, só podemos ligar os estados desse novo modelo epistêmico quando os estados do modelo epistêmico original e os estados do modelo de ação, que compõem esse novo estado, estão ligados. O modelo resultante é:*

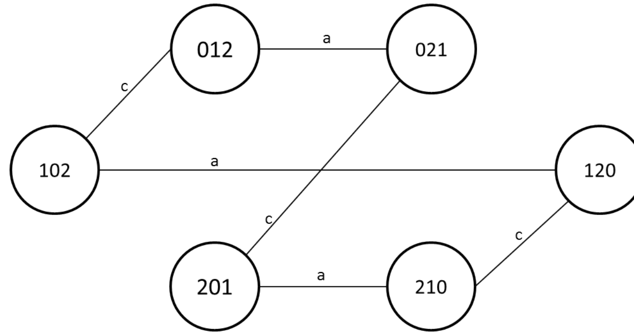

Figura 2.11: Modelo resultante da ação de Anne mostrar sua carta para Bill

Sistemas Axiomáticos

Axiomas

1. $[\mathsf{M}, \mathsf{j}]p \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow p)$,

2. $[\mathsf{M}, \mathsf{j}]\neg\phi \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow \neg[\mathsf{M}, \mathsf{j}]\phi)$,

3. $[\mathsf{M}, \mathsf{j}](\phi \wedge \psi) \leftrightarrow ([\mathsf{M}, \mathsf{j}]\phi \wedge [\mathsf{M}, \mathsf{j}]\psi)$,

4. $[\mathsf{M}, \mathsf{j}]K_a\phi \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow \bigwedge_{\mathsf{j} \sim_a \mathsf{k}} K_a[\mathsf{M}, \mathsf{k}]\phi)$,

5. $[[\mathsf{M}, \mathsf{j}] \cup [\mathsf{M}', \mathsf{j}']]\phi \leftrightarrow [\mathsf{M}, \mathsf{j}]\phi \wedge [\mathsf{M}', \mathsf{j}']\phi$,

6. $[\mathsf{M}, \mathsf{j}][\mathsf{M}', \mathsf{j}']\phi \leftrightarrow [(\mathsf{M}, \mathsf{j}); (\mathsf{M}', \mathsf{j}')]\phi$. (Composição de modelos de ação)

As provas dos axiomas descritos acima podem ser encontradas na seção 6.6 de [3].

2.3 Lógica Dinâmica Proposicional

Nessa seção, apresentaremos de maneira sucinta, a lógica proposicional dinâmica(PDL), apenas para contextualizar o leitor para próxima seção, onde mencionamos essa lógica.

2.3.1 Sintaxe e Semântica

Definição 11 *A linguagem da PDL consiste em um conjunto contável* Φ *de símbolos proposicionais, um conjunto contável de* Π *programas básicos, os conectivos booleanos* $\neg$ *e* $\wedge$*, os construtores de programas* ; *(concatenação sequencial),* $\cup$ *(escolha não determinística) e* **(iteração) e um operador modal* $\langle\pi\rangle$ *para cada programa* π*. Programas não básicos são construídos através dos construtores e de outros programas. As fórmulas dessa lógica são definidas a seguir:*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\pi\rangle\varphi, \ \textit{com} \ \pi ::= a \mid \pi_1;\pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^\star \mid \pi?,$$

onde $p \in \Phi$ *e* $a \in \Pi$*.*

Utilizamos a abreviação padrão para: $\bot \equiv \neg\top$, $\varphi \vee \phi \equiv \neg(\neg\varphi \wedge \neg\phi)$, $\varphi \rightarrow \phi \equiv \neg(\varphi \wedge \neg\phi)$ e $[\pi]\varphi \equiv \neg\langle\pi\rangle\neg\varphi$.

Definição 12 *Uma* estrutura *em PDL é uma tupla* $\mathcal{F} = (W, R_a)$ *onde:*

- W *é um conjunto não vazio de estados;*
- R_a *é uma relação binária sobre* W*, para cada programa básico* $a \in \Pi$*;*
- *é possível definir indutivamente uma relação binária* R_π*, para cada programa não básico* π*, como definido abaixo:*
  - $R_{\pi_1;\pi_2} = R_{\pi_1} \circ R_{\pi_2}$,
  - $R_{\pi_1 \cup \pi_2} = R_{\pi_1} \cup R_{\pi_2}$,
  - $R_{\pi^\star} = R_\pi^\star$, *onde* $R_\pi^\star$ *denota o fechamento transitivo de* R_π*.*

Definição 13 *Um* modelo *em PDL é um modelo de Kripke* $\mathcal{M} = (\mathcal{F}, \mathbf{V})$*, onde* $\mathcal{F}$ *é uma estrutura em PDL e* $\mathbf{V}$ *é uma função de valoração* $\mathbf{V} : \Phi \rightarrow 2^W$*.*

A definição semântica de satisfazibilidade em PDL é definida da seguinte forma:

Definição 14 *Seja* $\mathcal{M} = (\mathcal{F}, \mathbf{V})$ *um modelo. A noção de satisfazibilidade de uma fórmula* φ *em um modelo* $\mathcal{M}$ *em um estado* w*, representada pela notação* $\mathcal{M}, w \Vdash \varphi$*, pode ser definida indutivamente assim:*

- $\mathcal{M}, w \models p$ *sse* $w \in \mathbf{V}(p)$,
- $\mathcal{M}, w \models \top$ *sempre,*
- $\mathcal{M}, w \models \neg\varphi$ *sse* $\mathcal{M}, w \not\Vdash \varphi$,
- $\mathcal{M}, w \models \varphi_1 \wedge \varphi_2$ *sse* $\mathcal{M}, w \Vdash \varphi_1$ *e* $\mathcal{M}, w \Vdash \varphi_2$,
- $\mathcal{M}, w \models \langle\pi\rangle\varphi$ *sse existe* $w' \in W$ *tal que* $wR_\pi w'$ *e* $\mathcal{M}, w' \Vdash \varphi$.

Axiomatização

1. *Todas as instanciações das tautologias da lógica proposicional*,
2. $[\pi](\varphi \to \psi) \to ([\pi]\varphi \to [\pi]\psi)$,
3. $[\pi_1; \pi_2]\varphi \leftrightarrow [\pi_1][\pi_2]\varphi$,
4. $[\pi_1 \cup \pi_2]\varphi \leftrightarrow [\pi_1] \wedge [\pi_2]\varphi$,
5. $[\pi^\star]\varphi \leftrightarrow \varphi \wedge [\pi][\pi^\star]\varphi$,
6. $[\pi^\star](\varphi \to [\pi]\varphi) \to ([\pi]\varphi \to [\pi^\star]\varphi)$,

Regras de inferência

M.P. $\varphi, \varphi \to \psi/\psi$ U.G. $\varphi/[\pi]\varphi$ UB. $\varphi/\sigma\varphi$
onde σ mapeia uniformemente as fórmulas em variáveis proposicionais.

Lema 1 *(Correção e Completude) O sistema axiomático acima é correto e completo com relação a PDL.*

2.4 Lógica Epistêmica Dinâmica com Atribuições

No jogo das crianças sujas,"Muddy Children Problem", o que aconteceria se acrescentássemos a ação "lavar a criança"? E no jogo das cartas,"Russian Card Game", mostrado nas seções anteriores, o que aconteceria se tivéssemos a ação de "trocar a carta"? Embora a lógica epistêmica dinâmica nos permita lidar com mudanças nas informações dos agentes, ela não permite que um agente altere o valor (verdadeiro e falso) de uma proposição em um determinado estado ou no modelo como um todo. Lógica epistêmica dinâmica com atribuição é uma extensão da lógica epistêmica dinâmica para lidar com a atribuição de novas proposições aos estados dos agentes.

Vários trabalhos têm sido feitos nessa área (conhecimento + atribuição), como VAN DITMARSCH *et al.* [1], KOOI [4] e J. VAN BENTHEN e KOOI [5].

Em [5], é formulada um nova lógica chamada LCC (Logic of comunication and change) que adiciona o operador de conhecimento à PDL (Propositional dynamic logic). Nesse artigo é descrito também um modelo de atualização muito parecido com o modelo de ação da DEL, acrescido apenas da propriedade de substituição, isso permite que os valores das proposições sejam alterados. Essa propriedade de substituição mapeia todas as proposições em seus novos valores, se uma proposição não sofreu alteração, seu valor é mapeado para o seu valor anterior.

No entando, nessa seção iremos focar no trabalho de VAN DITMARSCH *et al.* [1], que faz uma extensão da lógica epistêmica dinâmica (DEL) para tratar atribuições.

A atribuição não é algo novo da lógica, mas é uma operação primitiva em todas as linguagens de programação. A operação $p = e$ pode ser traduzida em "a variável p recebe o valor da expressão e". Da mesma forma, em lógica epistêmica dinâmica com atribuições (DELWA) , podemos atribuir novos valores (verdadeiro ou falso) às proposições.

Para facilitar o entendimento, [1] divide a lógica epistêmica dinâmica com atribuições em duas partes: Atribuições públicas (utilizando como exemplo o "Muddy Children Problem") e atribuições atômicas (utilizando como exemplo o "Russian Card Game"). A primeira parte é simplesmente um caso especial da segunda, sendo tratada separadamente apenas para facilitar o entendimento.

2.4.1 Atribuições públicas

Vamos pensar no exemplo das crianças sujas. Suponha que existam 3 crianças: Anne, Bill e Cath, onde 2 delas têm a testa suja. É de conhecimento comum que as crianças só conseguem ver a testas das outras crianças, ou seja, não conseguem ver a própria testa. O pai das crianças diz: existe pelo menos uma criança com a testa suja. Nesse momento, antes das crianças se pronunciarem, ele joga um balde de água na Anne. A ação de jogar um balde de água em Anne é uma ação de atribuição, que atribui o valor "criança com a testa limpa" a Anne, independente do valor anterior.

Atribuições públicas são as consequências do anúncio público de uma fórmula.

Exemplo 7 (Atribuições públicas)

Representação do jogo descrito no inicio dessa seção. Modelo epistêmico inicial desse exemplo é o mesmo que foi mostrado anteriormente, onde os rótulos dos estados do modelo epistêmico são representados por xyz onde x,y,z $\in$ *0,1 e 0 indica que a criança está limpa e 1 indica que a criança está suja. O estado real é o 110, onde a Anne e o Bill estão sujos e a Cath está limpa.*

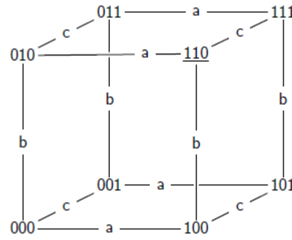

Figura 2.12: DELWA - Crianças Sujas - Modelo inicial

Ação do pai afirmar que existe pelo menos uma criança suja. Até aqui, o exemplo é exatamente igual ao mostrado anteriormente.

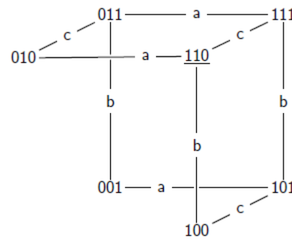

Figura 2.13: DELWA - Crianças Sujas - pelo menos uma criança suja

Agora que as crianças já sabem que tem pelo menos uma suja, o pai executa a ação de jogar um balde de água em Anne. Essa ação altera o resultado final do exemplo? Essa ação faz com que as crianças não saibam mais se tem pelo menos uma criança suja, pois Anne poderia ser a única criança suja e agora ela está limpa.

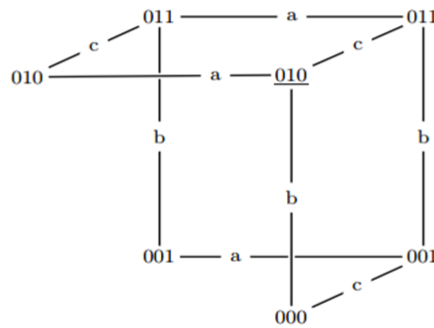

Figura 2.14: DELWA - Crianças Sujas - pai joga um balde de água em Anne

O pai agora começa a perguntar se alguém já sabe se sua testa está suja ou limpa. Independente de quantas vezes o pai pergunte, Bill e Cath nunca vão ter certeza do estado real do sistema.

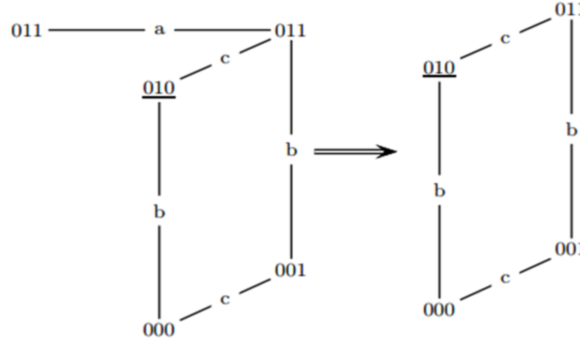

Figura 2.15: DELWA - Crianças Sujas - "Alguém sabe o estado real do sistema?"

Podemos notar que o fato do pai jogar um balde de água em Anne faz com que Bill e Cath nunca saibam se estão sujos ou limpos, isso porque eles não sabem se a Anne afirmou que sabia se a sua testa estava limpa ou suja por causa da ação do pai de jogar um balde de água ou se foi pelo fato dela ver apenas uma criança suja.

Sintaxe e Semântica

Definição 15 *A linguagem das atribuições públicas consiste em um conjunto contável* Φ *de símbolos proposicionais, um conjunto finto* $\mathcal{A}$ *de agentes, os conectivos booleanos* $\neg$ *e* $\wedge$*, o operador* K_a *para cada agente* a *e o operador de atribuição. As fórmulas são definidas como segue:*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid K_a\varphi \mid [p := \phi]\varphi$$

onde $[p := \phi]\varphi$ *indica que depois que* p *recebe o valor de* ϕ *vale* φ *.*

Definição 16 *Dado um modelo epistêmico* $\mathcal{M} = \langle S, \sim_a, V\rangle$ *e um estado* $s \in S$*. A noção de satisfação* $\mathcal{M}, s \models \varphi$ *é definida a seguir:*

$$\begin{array}{lll} \mathcal{M}, s \models p & sse & s \in V(p), \\ \mathcal{M}, s \models \neg\phi & sse & \mathcal{M}, s \not\models \phi, \\ \mathcal{M}, s \models \phi \wedge \psi & sse & \mathcal{M}, s \models \phi \text{ e } \mathcal{M}, s \models \psi, \\ \mathcal{M}, s \models K_a\phi & sse & \text{para todo } s' \in S : s \sim_a s' \text{ implica } \mathcal{M}, s' \models \phi, \\ \mathcal{M}, s \models [p := \phi]\psi & sse & \mathcal{M}_{p:=\phi}, s \models \psi. \end{array}$$

onde, $\mathcal{M}_{p:=\phi} = \langle S, \sim_a, V'\rangle$ *é igual a* $\mathcal{M}$*, com exceção de* $V'_p = [\phi]_{\mathcal{M}}$*, ou seja, para todo* $q \neq p$ *temos que* $V'_q = V_q$*.*

2.4.2 Atribuições atômicas

Vimos que, nas atribuições públicas, as alterações são feitas em todo o modelo, ou seja, o conhecimento de todos os agentes é alterado. As atribuições atômicas alteram

apenas o conhecimento dos agente envolvidos nas ações.

Vamos voltar ao exemplo das cartas, o que aconteceria se os jogadores A e B trocassem de cartas?

Exemplo 8 (Atribuições atômicas)

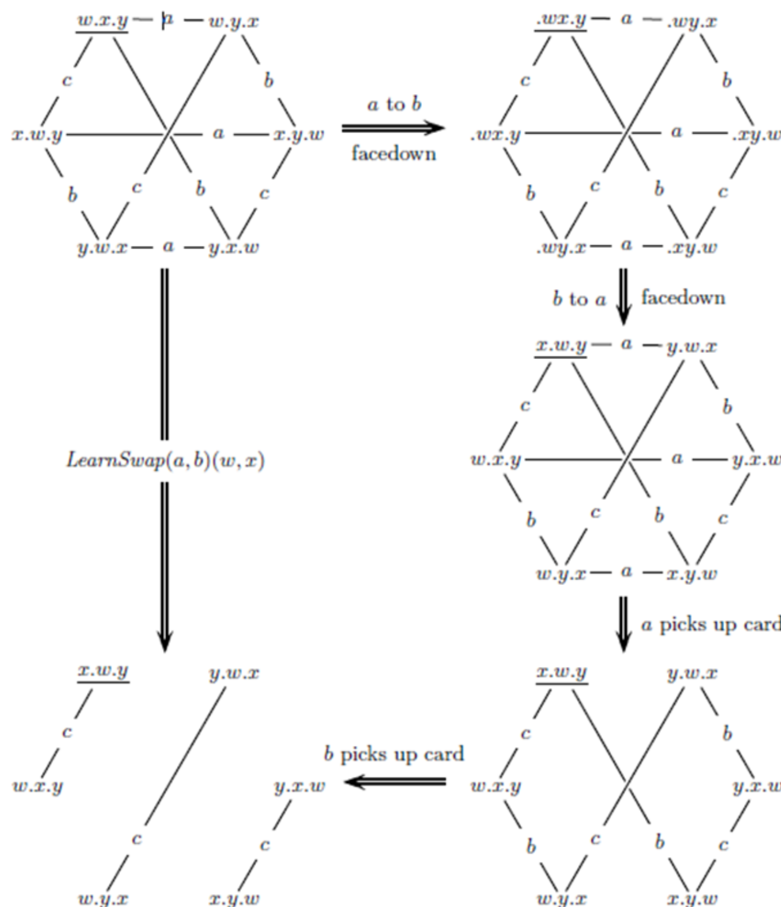

Figura 2.16: Russian Cards

Primeiro modelo epistêmico, em cima e mais a esquerda, mostra três jogadores Anne, Bill e Cath (a,b,c) onde cada um está segurando uma carta (w,x,y). Inicialmente, cada um só conhece a sua carta e sabe que os outros jogadores também só conhecem a sua própria carta. O estado real é o estado sublinhado. Continuando, no sentido horário, Anne coloca sua carta, de cabeça para baixo, na frente de Bill, depois Bill coloca sua carta, de cabeça para baixo, na frente de Anne. Nesse momento Anne e Bill pegam a carta que está na sua frente, ou seja, ocorre a troca de cartas entre Anne e Bill. Agora Anne e Bill não tem mais dúvidas sobre qual é o estado real, enquanto Cath ainda tem dúvidas. Cath sabe que Anne e Bill trocaram de carta mas não sabe qual foi a troca, visto que ela não sabia que carta cada um deles tinha e por isso não sabe quais atribuições foram feitas.

Sintaxe e Semântica

Definição 17 *A linguagem das atribuições atômicas consiste em um conjunto contável* Φ *de símbolos proposicionais, um conjunto finito* $\mathcal{A}$ *de agentes, os conectivos booleanos* $\neg$ *e* $\wedge$*, o operador* K_a *para cada agente a e os operadores de teste (?), de atribuição (:=) , de aprendizado (* L_G *) , execução sequencial (;) e de escolhas não determinísticas (!,¡,U) . As fórmulas são definidas como segue:*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid K_a\varphi \mid [\alpha]\varphi$$

$$\alpha ::= ?\varphi \mid p := \varphi \mid L_G\alpha \mid (\alpha!\beta) \mid (\alpha¡\beta) \mid (\alpha;\beta) \mid (\alpha \cup \beta)$$

Onde, $?\varphi$ *é um teste,* $p := \varphi$ *é uma atribuição atômica ,* $L_G\alpha$ *produz a ação "grupo G aprende* α*",* $(\alpha;\beta)$ *ação sequencial ,* $(\alpha \cup \beta)$ *escolha não determinística,* $(\alpha!\beta)$ *e* $(\alpha¡\beta)$ *transformam a ação não determinística em determinística para os agentes envolvidos (agentes em* α *e* β*), a escolha continua não determinística para os outros agentes. Na primeira escolhe-se* α *e na segunda* β *.*

Definição 18 *Dado um modelo epistêmico* $\mathcal{M} = \langle S, \sim_a, V\rangle$ *e um estado* $s \in S$*. As noções de satisfação* $\models$ *de uma fórmula* φ *em* $(\mathcal{M}, s)$ *e de satisfação* $[\![.]\!]$ *de uma ação* α *entre estados epistêmicos são definidas a seguir:*

$\mathcal{M}, s \models p$	*sse*	$s \in V(p)$
$\mathcal{M}, s \models \neg\phi$	*sse*	$\mathcal{M}, s \not\models \phi$
$\mathcal{M}, s \models \phi \wedge \psi$	*sse*	$\mathcal{M}, s \models \phi$ *e* $\mathcal{M}, s \models \psi$
$\mathcal{M}, s \models K_a\phi$	*sse*	*para todo* $s' \in S : s \sim_a s'$ *implica* $\mathcal{M}, s' \models \phi$
$\mathcal{M}, s \models [\alpha]\phi$	*sse*	*para todo* $(\mathcal{M}', s') : (\mathcal{M}, s)[\![\alpha]\!](\mathcal{M}', s')$ *implica* $(\mathcal{M}', s') \models \phi$
$(\mathcal{M}, s)[\![?\phi]\!](\mathcal{M}', s')$	*sse*	$\mathcal{M}' = \langle [\![\phi]\!]_M, \emptyset, V \cap [\![\phi]\!]_M\rangle$ *e* $s' = s$
$(\mathcal{M}, s)[\![p := \phi]\!](\mathcal{M}', s')$	*sse*	$\mathcal{M}' = \langle S, \emptyset, V'\rangle$ *e* $s' = s$
$(\mathcal{M}, s)[\![L_G\alpha]\!](\mathcal{M}', s')$	*sse*	$\mathcal{M}' = \langle S', \sim', V'\rangle$ *e* $(\mathcal{M}, s)[\![\alpha]\!]s'$
$[\![\alpha;\beta]\!]$	$=$	$[\![\alpha]\!] \circ [\![\beta]\!]$
$[\![\alpha \cup \beta]\!]$	$=$	$[\![\alpha]\!] \cup [\![\beta]\!]$
$[\![\alpha!\beta]\!]$	$=$	$[\![\alpha]\!]$
$[\![\alpha¡\beta]\!]$	$=$	$[\![\beta]\!]$

Na cláusula de atribuição, $V' = V$ *com exceção de* $V'_p = [\![\alpha]\!]\mathcal{M}$*. Na cláusula Learning (*L_G*),* $\mathcal{M}'$ *é tal que:* $S' = \{(\mathcal{M}'', t'') | \exists u \in \mathcal{M} : (\mathcal{M}, u)[\![t(\alpha)]\!](\mathcal{M}'', t'')\}$ *; para um agente arbitrário n: se* $(\mathcal{M}, s)[\![t(\alpha)]\!](\mathcal{M}''_1, s'')$ *e* $(\mathcal{M}, \sqcup)[\![t(\alpha)]\!](\mathcal{M}''_2, t'')$ *então* $(\mathcal{M}''_1, s'') \sim'_n (\mathcal{M}''_2, t'')$ *sse* $(\mathcal{M}''_1, s'') \sim_n (\mathcal{M}''_2, t'')$ *(onde* $\sim_n$ *significa equivalência de estados epistêmicos) ou (* $n \notin gr(\mathcal{M}''_1) \cup gr(\mathcal{M}''_2)$ *e* $s \sim_n t$ *) [onde* $gr(\mathcal{M})$ *é um conjunto de agentes]; e para um átomo arbitrário p e para um estado* $(\mathcal{M}''_2, s'')$ *(com*

valoração V″) no domínio de $\mathcal{M}'$: $(\mathcal{M}''_2, s'') in V'_p$ sse $s'' \in V''_p$.

O resultado de uma ação de teste (? α) é um modelo epistêmico sem nenhuma aresta, e apenas com os estados onde α é verdadeiro. O resultado de uma ação de atribuição atômica é, igual ao teste, um modelo epistêmico sem nenhuma aresta, porém altera o valor dos átomos.

A ação onde Anne troca a carta q pela carta q' de Bill, de modo que Cath não saiba quais cartas foram trocadas pode ser descrita como:

- $Swap(a,b)(q,q') = ?(q_a \wedge q'_b); q_a = \bot; q_b = \top; q'_a = \top; q'_b = \bot$
- LearnSwap(a,b)(w,x) = *Labc*(*Lab* Swap(a,b)(w,x) ! (*Lab* Swap(a,b)(w,y) $\cup$ *Lab* Swap(a,b)(x,w) $\cup$ *Lab* Swap(a,b)(x,y) $\cup$ *Lab* Swap(a,b)(y,w) $\cup$ *Lab* Swap(a,b)(y,x)))

Lemos essa ação da seguinte maneira: Anne, Bill e Cath aprendem (*Labc*) que uma das seis alternativas é executada: ou Anne e Bill aprendem (*Lab*) que eles trocaram as cartas w e x (Swap(a,b)(w,x)), ou ... , ou ... ; A primeira alternativa é a que realmente aconteceu porém apenas Anne e Bill, agentes envolvidos na ação *Lab* sabem isso. A ação *Swap* consiste em 5 partes, a primeira parte é para verificar se Anne tem a carta q e Bill tem a carta q', se isso for verdade as outras quatro ações são executadas simultaneamente, as quais são as ações de troca.

2.5 Verificador de Modelos Epistêmicos Dinâmicos

Segundo VAN DITMARSCH *et al.* [3], atualmente ainda não existe um provador de teoremas automático para lógica epistêmica dinâmica de múltiplos agentes que incorpore todos os operadores presentes na lógica, existem alguns provadores porém eles limitam as fórmulas a um conjunto finito. O maior problema da prova é a presença de diversos operadores modais, como conhecimento comum, que não permitem uma redução direta das regras que os removem, como por exemplo o axioma $C_B\phi \leftrightarrow \phi \wedge E_B C_B \phi$, nesse caso o operador de conhecimento comum aparece dos dois lados, impedindo a sua eliminação.

Uma alternativa para provar se uma fórmula é consequência de um teorema, é verificar se a fórmula é verdadeira em um modelo que incorpore características suficientes do teorema. A maioria dos verificadores de modelos epistêmicos existentes tendem a modelar as características dinâmicas do modelo de forma temporal, diferentemente do que acontece na lógica epistêmica dinâmica onde o tempo não é explícito. Recentemente, por volta de 2006, surgiu um verificador de modelos epistêmicos realmente dinâmico, chamado DEMO (Dynamic Epistemic MOdeling)[9].

2.5.1 DEMO

O DEMO foi implementado em Haskell e é uma ferramenta muito poderosa. Dentre suas diversas funcionalidades, estão presentes: modelagem das atualizações epistêmicas, geração gráfica do modelo resultante das atualizações, geração gráfica do modelo de ação, avaliação de fórmulas no modelo epistêmico, tradução de fórmulas epistêmicas dinâmicas para fórmulas PDL.

Primeiro vamos apresentar um pouco da sintaxe do DEMO e depois vamos mostrar como modelar os exemplos vistos no início desse capítulo (Muddy Children e Russian Cards) no DEMO.

Modelos Epistêmicos

Para gerar um modelo epistêmico simples, podemos utilizar o seguinte comando:

$$em0 = initE([P0, Q0][a, b, c])$$

Esse comando gera um modelo epistêmico com 3 agentes (a,b e c) e 2 proposições(p e q). O número de estados do modelo será igual ao numero de combinações dessas proposições, nesse caso são 4 estados ([], [p], [q] e [p,q]). Como não foi especificado nada sobre a acessibilidade, o grafo é completo, ou seja, os agentes têm dúvidas entre todos os estados. Esse modelo epistêmico fica salvo na variável em0 e será utilizado mais à frente.

Às vezes, queremos especificar o número de estados e as funções de acessibilidade como parte do nosso modelo epistêmico. Para isso, podemos criar o modelo epistêmico da seguinte forma:

$$\begin{aligned} cards0 &:: EpistM \\ cards0 &= (Pmod[0..5]\ val\ acc\ [0]) \end{aligned}$$

$$
\begin{array}{lcl}
where & & \\
val & = & [(0,[P1,Q2,R3]),(1,[P1,R2,Q3]), \\
 & & (2,[Q1,P2,R3]),(3,[Q1,R2,P3]), \\
 & & (4,[R1,P2,Q3]),(5,[R1,Q2,P3])] \\
acc & = & [(a,0,0),(a,0,1),(a,1,0),(a,1,1), \\
 & & (a,2,2),(a,2,3),(a,3,2),(a,3,3), \\
 & & (a,4,4),(a,4,5),(a,5,4),(a,5,5), \\
 & & (b,0,0),(b,0,5),(b,5,0),(b,5,5), \\
 & & (b,2,2),(b,2,4),(b,4,2),(b,4,4), \\
 & & (b,3,3),(b,3,1),(b,1,3),(b,1,1), \\
 & & (c,0,0),(c,0,2),(c,2,0),(c,2,2), \\
 & & (c,3,3),(c,3,5),(c,5,3),(c,5,5), \\
 & & (c,4,4),(c,4,1),(c,1,4),(c,1,1)]
\end{array}
$$

onde:

- cards0 :: EpistM define que o objeto criado é um modelo epistêmico.
- Pmod[0..5] define que teremos 6 estados nesse modelo, numerados de zero a cinco.
- "val" especifica as proposições válidas em cada um dos estados, exemplo (0,[P 1,Q 2,R 3]) diz que p1,q2 e r3 são verdadeiros no estado 0.
- "acc" especifica a acessibilidade de cada agente a cada estado, exemplo (b,3,1) cria um aresta entre o estado 3 e o estado 1 para o agente b, ou seja (3,1) $\in R_b$.
- o último argumento especifica o estado real do sistema.

Essa forma dá um pouco mais de trabalho, mas temos como descrever exatamente o nosso modelo.

Para visualizar o modelo criado podemos utilizar o comando showM (modelo). Exemplo: showM (em0), gera a seguinte saída:

$$
\begin{array}{l}
[0,1,2,3] \\
(0,[])(1,[p])(2,[q])(3,[p,q]) \\
(a,[[0,1,2,3]]) \\
(b,[[0,1,2,3]]) \\
(c,[[0,1,2,3]])
\end{array}
$$

onde a primeira linha representa os estados do modelo, a segunda linha a valoração em cada estado e as outras linhas representam as arestas de cada agente.

Dependendo do tamanho do modelo, fica inviável analisá-lo através do showM. Por isso, o DEMO possuiu um comando que gera um arquivo com a imagem do modelo epistêmico.

$$writeP \text{ ``}filename\text{''}(em0)$$

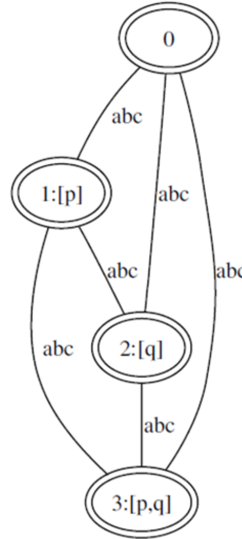

Figura 2.17: Grafo que representa o modelo em0

Modelos de Ação

O DEMO já tem alguns modelos de ação pré-definidos, como por exemplo:

- pulic ϕ - anuncia ϕ para todos os agentes.
- groupM [Agentes] ϕ - anuncia ϕ para os agentes do grupo.
- message Agente A ϕ - anuncia ϕ para o agente A.

Para atualizar um modelo epistêmico com um modelo de ação, utilizamos o comando upd ModeloEpistêmico (ModeloAção). Exemplo: atualizando o modelo epistêmico em0 com a ação pré-definida message.

$$em1 = upd\ em0\ (messageap)$$

onde:

- em0 = modelo epistêmico criado anteriormente
- message a p = mensagem que avisa ao agente 'a' que a proposição 'p' é verdadeira
- em1 = variável que guarda o modelo atualizado

Resultado do modelo epistêmico (em0) atualizado pela ação message a p:

$$
\begin{array}{l}
[0,1,2,3,4,5] \\
(0,[])(1,[p])(2,[p])(3,[q])(4,[p,q]) \\
(5,[p,q]) \\
(a,[[0,2,3,5],[1,4]]) \\
(b,[[0,1,2,3,4,5]]) \\
(c,[[0,1,2,3,4,5]])
\end{array}
$$

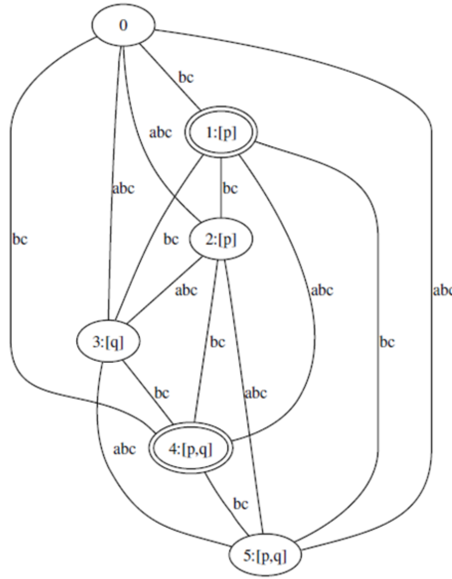

Figura 2.18: Grafo que representa o modelo em1

Assim como nos modelos epistêmicos, podemos descrever cada parte do nosso modelo de ação da seguinte forma:

$$
\begin{array}{lcl}
read & :: & PoAM \\
read & = & (Pmod[0,1]\ pre\ acc\ [1])
\end{array}
$$

$$
\begin{array}{lcl}
where & & \\
pre & = & [(0, Neg\ p), (1, p)] \\
acc & = & [(a,0,0),(a,1,1) \\
 & & (b,0,0),(b,0,1),(b,1,0),(b,1,1)]
\end{array}
$$

onde:

- read :: PoAM define que o objeto criado é um modelo de ação.
- Pmod[0,1] define que teremos 2 estados nesse modelo, onde o 0 é o np e o 1 é o p.

- “pre” especifica as pré-condições de cada ação, exemplo (0,Neg p) diz que pré-condição para a ação 0 ocorrer é p ser falso.
- “acc” especifica a acessibilidade de cada agente a cada ação, exemplo (b,0,1) cria um aresta entre a ação 0 e a ação 1 para o agente b, ou seja (0,1) $\in R_b$.
- o último argumento especifica a ação que foi executada.

Esses modelos de ação também podem ser visualizados através do comando showM. Exemplo: showM read

$$\begin{array}{l} [0,1] \\ (0,[-p])(1,[p]) \\ (a,[[0],[1]]) \\ (b,[[0,1]]) \end{array}$$

Podemos utilizar o DEMO para checar se determinadas condições são verdadeiras no modelo resultante.

Exemplo:

$$\begin{array}{lcl} letter & :: & EpistM \\ letter & = & (Pmod[01]\ val\ acc\ [1]) \end{array}$$

$$\begin{array}{lcl} where & & \\ val & = & [(0,[]),(1,[P0])] \\ acc & = & [(a,0,0),(a,0,1),(a,1,0),(a,1,1), \\ & & (b,0,0),(b,0,1),(b,1,0),(b,1,1)] \end{array}$$

isTrue (upd letter read) (K a p)

Esse comando verifica se depois que o modelo epistêmico “letter” for atualizado com o modelo de ação “read” K a p é verdadeiro, onde K a p = $K_a p$, que significa “Anne sabe p”.

Pode-se verificar sentenças mais complexas, como por exemplo: isTrue (upd letter read) (CK [a,b] Disj[K a p , K a (Neg p)]) , onde a sentença CK [a,b] Disj[K a p , K a (Neg p)] = $C_{ab}(K_a p \vee K_a \neg p)$ que significa “É de conhecimento comum para Anne e Bill que Anne sabe se p é verdadeiro ou falso”.

Muddy Children

Abaixo, a figura do exemplo das crianças sujas implementado no DEMO. Assim como no exemplo descrito do inicio desse capítulo, Bill e Anne estão sujos e Cath está limpa.

```
module Muddy
where

import DEMO

ab_dirty = Conj [ p1, p2, Neg p3]

awareness = [info [b,c] p1,
             info [a,c] p2,
             info [a,b] (Neg p3) ]

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)]


mu0 = upd (initE [P 1, P 2, P 3]) (test ab_dirty)
mu1 = upds mu0 awareness
mu2 = upd  mu1 (public (Disj [p1, p2, p3]))
mu3 = upd  mu2 (public (Conj[Neg aK, Neg bK, Neg cK]))
mu4 = upd  mu3 (public (Conj[Neg aK, Neg bK, Neg cK]))
mu5 = upds mu4 [public (Conj[bK, aK])]
```

Figura 2.19: DEMO - Muddy Children

- a linha 19 : anúncio do pai de que tem pelo menos uma criança com a testa suja;
- as linhas 20 e 21 : anúncios do pai perguntando se alguma criança já sabe se a sua testa está limpa ou suja e a resposta das crianças de que elas ainda não sabem.
- a linha 22 : nesse ponto, Anne e Bill já sabem que suas respectivas testas estão sujas.

Russian Cards

Exemplo do jogo das cartas implementado em DEMO. Todas as premissas são iguais às descritas no início desse capítulo.

```
 1  module Cards
 2  where
 3
 4  import DEMO
 5
 6  cards0 :: EpistM
 7  cards0 = (Pmod [0..5] val acc [0])
 8    where
 9    val    = [(0,[P 1,Q 2,R 3]),(1,[P 1,R 2,Q 3]),
10              (2,[Q 1,P 2,R 3]),(3,[Q 1,R 2,P 3]),
11              (4,[R 1,P 2,Q 3]),(5,[R 1,Q 2,P 3])]
12    acc    = [(a,0,0),(a,0,1),(a,1,0),(a,1,1),
13              (a,2,2),(a,2,3),(a,3,2),(a,3,3),
14              (a,4,4),(a,4,5),(a,5,4),(a,5,5),
15              (b,0,0),(b,0,5),(b,5,0),(b,5,5),
16              (b,2,2),(b,2,4),(b,4,2),(b,4,4),
17              (b,3,3),(b,3,1),(b,1,3),(b,1,1),
18              (c,0,0),(c,0,2),(c,2,0),(c,2,2),
19              (c,3,3),(c,3,5),(c,5,3),(c,5,5),
20              (c,4,4),(c,4,1),(c,1,4),(c,1,1)]
21
22  showABp :: PoAM
23  showABp = (Pmod [0,1] pre susp [0])
24    where
25    pre  = [(0,p1),(1,q1)]
26    susp = [(a,0,0),(a,1,1),
27            (b,0,0),(b,1,1),
28            (c,0,0),(c,0,1),
29            (c,1,0),(c,1,1)]
30
31  revealAB = reveal [b] [p1,q1,r1]
32
33  result   = upd cards0 showABp
```

Figura 2.20: DEMO - Russian Cards

- card0 : modelo epistêmico inicial do jogo.
- showABp : ação de Anne mostrar a carta p para Bill.
- a linha 33 : Anne mostra sua carta para Bill.

Capítulo 3

Modelo Proposto

A inclusão da operação de atribuição em lógica epistêmica dinâmica é bem recente, sendo encontrada em VAN DITMARSCH *et al.* [1], KOOI [4] e J. VAN BENTHEN e KOOI [5].

O objetivo desse trabalho é formalizar uma extensão do modelo de ação para que seja possível realizar atribuições. A principal diferença para outros trabalhos existentes na área, como o mostrado no capítulo anterior, é a utilização de modelos de ação para realizar as atribuições às proposições, ao invés de utilizar outros mecanismos para realizar as atribuições. Iremos nos restringir a operações de atribuições booleanas, ou seja, uma proposição só pode receber os valores verdadeiro ou falso.

Apresentaremos também uma extensão do DEMO, verificador de modelos para lógica epistêmica dinâmica, para trabalhar com atribuições. No próximo capítulo, iremos apresentar alguns cenários, muito comuns na literatura dessa área, para esse novo modelo de ação.

3.1 Modelo de Ação com Atribuição

Como visto no capítulo anterior, modelo de ação tem uma estrutura que se assemelha com o modelo de Kripke, onde cada ação tem uma pré-condição que precisa ser satisfeita para a ação ser realizada. Nossa proposta é que além de uma pré-condição, cada ação deve ter também uma pós-condição, que seria uma lista de atribuições booleanas.

Exemplo 9 (Modelo de Ação com Atribuição)

Vamos voltar ao exemplo do jogo das cartas, visto no seção 2.4.2, onde, inicialmente, temos a seguinte configuração:

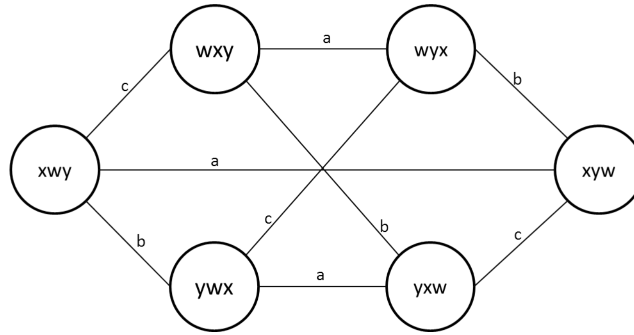

Figura 3.1: Russian Cards

Agora vamos supor que o jogador A troque de carta com o jogador B, como iremos modelar essa troca com modelos de ação? Vamos pensar primeiro em modelos de ação sem atribuição, onde A e B apenas mostram suas cartas um para o outro.

Nesse caso, temos o seguinte modelo de ação:

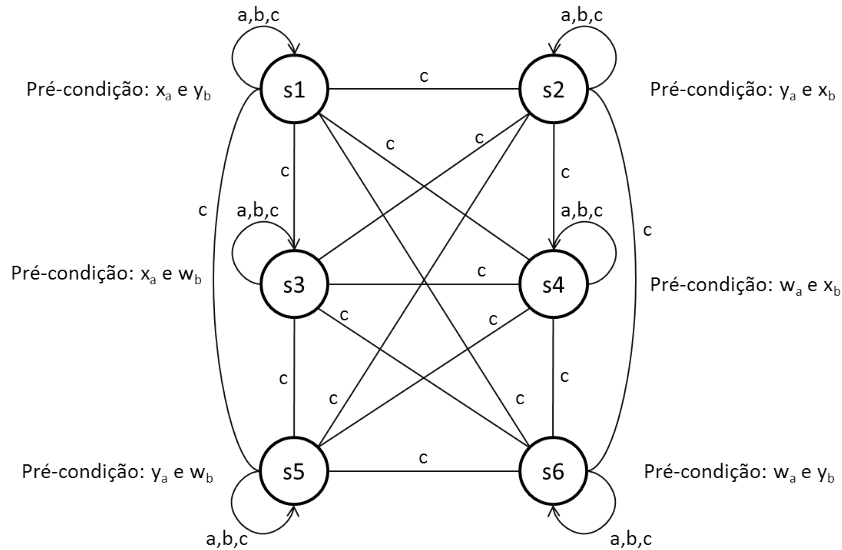

Figura 3.2: Modelo de ação sem atribuição - Russian Cards

Podemos notar que nesse modelo de ação os jogadores A e B não tem dúvidas entre as ações, porém o jogador C não sabe qual ação foi realizada.

Agora vamos pensar que além de mostrar a carta um para o outro, eles trocam as cartas. Suponha que o jogador A tem a carta x e o jogador B tem a carta y, a pré-condição para o jogador A mostrar a carta x para o jogador B é o jogador A ter a carta x, o mesmo ocorre com B e a carta y, a pós-condição de A dar a carta x para B é $x_a = falso$ e $x_b = verdadeiro$ e a pós-condição de B dar a carta y para A é $y_b = falso$ e $y_a = verdadeiro$. Estendendo esse conceito para todos os possíveis estados, temos o seguinte modelo de ação com atribuição:

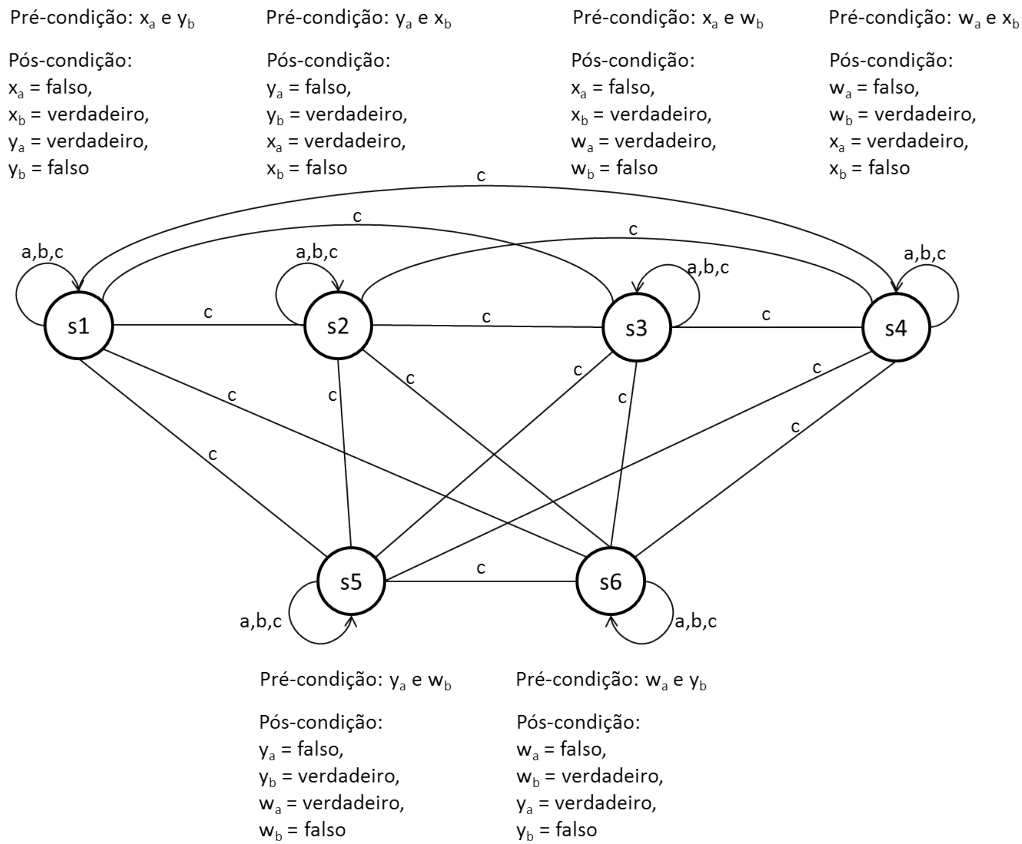

Figura 3.3: Modelo de ação com atribuição - Russian Cards

Esse modelo é igual ao modelo anterior, acrescido das pós-condições.

Faremos o produto cartesiano do modelo epistêmico inicial com o modelo de ação com atribuição. Para facilitar a visualização, iremos omitir os estados gerados pelo produto cartesiano que não atendem às pré-condições, visto que os mesmo serão eliminados de qualquer forma.

Mostraremos esse produto passo a passo, primeiro antes de aplicar a pós-condição e depois de aplicá-la.

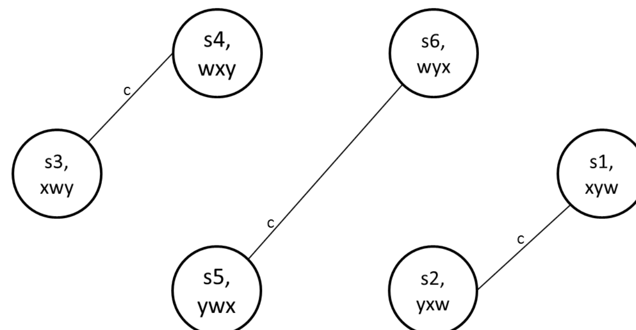

Figura 3.4: Modelo Epistêmico antes de aplicar as pós-condições

Podemos notar que as arestas 'a' e 'b' foram removidas pelo modelo de ação. Isso ocorreu porque A e B não tem mais dúvidas sobre qual é o estado real do sistema.

O jogador C sabe que os jogadores A e B sabem o estado real do sistema, mas ele não sabe qual é esse estado.

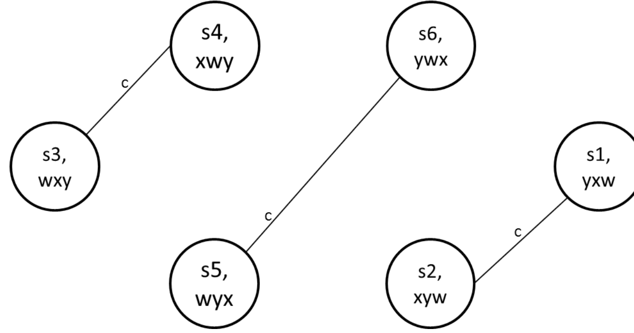

Figura 3.5: Modelo Epistêmico depois de aplicar as pós-condições

Nesse ponto, os jogadores A e B trocaram as cartas, o jogador C sabe que ocorreu uma troca mas não sabe qual foi. Podemos ver que todo o processo ocorre normalmente, como se fosse um modelo de ação normal, apenas no final são feitas as atribuições.

3.1.1 Sintaxe e Semântica

Definição 19 *A linguagem do modelo de ação com atribuição consiste em um conjunto finito* Φ $(p_1, p_2, ..., p_n)$ *de símbolos proposicionais, um conjunto finto* $\mathcal{A}$ *de agentes, os conectivos booleanos* $\neg$ *e* $\wedge$*, o operador* K_a *para cada agente* a *e o operador* $[\mathsf{M}, \mathsf{j}]$*. As fórmulas são definidas como segue:*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid K_a\varphi \mid [\mathsf{M}, \mathsf{j}]\varphi \mid [\alpha]\varphi$$

$$\alpha ::= (\alpha \cup \alpha) \mid (\alpha; \alpha)$$

Definição 20 *Um modelo de ação com atribuição* M *é uma estrutura* $\langle \mathsf{S}, \sim, \mathsf{pre}, \mathsf{pos} \rangle$*, onde:*

- S *é um domínio finito de pontos de ações ou eventos,*
- $\sim_a$ *é a relação de equivalência em* S*,*
- $\mathsf{pre} : \mathsf{S} \mapsto \mathcal{L}$ *é a função de pré-condição que atribui uma pré-condição para cada* $\mathsf{j} \in \mathsf{S}$*,*
- $\mathsf{pos}(\mathsf{j}) = \{(p, x) | \forall p \in \Phi \text{ e } x = V \text{ ou } F\}$.

A linguagem do modelo de ação com atribuição é igual à linguagem do modelo de ação sem atribuição. E continua valendo a premissa que se um agente consegue diferenciar as duas ações, por consequência ele consegue diferenciar os estados resultantes dessas ações.

Definição 21 *Dado um estado epistêmico* $(\mathcal{M}, s)$ *com* $\mathcal{M} = \langle S, \sim_a, V \rangle$ *e um modelo de ação* (M, j) *com* $\mathsf{M} = \langle \mathsf{S}, \sim, \mathsf{pre}, \mathsf{pos} \rangle$*, o resultado da execução* (M, j) *em* $(\mathcal{M}, s)$ *é* $(\mathcal{M} \otimes \mathsf{M}, (s, \mathsf{j}))$ *onde* $\mathcal{M} \otimes \mathsf{M} = \langle S', \sim', V' \rangle$ *tal que:*

1. $S' = \{(s, \mathsf{j})$ *tal que* $s \in S, \mathsf{j} \in \mathsf{S},$ *e* $\mathcal{M}, s \models \mathsf{pre}(\mathsf{j})\}$,
2. $(s, \mathsf{j}) \sim'_a (t, \mathsf{k})$ *sse* $(s \sim_a t$ *e* $\mathsf{j} \sim_a \mathsf{k})$,
3. $V'(p) = \{(s, \mathsf{j}) \mid (p, V) \in \mathsf{pos}(\mathsf{j})\}$.

Definição 22 *Dado um modelo de ação* (M, j) *com* $\mathsf{M} = \langle \mathsf{S}, \sim, \mathsf{pre}, \mathsf{pos} \rangle$*, a definição de fpos*$(\mathsf{j})$ *é dada a seguir:*

1. $L(\mathsf{j}) = \{p \mid (p, V) \in \mathsf{pos}(\mathsf{j})\}$ *, conjunto das proposições verdadeiras em* j.
2. $p_1, ..., p_h \in L(\mathsf{j})$.
3. $q_1, ..., q_m \notin L(\mathsf{j})$.
4. $fpos(\mathsf{j}) = p_1 \wedge ... \wedge p_h \wedge \neg q_1 \wedge ... \wedge \neg q_m$.

Definição 23 *Dado um estado epistêmico* $(\mathcal{M}, s)$ *com* $\mathcal{M} = \langle S, \sim_a, V \rangle$ *e um modelo de ação* (M, j) *com* $\mathsf{M} = \langle \mathsf{S}, \sim, \mathsf{pre}, \mathsf{pos} \rangle$*, a noção de satisfação* $\mathcal{M}, s \models \varphi$ *é definida a seguir:*

$$\begin{array}{lll}
\mathcal{M}, s \models p & sse & s \in V(p) \\
\mathcal{M}, s \models \neg\phi & sse & \mathcal{M}, s \not\models \phi \\
\mathcal{M}, s \models \phi \wedge \psi & sse & \mathcal{M}, s \models \phi \text{ e } \mathcal{M}, s \models \psi \\
\mathcal{M}, s \models K_a\phi & sse & \text{para todo } s' \in S : s \sim_a s' \text{ implica } \mathcal{M}, s' \models \phi \\
\mathcal{M}, s \models [\mathsf{M}, \mathsf{j}]\phi & sse & \mathcal{M}, s \models \mathsf{pre}(\mathsf{j}) \text{ implica } \mathcal{M} \otimes \mathsf{M}, (s, \mathsf{j}) \models (fpos(\mathsf{j}) \rightarrow \phi)
\end{array}$$

Composição de modelos de ação com atribuições

Nos modelos de ação sem atribuições é possível fazer a composição de modelos. Para que isso funcione também nos modelos de ação com atribuições, temos que fazer algumas adaptações. Inicialmente, iremos pensar de maneira informal sobre esse problema, depois formalizaremos a definição de composição de modelos de ação com atribuições.

Para facilitar o entendimento, vamos pensar em um modelo epistêmico (M1) simples, de apenas 3 estados, e 2 modelos de ação (A1 e A2) com apenas 2 ações, como mostrado abaixo:

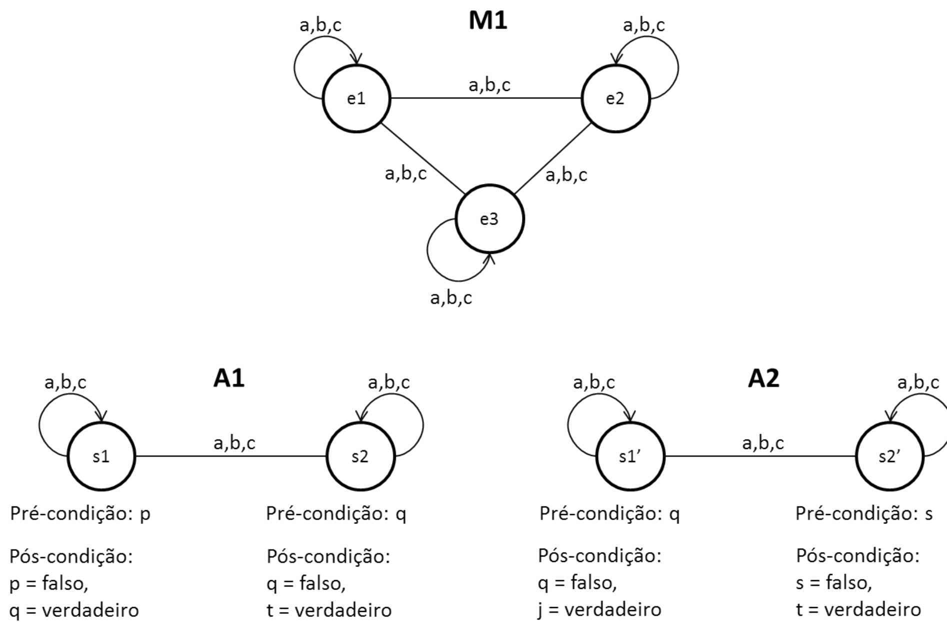

Figura 3.6: Composição de modelos de ação

Uma pergunta que surge ao se pensar em composição de modelos de ação é a seguinte: a ordem da composição importa? Como vimos no capítulo anterior, a ordem não importa se os modelos forem sem atribuição. Isso continua válido para modelos de ação com atribuições? Esquecendo um pouco a composição, a ordem em que os modelos de ação com atribuições são aplicados altera o resultado?

Suponha que apliquemos a ação A1 no modelo M1, gerando o modelo M2, e que depois apliquemos a ação A2 no modelo M2, gerando o modelo M3. Essa sequência pode ser vista na figura abaixo:

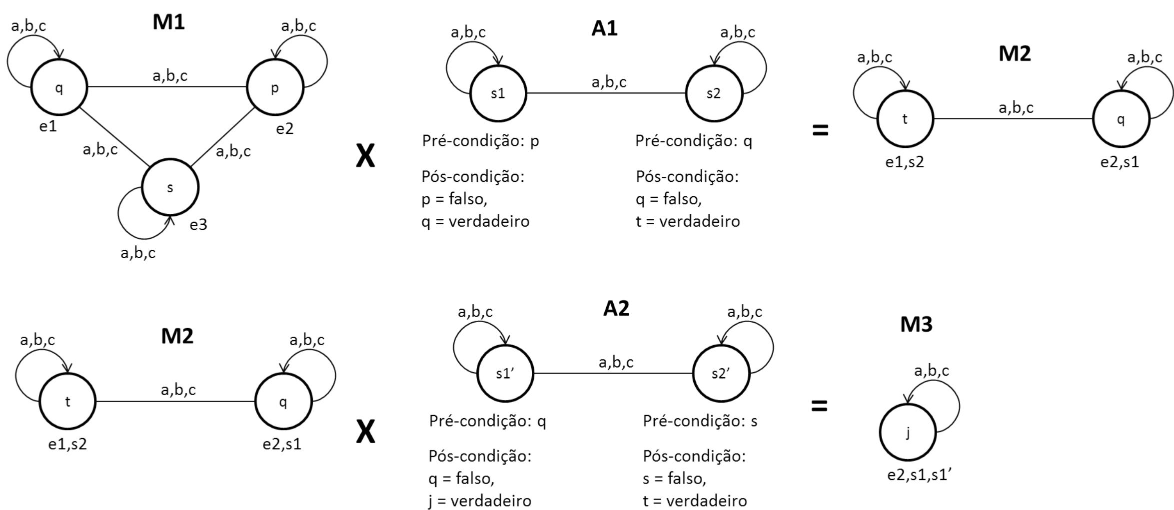

Figura 3.7: Atualização do modelo epistêmico M1 com os modelos de ação A1 e A2

Porém, se aplicarmos a ação A2 no modelo M1, gerando o modelo M2' e depois aplicarmos a ação A1 nesse modelo, temos o seguinte resultado:

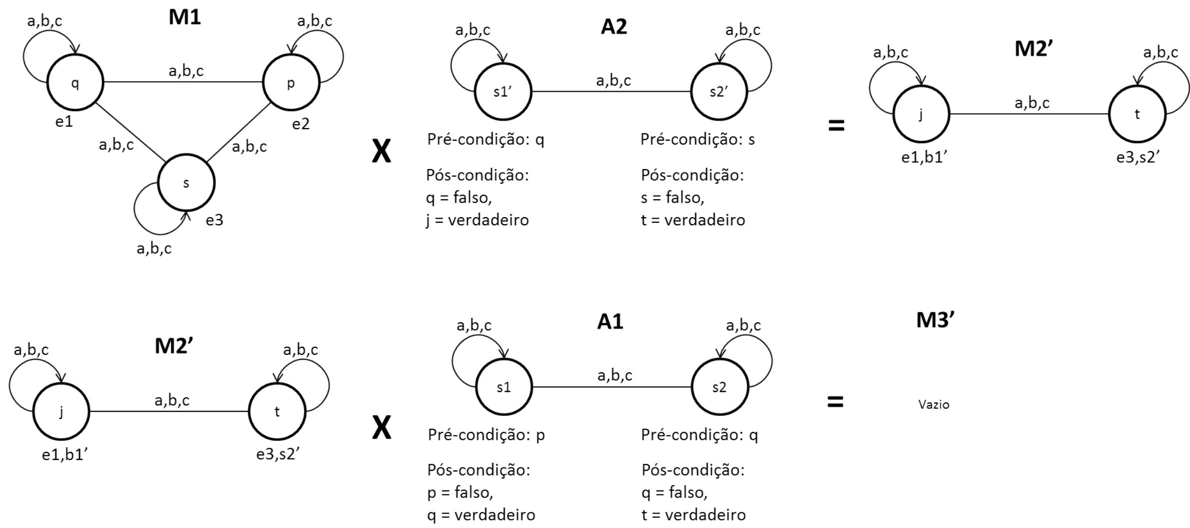

Figura 3.8: Atualização do modelo epistêmico M1 com os modelos de ação A2 e A1

Podemos concluir que, no caso da composição de modelos de ação com atribuições, a ordem importa, pois uma ação pode alterar uma proposição e assim tornar a outra ação, que era incompatível no modelo inicial, compatível.

Inicialmente, vamos adotar uma abordagem ingênua para juntar os dois modelos de ação(A1 e A2, nessa ordem), fazendo simplesmente o produto cartesiano delas e ignorando as pré e pós-condições. Com isso, temos o seguinte modelo de ação resultante:

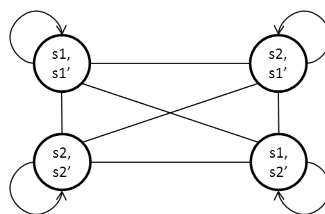

Figura 3.9: Composição dos modelos de ação A1 e A2 antes da eliminação dos estados incompatíveis

Agora, em cada estado(ação) resultante, verificamos se as pós-condições da ação do modelo A1 são compatíveis com as pré-condições da ação do modelo A2. Caso não sejam compatíveis, eliminamos o estado. Com isso, o estado (s2,s1') foi eliminado, resultando no seguinte modelo:

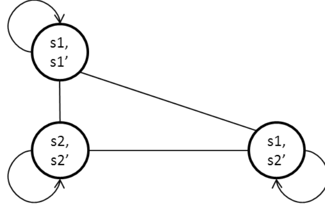

Figura 3.10: Composição dos modelos de ação A1 e A2 após a eliminação dos estados incompatíveis

Como iremos determinar quais são as pré e pós-condições? As pré-condições têm que contemplar as necessidades das duas ações, então uma ideia ingênua seria juntar as duas pré-condições. Como a pós-condição altera valores das proposições, também temos que levá-la em consideração. Por exemplo, se q = verdadeiro é pós-condição de s1 e q pré-condição de s1' ele não precisa estar nas pré-condições do estado s1,s1', pois sempre vai ser verdade visto que s1 torna q verdadeiro. Logo, a pré-condição do novo estado é formada pela junção da pré-condição do estado do modelo A1 com a pré-condição do estado do modelo A2, excluindo as proposições verdadeiras na pós-condição do modelo A1.

No caso das pós-condições é mais simples, como sempre temos atribuições para todas as proposições do modelo nas pós-condições, a pós-condição do estado(ação) resultante será a pós-condição da segunda ação.

Definição 24 *Dado os modelos de ações* (M, j) *com* $\mathsf{M} = \langle \mathsf{S}, \sim, \mathsf{pre}, \mathsf{pos} \rangle$ *e* $(\mathsf{M}', \mathsf{j}')$ *com* $\mathsf{M}' = \langle \mathsf{S}', \sim', \mathsf{pre}', \mathsf{pos}' \rangle$*, a composição deles é o modelo de ação* $(\mathsf{M}; \mathsf{M}', (\mathsf{j}, \mathsf{j}'))$ *com* $\mathsf{M}; \mathsf{M}' = \langle \mathsf{S}'', \sim'', \mathsf{pre}'', \mathsf{pos}'' \rangle$*:*

- $\mathsf{S}'' = \{(\mathsf{j}, \mathsf{j}') \text{ } tal \text{ } que \text{ } \mathsf{j} \in \mathsf{S}, \mathsf{j}' \in \mathsf{S}'\}$
- $(\mathsf{j}, \mathsf{j}') \sim''_a (\mathsf{k}, \mathsf{k}')$ *sse* $(\mathsf{j} \sim_a \mathsf{k}$ *e* $\mathsf{j}' \sim_a \mathsf{k}')$
- $\mathsf{pre}''(\mathsf{j}, \mathsf{j}') = \langle (\mathsf{M}, \mathsf{j}) \rangle \mathsf{pre}'(\mathsf{j}')$
- $\mathsf{pos}''(\mathsf{j}, \mathsf{j}') = \mathsf{pos}'(\mathsf{j}')$

Os estados incompatíveis são eliminados pela pré-condição, ou seja, estados onde a pré-condição é $\perp$.

Usando essa definição, a composição dos modelos de ação A1 e A2 (nessa ordem), gera o seguinte modelo de ação:

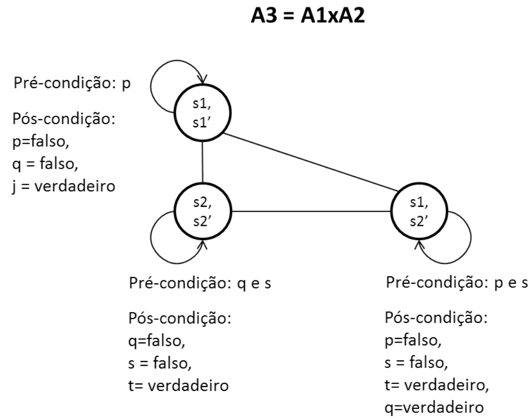

Figura 3.11: Composição de modelos de ação A1 e A2

Aplicando A3 em M1, temos o seguinte modelo epistêmico resultante:

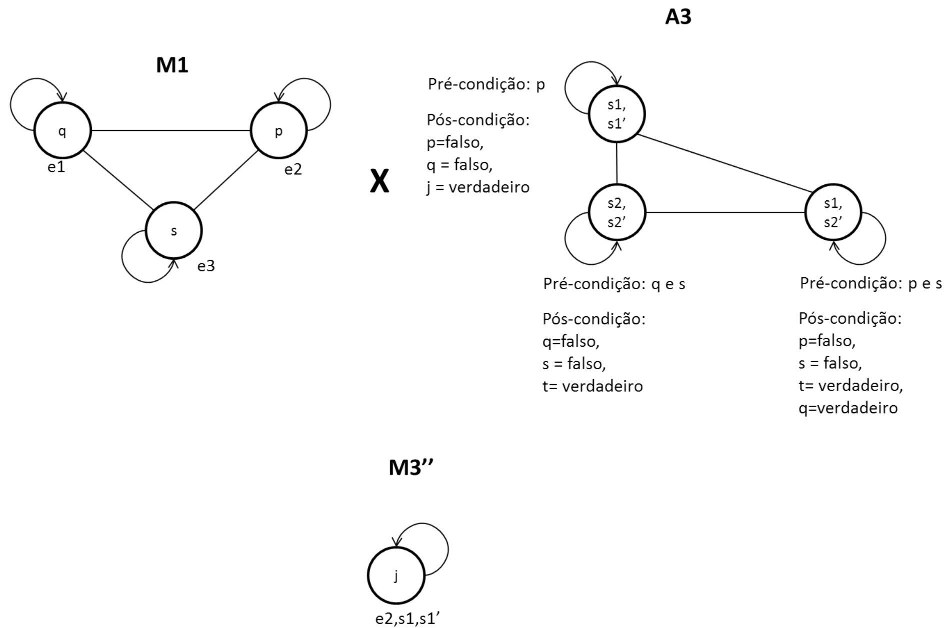

Figura 3.12: Atualização do modelo epistêmico M1 com o modelo de ação A3

Podemos notar que o resultado é o mesmo de quando aplicávamos as ações separadamente.

Sistemas Axiomáticos

Seja :

$$fpos(\mathsf{j}) = p_1 \wedge ... \wedge p_n \wedge \neg p_{n+1} \wedge ... \wedge \neg p_m.$$

Axiomas

1. $[\mathsf{M},\mathsf{j}]p \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow (fpos(\mathsf{j}) \rightarrow p))$,

2. $[\mathsf{M},\mathsf{j}]\neg\phi \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow \neg[\mathsf{M},\mathsf{j}]\phi)$,

3. $[\mathsf{M},\mathsf{j}](\phi \wedge \psi) \leftrightarrow ([\mathsf{M},\mathsf{j}]\phi \wedge [\mathsf{M},\mathsf{j}]\psi)$,

4. $[\mathsf{M},\mathsf{j}]K_a\phi \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow \bigwedge_{\mathsf{j}\sim_a \mathsf{k}} K_a[\mathsf{M},\mathsf{k}]\phi)$,

5. $[[\mathsf{M},\mathsf{j}] \cup [\mathsf{M}',\mathsf{j}']]\phi \leftrightarrow [\mathsf{M},\mathsf{j}]\phi \wedge [\mathsf{M}',\mathsf{j}']\phi$,

6. $[\mathsf{M},\mathsf{j}][\mathsf{M}',\mathsf{j}']\phi \leftrightarrow [(\mathsf{M},\mathsf{j});(\mathsf{M}',\mathsf{j}')]\phi$. (Composição de modelos de ações)

Correção

Para provar a correção, precisamos mostrar que nossos axiomas são válidos. A adição da propriedade de pós-condição altera apenas os axiomas 1 e 5. Iremos provar que os mesmos continuam valendo com a pós-condição.

Lema 2 $[\mathsf{M},\mathsf{j}]p \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$ *é válida.*

Prova 1 *Queremos provar* $[\mathsf{M},\mathsf{j}]p \leftrightarrow (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$.
Suponha, por absurdo, que a fórmula não seja verdadeira, logo temos 2 situações para essa fórmula ser falsa:

1. $\mathcal{M}, w \models [\mathsf{M},\mathsf{j}]p$ *e* $\mathcal{M}, w \not\models (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$
2. $\mathcal{M}, w \not\models [\mathsf{M},\mathsf{j}]p$ *e* $\mathcal{M}, w \models (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$

Da definição 23, temos que:
17.1 $\mathcal{M}, s \models \neg\phi$ *sse* $\mathcal{M}, s \not\models \phi$
17.5 $\mathcal{M}, s \models [\mathsf{M},\mathsf{j}]\phi$ *sse* $\mathcal{M}, s \models \mathsf{pre}(\mathsf{j})$ *implica* $\mathcal{M} \otimes \mathsf{M}, (s,\mathsf{j}) \models (\mathit{fpos}(\mathsf{j}) \rightarrow \phi)$

Provando a primeira parte:
Seja:
* $\mathcal{M}, w \models [\mathsf{M},\mathsf{j}]p$
* $\mathcal{M}, w \not\models (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$
Sabemos que para uma fórmula do tipo $A \rightarrow B$ ser falsa, A tem que ser verdadeiro e B tem que ser falso.
Se $\mathcal{M}, w \not\models (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$ *é verdadeiro, então* $(\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$ *é falso. Como dito acima, para* $(\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$ *ser falso* $\mathsf{pre}(\mathsf{j})$ *tem que ser verdadeiro e* $(\mathit{fpos}(\mathsf{j}) \rightarrow p)$ *tem que ser falso.*

Se $\mathcal{M}, w \models [\mathsf{M},\mathsf{j}]p$, *da definição 23, temos que* $pre(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p)$ *é verdadeiro. Absurdo pois para que o outro lado da fórmula seja válido,* $pre(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p)$ *tem que ser falso.*

Provando a segunda parte:

Seja:
* $\mathcal{M}, w \not\models [\mathsf{M},\mathsf{j}]p$
* $\mathcal{M}, w \models (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$

Se $\mathcal{M}, w \not\models [\mathsf{M},\mathsf{j}]p$*, utilizando 17.1 e 17.2, temos que* $\neg(\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$*, para que essa fórmula seja verdadeira,* $\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p)$ *deve ser falso. Como dito anteriormente para* $A \rightarrow B$ *ser falso,* A *é verdadeiro e* B *é falso, ou seja,* $pre(\mathsf{j})$ *é verdadeiro e* $(\mathit{fpos}(\mathsf{j}) \rightarrow p)$ *é falso.*

Para $\mathcal{M}, w \models (\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p))$ *ser verdadeiro,* $(\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p)$ *tem que ser verdadeiro. Absurdo, pois para a outra fórmula ser verdadeira,* $(\mathsf{pre}(\mathsf{j}) \rightarrow (\mathit{fpos}(\mathsf{j}) \rightarrow p)$ *tem que ser falso.*

Lema 3 $[\mathsf{M},\mathsf{j}][\mathsf{M}',\mathsf{j}']\phi \leftrightarrow [(\mathsf{M},\mathsf{j});(\mathsf{M}',\mathsf{j}')]\phi$ *é válida.*

Prova 2 *Queremos provar* $[\mathsf{M},\mathsf{j}][\mathsf{M}',\mathsf{j}']\phi \leftrightarrow [(\mathsf{M},\mathsf{j});(\mathsf{M}',\mathsf{j}')]\phi$.
Prova baseada na prova de 6.9 VAN DITMARSCH et al. *[3].*

Seja $(\mathcal{M}, t)$ *um modelo arbitrário. Temos que mostrar que* $\mathcal{M}, t \models [\mathsf{M},\mathsf{j}][\mathsf{M}',\mathsf{j}']\phi$ *sse* $\mathcal{M}, t \models [(\mathsf{M},\mathsf{j});(\mathsf{M}',\mathsf{j}')]\phi$*. Para isso, é suficiente mostrar que* $\mathcal{M} \otimes (\mathsf{M};\mathsf{M}')$ *e* $((\mathcal{M} \otimes \mathsf{M}) \otimes \mathsf{M}')$ *são isomórficos. Logo, temos que mostrar que os estados, as arestas e as funções de valoração do modelo final são iguais.*

Vamos mostrar que os modelos finais tem os mesmos estados. A única propriedade do modelo de ação que elimina estados é a pré-condição, logo, temos que mostrar que as pré-condições são iguais. Seja $(t, (\mathsf{j},\mathsf{j}')) \in D(\mathcal{M} \otimes (\mathsf{M};\mathsf{M}'))$ [1]*, então temos que* $\mathcal{M}, t \models \mathsf{pre}''(\mathsf{j},\mathsf{j}')$*, ou seja,* $\mathcal{M}, t \models \langle(\mathsf{M},\mathsf{j})\rangle\mathsf{pre}'(\mathsf{j}')$*. Essa última é equivalente a* $\mathcal{M}, t \models \mathsf{pre}(\mathsf{j}) \wedge [\mathsf{M},\mathsf{j}]\mathsf{pre}'(\mathsf{j}')$ *ou seja* $\mathcal{M}, t \models \mathsf{pre}(\mathsf{j}) \wedge \mathcal{M}, t \models \mathsf{pre}(\mathsf{j})[\mathsf{M},\mathsf{j}]\mathsf{pre}'(\mathsf{j}')$*. De* $\mathcal{M}, t \models \mathsf{pre}(\mathsf{j})$ *temos que* $(t, \mathsf{j}) \in D(\mathcal{M} \otimes \mathsf{M})$ *e de* $\mathcal{M}, t \models \mathsf{pre}(\mathsf{j})[\mathsf{M},\mathsf{j}]\mathsf{pre}'(\mathsf{j}')$ *temos que* $((t,\mathsf{j}),\mathsf{j}') \in D((\mathcal{M} \otimes \mathsf{M}) \otimes \mathsf{M})$*. Esse argumento vale para ambos os lados.*

Considerando agora a acessibilidade temos que $(t, (\mathsf{j},\mathsf{j}')) \sim_a (t_1, (\mathsf{j}_1,\mathsf{j}_1'))$ *sse* $(t \sim_a t_1$ *e* $\mathsf{j} \sim_a \mathsf{j}_1$ *e* $\mathsf{j}' \sim_a \mathsf{j}_1')$ *sse* $((t,\mathsf{j}),\mathsf{j}') \sim_a ((t_1,\mathsf{j}_1),\mathsf{j}_1')$.

A função de valoração para a tripla $(t, (\mathsf{j},\mathsf{j}'))$ *corresponde a* $V''(p) = \{(t, (\mathsf{j},\mathsf{j}')) \mid (p, V) \in \mathsf{pos}(\mathsf{j},\mathsf{j}')\}$*, por 24 sabemos que* $\mathsf{pos}(\mathsf{j},\mathsf{j}') = \mathsf{pos}(\mathsf{j}')$*, logo* $V''(p) = \{(t, (\mathsf{j},\mathsf{j}')) \mid (p, V) \in \mathsf{pos}(\mathsf{j}')\}$*. A função de valoração para a tripla* $((t,\mathsf{j}),\mathsf{j}')$ *corresponde a* $V'''(p) = \{((t,\mathsf{j}),\mathsf{j}') \mid (p, V) \in \mathsf{pos}(\mathsf{j}')\}$*. Logo podemos ver que em ambos os lados a valoração é definida por* $\mathsf{pos}(\mathsf{j}')$.

[1] $D(\mathcal{M})$ (domínio de $\mathcal{M}$) é o conjunto de estados de um modelo $\mathcal{M}$, também conhecido como S.

Completude

A completude segue direto dos axiomas, definindo-se uma tradução/redução do modelo de ação com atribuição para o S5. Essa prova foi baseada na prova da seção 7.6 do VAN DITMARSCH *et al.* [3].

A prova de completude para modelos de ação(AM) é parecida com a prova de completude para anúncios públicos (PA). Provamos uma tradução das fórmulas que contêm modalidades de ação para fórmulas que não contêm essas modalidades. Essa tradução segue os axiomas do sistema de prova. Os axiomas que descrevem a interação dos modelos de ação com outros operadores lógicos são bem similares aos axiomas do PA, que descrevem a interação dos anúncios públicos e os outros operadores lógicos. De fato podemos notar que esses axiomas do PA são casos especiais dos AM.

Definição 25 *A tradução t:* $\mathcal{L}_{K\otimes} \rightarrow \mathcal{L}_K$ *é definida a seguir:*

$$
\begin{array}{lcl}
t(p) & = & p \\
t(\neg\varphi) & = & \neg t(\varphi) \\
t(\varphi \wedge \psi) & = & t(\varphi) \wedge t(\psi) \\
t(K_a\varphi) & = & K_a t(\varphi) \\
t([\mathsf{M},\mathsf{j}]p) & = & t(\mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p)) \\
t([\mathsf{M},\mathsf{j}]\neg\varphi) & = & t(\mathsf{pre}(\mathsf{j}) \rightarrow \neg[\mathsf{M},\mathsf{j}]\varphi) \\
t([\mathsf{M},\mathsf{j}](\varphi \wedge \psi)) & = & t([\mathsf{M},\mathsf{j}]\varphi \wedge [\mathsf{M},\mathsf{j}]\psi) \\
t([\mathsf{M},\mathsf{j}]K_a\varphi) & = & t(\mathsf{pre}(\mathsf{j}) \rightarrow K_a[\mathsf{M},\mathsf{j}]\varphi) \\
t([\mathsf{M},\mathsf{j}][\mathsf{M}',\mathsf{j}']\varphi) & = & t([\mathsf{M},\mathsf{j};\mathsf{M}',\mathsf{j}']\mathcal{X}) \\
t([\alpha \cup \alpha']\varphi) & = & t([\alpha]\varphi) \wedge t([\alpha']\varphi)
\end{array}
$$

Pela robustez do sistema de prova essa tradução preserva o significado de uma fórmula. Resta mostrar que cada fórmula é, comprovadamente, equivalente à sua tradução. Para mostrar isso precisaremos de condição diferente na linguagem para que possamos aplicar a hipótese de indução em fórmulas que não sejam subfórmulas das fórmulas que temos. [3] estende a medida de complexidade para $\mathcal{L}_{K[]}$ para a linguagem com modelos de ação e mostra que ela tem as propriedades desejadas. Ele também atribui complexidade a modelos de ação, fazendo este ser o máximo da complexidade das pré-condições do modelo de ação.

Definição 26 *A complexidade* $c : \mathcal{L}_{K\otimes} \rightarrow \mathcal{N}$ *é definida a seguir:*

$$
\begin{array}{lcl}
c(p) & = & 1 \\
c(\neg\varphi) & = & 1 + c(\varphi) \\
c(\varphi \wedge \psi) & = & 1 + max(c(\varphi), c(\psi)) \\
c(K_a\varphi) & = & 1 + c(\varphi) \\
c([\alpha]\varphi) & = & (6 + c(\alpha)) * c(\varphi) \\
c(\mathsf{M}, \mathsf{j}) & = & max\{c(\mathsf{pre}(t) \ + \ c(pos(t)) \mid t \in \mathsf{M}\} \\
c(\alpha \cup \alpha') & = & 1 + max(c(\alpha), c(\alpha'))
\end{array}
$$

Segundo [3] podemos, seguramente, usar o máximo da complexidade das pré-condições no modelo de ação, visto que modelos de ação são finitos. Portanto a complexidade de uma fórmula ou de um modelo de ação será sempre um número natural. O número 4 aparece na cláusula para modelo de ação pois dá as seguintes propriedades.

Lema 4 *Para todo φ, ψ e $\mathcal{X}$:*

1. $c(\psi) \geq c(\varphi)$ *se* $\varphi \in Sub(\psi)$,
2. $c([\mathsf{M}, \mathsf{j}]p) > c(\mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p))$,
3. $c([\mathsf{M}, \mathsf{j}]\neg\varphi) > c(\mathsf{pre}(\mathsf{j}) \rightarrow \neg[\mathsf{M}, \mathsf{j}]\varphi)$,
4. $c([\mathsf{M}, \mathsf{j}](\varphi \wedge \psi)) > c([\mathsf{M}, \mathsf{j}]\varphi \wedge [\mathsf{M}, \mathsf{j}]\psi)$,
5. $c([\mathsf{M}, \mathsf{j}]K_a\varphi) > c(\mathsf{pre}(\mathsf{j}) \rightarrow K_a[\mathsf{M}, \mathsf{j}]\varphi)$,
6. $c([\mathsf{M}, \mathsf{j}][\mathsf{M}', \mathsf{j}']\varphi) > c([\mathsf{M}, \mathsf{j}; \mathsf{M}', \mathsf{j}']\varphi)$,
7. $c([\alpha \cup \alpha']\varphi) > c([\alpha]\varphi \wedge [\alpha']\varphi)$.

Prova 3 *A prova dos itens 1, 4 e 7 é direta da definição 26.*

2. *Desenvolvendo-se os dois lados e observado-se que* $c(\phi \rightarrow \psi) = 2 + c(\phi) + c(\psi)$

   $c([\mathsf{M}, \mathsf{j}]p) = (6 + c(\mathsf{M}, \mathsf{j})) * c(p) = 6 + max\{c(\mathsf{pre}(t) \ + \ c(pos(t)) \mid t \in \mathsf{M}\}$
   $c(\mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p)) = 2 + c(\mathsf{pre}(\mathsf{j})) + 2 + c(pos(\mathsf{j})) + 1 =$
   $= 5 + c(\mathsf{pre}(\mathsf{j})) + c(pos(\mathsf{j}))$

   Portanto, $c([\mathsf{M}, \mathsf{j}]p) > c(\mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p))$

Agora pode-se provar o lema que diz que toda fórmula é equivalente a sua tradução.

Lema 5 *Para todas as fórmulas $\varphi \in \mathcal{L}_{K\otimes}$vale :*

$$\vdash \varphi \leftrightarrow t(\varphi)$$

Prova 4 *Por indução em* $c(\varphi)$.
Hipótese de Indução: *Suponha que*

$$\vdash \varphi \leftrightarrow t(\varphi)$$

vale para fórmulas φ *onde* $c(\varphi) < n$

1. $\varphi = p$ *direto do fato que* $\vdash p \leftrightarrow p$;

2. $\varphi = \neg\psi,\ \psi_1 \wedge \psi_2\ K_a\psi$: *direto da H. I.;*

3. $\varphi = [\mathsf{M}, \mathsf{j}]p$:

   $t([\mathsf{M}, \mathsf{j}]p) = t(\mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p))$, *pela H. I.*

   $\vdash t(\mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p)) \leftrightarrow \mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p)$, *mas*

   $\vdash \mathsf{pre}(\mathsf{j}) \rightarrow (pos(\mathsf{j}) \rightarrow p) \leftrightarrow [\mathsf{M}, \mathsf{j}]p$, *e portanto*

   $\vdash t([\mathsf{M}, \mathsf{j}]p) \leftrightarrow [\mathsf{M}, \mathsf{j}]p$

4. $\varphi = [\mathsf{M}, \mathsf{j}]\neg\varphi$: *segue análogo ao caso 3. usando-se axioma 2.*

5. $\varphi = [\mathsf{M}, \mathsf{j}](\varphi \wedge \psi)$: *segue análogo ao caso 3. usando-se axioma 3.*

6. $\varphi = [\mathsf{M}, \mathsf{j}]K_a\varphi$: *segue análogo ao caso 3. usando-se axioma 4.*

7. $\varphi = [\mathsf{M}, \mathsf{j}][\mathsf{M}', \mathsf{j}']\varphi$: *segue análogo ao caso 3. usando-se axioma 5.*

8. $\varphi = [\alpha \cup \alpha']\varphi$: *segue análogo ao caso 3. usando-se axioma 6.*

A completude segue automaticamente.

Definição 27 *Completude*

Para todo $\varphi \in \mathcal{L}_{K\otimes}$

$$\models \varphi \text{ implica } \vdash \varphi$$

Prova 5 *Suponha* $\models \varphi$. *Pelo lema 5 nós temos que* $\vdash \varphi \leftrightarrow t(\varphi)$, *pela Correção nós podemos concluir que* $\models t(\varphi)$. *Mas como* $t(\varphi)$ *não possui nenhuma ação, ela é uma fórmula de* **S5** *e como* **S5** *é completo nós temos que* $\vdash_{S5} t(\varphi)$, *mas como* **S5** *está contido em* **AMa**, *nós temos que* $\vdash_{Ama} t(\varphi)$.

3.2 Extensão do DEMO

Na seção anterior, apresentamos um novo modelo de ação. Para que seja possível verificar modelos que utilizem esse novo modelo de ação tivemos, que criar novas funcionalidades no DEMO, adaptando-o para funcionar com a nova propriedade de pós-condição. Essas novas funcionalidades foram implementadas em Haskell (assim como o DEMO) e sua implementação pode ser vista no apêndice. Nessa seção, iremos descrever essas novas funcionalidades.

Segue abaixo a lista das funções implementadas e suas descrições:

- updTrueAssignment (ModeloEpistêmico) (Proposição) - Dado um modelo epistêmico e uma proposição, essa função atualiza o valor dessa proposição para verdadeiro em todos os estados do modelo. A função retorna o modelo resultando dessa atualização.

- updFalseAssignment (ModeloEpistêmico) (Proposição) - Dado um modelo epistêmico e uma proposição, essa função atualiza o valor dessa proposição para falso em todos os estados do modelo. A função retorna o modelo resultando dessa atualização.

- publicAssignment (ModeloEpistêmico) (tipo) (Proposição) - Dado um modelo epistêmico, um tipo de atribuição ('+' para verdadeiro e '-' para falso) e uma proposição, essa função atualiza o valor dessa proposição para o tipo escolhido em todos os estados do modelo. A função retorna o modelo resultando dessa atualização.

- publicAssign (ModeloEpistêmico) (condição) (Proposição) - Dado um modelo epistêmico, uma condição e uma proposição, essa função atualiza o valor dessa proposição para o resultado da condição em todos os estados do modelo. Ex. publicAssign (M) (K a p) (Q 0) , se K p for verdadeiro no modelo M, a proposição q será atualizada com o valor verdadeiro em todos os estados do modelo, caso contrário, será atualizado com o valor falso. A função retorna o modelo resultando dessa atualização.

- updWA (ModeloEpistêmico) (ModeloDeAçãoComAtribuição) - Dado um modelo epistêmico e um modelo de ação com atribuição fazemos o produto cartesiano dos dois modelos, respeitando sempre as pré e pós-condições. A função retorna o modelo resultando dessa atualização.[Atualmente, essa função ainda contêm alguns erros de execução, devido a falta de habilidade com a linguagem Haskell.]

Outras funções foram criadas para realizar algumas operações internas, a implementação dessas funções pode ser encontrada no apêndice. Exemplos utilizando algumas dessas novas funcionalidades do DEMO são descritos no próximo capítulo.

Capítulo 4

Aplicações do modelo proposto

Nesse capítulo, apresentaremos alguns exemplos de aplicações para o modelo proposto no capítulo anterior.

4.1 Jogo das crianças sujas

Vamos voltar ao jogo das crianças sujas, apresentado no capítulo 2.

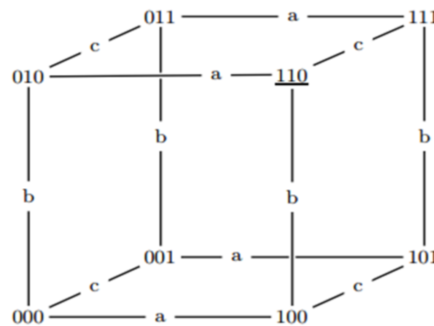

Figura 4.1: Crianças sujas

A descrição do jogo é a mesma feita na seção 2.3.1.

A ação do pai de jogar o balde de água na Anne é representada pelo seguinte modelo de ação.

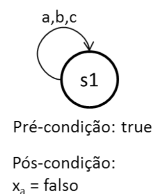

Figura 4.2: Atualização: Pai joga um balde em Anne

Como a pré-condição dessa ação é TRUE, ela será aplicada em todos os estados do modelo, e como a pós-condição dela é que a testa da Anne está limpa, todos os estados ficaram com testa de Anne limpa, como na imagem abaixo.

Para representar essa ação no DEMO, podemos escrever de uma das seguintes formas:

- updFalseAssignment (mm1) (P 0)
- publicAssignment (mm1) ('-') (P 0)
- publicAssign (mm1) (false) (P 0)

onde mm1 é o modelo inicial e P 0 é a proposição que diz se a testa da Anne está suja.

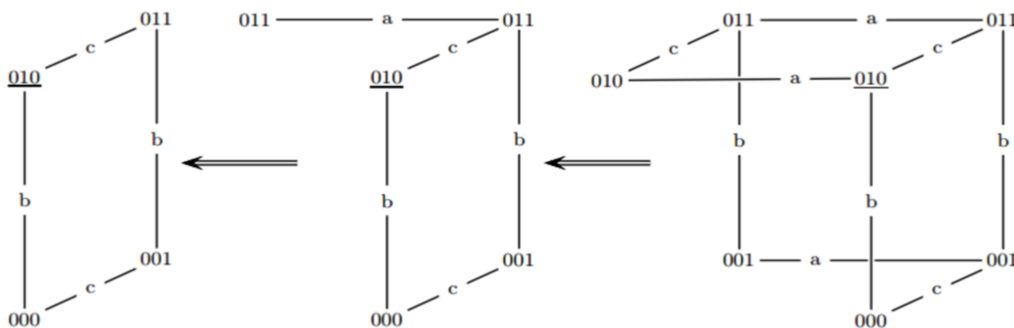

Figura 4.3: Crianças sujas, após a atualização

Como podemos notar, o jogo se comporta exatamente igual ao apresentado na seção 2.3.1.

4.2 Carta no baú

Esse jogo é parecido com o jogo das cartas, descrito anteriormente, porém com 3 cartas(w,x,y), 2 jogadores e 1 baú. Nesse jogo temos 2 crianças, Anne e Bill, cada um com uma carta e um baú, inicialmente fechado, com a outra carta. Igualmente ao jogo das cartas anterior, os jogadores têm que descobrir qual a distribuição das cartas no jogo. Os jogadores podem executar a ação de espiar o baú, porém essa ação só é efetiva se o baú estiver aberto, caso contrário eles continuam sem conseguir ver dentro do baú. De tempos em tempos, é executada, por uma terceira pessoa, a ação de abrir ou fechar o baú. Anne percebe quando Bill espia o baú e Bill percebe quando Anne espia o baú, porém não conseguem saber qual a carta que está no baú.

O modelo epistêmico inicial do jogo é representado pela figura abaixo.

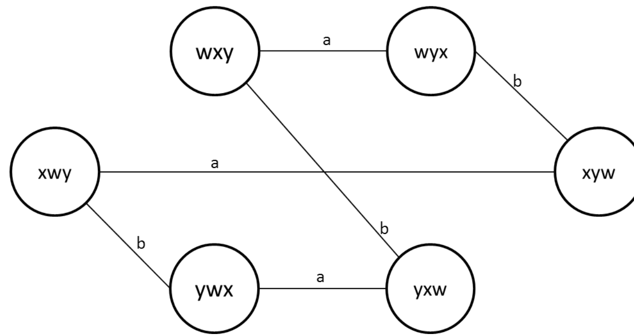

Figura 4.4: Jogo carta no baú

A ação de abrir e fechar o baú é representada pelo modelo de ação a seguir. É apenas uma ação de troca de valores.

Existe a aresta ligando os dois estados pois os jogadores não sabem se o baú está fechado ou aberto e por isso não sabem qual ação (abrir ou fechar o baú) será executada.

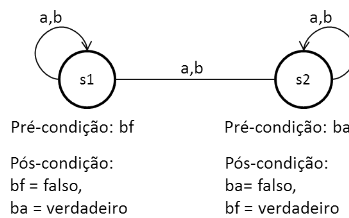

Figura 4.5: Ação de abrir/fechar o baú

As ações espiarA e espiarB são parecidas, a única diferença é que quando A espia, B não sabe qual carta A viu e, quando B espia, A não sabe qual carta B viu, porém eles sabem se o outro conseguiu abrir o baú (ver alguma carta) ou não.

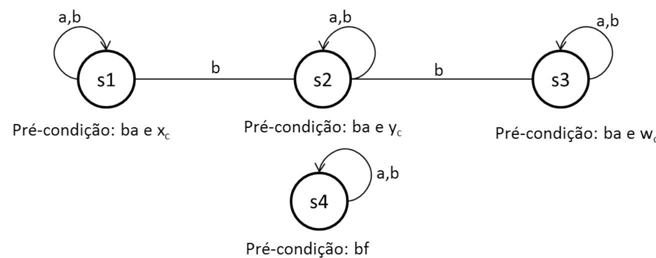

Figura 4.6: Ação de Anne espiar o baú

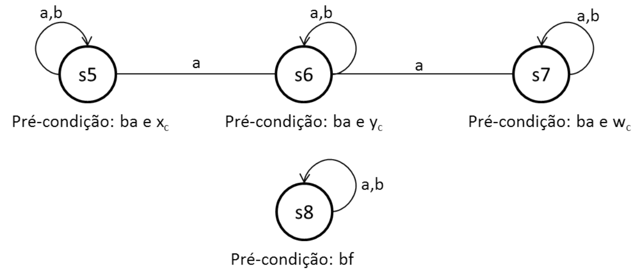

Figura 4.7: Ação de Bill espiar o baú

4.3 Ações Possíveis

Com esse novo modelo de ação podemos criar ações como "ligar/desligar a luz" , "abrir/fechar a porta", que são utilizadas em J. VAN BENTHEN e KOOI [5]. Mas diferentemente de [5], no nosso modelo só podemos fazer atribuições booleanas, logo o modelo de ação fica um pouco diferente.

Abaixo, a representação gráfica do modelo de ação que leva em conta se a luz está desligada para ligar e vice-versa.

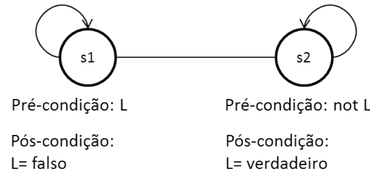

Figura 4.8: Modelo de ação de ligar/desligar a luz

Podemos modelar também duas ações diferentes que, independentemente do estado atual da luz, vai para o estado de desligada ou vai para o estado de ligada.

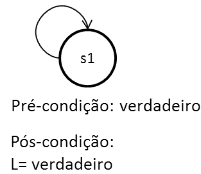

Figura 4.9: Modelo de ação de ligar a luz

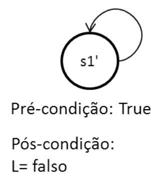

Figura 4.10: Modelo de ação de desligar a luz

A diferença entre o primeiro modelo de ação, onde tínhamos as duas ações de ligar e desligar no mesmo modelo, e esses outros dois modelos de açã,o é que no primeiro podemos modelar coisas do tipo: “O jogador 1 consegue ver quando a luz foi ligada, porém o jogador 2 não consegue”. Já nos outros dois modelos de ação, as ações podem ser classificadas como atribuições públicas, logo todos os jogadores do sistema vão saber que a luz está ligada depois que for executada a ação de ligar a luz, assim como todos vão saber que a luz está desligada quando for executada a ação de apagar a luz, e também todos vão saber que todos sabem que a luz está apagada.

A representação gráfica do modelo de ação de “abrir/fechar a porta” é semelhante aos modelos de ação mostrados acima para as ações de “ligar/desligar a luz”.

Capítulo 5

Conclusões

Nessa dissertação, discutimos sobre atribuições em lógicas epistêmicas dinâmicas. Para tal, redefinimos o *framework* do modelo de ação, encontrado na lógica epistêmica dinâmica, para contemplar propriedades que permitissem que fossem feitas, em cada ação, atribuições booleanas às proposições. Chamamos essa propriedade de pós-condição de uma ação, visto que essa propriedade era a última coisa da ação a ser executada.

Apresentamos no capítulo 2 um método já existente para atribuir valores às proposições em lógica epistêmica dinâmica, porém, como mostrado no capítulo 3, essa abordagem não utilizava o *framework* do modelo de ação para realizar essa atribuição, que é amplamente utilizada na lógica epistêmica dinâmica e, inclusive, é utilizado pelo DEMO (verificador de modelos epistêmicos).

Mostramos, no capítulo 3, que todas as definições, com suas respectivas adaptações, de modelo de ação da lógica epistêmica dinâmica também são válidas para o novo modelo de ação com atribuições, porém, algumas das definições contêm ressalvas. Por exemplo, na questão da composição de modelos de ação com atribuição a ordem em que é feita a composição importa, visto que cada ação modifica os valores das proposições, o que não acontecia nos modelos de ação sem atribuição, pois os valores nunca eram alterados. Mostramos também que a pesquisa dessa dissertação resultou na implementação de novas funcionalidades ao DEMO, verificador de modelos epistêmicos. Tais funcionalidades visam permitir que sejam verificados, através do DEMO, modelos de ações com atribuições. No capítulo 4, apresentamos alguns cenários onde podemos aplicar esse novo *framework* do modelo de ação, mostrando como é intuitivo utilizar esse *framework*.

Uma restrição imposta nessa dissertação, para o modelo de ação com atribuições, é que as atribuições seriam apenas booleanas. Essa limitação foi colocada para facilitar a formalização da composição de modelos de ação e a implementação das novas funcionalidades do DEMO. Diante disso, podemos indicar como direção dos trabalhos futuros a remoção dessa limitação para que possa ser feito qualquer tipo

de atribuição, inclusive de fórmulas que utilizem os conectivos and e or. Podemos também gerar cenários mais reais para os modelos de ação com atribuição.

Outra direção para trabalhos futuros é focar na implementação de novas funcionalidades do DEMO e na correção da função updWA.

Referências Bibliográficas

[1] VAN DITMARSCH, H., VAN DER HOEK, W., KOOI, B. “Dynamic Epistemic Logic with Assignment”, *AAMAS*, v. 4, n. 1, pp. 141–148, jul. 2005.

[2] HINTIKKA, J. *Knowledge and Belief*. Ithaca, N.Y, Cornell University Press, 1962.

[3] VAN DITMARSCH, H., VAN DER HOEK, W., KOOI, B. *Dynamic Epistemic Logic*. Springer, 2008.

[4] KOOI, B. “Expressivity and completeness for public update logic via reduction axioms”, *Journal of Applied Non-Classical Logics*, v. 17, n. 2, pp. 231–253, 2007.

[5] J. VAN BENTHEN, J. V. E., KOOI, B. “Logics of communication and change”, *Information and Compatation*, v. 204, n. 11, pp. 1620–1662, 2006.

[6] FAGIN, R., HALPERN, J. Y., MOSES, Y., et al. *Reasoning About Knowledge*. The MIT Press, 1995.

[7] DELGADO, C. A. D. M. *Modelagem e verificação de propriedades epistêmicas em sistemas multi-agentes*. Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2007.

[8] H. VAN DITMARSCH, A. H., DE LIMA, T. “Public announcements, public assignments and the complexity of their logic”, *Journal of Applied Non-Classical Logics*, v. 22, n. 3, pp. 249–273, 2012.

[9] VAN EIJCK, J. “DEMO - A Demo of Epistemic Modelling”, nov. 2006.

[10] H. VAN DITMARSCH, W. V. D. H., RUAN, J. “Connection Dynamic Epistemic and Temporal Epistemic Logics”, *Journal of the IGPL*, v. 21, n. 3, pp. 380–403, 2013.

[11] B. RENNE, J. S., YAP, A. “Dynamic Epistemic Temporal Logic”, *Second International Workshop, LORI*, v. 2, pp. 263–277, 2009.

[12] SIETSMA, F., VAN EIJCK, J. “Model Checking for Dynamic Epistemic Logic with Factual Change”. CWI, Amsterdam, 2007.

[13] KOOI, B. “Probabilistic Dynamic Epistemic Logic”, *Journal of Logic, Language and Information*, v. 12, n. 4, pp. 381–408, 2003.

[14] SACK, J. “Extending Probabilistic Dynamic Epistemic Logic”, *Synthese*, v. 169, n. 2, pp. 241–257, 2009.

[15] H. VAN DITMARSCH, J. R., VAN DER HOEK, W. “Model checking dynamic epistemics in branching time”. In: *Proceedings of Workshop on Formal Approaches to Multi-Agent Systems*, Durham, UK, 2007.

[16] LENZEN, W. “Recent work in epistemic logic”, *Acta Philosophica Fennica*, v. 30, pp. 1–219, 1978.

[17] AUMANN, R., BRANDENBURGER, A. “Epistemic conditions for Nash equilibrium”, *Econometrica*, v. 63, n. 5, pp. 1161–1180, 1995.

[18] MEYER, J., VAN DER HOEK, W. *Epistemic logic for AI and Computer Science*. Cambridge, Cambridge University Press, 1995.

[19] KRIPKE, S. A. “A Completeness Theorem in Modal Logic”, *Journal of Symbolic Logic*, v. 24, n. 1, pp. 1–14, mar. 1959.

Apêndice A

Código

A.1 DEMO

A.1.1 DEMO.hs

```
module DEMO
    (
     module Data.List,
     module Data.Char,
     module Models,
     module Display,
     module MinBis,
     module ActEpist,
     module MinAE,
     module DPLL,
     module Semantics,
 module SemanticsPA
    )
    where

import Data.List
import Data.Char
import Models
import Display
import MinBis
import ActEpist
import MinAE
import DPLL
import Semantics
```

```
import SemanticsPA

version :: String
version = "DEMO 1.03, Summer 2005"

initM = initE [P 1, P 2, P 3, P 4]
upM   = upd initM (groupM [a,b] p1)

measure :: (Eq a,Ord a) => (Model a b,a) -> Maybe [Int]
measure (m,w) =
  let
    f           = filter (\ (us,vs) -> elem w us || elem w vs)
    g [(xs,ys)] = length ys - 1
  in
    case kd45 (domain m) (access m) of
      Just a_balloons -> Just
         ( [ g (f balloons) | (a,balloons) <- a_balloons  ])
      Nothing         -> Nothing
```

A.1.2 SemanticsPA.hs

```
module SemanticsPA
where

import Data.List
import Data.Char
import Models
import Display
import MinBis
import ActEpist
import MinAE
import DPLL
import Semantics

type PoAMWPA = Prop

data PoPA = Char Prop
```

```
upPA :: EpistM -> PoAM -> Pmod (State,State) [Prop]
upPA  m@(Pmod worlds val acc points) am@(Pmod states pre susp actuals) =
   Pmod worlds' val' acc' points'
   where
   worlds' = [ (w,s) | w <- worlds, s <- states,
                       formula <- maybe [] (\ x -> [x]) (lookup s pre),
                       isTrAt 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'                      ]


updPA ::  EpistM -> PoAM -> EpistM
updPA sm am = (bisimPmod (sameVal) . convPmod) (upPA sm am)

updTest :: EpistM -> PoAMWPA -> EpistM
updTest m@(Pmod worlds val acc points) am  =
Pmod worlds val' acc points
where
val' = [ if any (==am) props then (w,props) else (w,am:props) | (w,props) <- val ]

updTrueAssignment :: EpistM -> PoAMWPA -> EpistM
updTrueAssignment m@(Pmod worlds val acc points) am  =
Pmod worlds val' acc points
where
val' = [ if any (==am) props then (w,props) else (w,am:props) | (w,props) <- val ]

updFalseAssignment :: EpistM -> PoAMWPA -> EpistM
updFalseAssignment m@(Pmod worlds val acc points) am  =
Pmod worlds val' acc points
```

```
where
val' = [ (w, (( filter (<am) props) ++ ( filter (>am) props)) ) | (w,props) <- val

publicAssignment :: EpistM -> Char -> Prop -> EpistM
publicAssignment m@( Pmod worlds val acc points ) bt am = Pmod worlds val' acc poin
where
val' = [  if bt == '-'
then (w, (( filter (<am) props) ++ ( filter (>am) props)) )
else
if any (==am) props then (w,props) else (w,am:props)
| (w,props) <- val ]

publicAssign :: EpistM -> Form -> Prop -> EpistM
publicAssign m@( Pmod worlds val acc points ) bF am = Pmod worlds val' acc points
where
val' = [  if isTrue ( m ) ( bF )
then
if any (==am) props then (w,props) else (w,am:props)
else
(w, (( filter (<am) props) ++ ( filter (>am) props)) )

| (w,props) <- val ]
```

A.1.3 Semantics.hs

```
module Semantics
where

import Data.List
import Data.Char
import Models
import Display
import MinBis
import ActEpist
import MinAE
import DPLL

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

```
  (nub . sort . concat) [ alternatives rel a current | a <- agents ]

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

up :: EpistM -> PoAM -> Pmod (State,State) [Prop]
up  m@(Pmod worlds val acc points) am@(Pmod states pre susp actuals) =
   Pmod worlds' val' acc' points'
   where
   worlds' = [ (w,s) | w <- worlds, s <- states,
                       formula <- maybe [] (\ x -> [x]) (lookup s pre),
                       isTrAt 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'                      ]

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

upd ::  EpistM -> PoAM -> EpistM
upd sm am = (bisimPmod (sameVal) . convPmod) (up sm am)

upds  :: EpistM -> [PoAM] -> EpistM
upds = foldl upd

reachableAut :: SM -> NFA State -> State -> [State]
reachableAut model nfa@(NFA start moves final) w =
  acc model nfa [(w,start)] []
  where
    acc :: SM -> NFA State -> [(State,State)] -> [(State,State)] -> [State]
    acc model (NFA start moves final) [] marked =
```

```
      (sort.nub) (map fst (filter (\ x -> snd x == final) marked))
   acc m@(Mo states _ rel) nfa@(NFA start moves final)
         ((w,s):pairs) marked =
     acc m nfa (pairs ++ (cs \\ pairs)) (marked ++ cs)
     where
       cs = nub ([ (w, s') | Move t (Tst f) s' <- moves,
                             t == s, notElem (w,s') marked,
                             isTrueAt w m f ]
                ++
               [ (w',s') | Move t (Acc ag) s' <- moves, t == s,
                           w' <- states,
                           notElem (w',s') marked,
                           elem (ag,w,w') rel ])

isTrueAt :: State -> SM -> Form -> Bool
isTrueAt w m Top = True
isTrueAt w m@(Mo worlds val acc) (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)

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

isTrueAt w m (Up am f) =
  and [ isTrueAt w' m' f |
        (m',w') <- decompose (upd (mod2pmod m [w]) am) ]
isTrueAt w m (Aut nfa f) =
  and [ isTrueAt w' m f | w' <- reachableAut m nfa w ]

isTrAt :: State -> EpistM -> Form -> Bool
isTrAt w (Pmod worlds val rel pts) = isTrueAt w (Mo worlds val rel)

isTrue :: EpistM -> Form -> Bool
```

```
isTrue (Pmod worlds val rel pts) form =
   and [ isTrueAt w (Mo worlds val rel) form | w <- pts ]

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

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)

e00 :: EpistM
e00 = initE [P 0]

e0 :: EpistM
e0 = initE [P 0,Q 0]

public :: Form -> PoAM
public form =
   (Pmod [0] [(0,form)] [ (a,0,0) | a <- all_agents ] [0])

groupM :: [Agent] -> Form -> PoAM
groupM agents form =
  if (sort agents) == all_agents
    then public form
    else
      (Pmod
```

```
          [0,1]
          [(0,form),(1,Top)]
          ([ (a,0,0) | a <- all_agents ]
            ++ [ (a,0,1) | a <- all_agents \\ agents ]
            ++ [ (a,1,0) | a <- all_agents \\ agents ]
            ++ [ (a,1,1) | a <- all_agents           ])
          [0])

message :: Agent -> Form -> PoAM
message agent form = groupM [agent] form

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

test :: Form -> PoAM
test = secret []

reveal :: [Agent] -> [Form] -> PoAM
reveal ags forms =
  (Pmod
     states
     (zip states forms)
     ([ (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
```

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

one :: PoAM
one = public Top

cmpP :: PoAM -> PoAM ->
                 Pmod (State,State) Form
cmpP m@(Pmod states pre susp ss) (Pmod states' pre' susp' ss') =
  (Pmod nstates npre nsusp npoints)
       where
         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'                 ]

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

cmpPoAM :: PoAM -> PoAM -> PoAM
cmpPoAM pm pm' = aePmod (cmpP pm pm')

cmp :: [PoAM] -> PoAM
cmp = foldl cmpPoAM one

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

pow :: Int -> PoAM -> PoAM
pow n am = cmp (take n (repeat am))

vBtest :: EpistM -> PoAM -> [EpistM]
vBtest m a = map (upd m) (star one cmpPoAM a)
```

```
star :: a -> (a -> a -> a) -> a -> [a]
star z f a = z : star (f z a) f a

vBfix :: EpistM -> PoAM -> [EpistM]
vBfix m a = fix (vBtest m a)

fix :: Eq a => [a] -> [a]
fix (x:y:zs) = if x == y then [x]
                         else x: fix (y:zs)

m1  = initE [P 1,P 2,P 3]
phi = Conj[p1,Neg (Conj[K a p1,Neg p2]),
              Neg (Conj[K a p2,Neg p3])]
a1  = message a phi

ndSum' :: PoAM -> PoAM -> PoAM
ndSum' m1 m2 = (Pmod states val acc ss)
  where
       (Pmod states1 val1 acc1 ss1) = convPmod m1
       (Pmod states2 val2 acc2 ss2) = convPmod m2
       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'

am0 = ndSum' (test p) (test (Neg p))

am1 = ndSum' (test p) (ndSum' (test q) (test r))

ndSum :: PoAM -> PoAM -> PoAM
ndSum m1 m2 = aePmod (ndSum' m1 m2)

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

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

A.1.4 ActEpist.hs

```
{-# LANGUAGE DatatypeContexts #-}

module ActEpist
where

import Data.List
import Models
import MinBis
import DPLL

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

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

data Form = Top
          | Prop Prop
          | Neg  Form
          | Conj [Form]
          | Disj [Form]
          | Pr Program Form
          | K Agent Form
          | EK [Agent] Form
          | CK [Agent] Form
          | Up PoAM Form
          | Aut (NFA State) Form
          deriving (Eq,Ord)
```

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

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]

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

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 (Aut aut f)   = '[': show aut ++ "]" ++ show f

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 ++ ")*"

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

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'

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)

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)

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'

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'

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

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 ::  PoAM -> Form -> Form

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

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)]
  collect (Aut nfa f) = if nfa == nullAut || canonF f == Top
                        then [] else [Aut nfa (canonF f)]

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 (Aut nfa f)      = if nfa == nullAut || canonF f == Top then []
                        else [[g (Aut nfa (canonF f))]]
cf g (Conj fs)        = concat (map (cf g) fs)
cf g (Disj fs)        = deMorgan (map (cf g) fs)

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 (Aut nfa f))= if nfa == nullAut || canonF f == Top then [[]]
                        else [[- g (Aut nfa (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

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
```

```
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)

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

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

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

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

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)

type State = Integer

type SM = Model State [Prop]

type EpistM = Pmod State [Prop]

valuation  :: EpistM -> [(State,[Prop])]
valuation pmod = eval $ fst (pmod2mp pmod)

type AM = Model State Form

type PoAM = Pmod State Form

preconditions :: PoAM -> [Form]
preconditions (Pmod states pre acc points) =
   map (table2fct pre) points

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

zero :: PoAM
zero = Pmod [] [] [] []

gsmPoAM :: PoAM -> PoAM
gsmPoAM (Pmod states pre 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
    else gsm (Pmod states' pre' acc' points')

transf :: PoAM -> Integer -> Integer -> Program -> Program
transf am@(Pmod states pre 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@(Pmod states pre 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@(Pmod states pre 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@(Pmod states pre acc points) i j (Conc [])  =
  transf am i j (Test Top)
transf am@(Pmod states pre acc points) i j (Conc [p]) = transf am i j p
transf am@(Pmod states pre 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@(Pmod states pre acc points) i j (Sum [])  =
  transf am i j (Test (Neg Top))
transf am@(Pmod states pre acc points) i j (Sum [p]) = transf am i j p
transf am@(Pmod states pre acc points) i j (Sum ps)  =
  Sum [ transf am i j p | p <- ps ]
transf am@(Pmod states pre acc points) i j (Star p) = kleene am i j n p
```

```
  where n = toInteger (length states)

kleene ::  PoAM -> 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

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

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

step :: PoAM -> 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)

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

t' :: PoAM -> 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))

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

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)

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

data Symbol = Acc Agent | Tst Form deriving (Eq,Ord,Show)
```

```
data (Eq a,Ord a,Show a) => Move a = Move a Symbol a deriving (Eq,Ord,Show)

data (Eq a,Ord a,Show a) => NFA a = NFA a [Move a] a deriving (Eq,Ord,Show)

states :: (Eq a,Ord a,Show a) => NFA a -> [a]
states (NFA s delta f) = (sort . nub) (s:f:rest)
  where rest = [ s' | Move s' a t' <- delta ]
               ++
              [ t' | Move s' a t' <- delta ]

symbols :: (Eq a,Ord a,Show a) => NFA a -> [Symbol]
symbols (NFA s moves f) = (sort . nub) [ symb | Move s symb t <- moves ]

recog :: (Eq a,Ord a,Show a) => NFA a -> [Symbol] -> Bool
recog (NFA start moves final) [] = start == final
recog (NFA start moves final) (symbol:symbols) =
  any (\ aut -> recog aut symbols)
     [ NFA new moves final |
           Move s symb new <- moves, s == start, symb == symbol ]

reachable :: (Eq a,Ord a,Show a) => NFA a -> [a]
reachable (NFA start moves final) = acc moves [start] []
  where
  acc :: (Show a,Ord a) => [Move a] -> [a] -> [a] -> [a]
  acc moves [] marked = marked
  acc moves (b:bs) marked = acc moves (bs ++ (cs \\ bs)) (marked ++ cs)
     where
     cs  = nub [ c | Move b' symb c <- moves, b' == b, notElem c marked ]

accNFA :: (Eq a,Ord a,Show a) => NFA a -> NFA a
accNFA nfa@(NFA start moves final) =
  if
    notElem final fromStart
  then
    NFA start [] final
  else
    NFA start moves' final
  where
```

```
   fromStart = reachable nfa
   moves' = [ Move x symb y | Move x symb y <- moves, elem x fromStart ]

initPart :: (Eq a,Ord a,Show a) => NFA a -> [[a]]
initPart nfa@(NFA start moves final) = [states nfa \\ [final], [final]]

refinePart :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> [[a]]
refinePart nfa p = refineP nfa p p
  where
  refineP :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> [[a]] -> [[a]]
  refineP nfa part [] = []
  refineP nfa@(NFA start moves final) part (block:blocks) =
     newblocks ++ (refineP nfa part blocks)
       where
         newblocks =
           rel2part block (\ x y -> sameAccBl nfa part x y)

sameAccBl :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> a -> a -> Bool
sameAccBl nfa part s t =
    and [ accBl nfa part s symb == accBl nfa part t symb |
                                              symb <- symbols nfa ]

accBl :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> a -> Symbol -> [[a]]
accBl nfa@(NFA start moves final) part s symb =
   nub [ bl part y | Move x symb' y <- moves, symb' == symb, x == s ]

compress :: (Eq a, Ord a, Show a) => NFA a -> [[a]]
compress nfa = compress' nfa (initPart nfa)
  where
  compress' :: (Eq a, Ord a, Show a) => NFA a -> [[a]] -> [[a]]
  compress' nfa part = if rpart == part
                       then part
                       else compress' nfa rpart
   where rpart = refinePart nfa part

minimalAut' :: (Eq a, Ord a, Show a) => NFA a -> NFA [a]
minimalAut' nfa@(NFA start moves final) = NFA start' moves' final'
  where
   (NFA st mov fin) = accNFA nfa
```

```
   partition    = compress (NFA st mov fin)
   f            = bl partition
   g (Acc ag)   = Acc ag
   g (Tst frm) = Tst (canonF frm)
   start'       = f st
   final'       = f fin
   moves'       = (nub.sort)
                      (map (\ (Move x y z) -> Move (f x) (g y) (f z)) mov)

convAut :: (Eq a,Ord a,Show a) => NFA a -> NFA State
convAut aut@(NFA s delta t) =
    NFA
    (f s)
    (map (\ (Move x symb y) -> Move (f x) symb (f y)) delta)
    (f t)
    where f = convert (states aut)

minimalAut :: (Eq a, Ord a, Show a) => NFA a -> NFA State
minimalAut = convAut . minimalAut'

nullAut = (NFA 0 [] 1)

genKnown :: [Agent] -> NFA State
genKnown agents = (NFA 0 [Move 0 (Acc a) 1 | a <- agents ] 1)

relCknown :: [Agent] -> Form -> NFA State
relCknown agents form = (NFA 0 (Move 0 (Tst form) 1 :
                               [Move 1 (Acc a) 0 | a <- agents]) 0)

cKnown :: [Agent] -> NFA State
cKnown agents = (NFA 0 [Move 0 (Acc a) 0 | a <- agents] 0)

aut' :: (Show a,Ord a) =>
           PoAM -> State -> State -> NFA a -> NFA (State,Int,a)
aut' (Pmod sts pre acc _) s t (NFA start delta final) =
  (NFA (s,0,start) delta' (t,1,final)) where
    delta' = [ Move (u,1,w) (Acc a) (v,0,x) |
                (a,u,v) <- acc,
                Move w (Acc a') x <- delta,
```

```
                a == a' ]
               ++
             [ Move (u,0,w) (Tst (table2fct pre u)) (u,1,w) |
                u <- sts,
                w <- states (NFA start delta final) ]
               ++
             [ Move (u,1,v)
                (Tst (Neg (Up (Pmod sts pre acc [u])
                                      (Neg form)))) (u,1,w) |
               u <- sts,
               Move v (Tst form) w <- delta ]

aut :: (Show a,Ord a) => PoAM -> State -> State -> NFA a -> NFA State
aut am s t nfa = minimalAut (aut' am s t nfa)

tr' :: Form -> Form
tr' Top = Top
tr' (Prop p) = Prop p
tr' (Neg form) = Neg (tr' form)
tr' (Conj forms) = Conj (map tr' forms)
tr' (Disj forms) = Disj (map tr' forms)
tr' (K agent form) = K agent (tr' form)
tr' (EK agents form) = Aut (genKnown agents) (tr' form)
tr' (CK agents form) = Aut (cKnown agents) (tr' form)

tr' (Aut nfa form) = Aut (tAut nfa) (tr' form)
tr' (Up (Pmod sts pre rel []) form) = Top
tr' (Up (Pmod sts pre rel [s]) Top) = Top
tr' (Up (Pmod sts pre rel [s]) (Prop p)) =
  impl (tr' (table2fct pre s)) (Prop p)
tr' (Up (Pmod sts pre rel [s]) (Neg form)) =
  impl (tr' (table2fct pre s))
    (Neg (tr' (Up (Pmod sts pre rel [s]) form)))
tr' (Up (Pmod sts pre rel [s]) (Conj forms)) =
  Conj [ tr' (Up (Pmod sts pre rel [s]) form) | form <- forms ]
tr' (Up (Pmod sts pre rel [s]) (Disj forms)) =
  Disj [ tr' (Up (Pmod sts pre rel [s]) form) | form <- forms ]
tr' (Up (Pmod sts pre rel [s]) (K agent form)) =
  impl (tr' (table2fct pre s))
```

```
    (Conj [ K agent (tr' (Up (Pmod sts pre rel [t]) form)) |
             t <- sts ])
tr' (Up (Pmod sts pre rel [s]) (EK agents form)) =
  tr' (Up (Pmod sts pre rel [s]) (Aut (genKnown agents) form))
tr' (Up (Pmod sts pre rel [s]) (CK agents form)) =
  tr' (Up (Pmod sts pre rel [s]) (Aut (cKnown agents) form))

tr' (Up (Pmod sts pre rel [s]) (Aut nfa form)) =
  Conj [ tr' (Aut (aut (Pmod sts pre rel [s]) s t  nfa)
              (Up  (Pmod sts pre rel [t]) form)) |  t <- sts ]
tr' (Up (Pmod sts pre rel [s]) (Up (Pmod sts' pre' rel' points) form)) =
  tr' (Up (Pmod sts pre rel [s])
    (tr' (Up (Pmod sts' pre' rel' points) form)))
tr' (Up (Pmod sts pre rel points) form) =
  Conj [ tr' (Up (Pmod sts pre rel [s]) form) | s <- points ]

kvbtr :: Form -> Form
kvbtr = canonF . tr'

tAut :: NFA State -> NFA State
tAut (NFA s delta f) = NFA s (map trans delta) f
 where trans (Move u (Acc x) v)     = Move u (Acc x) v
       trans (Move u (Tst form) v) = Move u (Tst (kvbtr form)) v
```

A.1.5 Display.hs

```
module Display
where

import Data.Char
import Data.List
import Models

accFor :: Eq a => a -> [(a,b,b)] -> [(b,b)]
accFor label triples = [ (x,y) | (label',x,y) <- triples, label == label' ]

containedIn :: Eq a => [a] -> [a] -> Bool
```

```
containedIn [] ys     = True
containedIn (x:xs) ys = elem x ys && containedIn xs ys

idR :: Eq a => [a] -> [(a,a)]
idR = map (\x -> (x,x))

reflR :: Eq a => [a] -> [(a,a)] -> Bool
reflR xs r = containedIn (idR xs) r

symR :: Eq a => [(a,a)] -> Bool
symR [] = True
symR ((x,y):pairs) | x == y    = symR (pairs)
                   | otherwise = elem (y,x) pairs
                                 && symR (pairs \\ [(y,x)])

transR :: Eq a => [(a,a)] -> Bool
transR [] = True
transR s = and [ trans pair s | pair <- s ]
   where
   trans (x,y) r = and [ elem (x,v) r | (u,v) <- r, u == y ]

equivalenceR :: Eq a => [a] -> [(a,a)] -> Bool
equivalenceR xs r = reflR xs r && symR r && transR r

isS5 :: (Eq a) => [a] -> [(Agent,a,a)] -> Bool
isS5 xs triples =
  all (equivalenceR xs) rels
    where rels = [ accFor i triples | i <- all_agents ]

pairs2rel :: (Eq a, Eq b) => [(a,b)] -> a -> b -> Bool
pairs2rel pairs = \ x y -> elem (x,y) pairs

equiv2part :: Eq a => [a] -> [(a,a)] -> [[a]]
equiv2part xs r = rel2part xs (pairs2rel r)

euclideanR :: Eq a => [(a,a)] -> Bool
euclideanR s = and [ eucl pair s | pair <- s ]
  where
  eucl (x,y) r = and [ elem (y,v) r | (u,v) <- r, u == x ]
```

```
serialR :: Eq a => [a] -> [(a,a)] -> Bool
serialR [] s = True
serialR (x:xs) s = any (\ p -> (fst p) == x) s && serialR xs s

kd45R :: Eq a => [a] -> [(a,a)] -> Bool
kd45R xs r = transR r && serialR xs r && euclideanR r

k45R :: Eq a => [(a,a)] -> Bool
k45R r = transR r && euclideanR r

isolated  :: Eq a =>  [(a,a)] -> a -> Bool
isolated r x = notElem x (map fst r ++ map snd r)

k45PointsBalloons :: Eq a => [a] -> [(a,a)] -> Maybe ([a],[([a],[a])])
k45PointsBalloons xs r =
  let
     orphans = filter (isolated r) xs
     ys = xs \\ orphans
  in
    case kd45Balloons ys r of
      Just balloons -> Just (orphans,balloons)
      Nothing       -> Nothing

entryPair :: Eq a => [(a,a)] -> (a,a) -> Bool
entryPair r = \ (x,y) -> notElem (y,x) r

kd45Balloons :: Eq a => [a] -> [(a,a)] -> Maybe [([a],[a])]
kd45Balloons xs r =
  let
     (s,t)          = partition (entryPair r) r
     entryPoints    = map fst s
     nonentryPoints = xs \\ entryPoints
     s5part xs r = if equivalenceR xs r
                      then Just (equiv2part xs t)
                      else Nothing
  in
    case s5part nonentryPoints t of
      Just part ->
```

```
        Just [ (nub (map fst (filter (\ (x,y) -> elem y block) s)),
                block) |     block <- part                              ]
      Nothing   ->
        Nothing

k45 :: (Eq a, Ord a) => [a] ->
    [(Agent,a,a)] -> Maybe [(Agent,([a],[([a],[a])]))]
k45 xs triples =
  if and [ maybe False (\ x -> True) b | (a,b) <- results  ]
     then Just [ (a, maybe undefined id b) | (a,b) <- results  ]
     else Nothing
       where rels     = [ (a, accFor a triples)  | a     <- all_agents ]
             results  = [ (a, k45PointsBalloons xs r) | (a,r) <- rels   ]

kd45 :: (Eq a, Ord a) => [a] -> [(Agent,a,a)] -> Maybe [(Agent,[([a],[a])])]
kd45 xs triples =
  if and [ maybe False (\ x -> True) b | (a,b) <- balloons ]
     then Just [ (a, maybe undefined id b) | (a,b) <- balloons ]
     else Nothing
       where rels     = [ (a, accFor a triples)  | a     <- all_agents ]
             balloons = [ (a, kd45Balloons xs r) | (a,r) <- rels       ]

kd45psbs2balloons :: (Eq a, Ord a) =>
  [(Agent,([a],[([a],[a])]))] -> Maybe [(Agent,[([a],[a])])]
kd45psbs2balloons psbs =
  if all (\ x -> x == []) entryList
     then Just balloons
     else Nothing
  where
    entryList  = [ fst bs      | (a,bs) <- psbs ]
    balloons   = [ (a, snd bs) | (a,bs) <- psbs ]

s5ball2part :: (Eq a, Ord a) =>
  [(Agent,[([a],[a])])] -> Maybe [(Agent,[[a]])]
s5ball2part balloons =
  if all (\ x -> x == []) entryList
     then Just partitions
     else Nothing
  where
```

```
    entryList  = [ concat (map fst bs) | (a,bs) <- balloons ]
    partitions = [ (a, map snd bs)     | (a,bs) <- balloons ]

display :: Show a => Int -> [a] -> IO()
display n = if n < 1 then error "parameter not positive"
                     else display' n n
  where
  display' ::  Show a => Int -> Int -> [a] -> IO()
  display' n m [] = putChar '\n'
  display' n 1 (x:xs) =   do (putStr . show) x
                             putChar '\n'
                             display' n n xs
  display' n m (x:xs) =   do (putStr . show) x
                             display' n (m-1) xs

showMo :: (Eq state, Show state, Ord state, Show formula) =>
                Model state formula -> IO()
showMo = displayM 10

showM :: (Eq state, Show state, Ord state, Show formula) =>
                              Pmod state formula -> IO()
showM (Pmod sts pre acc pnts) = do putStr "==> "
                                   print pnts
                                   showMo (Mo sts pre acc)

showMs :: (Eq state, Show state, Ord state, Show formula) =>
                              [Pmod state formula] -> IO()
showMs ms = sequence_ (map showM ms)

displayM :: (Eq state, Show state, Ord state, Show formula) =>
                Int -> Model state formula -> IO()
displayM n (Mo states pre rel) =
  do print states
     display (div n 2) pre
     case (k45 states rel) of
       Nothing       -> display n rel
       Just psbs     -> case kd45psbs2balloons psbs of
         Nothing       -> displayPB (div n 2) psbs
         Just balloons -> case s5ball2part balloons of
```

```
          Nothing          ->  displayB (div n 2) balloons
          Just parts       ->  displayP (2*n) parts

displayP :: Show a => Int -> [(Agent,[[a]])] -> IO()
displayP n parts = sequence_ (map (display n) (map (\x -> [x]) parts))

displayB :: Show a => Int -> [(Agent,[([a],[a])])] -> IO()
displayB n balloons = sequence_ (map (display n) (map (\x -> [x]) balloons))

displayPB :: Show a => Int ->  [(Agent,([a],[([a],[a])]))] -> IO()
displayPB n  psbs = sequence_ (map (display n) (map (\x -> [x]) psbs))

class Show a => GraphViz a where
  graphviz :: a -> String

glueWith :: String -> [String] -> String
glueWith _ []     = []
glueWith _ [y]    = y
glueWith s (y:ys) = y ++ s ++ glueWith s ys

listState  :: (Show a, Show b, Eq a, Eq b) => a -> [(a,b)] -> String
listState w val =
    let
      props = head (maybe [] (\ x -> [x]) (lookup w val))
      label = filter (isAlphaNum) (show props)
    in
      if null label
         then show w
         else show w
              ++ "[label =\"" ++ (show w) ++ ":" ++ (show props) ++ "\"]"

links :: (Eq a, Eq b) => [(a,b,b)] -> [(a,b,b)]
links [] = []
links ((x,y,z):xyzs) | y == z    = links xyzs
                     | otherwise =
                       (x,y,z): links (filter (/= (x,z,y)) xyzs)

cmpl :: Eq b => [(Agent,b,b)] -> [([Agent],b,b)]
cmpl [] = []
```

```
cmpl ((x,y,z):xyzs) = (xs,y,z):(cmpl xyzs')
  where xs = x: [ a | a  <- all_agents, elem a (map f xyzs1) ]
        xyzs1 = filter (\ (u,v,w) ->
                       (v == y && w == z)
                               ||
                       (v == z && w == y)) xyzs
        f (x,_,_) = x
        xyzs' = xyzs \\ xyzs1

instance (Show a, Show b, Eq a, Eq b) => GraphViz (Model a b) where
  graphviz (Mo states val rel) = if isS5 states rel
   then
    "digraph G { "
      ++
      glueWith  " ; " [ listState s val | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s')
                                ++ " [label="
                                ++ (filter isAlpha (show ags))
                                ++ ",dir=none ]"  |
                         s <- states, s' <- states,
                         (ags,t,t') <- (cmpl . links) rel,
                         s == t, s' == t'                         ]
      ++ " }"
   else
    "digraph G { "
      ++
      glueWith  " ; " [ listState s val | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s')
                                ++ " [label=" ++ (show ag) ++ "]"  |
                         s <- states, s' <- states,
                         (ag,t,t') <- rel,
                         s == t, s' == t'                         ]
      ++ " }"

listPState  :: (Show a, Show b, Eq a, Eq b) =>
                a -> [(a,b)] -> Bool -> String
listPState w val pointed =
```

```
    let
      props = head (maybe [] (\ x -> [x]) (lookup w val))
      label = filter (isAlphaNum) (show props)
    in
      if null label
         then if pointed then show w ++ "[peripheries = 2]"
              else            show w
         else if pointed then
              show w
              ++ "[label =\"" ++ (show w) ++ ":" ++ (show props) ++
                 "\",peripheries = 2]"
              else show w
              ++ "[label =\"" ++ (show w) ++ ":" ++ (show props) ++ "\"]"

instance (Show a, Show b, Eq a, Eq b) => GraphViz (Pmod a b)  where
  graphviz (Pmod states val rel points) = if isS5 states rel
   then
    "digraph G { "
      ++
      glueWith  " ; " [ listPState s val (elem s points) | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s')
                          ++ " [label="
                          ++ (filter isAlpha (show ags))
                          ++  ",dir=none ]"  |
                     s <- states, s' <- states,
                     (ags,t,t') <- (cmpl . links) rel,
                     s == t, s' == t'                         ]
      ++ " }"
   else
    "digraph G { "
      ++
      glueWith  " ; " [ listPState s val (elem s points) | s <- states ]
      ++  " ; " ++
      glueWith " ; " [ (show s) ++ " -> " ++ (show s')
                          ++ " [label=" ++ (show ag) ++ "]"  |
                     s <- states, s' <- states,
                     (ag,t,t') <- rel,
                     s == t, s' == t'                         ]
```

```
      ++ " }"

writeGraph :: String -> IO()
writeGraph cts = writeFile "graph.dot" cts

writeGr :: String -> String -> IO()
writeGr name cts = writeFile name cts

writeModel :: (Show a, Show b, Eq a, Eq b) => Model a b -> IO()
writeModel m = writeGraph (graphviz m)

writePmod :: (Show a, Show b, Eq a, Eq b) => (Pmod a b) -> IO()
writePmod m = writeGraph (graphviz m)

writeP :: (Show a, Show b, Eq a, Eq b) => String -> (Pmod a b) -> IO()
writeP name m = writeGr (name ++ ".dot") (graphviz m)
```

A.1.6 DPLL.hs

```
module DPLL

where

import Data.List

type Clause    = [Integer]
type ClauseSet = [Clause]

type Valuation = [Integer]

rSort :: ClauseSet -> ClauseSet
rSort = (srt1.nub) . (map (srt2. nub))
  where srt1 = sortBy cmp
        cmp [] (_:_)  = LT
        cmp [] []     = EQ
        cmp (_:_) []  = GT
        cmp (x:xs) (y:ys) | (abs x) < (abs y)  = LT
```

```
                    | (abs x) > (abs y)  = GT
                    | (abs x) == (abs y) = cmp xs ys
        srt2 = sortBy (\ x y -> compare (abs x) (abs y))

trivialC :: Clause -> Bool
trivialC [] = False
trivialC (i:is) = elem (-i) is || trivialC is

clsNub :: ClauseSet -> ClauseSet
clsNub = filter (not.trivialC)

data Trie = Nil | End | Tr Integer Trie Trie Trie deriving (Eq,Show)

nubT :: Trie -> Trie
nubT (Tr v p n End)   = End
nubT (Tr v End End r) = End
nubT (Tr v Nil Nil r) = r
nubT tr               = tr

trieMerge :: Trie -> Trie -> Trie
trieMerge End _ = End
trieMerge _ End = End
trieMerge t1 Nil = t1
trieMerge Nil t2 = t2
trieMerge t1@(Tr v1 p1 n1 r1) t2@(Tr v2 p2 n2 r2)
  | v1 == v2 = (Tr
                v1
                (trieMerge p1 p2)
                (trieMerge n1 n2)
                (trieMerge r1 r2)
                )
  | v1 < v2 = (Tr
                v1
                p1
                n1
                (trieMerge r1 t2)
                )
  | v1 > v2 = (Tr
                v2
```

```
                p2
                n2
                (trieMerge r2 t1)
                )

cls2trie :: ClauseSet -> Trie
cls2trie []             = Nil
cls2trie ([]:_)         = End
cls2trie cls@((i:is):_) =
  let j = abs i in
   (Tr
    j
    (cls2trie [ filter (/= j)  cl | cl <- cls, elem   j  cl ])
    (cls2trie [ filter (/= -j) cl | cl <- cls, elem (-j) cl ])
    (cls2trie [ cl | cl <- cls, notElem j cl, notElem (-j) cl ])
   )

trie2cls :: Trie -> ClauseSet
trie2cls Nil = []
trie2cls End = [[]]
trie2cls (Tr i p n r) =
    [ i:rest | rest <- trie2cls p ]
    ++
    [ (-i):rest | rest <- trie2cls n ]
    ++
    trie2cls r

units :: Trie -> [Integer]
units Nil  = []
units End  = []
units (Tr i End n r) = i : units r
units (Tr i p End r) = -i: units r
units (Tr i p n r)     = units r

unitProp :: (Valuation,Trie) -> (Valuation,Trie)
unitProp (val,tr) = (nub (val ++ val'), unitPr val' tr)
  where
  val' = units tr
  unitPr :: Valuation -> Trie -> Trie
```

```
  unitPr [] tr = tr
  unitPr (i:is) tr = unitPr is (unitSR i tr)

unitSR :: Integer -> Trie -> Trie
unitSR i = (unitR pol j) . (unitS pol j)
  where pol = i>0
        j   = abs i

unitS :: Bool -> Integer -> Trie -> Trie
unitS pol i Nil = Nil
unitS pol i End = End
unitS pol i tr@(Tr j p n r) | i == j = if pol
                                       then nubT (Tr j Nil n r)
                                       else nubT (Tr j p Nil r)
                            | i < j  = tr
                            | i > j  = nubT (Tr
                                        j
                                        (unitS pol i p)
                                        (unitS pol i n)
                                        (unitS pol i r)
                                       )

unitR ::  Bool -> Integer -> Trie -> Trie
unitR pol i Nil = Nil
unitR pol i End = End
unitR pol i tr@(Tr j p n r) | i == j = if pol
                                       then
                                       nubT (Tr
                                        j
                                        p
                                        Nil
                                        (trieMerge n r)
                                       )
                                       else
                                       nubT (Tr
                                        j
                                        Nil
                                        n
                                        (trieMerge p r)
```

```
                                  )
                       | i < j  = tr
                       | i > j  = nubT (Tr
                                   j
                                   (unitR pol i p)
                                   (unitR pol i n)
                                   (unitR pol i r)
                                  )

setTrue :: Integer -> Trie -> Trie
setTrue i Nil = Nil
setTrue i End = End
setTrue i tr@(Tr j p n r) | i == j = trieMerge n r
                          | i < j  = tr
                          | i > j  = (Tr
                                       j
                                       (setTrue i p)
                                       (setTrue i n)
                                       (setTrue i r)
                                      )

setFalse :: Integer -> Trie -> Trie
setFalse i Nil = Nil
setFalse i End = End
setFalse i tr@(Tr j p n r) | i == j = trieMerge p r
                           | i < j  = tr
                           | i > j  = (Tr
                                       j
                                       (setFalse i p)
                                       (setFalse i n)
                                       (setFalse i r)
                                      )

split :: (Valuation,Trie) -> [(Valuation,Trie)]
split (v,Nil) = [(v,Nil)]
split (v,End) = []
split (v, tr@(Tr i p n r)) = [(v++[i], setTrue i tr),
                              (v++[-i],setFalse i tr)]
```

```
dpll :: (Valuation,Trie) -> [(Valuation,Trie)]
dpll (val,Nil) = [(val,Nil)]
dpll (val,End) = []
dpll (val,tr) =
  concat [ dpll vt | vt <- (split.unitProp) (val,tr) ]

dp :: ClauseSet -> [(Valuation,Trie)]
dp cls = dpll ([], (cls2trie . rSort) (clsNub cls))
```

A.1.7 MinAE.hs

```
module MinAE
where

import Data.List
import Models
import MinBis
import ActEpist

unfold :: PoAM -> PoAM
unfold (Pmod states pre acc [])     = zero
unfold am@(Pmod states pre acc [p]) = am
unfold (Pmod states pre acc points) =
  Pmod states' pre' acc' points'
  where
  len = toInteger (length states)
  points' = [ p + len | p <- points ]
  states' = states ++ points'
  pre'    = pre ++ [ (j+len,f) | (j,f) <- pre, k <- points, j == k ]
  acc'    = acc ++ [ (ag,i+len,j) | (ag,i,j) <- acc, k <- points, i == k ]

preds, sucs :: (Eq a, Ord a, Eq b, Ord b) => [(a,b,b)] -> b -> [(a,b)]
preds rel s = (sort.nub) [ (ag,s1) | (ag,s1,s2) <- rel, s == s2 ]
sucs  rel s = (sort.nub) [ (ag,s2) | (ag,s1,s2) <- rel, s == s1 ]

psPartition :: (Eq a, Ord a, Eq b) => Model a b -> [[a]]
psPartition (Mo states pre rel) =
```

```
  rel2part states (\ x y -> preds rel x == preds rel y
                             &&
                             sucs rel x == sucs rel y)

minPmod :: (Eq a, Ord a) => Pmod a Form -> Pmod [a] Form
minPmod pm@(Pmod states pre rel pts) =
  (Pmod states' pre' rel' pts')
     where
     m         = Mo states pre rel
     partition = refine m (psPartition m)
     states'   = partition
     f         = bl partition
     g         = \ xs -> canonF (Disj (map (table2fct pre) xs))
     rel'      = (nub.sort) (map (\ (x,y,z) -> (x, f y, f z)) rel)
     pre'      = zip states' (map g states')
     pts'      = map (bl states') pts

aePmod ::  (Eq a, Ord a, Show a) => Pmod a Form -> Pmod State Form
--aePmod ::  PoAM -> PoAM
aePmod = (bisimPmod propEquiv) . minPmod . unfold .
                           (bisimPmod propEquiv) . gsmPoAM . convPmod
```

A.1.8 MinBis.hs

```
module MinBis where

import Data.List
import Models

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

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

```
initPartition (Mo states pre rel) r =
  rel2part states (\ x y -> lookupFs x y pre r)

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@(Mo states pre rel) part (block:blocks) =
     newblocks ++ (refineP m part blocks)
       where
         newblocks =
           rel2part block (\ x y -> sameAccBlocks m part x y)

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

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

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

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

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

minimalModel :: (Eq a, Ord a, Eq b, Ord b) =>
                 (b -> b -> Bool) -> Model a b -> Model [a] b
minimalModel r m@(Mo states pre rel) =
```

```
  (Mo states' pre' rel')
     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)

minimalPmod :: (Eq a, Ord a, Eq b, Ord b) =>
                  (b -> b -> Bool) -> Pmod a b -> Pmod [a] b
minimalPmod r (Pmod sts pre rel pts) = (Pmod sts' pre' rel' pts')
  where (Mo sts' pre' rel') = minimalModel r (Mo sts pre rel)
        pts' = map (bl sts') pts

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 Data.List")
  convrt n (y:ys) x | x == y    = n
                    | otherwise = convrt (n+1) ys x

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

convPmod ::  (Eq a, Show a) => Pmod a b -> Pmod Integer b
convPmod (Pmod sts pre rel pts) = (Pmod sts' pre' rel' pts')
   where (Mo sts' pre' rel') = conv (Mo sts pre rel)
         pts' = nub (map (convert sts) pts)

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)

bisimPmod ::  (Eq a, Ord a, Show a, Eq b, Ord b) =>
```

```
            (b -> b -> Bool) -> Pmod a b -> Pmod Integer b
bisimPmod r = convPmod . (minimalPmod r)
```

A.1.9 Models.hs

```
module Models where

import Data.List

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

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

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

all_agents :: [Agent]
all_agents = [a .. last_agent]

last_agent :: Agent
--last_agent = a
--last_agent = b
last_agent = c
--last_agent = d
--last_agent = e

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

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

mod2pmod :: Model state formula -> [state] -> Pmod state formula
mod2pmod (Mo states prec accs) points = Pmod states prec accs points

pmod2mp :: Pmod state formula -> (Model state formula, [state])
pmod2mp (Pmod states prec accs points)  = (Mo states prec accs, points)

decompose ::  Pmod state formula -> [(Model state formula, state)]
decompose (Pmod states prec accs points) =
   [(Mo states prec accs, point) | point <- points ]

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

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)

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

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

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

points :: Pmod state formula -> [state]
points (Pmod _ _ _ pnts) = pnts

gsm :: Ord state => Pmod state formula ->  Pmod state formula
```

```
gsm (Pmod states pre rel points) = (Pmod states' pre' rel' points)
   where
   states' = closure rel all_agents points
   pre'    = [(s,f)     | (s,f)     <- pre,
                          elem s states'                  ]
   rel'    = [(ag,s,s') | (ag,s,s') <- rel,
                          elem s states',
                          elem s' states'                 ]

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))

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

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