ESPECIFICAÇÃO DE SISTEMAS MULTI-AGENTES
BASEADOS EM CONHECIMENTO

Michel de Almeida Carlini

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA 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.

Aprovada por:

Prof. Mario Roberto Folhadela Benevides, Ph.D.

Profª. Sheila Regina Murgel Veloso, D.Sc.

Profª. Renata Wassermann, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
MAIO DE 2002

CARLINI, MICHEL DE ALMEIDA

Especificação de Sistemas Multi-Agentes Baseados em Conhecimento [Rio de Janeiro] 2002

XI, 105p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2002)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Lógica de Conhecimento
2. Sistemas Distribuídos

I. COPPE/UFRJ II. Título(série)

Agradecimentos

Ao meu caro orientador, Professor Mario Benevides, por sua sábia orientação e incentivo permanente, a quem também devo meu respeito e minha admiração pela paciência e confiança que me foram destinadas.

A toda minha família, especialmente meus pais, Tercílio Carlini Sobrinho e Regina Celi de Almeida Carlini, por todo o apoio e dedicação oferecidos a mim e que espero sempre poder retribuir ao longo de minha vida.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pelo importante suporte financeiro direcionado à realização de pesquisa científica no país que muito me beneficiou.

A Carla Delgado, por sua amizade e o desenvolvimento de uma brilhante pesquisa, que tanto me auxiliaram na realização deste trabalho, principalmente em meus primeiros passos.

A Kate Revoredo, por sua amizade e companhia em muitas horas de estudo, que muito me ajudaram a esclarecer várias dúvidas durante toda a elaboração desta tese.

Aos colegas Ricardo Mesquita, Vera Prudência, Vania Costa e Fernanda Baião, que por seus conselhos ou palavras amigas, tornaram mais fácil a minha tarefa para concluir este trabalho.

Aos professores, funcionários e alunos do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ que contribuíram de alguma forma para que esta tese pudesse ser feita.

Por fim, agradeço ao Criador por ter alcançado mais esta vitória em minha vida, pois "nenhuma folha cai de uma árvore se não for por vontade de Deus".

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

ESPECIFICAÇÃO DE SISTEMAS MULTI-AGENTES BASEADOS EM CONHECIMENTO

Michel de Almeida Carlini

Maio/2002

Orientador: Mario Roberto Folhadela Benevides

Programa: Engenharia de Sistemas e Computação

Este trabalho apresenta uma proposta para construir programas baseados em conhecimento (*Knowledge-Based Programs*), utilizando a lógica dinâmica de conhecimento para modelar sistemas multi-agentes. É construído e estudado como exemplo, um KBP que descreve o conhecido problema das crianças com lama na testa, de acordo com as especificações desejadas para o sistema. Através de conceitos ligados a sistemas distribuídos e lógica de conhecimento, foram desenvolvidos uma arquitetura e uma linguagem que especificam uma estrutura básica para um KBP, com exemplos de funcionamento e provas de que os programas executam corretamente. A seguir, são apresentados dois métodos de tradução. Um para extrair as regras lógicas presentes em um KBP, o que possibilita a verificação de determinadas propriedades existentes na lógica deste programa. E outro para obter o procedimento inverso, através de regras lógicas, construir um KBP que implemente estas regras seguindo a especificação de um determinado sistema.

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

SPECIFICATION OF KNOWLEGDE-BASED
MULTI-AGENT SYSTEMS

Michel de Almeida Carlini

May/2002

Advisor: Mario Roberto Folhadela Benevides

Department: Systems Engineering and Computer Science

This work presents a proposal to build programs known as *Knowledge-Based Programs*, using dynamic knowledge logics in order to model multi-agent systems. A KBP was built and studied to describe the well-known Muddy Children Puzzle, in accordance with the desired specifications of the system behavior. Using notions from distributed systems and knowledge logics, an architecture and a language were proposed to specify a KBP's basic structure. Also, some examples are presented with their proofs of correct execution. Next, two translation methods are showed. One to extract the logical formulas from a KBP, this can be useful to verify some logical properties of the program. And another to build a KBP from a set of logical formulas (the system specification) written in our language.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este trabalho apresenta um estudo de sistemas multi-agentes, e sua modelagem através de programas baseados em conhecimento (*Knowledge-Based Programs*). São utilizados para isto os conceitos e propriedades obtidos através do formalismo presente na lógica dinâmica de conhecimento. Os programas baseados em conhecimento (deste ponto em diante mencionados apenas como KBP's) são algoritmos designados para modelar sistemas com múltiplos agentes (que também podem ser chamados de sistemas distribuídos) que, além de fazer as computações procedurais convencionais, são capazes de lidar com noções de conhecimento e crença para manipular o conhecimento envolvido no sistema. Mais especificamente, os KBP's lidam com agentes que realizam ações de acordo com o conhecimento que possuem sobre o sistema em que estão inseridos.

Para expressar o comportamento epistemológico e procedural dos KBP's faz-se necessário definir uma arquitetura distribuída específica onde os agentes sejam capazes de realizar as ações computacionais convencionais e fazer inferências sobre sua base local de conhecimento. Para que haja colaboração e o conhecimento possa evoluir ao longo do processamento, os agentes devem se comunicar uns com os outros. Os KBP's devem ser elaborados segundo um modelo e uma linguagem bem definida, que tenha poder para expressar as conclusões lógicas que os agentes atingem utilizando sua máquina de inferência. O modelo deve definir como as ações são determinadas pelas informações que o agente tem acerca do estado do sistema e a comunicação entre os agentes como forma de disseminar o conhecimento no sistema distribuído.

A máquina de inferência de um agente trabalha sobre a lógica de conhecimento e crença baseada em modelos de Kripke. Através dela pode-se expressar propriedades sobre o conhecimento de um agente e também sobre conhecimento em um grupo de agentes. Porém a semântica dos modelos de Kripke limita-se a modelar relações entre estados de conhecimento, mas não a evolução temporal dos estados de conhecimento nem do sistema, nem de cada agente individualmente. Os KBP's por sua vez podem tratar o conhecimento evoluindo através do sistema distribuído. Para obter uma lógica de conhecimento com poder de expressão semelhante ao dos KBP's, são acrescentadas modalidades de ação à lógica de conhecimento para múltiplos agentes. Definindo então uma lógica de conhecimento que esteja de acordo com a linguagem utilizada nos KBP's, pode-se aplicar uma transformação a um programa para gerar os axiomas de uma lógica de conhecimento para múltiplos agentes modelando o problema representado no programa. Ao expressar um problema através da lógica dinâmica de conhecimento torna-se possível verificar formalmente as propriedades presentes no programa, e consequentemente saber se este programa está de acordo com a especificação do problema. Tendo um método que define passo a passo a tradução de uma forma para a outra, abre-se a possibilidade de se construir uma ferramenta computacional capaz de realizar esta tarefa.

Com o intuito de se construir um aplicativo capaz de implementar os métodos de tradução de regras lógicas para um KBP e vice-versa, é proposto um procedimento que fornece as condições necessárias de entrada de informação para o usuário. Principalmente para o caso da geração de um KBP, onde as regras lógicas relacionadas a um determinado problema devem ser introduzidas pelo usuário juntamente com informações sobre a estrutura de comunicação dos agentes presentes na situação em questão, dessa forma, o aplicativo deve ser capaz de construir um KBP que mapeia estas informações e as apresente de forma correta e concisa.

Também é apresentado um outro exemplo de problema tratado em um KBP, através do programa *Cluedo*, que analisa um jogo de tabuleiro (aqui conhecido como Detetive) que possui características muito interessantes no que diz respeito ao estudo de conhecimento entre agentes e como eles utilizam este conhecimento quando interagem entre si. Além de fornecer um contra-ponto em relação ao problema das

crianças com lama na testa, exemplificando outros detalhes que não estão presentes nesse problema. E ainda, é feito um estudo sobre um algoritmo para sistema distribuídos bastante conhecido, chamado PIF (*Propagation of Information with Feedback*).

Por fim, é indicada uma prova de corretude (conhecida como corretude forte) para o programa que trata do problema das crianças com lama na testa. Essa prova de corretude tem por objetivo validar especificações que sejam feitas durante o processo de construção ou análise de um sistema com vários agentes, fornecendo uma base formal para a especificação de um sistema que envolva conhecimento entre agentes.

A apresentação desses assuntos nesta tese é organizada da seguinte forma:

No Capítulo 2 são vistos os conceitos básicos sobre conhecimento e crença com a introdução formal da lógica de conhecimento. Seguem-se as seções sobre estruturas de Kripke, sintaxe e semântica da lógica de conhecimento, noções de conhecimento comum e distribuído e uma introdução às provas de corretude e completude para a lógica exibida. Também neste capítulo, é feita uma introdução preliminar sobre KBP's, os conceitos de ações, protocolos e contextos e também as seções sobre modelagem de KBP's e exemplo de aplicação. O capítulo é baseado em (FAGIN, HALPERN, et al., 1995; FAGIN, HALPERN, et al., 1997).

O Capítulo 3 ilustra o funcionamento dos sistemas distribuídos, a estrutura básica de um sistema de memória distribuída, noções de execuções e passos em um sistema, modelos síncrono e assíncrono, definição de um algoritmo e conceitos de eventos e ordens. O capítulo é baseado em (BARBOSA, 1996; FAGIN, HALPERN, et al., 1995).

O Capítulo 4 reapresenta a linguagem utilizada para modelagem de KBP's estendida com a utilização dos conceitos vistos em sistemas distribuídos. Para dar suporte ao desenvolvimento dos KBP's, também é introduzida a Lógica Dinâmica de Conhecimento, (DELGADO, BENEVIDES, 2001). Como exemplo de aplicação, é demonstrada a construção de um KBP que modela o conhecido problema das crianças com lama na testa, tanto para o caso síncrono quanto para o assíncrono, com a apresentação de exemplos e provas para os programas.

O Capítulo 5 introduz os chamados métodos de tradução de KBP's para a lógica e vice-versa. É utilizado como exemplo o programa que foi construído para modelar o problema das crianças com lama na testa. Também são enunciados dois teoremas simples sobre os métodos de tradução vistos.

O Capítulo 6 mostra uma possível implementação prática para os métodos de tradução vistos, com as indicações necessárias para a execução das transformações que realizam as traduções. E mais uma vez é demonstrado um exemplo de aplicação, que trata de um simples jogo de cartas.

O Capítulo 7 fornece mais dois exemplos de modelagem de KBP's. Primeiro, o programa $Cluedo_i$ que estuda o jogo de mesmo nome (aqui conhecido como Detetive), oferecendo mais um estudo interessante sobre sistemas multi-agentes envolvendo conhecimento. E depois, é considerado o algoritmo de sistemas distribuídos para PIF, nas versões síncrona e assíncrona, com um maior enfoque nesta última. É feito ainda um estudo da lógica envolvida dos KBP's criados para modelar a especificação desejada e também são apresentados exemplos de execução para uma situação escolhida.

O Capítulo 8 trata das conclusões e possíveis trabalhos futuros, enfatizando os pontos mais importantes no trabalho e oferecendo outras possibilidades de avanços para estudo.

O Apêndice A inicia o desenvolvimento de uma prova de corretude (chamada corretude forte) para o KBP construído para modelar o problema das crianças com lama na testa. Este apêndice é baseado em (FAGIN, HALPERN, et al., 1995).

O Apêndice B introduz alguns conceitos e definições interessantes, relacionados a especificação e propriedades de sistemas distribuídos em geral. Este apêndice é baseado em (HALPERN, ZUCK, 1992).

Capítulo 2

Conhecimento em Sistemas Distribuídos

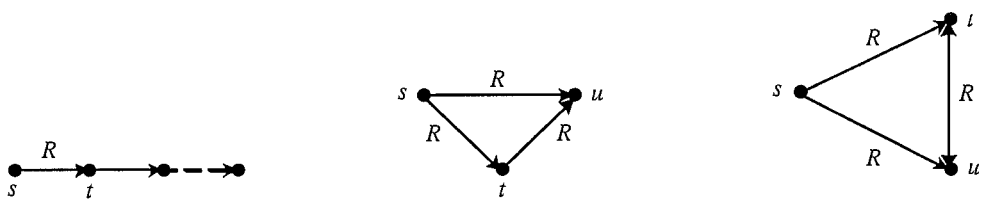
Para discutir o conceito de conhecimento em sistemas multi-agentes é necessário supor que os agentes devem ser capazes de raciocinar sobre o mundo e também sobre o raciocínio de outros agentes, (DELGADO, BENEVIDES, 2001). A lógica de conhecimento, em especial o sistema axiomático $S5$, é utilizada para representar conhecimento em sistemas multi-agentes, (FAGIN, HALPERN, et al., 1995). E com a aplicação do conceito de KBP's, (FAGIN, HALPERN, et al., 1997), é possível representar este conhecimento entre os agentes no sistema de forma elegante e consistente.

Com o objetivo de formalizar a evolução do conhecimento em um sistema, são necessários modelos semânticos de conhecimento. O método mais comum de se modelar conhecimento é o baseado na semântica de mundos possíveis de Kripke, (HINTIKKA, 1962). Neste método, a informação que um agente (ou um processo) possui sobre o sistema pode ser incompleta, ao invés de conhecer precisamente o estado atual em que o sistema se encontra, o agente pode somente saber que o estado atual do sistema pertence a um dado conjunto de estados possíveis (daí o nome mundos possíveis). O agente sabe um fato φ como verdadeiro se φ é verdadeiro em todos os estados que o agente imagina possíveis. A semântica de mundos possíveis é formalizada com o auxílio das estruturas de Kripke, (KRIPKE, 1963).

Estes modelos semânticos de conhecimento podem ser utilizados para interpretar fórmulas da lógica de conhecimento. Estas fórmulas pertencem à lógica proposicional modal, onde para cada agente i, existe uma modalidade K_i. Intuitivamente,

a fórmula $K_i\varphi$ indica que o "agente i sabe φ". Para entender melhor o conceito de conhecimento, é feita uma formalização do conhecimento através da axiomatização de fórmulas válidas, que são satisfeitas por todas as estruturas de conhecimento. Então é utilizado o sistema modal lógico $S5$ como uma correta e completa axiomatização para as estruturas de conhecimento, o que sugere a sua aplicação como um formalismo apropriado para o estudo de conhecimento em sistemas distribuídos, (FAGIN, VARDI, 1986).

Uma estrutura de Kripke M é uma tupla $(S, \pi, \mathcal{K}_1, ..., \mathcal{K}_m)$, onde S é um conjunto de estados, π é uma atribuição de valores-verdade para as primitivas de cada estado $s \in S$ (tal que $\pi(s,p) \in \{true, false\}$ para cada estado s), e $\mathcal{K}_i, i = 1, ..., m$ é uma relação binária em S que é serial, transitiva e euclidiana. Uma relação R é serial se para cada $s \in S$ existe algum $t \in S$ tal que $(s,t) \in R$; R é transitiva se $(s,u) \in R$ sempre que $(s,t) \in R$ e $(t,u) \in R$; R é euclidiana se $(t,u) \in R$ sempre que $(s,t) \in R$ e $(s,u) \in R$. Intuitivamente, $(s,t) \in \mathcal{K}_i$ se em um estado s, o agente i considera o estado t possível, isto é, se s fosse o verdadeiro estado do mundo em questão, o agente i poderia considerar t como um possível estado deste mundo. Será mostrado que as condições em $\mathcal{K}_i$ implicam em axiomas relacionados a conhecimento. Por exemplo, o fato de que uma relação $\mathcal{K}_i$ é serial significa que em todos os mundos, o agente i sempre considera algum mundo possível, disto pode-se afirmar que ele não acredita em um estado falso. Modificando essas condições, podem-se obter diferentes axiomas para conhecimento. As relações mais comuns estão demonstradas na Figura 2.1.



Relação serial | Relação transitiva | Relação euclidiana

Figura 2.1: Exemplos de relações entre estados

As estruturas de conhecimento podem ser vistas como modelos abstratos para conhecimento. Elas podem modelar todos os estados possíveis do conhecimento

sem se preocupar em como o conhecimento foi adquirido primeiramente. Mas para estudar formalmente o conhecimento em sistemas distribuídos, é preciso saber quais estados de conhecimento podem ser alcançados nestes sistemas. Considerando que os agentes em sistemas distribuídos se comunicam entre si exclusivamente através da troca se mensagens, é necessário saber quais estados de conhecimento são alcançáveis através desta comunicação em particular.

2.1 Lógica de Conhecimento em Sistemas Distribuídos

A lógica de conhecimento é uma variante da lógica modal proposicional, onde os operadores que lidam com as noções de necessidade e possibilidade são transformados em operadores que representam os conceitos de conhecimento e crença, respectivamente. Para explicar o significado desses operadores nos vários sistemas axiomáticos da lógica modal é preciso recorrer à noção de uma relação de acessibilidade, que está ligada à definição do sistema semântico, que é apresentado a seguir.

2.1.1 Sintaxe e Semântica

A linguagem utilizada é a da lógica modal proposicional para um número m de agentes. Os símbolos da lógica de conhecimento são, (COSTA, 1992):

- Um conjunto Φ enumerável de símbolos proposicionais;
- Pontuação: '(' e ')';
- Conectivos: '$\neg$', '$\wedge$', '$\vee$', '$\rightarrow$' e '$\leftrightarrow$';
- Operadores modais: K_i e B_i, para $i = 1, ..., m$ (um para cada agente).

Os conectivos seguem as definições da lógica proposicional, o operador modal K_i indica conhecimento em relação ao agente i, e o operador modal B_i indica crença em relação ao agente i, para $i = 1, ..., m$.

As fórmulas da linguagem (denotada por $\mathcal{L}_m$) são obtidas recursivamente através das regras:

1. Todo símbolo proposicional de Φ é uma fórmula, chamada fórmula atômica;
2. Se φ é uma fórmula, então $\neg\varphi$ também é uma fórmula;
3. Se φ e ψ são fórmulas, então $\varphi \wedge \psi | \varphi \vee \psi | \varphi \rightarrow \psi | \varphi \leftrightarrow \psi$ também são fórmulas;
4. Se φ é uma fórmula, então $K_i\varphi$ também é uma fórmula, para $i = 1, ..., m$;
5. Se φ é uma fórmula, então $B_i\varphi$ também é uma fórmula, para $i = 1, ..., m$;
6. Nada mais é uma fórmula, a não ser quando descrito pelos itens acima.

Através da relação de satisfabilidade $\models$, onde $M, s \models \varphi$ é lido como "φ é satisfatível no estado s pela estrutura M", são definidos os modelos semânticos:

- $M, s \models \mathit{true}$;
- $M, s \models p$ se e somente se $\pi(s, p) = \{\mathit{true}\}$, onde p é uma primitiva;
- $M, s \models \neg\varphi$ se e somente se $M, s \not\models \varphi$;
- $M, s \models \varphi \wedge \psi$ se e somente se $M, s \models \varphi$ e $M, s \models \psi$;
- $M, s \models \varphi \vee \psi$ se e somente se $M, s \models \varphi$ ou $M, s \models \psi$;
- $M, s \models K_i\varphi$ se e somente se para todo t tal que $(s, t) \in \mathcal{K}_i$, $M, t \models \varphi$;
- $M, s \models B_i\varphi$ se e somente se existe t tal que $(s, t) \in \mathcal{K}_i$ onde $M, t \models \varphi$.

Uma axiomatização correta e completa baseada no sistema modal $S5$, também referido como $S5_m$, considerando uma generalização para sistemas distribuídos (onde m é o número de agentes, $1, ..., m$), é a seguinte:

1. Todas as tautologias da lógica proposicional. (A1)
2. Axioma **K**: $K_i(\varphi \rightarrow \psi) \rightarrow (K_i\varphi \rightarrow K_i\psi)$, para $i = 1, ..., m$. (A2)
3. Axioma do conhecimento: $K_i\varphi \rightarrow \varphi$, para $i = 1, ..., m$. (A3)
4. Introspecção positiva: $K_i\varphi \rightarrow K_iK_i\varphi$, para $i = 1, ..., m$. (A4)
5. Introspecção negativa: $\neg K_i\varphi \rightarrow K_i\neg K_i\varphi$, para $i = 1, ..., m$. (A5)
6. Axioma Dual: $B_i\varphi \leftrightarrow \neg K_i\neg\varphi$ e $K_i\varphi \leftrightarrow \neg B_i\neg\varphi$, para $i = 1, ..., m$. (A6)

Existem também mais duas regras de inferência:

1. *Modus Ponens*: de φ e $\varphi \rightarrow \psi$ deriva-se ψ. (R1)

$$\frac{\varphi, \varphi \rightarrow \psi}{\psi}$$

2. Generalização do conhecimento: de φ deriva-se $K_i\varphi$. (R2)

$$\frac{\varphi}{K_i\varphi}$$

As expressões (A1) e (R1) são obtidas diretamente da lógica proposicional. (A2) informa que o conhecimento de um agente é fechado sob implicação. (A3) informa que apenas fatos verdadeiros são conhecidos para um agente. (A4) e (A5) são os axiomas de introspecção, intuitivamente, eles informam que cada agente tem conhecimento completo sobre o seu conjuntos de crenças. (A6) é o axioma dual importado da lógica modal, que permite a transformação entre os operadores modais.

A validade de (A3), (A4) e (A5) se deve ao fato da relação $\mathcal{K}_i$ ser atribuída como serial, transitiva e euclidiana. Mais precisamente, (A3) segue de uma relação $\mathcal{K}_i$ serial, (A4) de uma relação transitiva e (A5) de uma relação euclidiana. Essas atribuições de relação estão presentes no sistema modal de conhecimento $S5_m$.

Um ponto interessante é o que se refere às expressões (A2) e (R2), onde o conceito de mundos possíveis não possui influência alguma. Não importa o quanto se modifique as relações $\mathcal{K}_i$, a questão de que um agente conhece ou acredita em um fato somente se este fato é verdadeiro em todos os mundos em que o agente considera possíveis implica em uma situação onde o agente conhece todas as tautologias e o seu conhecimento é fechado sob implicação.

2.1.2 Exemplo de Estrutura de Conhecimento

Considerando um exemplo para a aplicação dos conceitos de mundos possíveis e a sua utilização em relações de conhecimento, é proposta agora, a análise de um simples jogo de cartas que utiliza um conjunto de três cartas chamadas de A, B e C. Os agentes 1 e 2 escolhem uma carta cada um, e a terceira carta é deixada com a face voltada para baixo. Um mundo possível é caracterizado através da descrição

das cartas que pertencem a cada agente. Por exemplo, no mundo (A, B), o agente 1 possui a carta A e o agente 2 possui a carta B (portanto a carta C é a que está com a face voltada para baixo). Existem então, seis mundos possíveis: (A, B), (A, C), (B, A), (B, C), (C, A) e (C, B). Além disso, considerando o mundo (A, B), o agente 1 imagina dois mundos possíveis: (A, B) propriamente e (A, C). O agente 1 sabe que ele possui a carta A, mas ele considera possível que o agente 2 possui ou a carta B ou a carta C. Da mesma forma, no mundo (A, B), o agente 2 também imagina dois mundos possíveis: (A, B) e (C, B). Em geral, em um mundo (x, y), o agente 1 considera (x, y) e (x, z) possíveis, e o agente 2 considera (x, y) e (z, y) possíveis, onde z é diferente de x e y.

A partir dessa descrição, pode-se construir facilmente as relações $\mathcal{K}_1$ e $\mathcal{K}_2$. E também é fácil perceber que elas são na verdade, relações de equivalência, como é requerido pelas definições. Isto é devido ao fato de que a relação de possibilidades de um agente é determinada pela informação que ele possui, no caso, a sua carta. Em qualquer situação onde a relação de possibilidades de um agente é determinada pela informação que ele possui (e isto ocorre quase em todos os casos), as relações de possibilidades são relações de equivalência.

A estrutura deste exemplo com as três cartas é apresentada na Figura 2.2, (FAGIN, HALPERN, et al., 1995). Como as relações entre os mundos são de equivalência, as pontas das setas foram omitidas por simplicidade.

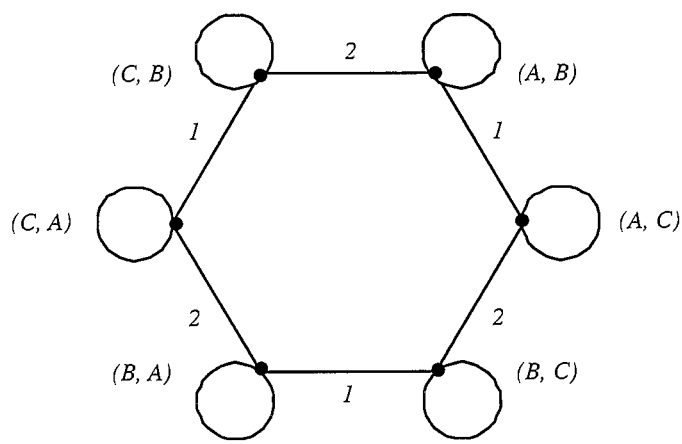


Figura 2.2: Estrutura de Kripke para o jogo das três cartas

Este exemplo demonstra a necessidade da existência de mundos que um agente não considera possíveis, incluídos na estrutura. Por exemplo, no mundo (A, B), o agente 1 sabe que o mundo (B, C) não pode ser considerado (pois ele sabe perfeitamente que A é a sua própria carta). No entanto, já que o agente 1 considera que o agente 2 considera que (B, C) é possível, este mundo deve ser incluído na estrutura. Isto é demonstrado pelo fato de não haver nenhuma relação na estrutura de (A, B) para (B, C) denotada por 1, mas existe uma relação assim para (A, C), e este possui uma relação denotada por 2 para (B, C). Além disso, as relações reflexivas existentes em cada mundo indicam a possibilidade óbvia de que os agentes também consideram o mundo atual como possível.

2.1.3 Conhecimento Comum e Conhecimento Distribuído

Um agente em um grupo precisa considerar não somente fatos que são verdadeiros sobre o mundo, mas também deve considerar o conhecimento dos outros agentes do grupo. Um exemplo subjetivo porém simples sobre um agente que deve imaginar o conhecimento de outro agente é a venda de um automóvel. O vendedor sabe o valor aproximado do carro, mas isso não é o único fator relevante para ele estipular seu preço. Também é levado em consideração o que ele pensa sobre o que o cliente sabe sobre o valor do carro, afinal é de seu interesse vender o carro pelo mais alto preço que ele consiga fazer o cliente pagar. O cliente, por sua vez, também tem algum conhecimento do valor do carro, e seu interesse é comprar pelo preço mais baixo, sabendo que o vendedor deve querer convencê-lo de que o carro vale mais do que realmente vale. Nesse caso, o preço de venda será estabelecido levando em consideração o conhecimento que os dois agentes têm sobre o sistema, e não será necessariamente o valor exato do carro, que pode inclusive não ser conhecido por nenhum dos dois agentes. As pessoas geralmente perdem a linha de raciocínio quando interpretam sentenças complexas tais como: "João não sabe se Maria sabe que João sabe que Maria sabe que Antônio esteve no parque." Mas este é justamente o tipo de raciocínio que se deseja fazer quando se analisa o conhecimento de agentes em um grupo.

Quando se está considerando uma situação que envolve vários agentes, existe

um grande número de estados de conhecimento onde é preciso saber que todos os agentes do grupo sabem um determinado fato. Como exemplo, pode-se citar o sistema de trânsito, onde todos os motoristas devem saber que o sinal vermelho significa pare. Assumindo que todos os motoristas sabem disso, é possível dizer que um motorista poderia sentir-se seguro? A resposta é não, pois ele poderia pensar que algum outro motorista talvez não soubesse dessa regra e pudesse avançar um sinal vermelho. Para que o sistema funcione, é necessário que todos os motoristas saibam que todos sabem que o sinal vermelho significa pare.

Mesmo a suposição de que "todos sabem que todos sabem algo" pode não ser suficiente para descrever todos os tipos de sistemas multi-agentes envolvendo conhecimento. Existem ainda outros sistemas desse tipo onde é necessário considerar um estado em que, simultaneamente, todos sabem um fato φ, todos sabem que todos sabem φ, todos sabem que todos sabem que todos sabem φ, e assim sucessivamente. É essa a noção de conhecimento comum entre os agentes de um grupo (no exemplo citado anteriormente, presume-se que a convenção de que o sinal verde significa "siga" e o sinal vermelho significa "pare" é de conhecimento comum entre todos os motoristas). Caracteriza-se intuitivamente o conhecimento comum como aquilo que "qualquer um" sabe.

Um ponto de interesse em relação ao conceito de conhecimento comum é o que diz respeito ao fechamento de um acordo entre um grupo de agentes. O conhecimento comum é atingido quando um fato é anunciado de forma que todos os agentes do grupo estão presentes ao mesmo tempo, e todos sabem que todos estão presentes, isto significa fazer com que todos tomem conhecimento do fato simultaneamente. E isto é muitas vezes um pré-requisito para o alcance de uma condição de parada em sistema distribuídos, o que é muito importante na análise de um grupo de agentes que interagem entre si.

Um outro conceito ligado a conhecimento e que é, de certa forma, oposto ao conceito de conhecimento comum é o de conhecimento distribuído. Um grupo possui o conhecimento distribuído de um fato φ se o conhecimento de φ está distribuído entre os agentes deste grupo, então reunindo o conhecimento destes agentes pode-se deduzir φ, mesmo se nenhum dos agentes do grupo individualmente sabe φ. Por

exemplo, se Alice sabe que Roberto está apaixonado ou por Carolina ou por Suzana, e Carlos sabe que Roberto não está apaixonado por Carolina, então Alice e Carlos tem juntos o conhecimento distribuído do fato de que Roberto está apaixonado por Suzana, embora nem Alice nem Carlos saibam disto individualmente. Enquanto que o conhecimento comum pode ser visto como o que "qualquer um" sabe, o conhecimento distribuído é visto como o que "Deus" sabe, alguém que possui o completo conhecimento sobre o que cada agente no grupo sabe, ou poderia saber.

O problema das crianças com lama na testa que será apresentado com detalhes no Capítulo 4, fornece um ambiente para estudo que demonstra a utilidade destes conceitos relativos a conhecimento na análise de situações que envolvem grupos de agentes.

A linguagem descrita anteriormente ainda não permite expressar as noções de conhecimento comum e distribuído, além de outros conceitos auxiliares envolvendo conhecimento em um sistema com múltiplos agentes. A seguir, são formalizados os estados de conhecimento relativos a um grupo G de agentes, como também são apresentados os axiomas que expressam as propriedades desses estados, (FAGIN, HALPERN, et al., 1995):

- Conhecimento de um agente, $A_G\varphi$:

  Conhecimento relativo a algum agente do grupo sobre um fato φ.

  $(M, s) \models A_G\varphi$ se e somente se existe i tal que $i \in G$ e para todo t, se $(s, t) \in \mathcal{K}_i$ então $(M, t) \models \varphi$.

- Conhecimento distribuído, $D_G\varphi$:

  Unindo o conhecimento individual de todos os agentes do grupo, pode-se deduzir φ.

  $(M, s) \models D_G\varphi$ se e somente se para todo t, se $(s, t) \in \bigcap_{i \in G} \mathcal{K}_i$ então $(M, t) \models \varphi$.

- Conhecimento de todos, $E_G\varphi$:

  Todos os agentes do grupo tem conhecimento do fato φ.

$(M, s) \models E_G\varphi$ se e somente se para todo i tal que $i \in G$ então $(M, s) \models K_i\varphi$.

- Conhecimento comum, $C_G\varphi$:

  Um fato φ é de conhecimento comum em um grupo se e somente se φ é verdadeiro e, todo mundo no grupo sabe φ, todo mundo sabe que todo mundo sabe φ, todo mundo sabe que todo mundo sabe que todo mundo sabe φ, etc.

  Seja ${E_G}^0\varphi$ uma representação para φ, e seja ${E_G}^{k+1}\varphi$ uma representação para $E_G{E_G}^k\varphi$, para $k \geq 1$. Em particular, ${E_G}^1\varphi$ é uma representação para $E_G\varphi$.

  $(M, s) \models C_G\varphi$ se e somente se $(M, s) \models {E_G}^k\varphi$, para $k = 1, 2, 3, ...$

Considerando a adição dos operadores A_G, D_G, E_G e C_G na linguagem, a semântica para a lógica de conhecimento deve incluir as condições descritas acima, decorrentes da definição destes operadores. Como consequência, acrescentam-se os seguintes axiomas e regras no sistema axiomático:

- $K_i\varphi \rightarrow D_G\varphi$ (D1)

- $D_G\varphi \rightarrow D_{G'}\varphi$ se $G \subseteq G'$ (D2)

- $E_G\varphi \rightarrow \bigwedge_{i \in G} K_i\varphi$ (C1)

- $C_G\varphi \rightarrow E_G(\varphi \wedge C_G\varphi)$ (C2)

Existe ainda uma expressão que define uma hierarquia entre os operadores de estado de conhecimento:

- $C_G\varphi \rightarrow {E_G}^k\varphi \rightarrow E_G\varphi \rightarrow A_G\varphi \rightarrow D_G\varphi \rightarrow \varphi$ (H1)

E uma regra de inferência para conhecimento comum:

- Regra de Indução: de $\varphi \rightarrow E_G(\psi \wedge \varphi)$ deriva-se $\varphi \rightarrow C_G\psi$. (RC1)

$$\frac{\varphi \rightarrow E_G(\psi \wedge \varphi)}{\varphi \rightarrow C_G\psi}$$

Desta forma, a lógica de conhecimento permite expressar as relações de conhecimento e crença de um grupo de agentes sobre mundos predefinidos, o que viabiliza a modelagem estática do conhecimento dos agentes de um sistema. Ao se pensar em um sistema distribuído que evolui com o tempo, pode-se utilizar a lógica de conhecimento para representar o conhecimento dos agentes do sistema em um determinado instante no tempo.

2.1.4 Corretude e Completude

Nas seções anteriores, foram descritas propriedades básicas da lógica de conhecimento (como também os conceitos de conhecimento comum e conhecimento distribuído). Essas propriedades foram caracterizadas através de fórmulas válidas. É possível que existam propriedades adicionais que não sejam diretamente obtidas das propriedades listadas anteriormente. Assim, é necessário obter uma caracterização completa das propriedades do conhecimento através dos resultados de corretude e completude para o sistema modal de interesse neste estudo ($S5_m$).

Para caracterizar as propriedades de conhecimento nas estruturas de Kripke em termos de fórmulas válidas, é definida uma classe de modelos para as estruturas de Kripke de interesse (de acordo com as relações $\mathcal{K}_i$ descritas anteriormente). Esta classe de modelos possui a notação $\mathcal{M}_m(\Phi)$, onde m é o número de agentes sobre Φ, que por sua vez é o conjunto de primitivas da lógica, já descrito anteriormente. Ou simplesmente $\mathcal{M}_m$, por simplificação.

Supondo, por exemplo, que uma fórmula φ seja verdadeira em um estado s, de uma estrutura $M \in \mathcal{M}_m$. Então $(M, s) \vDash K_i\varphi$ se verifica caso φ também seja verdadeira em todos os estados t tal que $(s, t) \in \mathcal{K}_i$. De forma geral, é dito que φ é válida em relação ao modelo $\mathcal{M}$, escrito da forma $\mathcal{M} \vDash \varphi$, se φ é valida em todas as estruturas de $\mathcal{M}$. E analogamente, é dito que φ é satisfatível em relação ao modelo $\mathcal{M}$ se φ é satisfeita por alguma estrutura em $\mathcal{M}$.

Considerando a linguagem $\mathcal{L}_m$, é apresentado o seguinte teorema, bastante simples:

Teorema 2.1 *Para todas as fórmulas φ e $\psi \in \mathcal{L}_m$, estruturas $M \in \mathcal{M}_m$, e agentes $i = 1, ..., m$:*

(a) Se φ é uma tautologia da lógica proposicional então $\mathcal{M}_m \models \varphi$;

(b) Se $M \models \varphi$ e $M \models \varphi \rightarrow \psi$ então $M \models \psi$;

(c) $\mathcal{M}_m \models (K_i\varphi \wedge K_i(\varphi \rightarrow \psi)) \rightarrow K_i\psi$;

(d) Se $M \models \varphi$ então $M \models K_i\varphi$.

Prova 2.1 *A prova para os itens (a) e (b) segue diretamente da definição de $\models$ na lógica proposicional. E para os itens (c) e (d), a prova é obtida através das regras de inferência (R1 e R2) para a lógica de conhecimento.*

De acordo com o teorema 2.1, pode-se provar por indução (tendo como base φ), que se φ é provável em $S5_m$ então φ é válida em relação à $\mathcal{M}_m$. Assim é obtida a prova de corretude para $S5_m$ em relação à classe de modelos $\mathcal{M}_m$.

Para se obter a prova de completude, é preciso demonstrar que toda fórmula em $\mathcal{L}_m$ que é válida em relação à $\mathcal{M}_m$, é provável em $S5_m$. Para isso, basta provar que toda fórmula consistente com $S5_m$ em $\mathcal{L}_m$ é satisfatível em relação à $\mathcal{M}_m$. Supondo agora que seja possível provar isso, e que φ é uma fórmula válida em $\mathcal{L}_m$. Se φ não é provável em $S5_m$, então $\neg\neg\varphi$ também não pode ser provável em $S5_m$, assim, por definição, $\neg\varphi$ é consistente com $S5_m$. Segue então, que $\neg\varphi$ é satisfatível em relação à $\mathcal{M}_m$, contradizendo a validade de φ em relação à $\mathcal{M}_m$.

As provas completas de corretude e completude para vários sistemas axiomáticos da lógica de conhecimento podem ser vistas em (FAGIN, HALPERN, et al., 1995).

2.2 Programas Baseados em Conhecimento

Um programa para um sistema multi-agentes é uma coleção de entidades de código sequencial, cada uma sendo executada em um agente (ou mais de uma em um mesmo agente). Essas entidades correspondem ao que se costuma chamar de tarefas, *threads*, processos, etc. Em um sistema distribuído convencional, os agentes agem segundo programas que realizam suas ações de acordo com o resultado de testes que são aplicados a seu estado local. Esses testes, todavia, não envolvem

conhecimento dos agentes, e menos ainda as relações entre conhecimento e ações. O objetivo dos KBP's é permitir examinar as atividades de um sistema distribuído em relação ao conhecimento dos agentes presentes neste sistema, abstraindo detalhes de implementação. Um KBP oferece maior flexibilidade porque capta as relações entre conhecimento e ações, o que não é possível nos sistemas distribuídos comuns.

Para se construir um KBP, é preciso se modelar um protocolo que atenda uma especificação para um determinado sistema. Uma especificação, por sua vez, pode ser expressa de várias formas, através de um conjunto de propriedades a serem satisfeitas (geralmente relacionadas aos conceitos de *safety*, *fairness* e *liveness*), ou de uma maneira mais formal, utilizando um conjunto de expressões lógicas definidas de acordo com uma determinada linguagem, ou até mesmo através da definição de um conjunto de execuções a ser realizado pelo programa que implementa o protocolo. Todos esses conceitos serão abordados nas próximas seções e capítulos.

A cada instante de tempo, cada agente presente em um sistema possui um determinado estado. Esse estado é chamado de estado local do agente, o que é diferente do conceito de estado global do sistema. O estado global de um sistema é definido como uma tupla $(l_e, l_1, ..., l_m)$, onde l_e é o estado local do ambiente e l_i é o estado local do agente i, $i = 1, ..., m$. De forma que um estado global descreve um sistema em um determinado instante. Mas isso não é o suficiente, já que os sistemas multi-agentes não são entidades estáticas. Por isso, para estudar como esses sistemas sofrem alterações no tempo, é preciso criar um conceito que considere este fator. Assim, uma execução é uma função que relaciona o tempo com os estados globais do sistema e que será estudada no próximo capítulo.

2.2.1 Ações, Protocolos e Contextos

Nesta seção são apresentados alguns conceitos necessários ao desenvolvimento posterior de KBP's, com a introdução de alguns formalismos que serão utilizados para fornecer uma base para a estrutura de construção dos KBP's. Primeiramente, a partir de um estado inicial global, é intuitivo perceber que o sistema muda de estado através das ações executadas pelos agentes presentes atualmente no sistema e também pelo próprio ambiente. Além disso, os agentes executam as suas ações

seguindo um procedimento definido através de um protocolo, que pode ser visto então como uma regra para a escolha de ações do agente. E ainda, para descrever o comportamento geral do sistema, já que a influência do ambiente no sistema não é considerada pelos protocolos dos agentes, existe a necessidade de se estabelecer um "contexto" que irá definir todas as variáveis capazes de produzir alterações no sistema.

As ações dos agentes executadas em um sistema são geralmente relacionadas à troca de mensagens entre os mesmos, além de possivelmente algumas ações internas que são realizadas por cada agente. É definido que para cada agente i existe um conjunto ACT_i de ações que podem ser executadas por este agente. Por exemplo, em um sistema distribuído, a ação $send_j x$, corresponde ao envio do valor da variável x do agente i para o agente j. Esta ação poderia estar em ACT_i, se x realmente fosse uma variável local de i.

Tendo em vista o ambiente como um agente especial (mas lembrando que o seu estado de conhecimento não é de interesse), pode-se definir também um conjunto ACT_e de ações para o ambiente. Considerando as ações de troca de mensagens entre agentes, a entrega das mensagens poderia ser vista como uma ação do ambiente. E ainda existe a possibilidade de execução de uma ação especial para os agentes e o ambiente, chamada ação nula (Λ), que corresponde a nenhuma ação executada.

Conhecer somente as ações executadas por um agente em particular não é suficiente, na maioria dos casos, para se determinar o estado global do sistema. As ações executadas simultaneamente pelos diferentes agentes presentes no sistema interagem entre si e isto modifica o estado do sistema. Para tratar essas interações entre as ações, define-se uma ação conjunta (*joint action*). Uma ação conjunta é uma tupla da forma $(a_e, a_1, ..., a_m)$, onde a_e é uma ação executada pelo ambiente e a_i é uma ação executada pelo agente i.

As ações conjuntas mudam o estado do sistema através de um transformador de estado global, que simplesmente mapeia um estado global em outro. A mudança de estado do sistema ocorre então através da aplicação do transformador $\tau(a_e, a_1, ..., a_m)$ aplicado a um estado global $(l_e, l_1, ..., l_m)$, onde τ é uma função de transição aplicada a uma ação conjunta.

Como mencionado anteriormente, um protocolo para um agente i é a descrição das ações que este agente pode executar, como uma função do seu estado local. Formalmente, um protocolo P_i para um agente i é definido como sendo uma função do conjunto L_i de estados locais do agente i para conjuntos não-vazios de ações em ACT_i.

Assim como é considerada a ação realizada pelo ambiente, também será considerado um protocolo para o ambiente. O protocolo para o ambiente é definido como sendo uma função do conjunto L_e para sub-conjuntos não-vazios de ACT_e. Por exemplo, em um sistema de troca de mensagens, o protocolo do ambiente pode representar a possibilidade de que algumas mensagens se percam ou sejam entregues fora de ordem.

Um ponto a ser destacado é que um protocolo é uma função sobre estados locais, em vez de estados globais. Isto segue a intuição de que toda a informação que um agente possui está guardada em seu estado local. Assim, o que um agente realiza depende apenas do seu estado local, e não de todo o estado global. Além disso, é dito que enquanto um programa é considerado um objeto sintático (é descrito através de um texto), um protocolo é visto como um objeto semântico (é descrito através de uma função).

Os programas não rodam os protocolos isoladamente, isto é, a combinação de protocolos rodando para todos os agentes é que causa a evolução do sistema. Então é definido um protocolo conjunto (*joint protocol*) P como sendo a tupla $(P_1, ..., P_m)$ que consiste de cada protocolo P_i, para os agentes $i = 1, ..., m$. Deve-se notar que embora a ação do ambiente seja incluída na ação conjunta, o protocolo do ambiente não é incluído no protocolo conjunto. Isto acontece devido ao papel especial que o ambiente possui, já que os protocolos dos agentes são construídos e analisados de acordo com uma especificação, mas o protocolo de ambiente é predefinido e se mantém inalterado. Na verdade, o ambiente pode ser visto como uma adversidade quando se modela um sistema multi-agentes, fazendo com que este sistema se comporte de forma indesejada.

O protocolo conjunto P e o protocolo do ambiente descrevem o comportamento de todos os "participantes" do sistema, mas não é suficiente apenas descrever todas

as ações realizadas pelos agentes e pelo ambiente. Para determinar o comportamento do sistema, é preciso saber o "contexto" no qual o protocolo conjunto é executado. Então, o protocolo do ambiente P_e deve fazer parte deste contexto, já que ele determina a contribuição do ambiente nas ações conjuntas. A função de transição τ também deve ser incluída, porque ela descreve o resultado das ações conjuntas. Além disso, o contexto deve conter o conjunto $\mathcal{G}_0$ de estados globais iniciais, porque este conjunto descreve o sistema quando a execução do protocolo começa. Então agora já se pode descrever a variável chamada contexto, onde o protocolo conjunto P está sendo executado. Formalmente, um contexto γ é uma tupla $(P_e, \mathcal{G}_0, \tau, \Psi)$, onde Ψ é um conjunto de execuções admissíveis.

Dados P, P_e, τ e $\mathcal{G}_0$, pode-se gerar todas as execuções possíveis de (P_e, P), a partir de estados globais em $\mathcal{G}_0$. Porém, existem vezes em que não se deseja considerar todas as execuções geradas. Esse é o caso, por exemplo, de uma suposição tal como "todas as mensagens são entregues aos destinatários". Não se deseja considerar as execuções onde alguma mensagem não é entregue. Desta forma, se especifica um conjunto de execuções admissíveis, que garante que as condições apropriadas ocorram (por exemplo, todas as mensagens são entregues). Esse é o papel da condição Ψ no contexto.

Somente através de um contexto, é que um protocolo conjunto descreve o comportamento de um sistema. A combinação de um contexto γ e de um protocolo conjunto P para os agentes determina de forma única um conjunto de execuções. Essa formalização de conceitos será útil mais adiante quando será proposta uma espécie de prova de corretude (conhecida como corretude forte) para os KBP's, no Apêndice A.

Uma observação interessante citada em (HALPERN, 2000) diz que um programa com testes de conhecimento, ou seja, um KBP, pode ser visto como o resultado da especificação de um conjunto de sistemas que satisfazem uma determinada propriedade, enquanto que um programa *standard* (que não possui os testes de conhecimento) pode ser visto como o resultado da especificação de um conjunto de execuções que são consistentes com o programa.

2.2.2 Modelagem de KBP's

Uma linguagem simples para modelar programas que é capaz de descrever protocolos de agentes em sistemas que envolvem conhecimento e que também possui uma sintaxe voltada para a interação destes agentes, que executam ações baseadas em testes aplicados aos seus estados locais é proposta nesta seção.

Um modelo de algoritmo que servirá de base para a construção de programas é o seguinte:

```
repeat
  case of
    if t1 and/or k1 then a1
    if t2 and/or k2 then a2
    ...
  end case
until final
```

Figura 2.3: Modelo de algoritmo

Onde t_x são testes padrão (booleanos), k_x são testes de conhecimento (combinação booleana de $K_i\varphi$, onde φ pode ser qualquer fórmula) e a_x são sequências de ações a seres executadas pelos agentes.

Um KBP engloba elementos lógicos e ações comuns de programação. A linguagem utilizada deve portanto oferecer elementos para viabilizar a estrutura da programação procedural, e também para expressar as sentenças lógicas que são manipuladas na máquina de inferência da base de conhecimento. Nesta linguagem estão presentes:

1. As estruturas convencionais utilizadas em programação: **repeat**, **case of**, **if — then — else**, etc.

2. Os símbolos, operadores e pontuação da lógica de conhecimento para múltiplos agentes descrita anteriormente.

3. As funções *send* 'msg' e rec_j 'msg', que tratam o envio e recebimento, respectivamente, das mensagens trocadas entre os agentes cujo conteúdo é passado

como parâmetro. No caso da função de recebimento de mensagem rec_j, o subscrito j indica o agente de origem da mensagem. A função *send* também pode indicar um envio de mensagem a somente um agente específico através do uso de um subscrito.

Para o problema que será tratado neste trabalho (e também na maior parte das situações envolvendo agentes em sistemas distribuídos), a função *send* é sempre feita em *broadcast* e a função *rec* deve estar sempre relacionada a um agente específico, indicado pelo subscrito j.

2.2.3 Exemplo de Aplicação

Um exemplo de fácil entendimento e bastante simples em relação à questão de troca de mensagens entre agentes e que pode demonstrar as vantagens da utilização do algoritmo para modelagem de KBP's apresentado anteriormente é descrito a seguir.

O problema conhecido como transmissão de um bit (*Sequence Transmission Problem*) descreve como dois processos, *Sender* (S) e *Receiver* (R) se comunicam através de um canal. O processo *Sender* inicia com um bit, 0 ou 1, que deve ser enviado ao processo *Receiver*, porém a linha de comunicação é falha e pode perder mensagens.

Dessa forma, existe um protocolo que funciona da seguinte maneira. O processo S permanece enviando o bit até receber uma mensagem de *acknowledgement*, ack de R, e R permanece enviando mensagens ack depois de receber o bit.

- t_s: **if** $\neg$ *rec* ack **then** *send* bit
- t_r: **if** *rec* bit **then** *send* ack

A função do ack é informar a S que o bit foi recebido por R. Logo, outra forma de descrever o comportamento de S é permanecer enviando o bit até saber que R o recebeu.

- k_s: **if** $\neg K_r$ *rec* bit **then** *send* bit
- k_r: **if** *rec* bit $\wedge \neg K_s$ *rec* ack **then** *send* ack

A vantagem é que se pode abstrair a forma como os agentes chegam a esse conhecimento. Por exemplo, se o canal de R para S não tiver falhas, R pode enviar apenas um ack a S. O modelo permite abstrair ainda mais, o motivo pelo qual R permanece enviando o bit para S é que R quer que S saiba que R sabe o valor do bit. Seja K_r bit uma abreviação para K_r (bit = 0) ou K_r (bit = 1). O comportamento de S pode ser descrito por:

- k_s: **if** $\neg K_s K_r$ bit **then** *send* bit
- k_r: **if** K_r bit $\wedge \neg K_r K_s K_r$ bit **then** *send* ack

Esse programa abstrai até a forma como S descobre que R sabe o valor do bit, e a forma como R descobre que S sabe que R sabe o valor do bit. Se há garantia de que as mensagens são entregues no mesmo turno em que são enviadas, S envia o bit apenas uma vez, e R não envia nenhum *acknowledgement*. Se o valor do bit é de conhecimento comum, S não manda nenhuma mensagem.

No Capítulo 4, onde é estudado o problema das crianças com lama na testa, uma modelagem de um KBP para este problema será apresentada estendendo as definições da linguagem descrita aqui, fornecendo assim um exemplo de aplicação destes conceitos.

Um outro exemplo interessante de algoritmo para um KBP é apresentado em (STULP, VERBRUGGE, 2000), que também trata o chamado *Sequence Transmission Problem*, estudando o envio e recebimento de pacotes de mensagens entre agentes considerando o protocolo TCP/IP, utilizado na Internet.

Capítulo 3

Arquitetura de Sistemas Distribuídos

No capítulo anterior foi visto como os sistemas distribuídos envolvendo conhecimento evoluem no tempo segundo modificações dos estados de conhecimento dos agentes presentes. Desta forma, esses agentes incrementam seu conhecimento sobre o real estado do sistema a medida que eventos ocorrem e a comunicação acontece durante a computação.

Neste capítulo será apresentada uma metodologia para especificação formal de sistemas distribuídos e para o desenvolvimento de algoritmos para esta classe de sistemas, os sistemas de memória distribuída, onde os agentes só se comunicam através de trocas de mensagens. Essa é uma das características desejáveis para esse estudo, pois os agentes envolvidos em uma computação distribuída devem ser completamente independentes do ponto de vista físico, ou seja, não compartilham nenhum recurso além dos canais de comunicação da rede.

Nas seções seguintes será apresentado um modelo teórico de arquitetura e descrição dos sistemas de memória distribuída. A abordagem aqui será fundamentalmente teórica, dando relevância apenas às características comuns ao modelo conhecido como sistema de memória distribuída, (BARBOSA, 1996).

3.1 Sistemas de Memória Distribuída

Um sistema de memória distribuída é formado por um conjunto de processadores interconectados de alguma forma por uma rede de canais de comunicação (a topologia da rede de comunicação é definida de acordo com a configuração do

sistema considerado). Por definição, os processadores não compartilham nenhuma memória fisicamente, consequentemente toda a comunicação entre eles deve necessariamente acontecer por intermédio de trocas de mensagens através dos canais de comunicação da rede.

Um sistema de memória distribuída pode ser representado por um grafo $G = (M, E)$, onde o conjunto de nós M do grafo representa o conjunto de processadores e o conjunto de arestas E é o conjunto de canais ponto a ponto interligando os processadores. Dois processadores se comunicam diretamente somente se existe um canal de comunicação entre eles. Caso contrário, a comunicação se dá em função do repasse de mensagens sobre um caminho entre os processadores de origem e destino, através dos canais da rede. Cada processador no sistema deve ser dotado de capacidades especiais para lidar com o tráfego de mensagens na rede que passa através de seus canais de comunicação adjacentes, sem interferir no processamento local.

A seguir, na Figura 3.1 é mostrado um exemplo de um grafo composto por processadores (ou nós) $M = \{1, 2, 3, 4, 5\}$ e canais de comunicação $E = \{(1, 2), (1, 4), (2, 3), (3, 5), (4, 1), (5, 2), (5, 4)\}$.

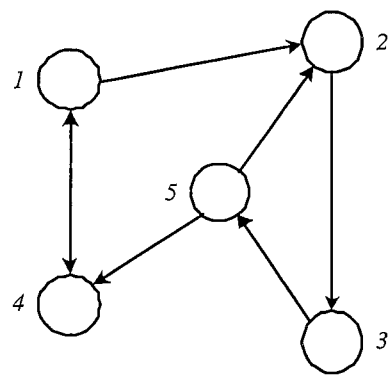


Figura 3.1: Grafo $G = (M, E)$ para um sistema de memória distribuída

Na arquitetura apresentada, o agente é representado como uma estrutura composta por dois processadores e uma base de conhecimento, onde o processador principal (*host*) é responsável pela execução dos processos, o segundo processador é responsável pelo gerenciamento da comunicação, e a base de conhecimento armazena todo o conhecimento do agente e é capaz de fazer inferência sobre os dados armazenados.

As mensagens caminham na rede através dos canais e dos processadores de comunicação, que encaminham as mensagens que recebem ou para o próprio processador principal do agente ou para outro processador de comunicação da rede, de acordo com o destino da mensagem. Esse repasse de mensagens acontece em cada processador de acordo com a função de roteamento definida, de forma que a comunicação seja efetiva e confiável, sem que o processador principal tome conhecimento do processo de comunicação. Por parte dos processos, tudo acontece como se houvesse um canal de comunicação FIFO (*first in, first out*) entre qualquer par de agentes do sistema.

A cada ponto no tempo, um agente está em algum estado local que reflete a informação disponível para ele. Como já visto no capítulo anterior, o estado global do sistema a cada instante é dado pelo conjunto de estados locais dos agentes e mais o estado do ambiente no instante considerado.

Cada processador executa individualmente seus programas, e a execução em paralelo de programas afins nos processadores dita o comportamento do sistema distribuído, o que fornece suporte necessário para a classe de sistemas em estudo neste trabalho, que envolve conhecimento entre agentes.

Um sistema distribuído pode ser síncrono ou assíncrono. Em sistemas síncronos todos os processadores recebem sinais de sincronismo de um relógio global, enquanto num sistema assíncrono não há nenhum agente no sistema provendo este tipo de informação.

3.2 Execuções e Passos

No capítulo anterior foram definidos conceitos como ações, protocolos e estados, sendo que estes últimos podem ser locais (para cada agente no sistema) e globais (para o sistema como um todo). Um estado global do sistema pode ser definido, segundo (FAGIN, HALPERN, et al., 1995), através de uma tupla $(l_e, l_1, ..., l_m)$, onde l_e é o estado do ambiente e l_i é o estado local para o agente i, $i = 1, ..., m$.

Para se descrever a evolução de um sistema no tempo, já que um estado global apenas referencia o sistema em um determinado instante no tempo, existe um conceito chamado execução (*run*). Uma execução é definida como uma função que

relaciona o tempo com os estados globais do sistema. Assim, uma execução é uma descrição completa de como o estado global do sistema evolui com o tempo. Considerando que o tempo varia de acordo com a sequência de estados globais do sistema, define-se $r(0)$ como o estado global inicial de uma possível execução r, $r(1)$ como o próximo estado global e assim por diante.

Um sistema pode ter quantas execuções forem possíveis, já que o seu estado global pode evoluir de várias maneiras possíveis a partir do estado global inicial. Assim, um sistema pode ser formalmente definido como um conjunto não-vazio de execuções. Isto pode abstrair inclusive a noção de sistema como um conjunto de agentes que interagem entre si. Assim, ao invés de se modelar diretamente o sistema, pode-se modelar os possíveis comportamentos do sistema. E como a definição de sistema requer um conjunto não-vazio de execuções, sempre existirá pelo menos um comportamento para o sistema.

Seja então L_e um conjunto dos estados possíveis para o ambiente e L_i um conjunto dos estados locais possíveis para o agente i, $i = 1, ..., m$. Dessa forma, $\mathcal{G} = L_e \times L_1 \times ... \times L_m$ é o conjunto dos estados globais possíveis. Então uma execução sobre $\mathcal{G}$ pode ser identificada como uma sequência de estados globais pertencentes a $\mathcal{G}$. Se $r(n) = (l_e, l_1, ..., l_m)$ é o estado global no ponto (r, n), então define-se $r_e(n) = l_e$ e $r_i(n) = l_i$ para $i = 1, ..., m$, ou seja, $r_i(n)$ é o estado local do agente i no ponto (r, n). Um passo (*round*) ocorre entre dois pontos consecutivos de uma mesma execução. Desta forma, é conveniente considerar que um agente executa uma ação em cada passo. Por fim, um sistema $\mathcal{R}$ em $\mathcal{G}$ é um conjunto execuções sobre $\mathcal{G}$. E (r, n) é um ponto no sistema $\mathcal{R}$ se $r \in \mathcal{R}$. No processo de modelagem de um sistema, um conjunto apropriado de execuções é escolhido para que as especificações desejadas para este sistema sejam satisfeitas.

3.3 Sistemas Síncronos

Neste tipo de sistema cada processador recebe o sinal do relógio global indicando o fim de um pulso de tempo e o início de outro, o que pressupõe que cada processo tem consciência de que todos os processadores envolvidos no sistema possuem a mesma informação temporal sobre o estado global do sistema, ou seja, todos

concordam sobre o pulso em que estão, a cada pulso.

As seguintes propriedades temporais descrevem a arquitetura de um sistema síncrono:

- Todos os nós são controlados por uma mesma base de tempo global referida como o relógio global do sistema distribuído, que gera intervalos de tempo de tamanho fixo maior que zero;

- O tempo de entrega de uma mensagem entre nós vizinhos é necessariamente não nulo e estritamente menor que a duração de um intervalo do relógio global.

O início de cada intervalo de tempo é indicado por um "pulso" do relógio global (*clock*). Para um inteiro $s \geq 0$, o pulso s indica o início do passo s. No pulso $s = 0$ os nós que devem iniciar o processamento no sistema distribuído enviam mensagens a um sub-conjunto (possivelmente vazio) de seus canais de comunicação adjacentes. No pulso $s > 0$ todas as mensagens enviadas no pulso $s - 1$ já foram recebidas por seus destinatários (de acordo com a especificação do modelo), e então os nós do sistema podem realizar os seus processos e as devidas trocas de mensagens entre eles.

Para garantir as propriedades do modelo síncrono assume-se a hipótese de que o tempo de processamento realizado pelos nós durante um intervalo é nulo, ou seja, considera-se que os processadores dos agentes são máquinas perfeitas. Sem essa hipótese, a duração do intervalo poderia não ser suficiente para acomodar a entrega das mensagens e o processamento local.

Na arquitetura síncrona, um conjunto de nós pode estar apto a enviar mensagens no pulso $s = 0$, já que as propriedades deste modelo permitem que os agentes realizem processamento independentemente do recebimento de qualquer mensagem, pois a existência de um relógio global possibilita esta ação sem a necessidade de espera pela chegada de mensagens de outros agentes. Porém, para que o processamento global do sistema não seja simplesmente um mero paralelismo de tarefas, ao menos uma mensagem deve ser enviada por pelo menos um nó do sistema durante a execução de um algoritmo distribuído.

Apesar de possuir uma essência bastante teórica, o modelo síncrono desperta

um grande interesse no desenvolvimento de algoritmos distribuídos, pois a sua simplicidade viabiliza o desenvolvimento de algoritmos complexos. Outra propriedade interessante que pode ser utilizada no desenvolvimento de algoritmos no modelo síncrono é o fato de que os nós podem ganhar informação temporal acerca do estado do sistema apenas esperando, isto é, contando os pulsos. Isto significa que há troca de informação entre as tarefas mesmo que não haja troca de mensagens.

3.4 Sistemas Assíncronos

Neste caso, não existe uma entidade sincronizadora, o que torna os processos a princípio incapazes de um consenso sobre o tempo global. Para manter um estado global consistente, os programas para esse tipo de sistemas mantêm uma espécie de comunicação sincronizadora entre os processadores, informando uns aos outros o estado temporal local e esperando um retorno (*feedback*) dos estados temporais dos outros processadores.

A arquitetura de um sistema assíncrono é caracterizada pelas seguintes propriedades:

- Cada nó é controlado por sua própria base de tempo, denominada relógio local. Esta é uma noção temporal local do agente, e independe de qualquer outro fator do sistema;

- O tempo total de envio de uma mensagem de um nó a outro adjacente através de um dos canais de comunicação da rede é finito porém imprevisível, ou seja, as mensagens chegam em algum momento futuro após serem enviadas, mas não há como prever o atraso na entrega.

No modelo assíncrono, exceto possivelmente no início da execução do algoritmo, o processamento em um nó só acontece como consequência do recebimento de mensagens. No caso de ocorrer a chegada de mais de uma mensagem ao mesmo tempo, estas são aceitas de forma não determinística. Todos os agentes tem consciência de que após algum tempo não infinito no futuro as mensagens que foram enviadas serão recebidas por seus destinatários.

Nenhuma informação temporal é utilizada que referencie a noção de tempo

global do sistema (pois tal noção não existe no modelo assíncrono). Assim como no caso síncrono pode existir aqui a possibilidade de um conjunto de nós enviar mensagens no início do sistema, porém isto ocorre de forma mais restrita. O processamento em um nó pode ser entendido ou como um conjunto de ações a serem tomadas inicialmente se o nó em questão deve iniciar o seu processamento e enviar mensagens espontaneamente, ou como um conjunto de ações que devem ser realizadas quando do recebimento de alguma mensagem, além da verificação de uma determinada situação através de algumas condições booleanas.

O modelo assíncrono está mais próximo das condições reais de funcionamento dos sistemas de memória distribuída em computação, porém suas características temporais acarretam muitas limitações na modelagem de algoritmos para esta classe de sistemas.

Considerando as propriedades de ambos os modelos, pode-se concluir que todo algoritmo assíncrono é também um algoritmo síncrono, isto é, se um algoritmo foi projetado para o modelo assíncrono e funciona corretamente sob as hipóteses deste modelo, então ele também funcionará corretamente sob as hipóteses do modelo síncrono para a escolha de uma duração de intervalo apropriada (capaz de acomodar os processamentos dos nós). Isso acontece porque as condições em que a comunicação acontece em um modelo síncrono está contida em uma das infinitas possibilidades que o modelo assíncrono permite. O inverso desta implicação (isto é, que o algoritmo síncrono executa corretamente no modelo assíncrono) pode ser atingido com uma transformação apropriada no algoritmo.

3.5 Modelo de Algoritmo

É apresentado nesta seção um modelo genérico de alto nível para um algoritmo de um sistema de memória distribuída. A estrutura do modelo permite analisar o procedimento computacional e a comunicação das tarefas, e consequentemente demonstra como a execução de várias instâncias do algoritmo influi no funcionamento como um todo do sistema de memória distribuída. É importante perceber o papel que as trocas de mensagens representam nesta classe de sistemas, em especial no controle do fluxo de computação de uma tarefa.

Para simplificar a notação utilizada, um algoritmo distribuído será representado por um grafo conexo direcionado $G_T = (N_T, D_T)$, onde o conjunto N_T é um conjunto de n tarefas e o conjunto de arestas direcionadas D_T é um conjunto de canais unidirecionais de comunicação. Para uma tarefa t, define-se $In_t \subseteq D_T$ como o conjunto de arestas direcionadas a t, e $Out_t \subseteq t$ o conjunto de arestas direcionadas de t para outros nós.

O algoritmo *Task_t* (Figura 3.2) apresentado a seguir descreve o comportamento de uma tarefa genérica t. Apesar do processo de envio de mensagens estar ao final do processamento, isto não é necessário, o processamento e a troca de mensagens podem estar intercalados no conjunto de ações sequenciais da tarefa.

Uma tarefa t é reativa ou regida por mensagens no sentido que ela normalmente só executa algum processamento (incluindo o envio de mensagens para outras tarefas) em resposta ao recebimento de uma mensagem de outra tarefa. Uma exceção a essa regra é que pelo menos uma tarefa deve ser capaz de enviar mensagens espontaneamente no início de sua execução, para iniciar o processamento no sistema para as outras tarefas. Além disso, as tarefas podem inicialmente fazer processamentos de inicialização.

De acordo com o algoritmo a tarefa t é executada até que uma condição global de terminação seja alcançada. Enquanto isso não acontece, o algoritmo permanece realizando comandos no laço de repetição. A cada passo, o algoritmo executa algum processamento e também pode enviar mensagens. O processamento executado por um passo corresponde a um grupo de comandos agrupados sob uma guarda. E por sua vez, a partir de alguma condição booleana B_k, onde $1 \leq k \leq n$, uma guarda é uma condição da forma:

if *receive* mensagem em $c_k \in In_t$ **and** B_k **then**

Uma guarda é satisfeita quando existe uma mensagem disponível para recebimento imediato no canal c_k e além disto a condição booleana B_k é verdadeira. Essa condição pode ou não depender da mensagem disponível para recebimento. A cada passo do algoritmo, apenas uma guarda dentre as que possam ser satisfeitas deve ser executada. Se nenhuma guarda for satisfeita, a tarefa é suspensa até que alguma

```
algorithm Task_t
  processamento inicial
  send mensagem para todos os canais pertencentes a um sub-conjunto
                                                  (possivelmente vazio) de Out_t
  repeat
    case of
      if receive mensagem em c_1 ∈ In_t and B_1 then
        processamento
        send mensagem para todos os canais pertencentes a um
                                                  sub-conjunto (possivelmente vazio) de Out_t
      ...
      if receive mensagem em c_n ∈ In_t and B_n then
        processamento
        send mensagem para todos os canais pertencentes a um
                                                  sub-conjunto (possivelmente vazio) de Out_t
    end case
  until condição global de terminação
end
```

Figura 3.2: Modelo de algoritmo

possa ser satisfeita. Se mais de uma pode ser satisfeita, então uma dentre elas é escolhida arbitrariamente.

O algoritmo apresentado aborda todos os aspectos que devem ser levados em conta ao se desenvolver um sistema distribuído onde a comunicação é totalmente realizada através de troca de mensagens. O modelo genérico pode facilmente ser adaptado ao se reunir as informações sobre as peculiaridades do sistema a ser desenvolvido e as propriedades e limitações da plataforma de implementação escolhida.

3.6 Eventos e Ordens

É utilizado um formalismo baseado em eventos para descrever o processamento distribuído que irá permitir o tratamento de propriedades globais, principalmente no que se refere a propriedades temporais no caso assíncrono. O modelo a ser estudado inicialmente será o síncrono, pois este modelo é bem mais simples de ser descrito por conter uma restrição temporal no que se refere a ocorrência de eventos, podendo ser considerado inclusive como um caso particular do modelo assíncrono.

O conceito de evento aplicado aqui é corresponde a uma unidade fundamen-

tal de um processamento distribuído, que por sua vez corresponde a uma execução do sistema. Um processamento distribuído então é visto simplesmente como um conjunto de eventos, denotado por Ξ. E um evento ξ é uma tupla da forma $(n_i, t, \varphi, \sigma, \sigma', \Phi)$, onde:

- n_i é o nó onde o evento ocorre;
- t é o tempo dado pelo relógio local de n_i em que o evento ocorreu;
- φ é a mensagem, se houver alguma, que acionou o evento com seu recebimento em n_i;
- σ é o estado de n_i anterior à ocorrência do evento;
- σ' é o estado de n_i imediatamente posterior à ocorrência do evento;
- Φ é o conjunto de mensagens, se houver um, enviadas por n_i como consequência da ocorrência do evento.

O evento é considerado também como a unidade básica temporal do modelo assíncrono. São chamados de eventos internos os eventos que ocorrem sem nenhuma causa imediata externa, ou seja, $\varphi = 0$. Esse tipo de evento representa uma recepção de mensagem ou a iniciação espontânea do processamento em um determinado nó.

Já que os sistemas considerados aqui sempre possuem trocas de mensagens, os eventos então refletem o recebimento ou envio de uma mensagem, o que significa que um evento de envio de mensagem num modelo sem falhas de comunicação necessariamente acarreta um evento de recebimento de mensagem para outro nó.

Para estabelecer uma ordem temporal entre a ocorrência dos eventos em um processamento distribuído, é definida uma relação binária, $\prec$ da seguinte forma:

Sejam ξ_1 e ξ_2 dois eventos. Então $\xi_1 \prec \xi_2$ se e somente se:

(i) Ambos ξ_1 e ξ_2 ocorrem no mesmo nó, respectivamente nos instantes t_1 e t_2 tais que $t_1 < t_2$. Nenhum outro evento ocorre no mesmo nó em um instante t tal que $t_1 < t < t_2$.

(ii) Os eventos ξ_1 e ξ_2 ocorrem em nós vizinhos, e existe uma mensagem φ que é enviada em ξ_1 e recebida em ξ_2.

O significado da relação binária $\prec$ é "ξ_1 ocorreu imediatamente antes de ξ_2", o que só faz sentido quando são considerados eventos em uma mesma execução. Assim, a relação $\prec$ é acíclica.

Com isso, uma execução Ξ pode ser visualizada através de um grafo $H = (\Xi, \prec)$, chamado grafo de precedência. Os nós de H são eventos de Ξ, e as arestas direcionadas representam a relação $\prec$. Em consequência da definição da relação $\prec$, o grafo H é direcionado e acíclico.

É visto na Figura 3.3 a seguir um grafo de precedência com quatro nós. O grafo de precedência permite uma melhor visualização ao se representar os eventos associados a um mesmo nó em uma linha horizontal, na ordem dada pela relação $\prec$. Nessa representação, as arestas horizontais correspondem a eventos internos para um nó, enquanto as outras arestas relacionam eventos correspondentes de envio e recebimento de mensagens entre dois nós.

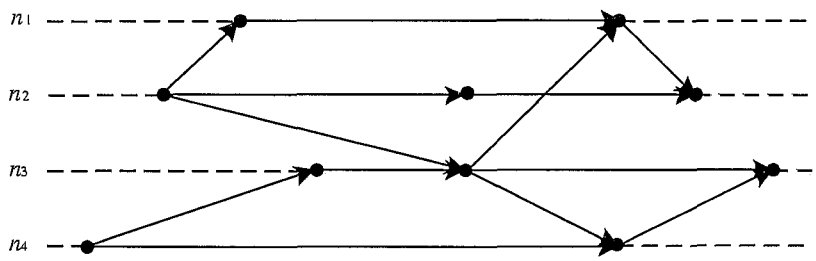


Figura 3.3: Grafo de precedência

Em função de $\prec$ pode-se definir uma segunda relação binária $\prec^+$, tal que $\prec^+$ é o fecho irreflexivo e transitivo de $\prec$, estabelecendo uma ordem parcial no conjunto de eventos Ξ. Dois eventos que não se relacionam em $\prec^+$ são ditos concorrentes.

E além de estabelecer o conceito de concorrência entre eventos, a relação $\prec^+$ pode ser usada para definir conceitos de grande interesse, tais como o passado e o futuro de um evento em relação a uma determinada execução. Assim, para um evento ξ, tem-se que:

$$Past(\xi) = \{\xi' | \xi' \in \Xi \, e \, \xi' \prec^+ \xi\}$$
$$Future(\xi) = \{\xi' | \xi' \in \Xi \, e \, \xi \prec^+ \xi'\}$$

Estes dois conjuntos introduzem duas regiões ao redor do evento ξ no grafo de precedência que demonstram os eventos que influenciam ξ e os eventos que são influenciados por ξ de forma causal. A Figura 3.4 representa dois conjuntos $\{\xi\} \cup Past(\xi)$ delimitado a esquerda e $\{\xi\} \cup Future(\xi)$ delimitado a direita, para o evento ξ escolhido no grafo de precedência.

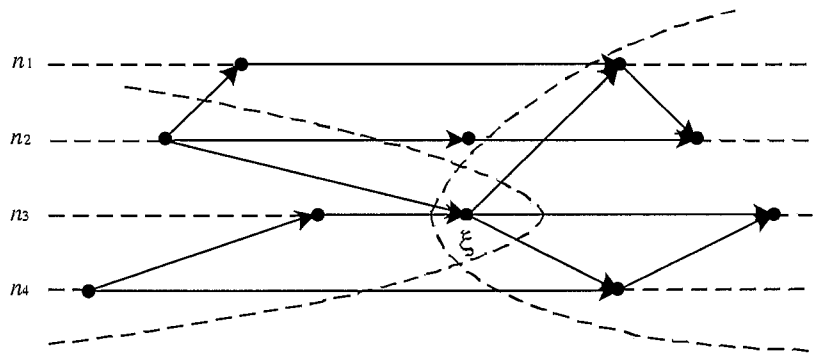


Figura 3.4: Representação de conjuntos em um grafo de precedência

Com a formalização apresentada para descrever as relações de precedência entre eventos de um processamento, é proposta uma nova definição para o estado global do sistema que envolve as propriedades temporais descritas, o que é de especial importância para o caso assíncrono.

Primeiramente, utiliza-se o conceito de estado do sistema, que é definido simplesmente como um conjunto de m estados locais, um para cada nó, e mais um estado para cada aresta. Os estados m_i dos nós são os estados locais de cada agente, retirados diretamente da sequência de estados produzida no processamento distribuído do sistema. Os estados das arestas $(m_i \rightarrow m_j)$ são formados por um conjunto de mensagens que representam a mudança de estado do sistema, ou seja, as mensagens que foram enviadas por m_i na aresta $(m_i \rightarrow m_j)$ mas que ainda não foram recebidas por m_j. Este conceito não possui muita abrangência, mas servirá como auxiliar na definição de estado global do sistema, que é apresentada agora.

Para definir um estado global consistente, é necessário estabelecer uma ordem total $<$ em Ξ consistente com $\prec^+$. Os $\Xi - 1$ pares de eventos consecutivos $(\xi_1, \xi_2) \in <$

podem ser identificados de forma que todo evento $\xi \neq \xi_1$, ξ_2 é tal que $\xi < \xi_1$ ou $\xi_2 < \xi$. Associado a cada par (ξ_1, ξ_2) de eventos consecutivos em $<$ está um estado do sistema denotado por (ξ_1, ξ_2) com as seguintes características:

- Para cada nó m_i, o seu estado é resultante da ocorrência do evento mais recente (com o maior tempo de ocorrência) em m_i, por exemplo ξ, tal que $\xi_1 \not< \xi$ (inclusive $\xi = \xi_1$).

- Para cada aresta $(m_i \rightarrow m_j)$, o seu estado é o conjunto de mensagens enviadas em conexão com um evento ξ tal que $\xi_1 \not< \xi$ (inclusive $\xi = \xi_1$) e recebidas em conexão com um evento ξ' tal que $\xi' \not< \xi_2$ (inclusive $\xi' = \xi_2$).

Um estado do sistema é global se e somente se ou todos os nós estão em seu estado inicial e todas as arestas estão vazias, ou todos os nós estão em seu estado final e todas as arestas estão vazias, ou existe uma ordem total $<$ consistente com $\prec^+$ na qual este estado do sistema é formado por um par (ξ_1, ξ_2) de eventos consecutivos.

Uma outra definição para estado global pode ser construída da seguinte forma, um estado do sistema é global se e somente se ele pode ser representado por uma partição (Ξ_1, Ξ_2) de Ξ tal que $Past(\xi) \subseteq \Xi_1$ e $Future(\xi) \subseteq \Xi_2$.

Por simplicidade, o estado global pode também ser referido por sua partição. As partições que obedecem a esta restrição são chamadas de cortes consistentes.

De forma análoga a definição anterior de estado global do sistema, pode-se também definir um estado global consistente representado pela partição (Ξ_1, Ξ_2), quando esta é um corte consistente, da seguinte forma:

- Para cada agente m_i, o seu estado local é decorrente do evento de Ξ_1 mais recente ocorrendo em n_i

- Para cada aresta $(m_i \rightarrow m_j)$, o seu estado local é formado por um conjunto de mensagens enviadas por m_i através de eventos de Ξ_1 e recebidas por m_j através de eventos de Ξ_2.

Os cortes consistentes apresentados sobre um grafo de precedência podem ser vistos na Figura 3.5. Nesta figura são identificadas duas partições Ξ_a e Ξ_b, sendo que a primeira corresponde a um corte consistente, enquanto que na segunda há

uma aresta do futuro para o passado, de forma que esta não pode ser considerada um corte consistente.

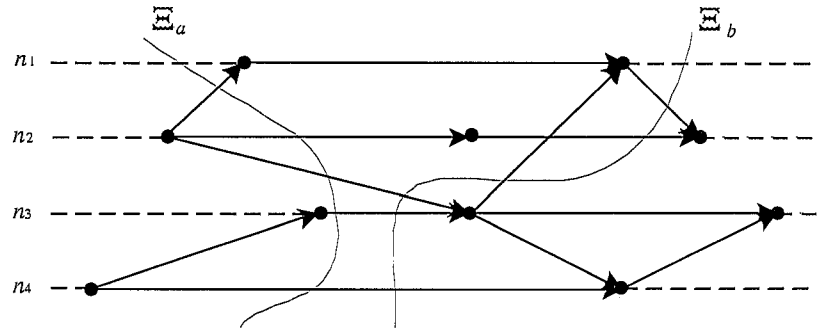


Figura 3.5: Representação de partições em um grafo de precedência

A partir das definições de estado global consistente, pode-se estender o conceito de passado e futuro de um evento para passado e futuro em relação a um estado global Ψ:

$$Past(\Psi) = \bigcup_{\xi \in \Xi_1} [\{\xi\} \cup Past(\xi)]$$
$$Future(\Psi) = \bigcup_{\xi \in \Xi_2} [\{\xi\} \cup Future(\xi)]$$

Onde Ψ é o estado global do sistema representado pela partição (Ξ_1, Ξ_2). Assim, pode-se afirmar que um estado global Ψ_1 antecede um outro estado global Ψ_2 em uma execução Ξ se e somente se $Past(\Psi_1) \subset Past(\Psi_2)$ ou $Future(\Psi_2) \subset Future(\Psi_1)$.

Para ilustrar a funcionalidade dos conceitos de execuções e cortes consistentes sobre um grafo de precedência, é visto agora um exemplo de sistema multi-agentes bem conhecido, com três agentes rodando o algoritmo distribuído para propagação da informação com realimentação (PIF - *Propagation of Information with Feedback*).

O algoritmo distribuído para PIF é bastante simples e funciona basicamente da seguinte forma. O agente iniciador (a_1) envia uma mensagem φ a todos os outros agentes do sistema. Cada agente, ao receber φ, envia novamente φ para todos os agentes, exceto aquele do qual recebeu φ.

Primeiramente, é definido o conjunto $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ de eventos que caracterizam o envio de mensagens entre os agentes (os eventos têm o mesmo

significado independente da execução). Os conjuntos E_m de eventos possíveis para um agente m são $E_1 = \{e_1, e_6, e_7\}, E_2 = \{e_2, e_5\}, E_3 = \{e_3, e_4\}$.

Tendo o agente a_1 como iniciador, são obtidas seis execuções possíveis $(r_1, r_2, r_3, r_4, r_5, r_6)$, conforme a Figura 3.6. As execuções resultam das possíveis ordenações de acontecimentos dos eventos para cada agente.

Assim, os possíveis estados globais (ou cortes consistentes) que ocorrem nas seis execuções e as suas respectivas visões passadas e futuras são:

$$
\begin{aligned}
&c_0 : V_p = \{\,\}; V_f = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}\\
&c_1 : V_p = \{e_1\}; V_f = \{e_2, e_3, e_4, e_5, e_6, e_7\}\\
&c_2 : V_p = \{e_1, e_2\}; V_f = \{e_3, e_4, e_5, e_6, e_7\}\\
&c_3 : V_p = \{e_1, e_2, e_3\}; V_f = \{e_4, e_5, e_6, e_7\}\\
&c_4 : V_p = \{e_1, e_2, e_3, e_4\}; V_f = \{e_5, e_6, e_7\}\\
&c_5 : V_p = \{e_1, e_2, e_3, e_4, e_5\}; V_f = \{e_6, e_7\}\\
&c_6 : V_p = \{e_1, e_2, e_3, e_4, e_5, e_6\}; V_f = \{e_7\}\\
&c_7 : V_p = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}; V_f = \{\,\}\\
&c_8 : V_p = \{e_1, e_2, e_3, e_6\}; V_f = \{e_4, e_5, e_7\}\\
&c_9 : V_p = \{e_1, e_2, e_3, e_6, e_4\}; V_f = \{e_5, e_7\}\\
&c_{10} : V_p = \{e_1, e_2, e_3, e_4, e_5, e_7\}; V_f = \{e_6\}\\
&c_{11} : V_p = \{e_1, e_4\}; V_f = \{e_2, e_3, e_5, e_6, e_7\}\\
&c_{12} : V_p = \{e_1, e_2, e_4, e_5\}; V_f = \{e_3, e_6, e_7\}\\
&c_{13} : V_p = \{e_1, e_2, e_4\}; V_f = \{e_3, e_5, e_6, e_7\}\\
&c_{14} : V_p = \{e_1, e_2, e_4, e_5, e_7\}; V_f = \{e_3, e_6\}\\
&c_{15} : V_p = \{e_1, e_4, e_5\}; V_f = \{e_2, e_3, e_6, e_7\}\\
&c_{16} : V_p = \{e_1, e_4, e_5, e_7\}; V_f = \{e_2, e_3, e_6\}
\end{aligned}
$$

Uma vez definidos os estados globais, é possível estabelecer estruturas Kripke sobre sistemas distribuídos síncronos, como visto no Capítulo 2. Para definir estruturas Kripke sobre modelos assíncronos, devem ser utilizadas as definições de estados globais e ordenação parcial apresentadas neste capítulo, o que torna o modelo mais complexo.

Os conceitos demonstrados neste capítulo introduzem uma noção necessária para o entendimento do comportamento de agentes em sistemas distribuídos envolvendo conhecimento. Este estudo será útil mais a frente, na apresentação de uma nova linguagem para a construção de KBP's e na introdução dos métodos de tradução vistos no Capítulo 5.

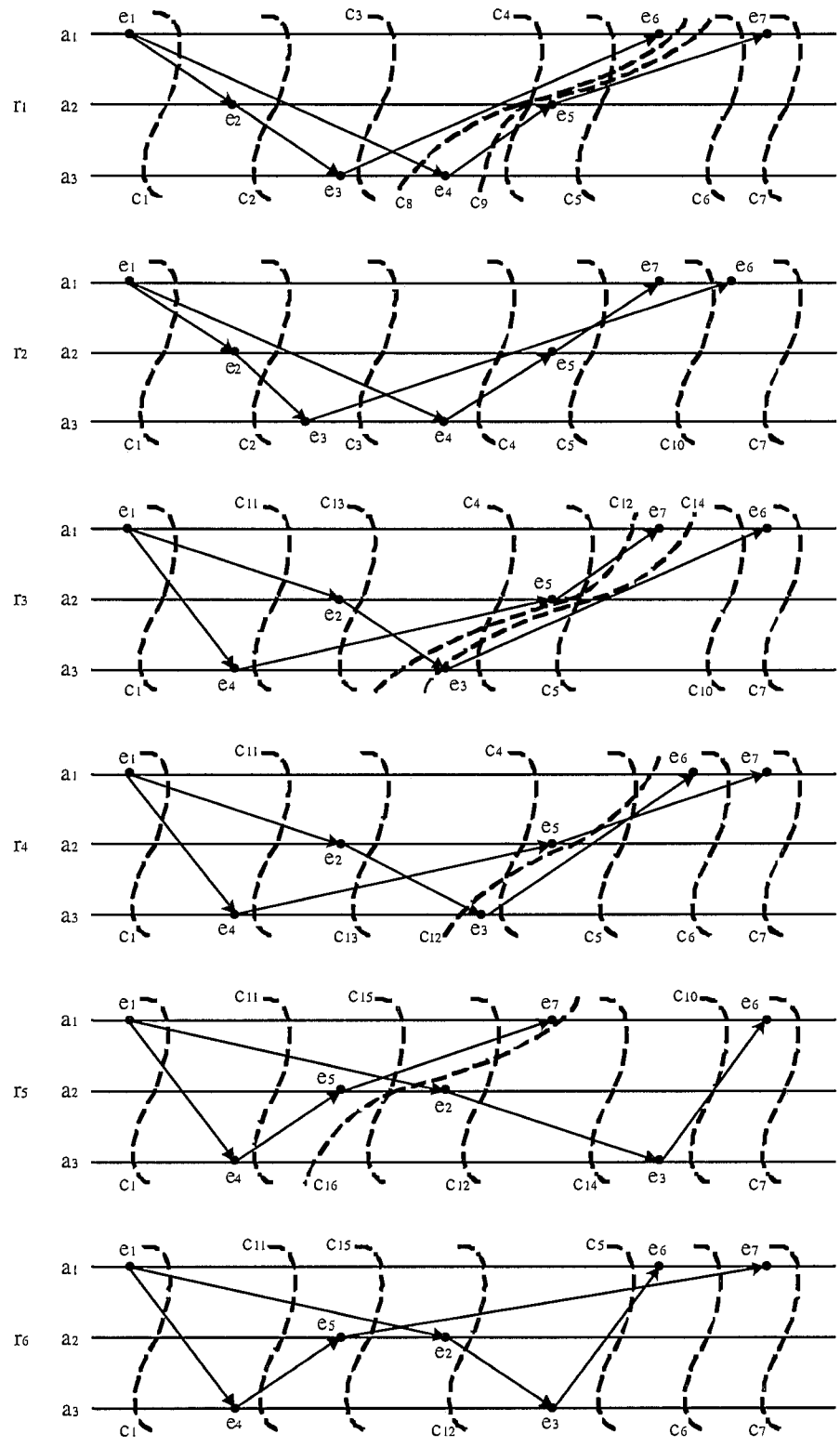


Figura 3.6: Possíveis execuções de PIF para três agentes

Capítulo 4

Especificação de Sistemas Multi-Agentes

Neste capítulo é introduzida a estrutura que será utilizada na construção de KBP's para especificação de sistemas multi-agentes baseados em conhecimento. Para isso, é estudado o problema das crianças com lama na testa (*Muddy Children Puzzle*), que descreve um exemplo clássico desse tipo de sistema.

4.1 O Problema das Crianças com Lama na Testa

O problema das crianças com lama na testa é bastante conhecido na área de lógica epistemológica, sendo as vezes apresentado também através de uma variante conhecida como o problema dos homens sábios (*Wise-Men Puzzle*). O seu enunciado é o seguinte, (FAGIN, HALPERN, et al., 1995):

> Imagine n crianças brincando juntas. A mãe destas crianças disse a elas para não se sujarem, ou elas seriam punidas. Acontece que durante a brincadeira algumas crianças, k delas, se sujam na testa. Cada uma delas pode ver a lama na testa das outras crianças, mas não na sua própria testa. Então ninguém fala nada. Assim, o pai das crianças chega e diz: "Pelo menos um de vocês está com a testa suja", expressando um fato já conhecido por cada criança (se $k > 1$). O pai a seguir faz a seguinte pergunta, repetidamente: "Alguém sabe se está com lama na sua própria testa?" Considerando que as crianças têm boa percepção, são inteligentes, não mentem e respondem às perguntas simultaneamente, o

que irá acontecer?

É possível provar que nas primeiras $k-1$ perguntas do pai, todas as crianças responderão 'Não', mas na k-ésima pergunta, as crianças que estiverem com lama na testa responderão 'Sim'.

Pode-se elaborar uma prova por indução no número de crianças com a testa suja k. Para $k = 1$ o resultado é imediato, a criança com a testa suja não vê nenhuma outra que também possua a testa suja, então ela própria se identifica como tendo a testa suja. Com $k = 2$, existem somente duas crianças com a testa suja, a e b. Elas respondem 'Não' na primeira pergunta, já que cada uma vê a testa suja da outra. Mas quando b diz 'Não', a descobre que ela deve estar com a testa suja, senão b saberia que estava com a testa suja e teria respondido 'Sim' na primeira pergunta. Assim a responde 'Sim' na segunda pergunta. E b também utiliza o mesmo raciocínio. Agora, com $k = 3$, existem três crianças, a, b e c, com a testa suja. A criança a imagina que não está com a testa suja. Então, pelo caso $k = 2$, ela também imagina que b e c responderão 'Sim' na segunda pergunta. Mas quando isto não acontece, ela descobre que a sua suposição era falsa e que ela também está com a testa suja, respondendo 'Sim' na terceira pergunta. O mesmo acontece para b e c. E similarmente para o caso geral.

Voltando às estruturas de Kripke estudadas no Capítulo 2, para se elaborar uma estrutura deste tipo para o problema das crianças, primeiro será considerada a situação inicial, antes do pai das crianças se expressar. Assim, pode-se descrever uma possível situação do problema através de uma tupla de 0's e 1's da forma $(x_1, ..., x_n)$, onde $x_i = 1$ se a criança i está com a testa suja, e $x_i = 0$ caso contrário. Assim, para $n = 3$, uma tupla da forma $(1, 0, 1)$ indicaria que as crianças 1 e 3 estariam com a testa suja. Supondo a situação descrita por essa tupla e considerando que o pai das crianças ainda não fez a primeira pergunta, qual situação a criança 1 imaginaria possível para o sistema? Já que a criança 1 pode ver a testa das outras crianças, duvidando apenas do seu próprio estado, existem duas situações possíveis. São elas, $(1, 0, 1)$ (o estado atual) e $(0, 0, 1)$. Da mesma forma, a criança 2 imaginaria possível

as situações $(1,0,1)$ e $(1,1,1)$. Em geral, a criança i possui a mesma informação em dois mundos possíveis, exceto pela i-ésima posição na tupla.

O sistema pode ser modelado em uma estrutura de Kripke M que consiste de 2^n estados. E já que a informação dos mundos possíveis considerados pelos agentes são equivalentes, exceto por um componente, pode-se definir que $(s,t) \in \mathcal{K}_i$ sempre que s e t concordam em todos os seus componentes, exceto um (o i-ésimo componente). E isto torna a relação $\mathcal{K}_i$ uma relação de equivalência. Para completar a descrição da estrutura M, é preciso definir uma função de valoração π. Para isso, é definida uma proposição primitiva para a linguagem. As primitivas são definidas na forma $p_1, ..., p_n, p$, onde p_i significa que a criança i tem lama na testa e p informa que pelo menos uma criança tem lama na testa. O que permite definir então π de forma que $(M,(x_1,...,x_n)) \vDash p_i$ se e somente se $x_i = 1$ e $(M,(x_1,...,x_n)) \vDash p$ se e somente se $x_j = 1$ para algum j. A primitiva p foi inserida apenas pela conveniência de se descrever a declaração inicial do pai das crianças, visto que ela é equivalente a $p_1 \vee ... \vee p_n$. Com isso, a definição da estrutura M está completa.

Embora se possa pensar que essa estrutura seja complicada, ela possui uma representação gráfica bem simples e elegante. Formando uma estrutura com 2^n nós cada um descrito por uma tupla e onde dois nós são ligados por uma aresta que representa exatamente o componente que difere entre as tuplas definidas pelos nós. O caso $n = 3$ possui um formato de cubo e está ilustrado na Figura 4.1.

A noção de que cada criança sabe o estado das outras é representada, de acordo com a definição formal que foi vista, do seguinte modo. Por exemplo, em uma situação definida pela tupla $(1,0,1)$, é dito que $(M,(1,0,1)) \vDash K_1 \neg p_2$, já que em ambos os mundos que a criança 1 considera possível, ela sempre vê se a criança 2 (como também todas as outras crianças) possui a testa suja ou não. Da mesma forma, $(M,(1,0,1)) \vDash K_1 p_3$, a criança 1 sabe que a criança 3 está com a testa suja. Entretanto, $(M,(1,0,1)) \vDash \neg K_1 p_1$, ou seja, a criança 1 não sabe se está com a sua própria testa suja.

Considerando o que acontece após a afirmação feita pelo pai das crianças, o fato de que todas as crianças sabem que pelo menos uma delas está com lama na testa se torna conhecimento comum, ou seja, $C_G p$. Isto implica em uma mudança

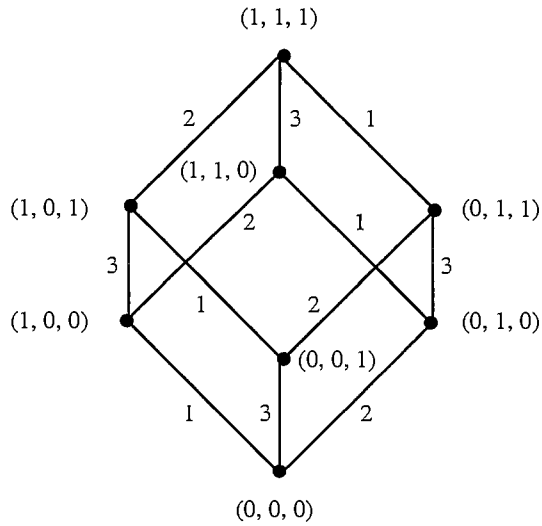


Figura 4.1: Estrutura de Kripke para o problema das crianças

no estado de conhecimento do grupo, mesmo que todas as crianças já soubessem p antes do pai se expressar. Voltando agora ao exemplo para $n = 3$ e considerando uma situação inicial $(1, 0, 1)$, embora nesta situação todas as crianças saibam que existe pelo menos uma delas com lama na testa, a criança 1 poderia considerar a situação $(0, 0, 1)$ possível antes do pai se expressar. E nessa última situação, a criança 3 poderia considerar $(0, 0, 0)$ possível. Assim, antes do pai se expressar, a criança 1 poderia imaginar que a criança 3 considera possível que não exista nenhuma criança com lama na testa. E isto demonstra a importância do conhecimento comum que é adquirido após a primeira afirmação do pai das crianças. Pode-se representar graficamente essa mudança no estado de conhecimento do sistema através da retirada do nó $(0, 0, ..., 0)$ da estrutura de Kripke. Isto gera uma estrutura truncada que para o exemplo com $n = 3$ está representada na Figura 4.2.

A seguir, o que acontece é que cada vez que as crianças respondem 'Não' à pergunta do pai, o estado de conhecimento do grupo é alterado e a estrutura é truncada mais uma vez. De forma que após a primeira vez que as crianças respondem 'Não', todos os nós que contém apenas um componente 1 são retirados da estrutura. Isto ocorre de forma semelhante a que foi descrita na "prova" por indução apresentada no enunciado do problema no início desta seção.

E utilizando o mesmo argumento, pode-se estabelecer que se as crianças res-

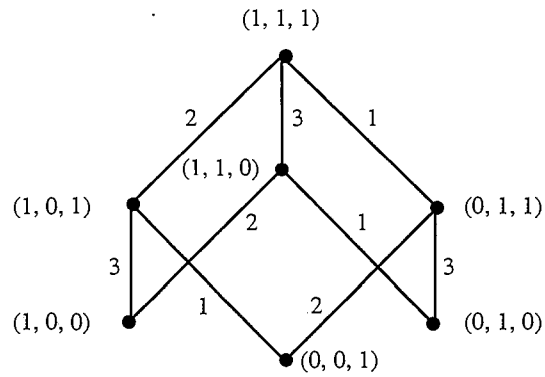


Figura 4.2: Nova estrutura de Kripke para o problema das crianças

pondem 'Não' m vezes, todos os nós com um número m de componentes 1's podem ser retirados da estrutura. Assim, é formada uma sequência de estruturas de Kripke que descrevem o conhecimento do grupo a cada passo do sistema.

4.2 Nova Definição de Linguagem para KBP's

Os programas apresentados nas seções posteriores simulam o raciocínio das crianças de acordo com a descrição do problema. Como já dito na seção anterior, a primitiva p_i significa que a criança i tem lama na testa. E as seguintes expressões e funções são utilizadas na construção do programa, atuando como palavras reservadas para KBP's:

- initial: informa que o programa está na condição inicial, no caso do problema das crianças, logo após a afirmação feita pelo pai das crianças.

As funções a seguir possuem fundamental importância na modelagem de qualquer KBP, pois tratam da evolução do conhecimento e das trocas de mensagens entre os agentes:

- *add*: adiciona uma expressão lógica na base de conhecimento de cada agente;
- $send_j$: representa a resposta dada por um agente, pode ser direcionada a um agente específico j ou não;
- rec_j: verifica qual foi a resposta enviada pelo agente j.

E o uso de condições de *input*, introduzidas no Capítulo 3, expressa uma descrição mais clara da forma de comportamento dos agentes nas trocas de mensagens, pois é tratado ali o recebimento das mensagens enviadas pelos agentes, especificando que uma determinada guarda será escolhida. Podem ser utilizados os seguintes quantificadores para identificação de agentes de acordo com o recebimento de uma mensagem específica:

- *for all*: para todos os agentes que tenham enviado uma dada mensagem;
- *there exists*: para os agentes, um ou mais, que enviaram uma dada mensagem, caso existam.

Considerando a estrutura de um sistema de memória distribuída, apresentada no Capítulo 3, os nós que formam o grafo que define o comportamento dos agentes no sistema podem ser agora visualizados de acordo com a Figura 4.3, que representa as partes integrantes no processamento de cada agente.

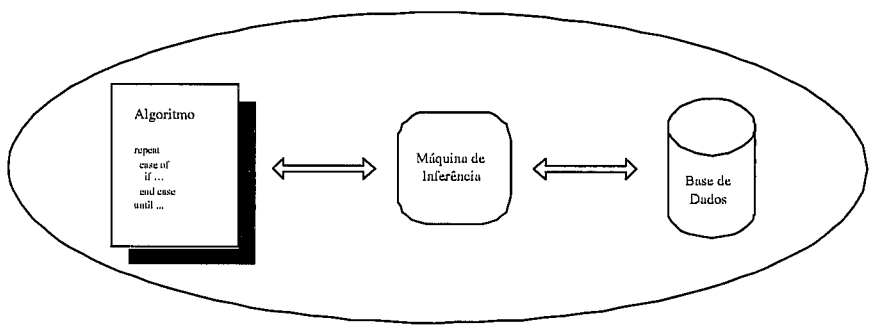


Figura 4.3: Processador interno de um agente

O processamento dos agentes pode então ser entendido como tendo em seu centro uma máquina de inferência, que interpreta as condições e comandos lidos no algoritmo e em consequência disto, faz consultas e registros na sua base de dados (que representa o conhecimento obtido pelo agente). Os sistemas multi-agentes podem ser então representados por um grafo que agrega vários destes processadores como nós, interligados por canais de comunicação. Para o exemplo das crianças com lama na testa, todos os agentes se comunicam entre si, ou seja, as mensagens são enviadas em *broadcast*. A Figura 4.4 demonstra a estrutura de comunicação no sistema para um exemplo com $n = 6$.

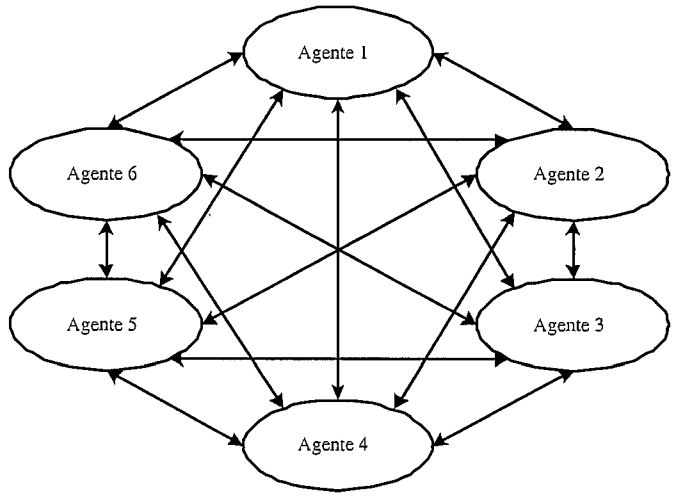


Figura 4.4: Estrutura de comunicação para o problema das crianças

Outro ponto de interesse em relação a forma de comunicação entre os agentes é o que diz respeito à existência, ou não, de um controle externo na troca de mensagens do sistema. Isto é, se o sistema é síncrono ou assíncrono. Para o caso síncrono, os agentes simplesmente executam as suas ações no instante determinado pelo *clock* do sistema, de acordo com as mensagens recebidas. No caso assíncrono, pode ser utilizado um artifício que tem por objetivo garantir que os agentes executem as suas ações com a defasagem de no máximo apenas uma fase. Isto é feito através do envio obrigatório de uma mensagem para cada agente sempre ao final do seu processamento. Assim, os agentes sempre ao início do seu processamento devem esperar o recebimento de mensagens de todos os outros agentes, o que funciona como um método de "sincronização" do sistema.

Ainda neste capítulo, são apresentadas duas versões de KBP's que exemplificam as diferenças existentes entre o caso síncrono e o caso assíncrono para o problema das crianças com lama na testa.

4.3 Conhecimento entre os Agentes

Para esta representação, o início do sistema se dá logo após o anúncio do pai das crianças de que há pelo menos uma delas com a testa suja. Todo o conhecimento e as crenças que as crianças adquirem até o instante que sucede o anúncio do pai é considerado como o conhecimento inicial dos agentes do sistema. Nesse ponto, cada

agente sabe:

1. Quais dentre as outras crianças estão com a testa suja, visto que ela pode ver a testa de todos, menos a sua. Considerando que existem k crianças com a testa suja:

   - Para uma criança i com a testa suja ($i \leq k$): $K_i p_j$, para $j \leq k$, $i \neq j$; $K_i \neg p_j$, para $j > k$.
   - Para uma criança i com a testa limpa ($i > k$): $K_i p_j$, para $j \leq k$; $K_i \neg p_j$, para $j > k$, $i \neq j$.

2. Que é de conhecimento comum que todo agente sabe a situação dos outros agentes, visto que qualquer agente pode ver todos os outros, exceto a si próprio:

   - $C_G(p_i \rightarrow K_j p_i)$ e $C_G(\neg p_i \rightarrow K_j \neg p_i)$, para $i \neq j$ e $i, j = 1, 2, ..., n$

3. Que o fato anunciado pelo pai é de conhecimento comum no grupo de agentes:

   - $C_G \bigvee_{i \in G} p_i \Rightarrow C_G p$

No estado inicial do sistema todos os agentes tem o "conhecimento inicial" que corresponde no enunciado do problema ao instante que sucede o anúncio do pai de que há pelo menos uma criança com a testa suja.

A partir de então, cada agente montará uma linha de raciocínio que evoluirá durante o andamento do sistema. Seja ik o número de crianças com a testa suja que um agente i vê (que será k ou $k-1$, conforme a situação de i). Nos estados epistêmicos do agente i, essas ik crianças são numeradas de $i1$ a ik. Essa identificação das crianças é válida somente nos estados epistêmicos de i, ou seja, cada agente identifica as crianças que vê com a testa suja segundo sua própria ordenação.

É importante ressaltar que a noção de conhecimento comum neste problema está implícita nas expressões de 'Sim' e 'Não' que as crianças respondem durante a evolução do sistema, de acordo com o conhecimento adquirido por cada uma delas. Isto ocorre na maioria das situações modeladas em KBP's, as fórmulas de conhecimento comum (e conhecimento distribuído também) são "traduzidas" por expressões que são utilizadas no envio de mensagens entre os agentes.

4.4 Lógica Dinâmica de Conhecimento

Para que o funcionamento dos KBP's possa receber um respaldo formal através da lógica, é preciso que sejam incorporadas novas funcionalidades na lógica de conhecimento. Pois a semântica dos modelos de Kripke permite que o estado do sistema seja representado em apenas um determinado instante no tempo. Assim, para modelar a evolução temporal dos estados de conhecimento dos agentes no sistema, são adicionadas modalidades de ação à lógica, (DELGADO, BENEVIDES, 2001).

A linguagem utilizada então é a da lógica de conhecimento (já apresentada no Capítulo 2) com a introdução das modalidades de ação $[send_{ij}]$, onde $i, j = 1, ..., m$ (um para cada agente) e $[tick]$. Outras modalidades de ação também podem ser inseridas, mas para o âmbito dos KBP's estudados neste trabalho, as duas ações modais vistas são suficientes.

A ação modal $[\text{send}_{ij}]$ indica o envio de uma mensagem do agente i para o agente j. Se o segundo subscrito for excluído, é considerado que a mensagem foi enviada para todos os outros agentes presentes no sistema (*broadcast*). Por exemplo, $[send_i]\ \varphi$ indica que a mensagem φ foi enviada pelo agente i para todos os agentes restantes.

E a ação modal $[tick]$ indica que "um pulso do relógio global do sistema aconteceu", de forma mais geral, para que se possa estender o conceito para os sistemas assíncronos, pode-se dizer que "um novo estado global do sistema foi atingido". Assim, $[tick]\ \varphi$ indica que φ se mantém em mundo possível atingido devido a ocorrência de um pulso global do sistema. E ainda, a expressão $[tick]\ \bot$ é utilizada para indicar o término do sistema, ou seja, o relógio global do sistema pára.

Aplicando a lógica dinâmica de conhecimento então ao problema das crianças com lama na testa, podem ser obtidas algumas expressões lógicas que representam (e especificam) o sistema. A seguir são apresentadas parte destas expressões, o conjunto completo de regras lógicas será mostrado no Capítulo 5.

De volta ao problema, quando uma criança/agente conclui que está com lama na própria testa, ela imediatamente anuncia este fato ao grupo. Quando isto acontece, todos os agentes do sistema adquirem conhecimento do fato anunciado e o

sistema atinge o seu término.

$K_i p_i \rightarrow [send_i]\ E_G K_i p_i \wedge [tick]\ \bot$, para $i = 1, 2, ..., n$.

Se uma criança/agente ainda não sabe se está ou não com lama na própria testa, e ela atinge o conhecimento de que alguma outra criança sabe que está com lama na própria testa, então a criança pode concluir que está limpa.

$K_i K_j p_j \wedge \neg K_i p_i \rightarrow K_i \neg p_i$, para $i \neq j$ e $i, j = 1, 2, ..., n$.

Existem também os chamados axiomas *frame* de conhecimento. Se um agente sabe um fato, esse fato continua sabido após a execução de qualquer modalidade de ação.

$K_i \varphi \rightarrow [send_{ij}]\ K_i \varphi$, para $i = 1, ..., m$ e $K_i \varphi \rightarrow [tick]\ K_i \varphi$.

Colocando de uma outra maneira, pode-se dizer que os axiomas *frame* de conhecimento garantem que nada do que um agente sabe pode tornar-se falso.

4.5 Caso Síncrono

Primeiramente, é introduzida uma expressão que descreve a transformação das crenças das crianças de acordo com o conhecimento adquirido por estas a cada vez que as perguntas do pai são respondidas. Assim, o agente i no estado inicial monta a seguinte linha de raciocínio, (DELGADO, BENEVIDES, 2001):

$B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge B_{i2}(\neg p_{i2} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$

O raciocínio da criança i pode ser acompanhado da seguinte forma: "Suponha que eu esteja com a testa limpa. Neste caso, existem k crianças com a testa suja. A criança 1 pode imaginar (assim como eu) um estado em que ela esteja com a testa limpa, e deste modo ela veria $k - 1$ crianças com a testa suja. Neste estado epistêmico de 1, a criança 2 vê $k - 2$ crianças com a testa suja, e imaginando que todas as crianças seguem sucessivamente esse raciocínio, a criança $k - 1$ imaginaria que a criança k não vê nenhuma criança com a testa suja, e portanto saberia que ela própria está com a testa suja (caso $k = 0$)". Quando o pai pergunta se alguém

já sabe se está com a testa suja (o que corresponde a um pulso no relógio global), o estado de conhecimento de cada agente muda para:

$$B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge B_{i2}(\neg p_{i2} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik}p_{ik}))...)))$$

Essa mudança corresponde ao conhecimento obtido quando todos respondem negativamente a uma pergunta feita pelo pai. Significa que existe pelo menos uma criança a mais com a testa suja do que o mínimo considerado na crença de uma das crianças com a testa suja (que pertence a ik), existente no estado epistêmico de cada criança i do sistema. Logo, a criança i deixa de acreditar nesse estado, e passa a considerar o caso do estado epistêmico da criança $i(k-1)$. Conforme dito anteriormente, os mundos possíveis são desta forma, gradativamente eliminados (após a k-ésima pergunta, os mundos com menos de k crianças com a testa suja são eliminados).

Esse mecanismo de transformação das crenças dos agentes, que pode ser visto também como a evolução do conhecimento de cada agente durante o andamento do sistema, é de grande importância para a modelagem de KBP's neste trabalho. No Capítulo 5 é mostrado que para qualquer tipo de situação estudada, uma expressão semelhante de transformação de crenças deverá ser construída para formar pelo menos uma guarda em um programa.

4.5.1 Programa

Para o caso síncrono, a entidade sincronizadora do sistema distribuído é o pai das crianças, que fica repetidamente enviando o sinal de relógio sob a forma da pergunta: "Alguém sabe que está com a testa suja?" (o programa que o agente pai executa é mostrado no Apêndice A).

program MC_i
  repeat
    case of
    (a) *input*: *for all* $j \neq i$, rec_j nil
      if initial $\wedge K_i p_i$ **then** *send* 'YES'
    (b) *input*: *there exists* $j \neq i$, rec_j 'YES'
      if $\neg K_i p_i$ **then** *add* $K_i \neg p_i$
    (c) *input*: *for all* $j \neq i$, rec_j nil
      if $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$ **then**
        add $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$
        if $K_i p_i$ **then** *send* 'YES'
    end case
  until $K_i p_i \vee K_i \neg p_i$
end

Figura 4.5: Programa MC_i (síncrono)

A seguir, são apresentados alguns exemplos de execução para este KBP.

4.5.2 Exemplos

Os exemplos de funcionamento do programa aqui apresentados tratam as condições mais expressivas consideradas para o problema das crianças com lama na testa. Considerando $n = 3$:

Ex. 1: Apenas uma criança com a testa suja: $p_1, \neg p_2, \neg p_3$; $k = \{1\}$

MC_1	MC_2	MC_3
Conhecimento inicial: $K_1\neg p_2, K_1\neg p_3,$ $K_1(p_1 \vee p_2 \vee p_3) \Rightarrow K_1 p_1$	Conhecimento inicial: $K_2 p_1, K_2\neg p_3$ Crença inicial: $B_2(\neg p_2 \wedge K_1 p_1)$	Conhecimento inicial: $K_3 p_1, K_3\neg p_2$ Crença inicial: $B_3(\neg p_3 \wedge K_1 p_1)$
Início: a) *send* 'YES' ■	Início: b) $K_2\neg p_2$ ■	Início: b) $K_3\neg p_3$ ■

Ex. 2: Duas crianças com a testa suja: $p_1, p_2, \neg p_3$; $k = \{1, 2\}$

MC_1	MC_2	MC_3
Conhecimento inicial: $K_1 p_2, K_1\neg p_3$ Crença inicial: $B_1(\neg p_1 \wedge K_2 p_2)$	Conhecimento inicial: $K_2 p_1, K_2\neg p_3$ Crença inicial: $B_2(\neg p_2 \wedge K_1 p_1)$	Conhecimento inicial: $K_3 p_1, K_3 p_2$ Crença inicial: $B_3(\neg p_3 \wedge B_1(\neg p_1 \wedge K_2 p_2))$
Início: c) $K_1(p_1 \wedge K_2 p_2)$ *send* 'YES' ■	Início: c) $K_2(p_2 \wedge K_1 p_1)$ *send* 'YES' ■	Início: c) $B_3(\neg p_3 \wedge K_1(p_1 \wedge K_2 p_2))$ b) $K_3\neg p_3$ ■

Ex. 3: Todas as três crianças com a testa suja: p_1, p_2, p_3; $k = \{1, 2, 3\}$

MC_1	MC_2	MC_3
Conhecimento inicial: $K_1 p_2, K_1 p_3$ Crença inicial: $B_1(\neg p_1 \wedge B_2(\neg p_2 \wedge K_3 p_3))$	Conhecimento inicial: $K_2 p_1, K_2 p_3$ Crença inicial: $B_2(\neg p_2 \wedge B_1(\neg p_1 \wedge K_3 p_3))$	Conhecimento inicial: $K_3 p_1, K_3 p_2$ Crença inicial: $B_3(\neg p_3 \wedge B_1(\neg p_1 \wedge K_2 p_2))$
Início: c) $B_1(\neg p_1 \wedge K_2(p_2 \wedge K_3 p_3))$ c) $K_1(p_1 \wedge K_2(p_2 \wedge K_3 p_3))$ *send* 'YES' ■	Início: c) $B_2(\neg p_2 \wedge K_1(p_1 \wedge K_3 p_3))$ c) $K_2(p_2 \wedge K_1(p_1 \wedge K_3 p_3))$ *send* 'YES' ■	Início: c) $B_3(\neg p_3 \wedge K_1(p_1 \wedge K_2 p_2))$ c) $B_3(p_3 \wedge K_1(p_1 \wedge K_2 p_2))$ *send* 'YES' ■

Tabela 4.1: Exemplos de execução para MC_i

A seção seguinte demonstra uma prova da correta execução do programa tendo o auxílio dos exemplos construídos aqui.

4.5.3 Prova

A prova de que MC_i (caso síncrono) está correto em relação à especificação do sistema é baseada na indução do número de passos p executados no programa, levando em consideração que $k \leq n$, para k crianças com a testa suja. O caso $p = 1$

é atípico, pois aqui apenas uma criança está com a testa suja, e o programa termina após a execução da guarda (a) para essa criança, o que ocorre somente neste caso.

Assim, é tomado como base para a prova de indução o caso $p = 2$. O programa para uma criança com a testa suja executa sempre a guarda (c) e pára. Para uma criança com a testa limpa o programa executa as guardas (c, b) e pára.

Suponha que MC_i funciona para um número de passos $p = x$, portanto existem $k = x$ crianças com a testa suja no sistema. Para um número de passos $p = x + 1$, MC_i funcionará da seguinte forma:

MC_i (criança com a testa suja)	MC_i (criança com a testa limpa)
Conhecimento inicial: $K_i p_j$, para todo $j \leqslant k$, $j \neq i$; $K_i \neg p_j$, para todo $j > k$.	Conhecimento inicial: $K_i p_j$, para todo $j \leqslant k$; $K_i \neg p_j$, para todo $j > k$, $j \neq i$.
Crença inicial: $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge$ $\wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$	Crença inicial (contém um termo ik a mais): $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge$ $\wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$
Início: c) $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge$ $\wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$ ...	Início: c) $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge$ $\wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$ ...

Tabela 4.2: Execução geral para o programa MC_i

A partir deste ponto, pode-se verificar que o programa sempre executa a guarda (c) até que k B_i's sejam eliminados, tanto para uma criança com a testa suja quanto para uma criança com a testa limpa, pois enquanto houver B_i's na crença de uma criança ela ainda estará incerta sobre o seu estado. Também deve ser observado que a fórmula de incerteza para uma criança com a testa limpa contém um termo B_i a mais.

A cada guarda (c) exatamente um termo B_i é eliminado da crença de uma criança, portanto após a primeira guarda (c) executada haverá uma crença com $x - 1$ B_i's para uma criança com a testa suja, e x B_i's para uma criança com a testa limpa. Do ponto de vista da execução do programa, o estado global da crença das crianças após a primeira guarda (c) executada é semelhante ao estado global do programa aplicado a x crianças com a testa suja antes da primeira guarda (c) executada.

Já que se sabe que o programa funciona para um número de passos $p = x$, é possível concluir que o programa funcionará também para o caso $p = x + 1$, e ainda que o programa executará uma guarda (c) a mais, para todas as crianças, até que a condição de parada seja atingida.

Assim, ao final da execução do programa, se p passos foram executados, então o número de crianças com a testa suja no sistema é igual a p. E todas estas crianças respondem 'Sim' na execução do passo p.

4.6 Caso Assíncrono

Neste caso não há sinal de sincronismo no sistema, portanto para que o sistema tenha noção de tempo global é preciso que cada agente simule o sincronismo através das respostas dos outros agentes. A solução adotada é que cada agente envia ao término do processamento de uma guarda uma mensagem 'YES' ou 'NO', que além do significado semântico dessa resposta no sistema, terá também uma função sincronizadora, como será visto a seguir.

4.6.1 Programa

Diferentemente do caso síncrono, aqui todas as guardas incluem um teste booleano de recebimento de uma mensagem de cada outro agente do sistema, excetuando-se as guardas que representam o início do sistema. Assim os agentes que concluem o processamento em uma fase, ficam "presos" para iniciar um novo processamento, aguardando mensagens dos outros agentes que indicam a conclusão do processamento da guarda anterior. Este processo bastante simples controla as defasagens de tempo entre os agentes.

program MC_i *Assync*
  repeat
    case of
    (a) *input*: *for all* $j \neq i$, rec_j nil
      if $\underline{\text{initial}} \wedge K_i p_i$ **then** *send* 'YES'
    (b) *input*: *for all* $j \neq i$, rec_j nil
      if $\underline{\text{initial}} \wedge \neg K_i p_i$ **then** *send* 'NO'
    (c) *input*: *for all* $j \neq i$, rec_j 'YES' or rec_j 'NO'
      if $\neg K_i p_i$ **then** *add* $K_i \neg p_i$
    (d) *input*: *for all* $j \neq i$, rec_j 'NO'
      if $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$ **then**
        add $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$
        if $K_i p_i$ **then** *send* 'YES'
        else *send* 'NO'
    end case
  until $K_i p_i \vee K_i \neg p_i$
end

Figura 4.6: Programa MC_i *Assync*

A execução dos exemplos ocorre de forma semelhante ao caso síncrono, não necessitando portanto, da sua reapresentação aqui.

4.6.2 Prova

A prova de que MC_i *Assync* está correto em relação à especificação do sistema que irá ser demonstrada é bastante parecida com a do caso síncrono, com alguns pontos diferentes a serem discutidos agora.

Neste caso, os sucessivos envios de mensagens mantêm os agentes com no máximo uma fase de diferença, pois a cada fase um agente espera obrigatoriamente até que receba uma mensagem de cada outro agente do sistema. Isto significa que a cada estado global do sistema todos os agentes estarão em uma das fases f ou $f+1$, sendo f um inteiro. Não é possível a um agente passar para a fase $f+2$ enquanto ainda houver um (ou mais de um) agente nesse mesmo estado global do sistema, ou seja, na fase f.

Segundo MC_i *Assync*, uma mensagem é sempre enviada ao final do processamento de uma guarda. Dito isso, pode-se montar a prova por indução no número m de mensagens enviadas pelo agente i, onde $i \leq n$.

Considerando inicialmente o caso $m = 1$, apenas uma guarda é satisfeita e o agente atinge a condição de parada. Esse caso representa o sistema onde há apenas uma criança com a testa suja. Duas execuções são possíveis para MC_i *Assync*:

1. A criança i está com a testa suja. De início, a guarda (a) é satisfeita e i envia uma mensagem 'YES' (todos os outros agentes entrarão na guarda (b)).

2. A criança i não está com a testa suja. De início a guarda (b) é satisfeita e MC_j *Assync* para alguma outra criança $j \neq i$ executará a guarda (a) (neste caso j é a criança com a testa suja), a seguir a guarda (c) é satisfeita e o programa para uma criança com a testa limpa termina.

Para um caso genérico $m = x$ tal que $x > 1$, há necessariamente mais de uma criança com a testa suja no sistema. Como todos enviaram 'NO' e $k > 1$, qualquer agente do sistema será capaz de satisfazer apenas a guarda (d) na próxima fase, após o recebimento de todas as mensagens. A partir deste ponto, pode-se ver que o programa sempre executará a guarda (d) até que k B_i's sejam eliminados, tanto para uma criança com a testa suja quanto para uma criança com a testa limpa, pois enquanto houver termos B_i's na crença de uma criança ela ainda estará incerta sobre o seu estado e consequentemente responderá 'Não' à pergunta do pai. A cada guarda (d) exatamente um termo B_i é eliminado da crença de uma criança, portanto após a primeiro guarda (d) executada, existirá uma crença com $x - 1$ termos B_i's para uma criança com a testa suja, e x termos B_i's para uma criança com a testa limpa. A última execução da guarda (d) em MC_i *Assync* ocorrerá quando alguma criança j do sistema concluir que está com a testa suja, ou seja, quando alguma mensagem 'YES' aparecer no sistema. Na fase seguinte, a guarda (c) será executada para as crianças com a testa limpa.

Supondo que MC_i *Assync* está correto para o caso $m = x$. Para $m = x + 1$ existirá uma mensagem a mais enviada por i. Isso só pode acontecer acrescentando-se uma mensagem 'NO' à execução de MC_i *Assync*, pois, por definição, quando um agente envia uma mensagem 'YES' no sistema que representa o problema das crianças com lama na testa, esta tem que ser a última mensagem enviada por ele. A única forma de acrescentar uma mensagem 'NO' à execução de MC_i *Assync* é

satisfazendo a guarda (d) uma vez a mais que no caso $m = x$. Isto só é possível acrescentando um termo B_i à crença inicial do agente i, o que ocorre como consequência de se aumentar em uma unidade o número de crianças com a testa suja no sistema. Dessa forma, uma mensagem 'NO' será acrescentada à execução de MC_i *Assync*, e então após o envio da $(x+1)$-ésima mensagem a guarda (c) será executada para as crianças com a testa limpa, tal como no caso anterior.

Capítulo 5

Traduções

As traduções referidas nas seções a seguir tratam de um primeiro passo na tentativa de obtenção de uma ferramenta computacional capaz de lidar com os KBP's e a lógica de conhecimento envolvida, que representam um determinado sistema multi-agentes. Entende-se por tradução, a descrição de um processo que pretende extrair as regras (expressões) lógicas que podem produzir informações relevantes sobre um problema a partir de um respectivo KBP apresentado, ou vice-versa, construir um KBP a partir de regras lógicas que representam a descrição de um dado problema.

Existe uma grande variedade de problemas que podem ser representados pelos KBP's. Isto implica na criação de métodos de tradução que devem possuir alguma flexibilidade em relação às informações de entrada e saída (regras lógicas e guardas de programa), o que é necessário para satisfazer várias condições diferentes.

Porém, ainda podem ser oferecidos mecanismos para validar os métodos propostos, como será visto ao final deste capítulo, com a apresentação de teoremas que expressam o fato de que um KBP sempre satisfaz uma especificação descrita através de regras lógicas, em ambos os métodos de tradução.

5.1 Geração da Lógica de Conhecimento a partir de um KBP

Utiliza-se como base um conjunto de regras para gerar os axiomas de uma lógica de conhecimento para múltiplos agentes a partir de um KBP para o problema. Ao expressar um problema através da lógica de conhecimento torna-se possível verificar formalmente as propriedades presentes no programa, e consequentemente saber

se o programa está de acordo com a especificação do problema.

5.1.1 Extração das Regras Lógicas

Este processo tem como princípio retornar um conjunto de regras lógicas basicamente através da interpretação das condições das guardas presentes no KBP em questão. Para extrair as regras lógicas que correspondem à evolução do conhecimento no sistema regido por um KBP aplicam-se as seguintes transformações ao programa:

1. As guardas do programa são os antecedentes das regras, e as ações correspondentes de cada guarda se transformam nos consequentes das regras, com a inserção da ação modal respectiva, no caso do envio de uma mensagem (*send*). Porém as ações de adição de fórmulas na base de conhecimento do agente (*add*) não possuem uma ação modal correspondente, as expressões contidas ali podem ser diretamente transcritas para a regra lógica.

2. Fazem parte da lógica, modalidades de ação que representem as operações que acontecem em um pulso do sistema distribuído, mas cujo resultado tem implicações apenas no próximo pulso do sistema, como por exemplo o envio de uma mensagem. Se um agente envia uma mensagem no pulso k, a mensagem só estará disponível para os agentes de destino no pulso $k+1$. A modalidade de ação garante que a informação representada pelo recebimento da mensagem só esteja disponível no sistema no mundo atingido pela ação modal $[send]$, e não esteja disponível no mundo atual.

3. As regras representadas pelas guardas que são aplicadas recursivamente (onde a ação resultante no programa valida a condição da própria guarda), como a guarda que trata as crenças B_i de um agente no KBP para o problema das crianças com lama na testa, devem conter a ação modal $[tick]$ aplicada ao consequente, pois as guardas são aplicadas uma vez para cada pulso no sistema (caso síncrono) e na lógica os pulsos são marcados com esta ação modal $[tick]$. Isto evita que uma ação do programa seja aplicada repetidas vezes em um mundo correspondente a um mesmo pulso do sistema. É importante ressaltar

que a expressão *tick* não aparece nas guardas, por questão de simplicidade, já que está implícito que cada guarda corresponde a um pulso (ou fase, para o caso assíncrono) do sistema.

4. A ação no KBP correspondente a um envio de mensagem (*send*) indica uma ação na lógica onde o significado do que é dito passa a ser do conhecimento de todos, quando a mensagem é enviada em *broadcast*. Esta classe de ações transforma-se em um consequente do tipo $[send]\ E_G\varphi$, onde φ é o significado da mensagem enviada para um agente qualquer do sistema. No caso do problema das crianças com lama na testa, uma mensagem 'YES' significa que a criança que enviou a mensagem já sabe que está com lama na testa, ou seja, $E_G K_i p_i$.

5. As expressões que representam mensagens recebidas (rec_j) pelos agentes, contidas nas condições de *input* são interpretadas da mesma forma que as presentes nas guardas, adicionando as informações ali contidas no antecedente da regra, necessitando porém, que seja feita a conversão de cada expressão em uma fórmula modal correspondente. Por exemplo, *input*: *for all* $j \neq i$, rec_j 'YES' resultaria na adição da expressão $K_i K_j p_j$ ao antecedente da regra.

6. A condição de parada é o antecedente da regra que tem como consequente, $\perp$, com a adição da modalidade de ação $[tick]$.

Deve-se ainda notar que sempre devem ser considerados os axiomas *frame* de conhecimento para todas as modalidades de ação presentes na lógica. Pois está implícito que durante a execução dos programas, nada do que é sabido por um agente pode tornar-se falso, por exemplo, $K_i\varphi \rightarrow [tick]\ K_i\varphi$.

A seguir é apresentado um exemplo de utilização deste método.

5.1.2 Exemplo de Utilização

Como exemplo, será utilizado o KBP construído para o problema das crianças com lama na testa apresentado no capítulo anterior.

Para se extrair as regras lógicas do KBP MC_i, primeiramente as expressões utilizadas nas trocas de mensagens entre os agentes, que estão presentes nas condições

de *input*, são devidamente inseridas nos antecedentes das regras que são extraídas nas guardas.

- Guardas:

  (a) $K_i p_i \rightarrow [send]\ E_G K_i p_i$

  (b) $\neg K_i p_i \rightarrow [send]\ E_G \neg K_i p_i$

  (c) $K_i K_j p_j \wedge \neg K_i p_i \rightarrow K_i \neg p_i$

  (d) $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge B_{i2}(\neg p_{i2} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...))) \rightarrow [tick]\ B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge B_{i2}(\neg p_{i2} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$

- Condição de Parada:

  $K_i p_i \vee K_i \neg p_i \rightarrow [tick]\ \bot$

  Então: $K_i p_i \rightarrow [tick]\ \bot$ e também $K_i \neg p_i \rightarrow [tick]\ \bot$

Observando o exemplo acima, feito a partir do KBP construído para o problema das crianças com lama na testa, é possível perceber a utilidade do método apresentado aqui para a extração das regras lógicas que podem exibir as características relacionadas à especificação de um sistema ou um problema a ser modelado. Desta forma, podem ser comprovadas determinadas propriedades que não seriam facilmente percebidas em uma visualização direta de um KBP. Essa dificuldade será analisada na próxima seção, quando o passo inverso será executado, a construção de um KBP tendo como ponto de partida as regras lógicas referentes a um determinado problema.

5.2 Geração de um KBP a partir de Regras Lógicas

Desta vez será proposta a execução do caminho inverso ao apresentado na seção anterior, que é a possibilidade de construção de um KBP tendo como base expressões da lógica de conhecimento que representam o funcionamento de um problema em um sistema distribuído. Basicamente, é necessário inverter as regras de extração para a lógica apresentadas anteriormente, porém, devido à complexidade existente na formação da estrutura de um KBP (que pode ser visto como uma apresentação

das regras lógicas de um sistema em mais alto nível), será necessário também impor algumas restrições no formato da apresentação das regras lógicas, bem como algumas informações adicionais que devem ser incluídas para auxiliar na criação das guardas a serem formadas.

5.2.1 Construção de um KBP

Primeiramente, com o objetivo de se obter principalmente as guardas e suas respectivas condições, é feita a inversão dos passos de extração das regras lógicas de um KBP, apresentadas anteriormente:

1. Os antecedentes das regras se transformam nas guardas do programa, e os consequentes se transformam nas ações da guarda correspondente, observando as ações modais contidas nestes últimos.

2. Deve estar presente nas regras lógicas, quando for o caso, a modalidade de ação que indica o envio de mensagem de um agente que acontece em pulso do sistema distribuído, se um agente envia uma mensagem no pulso k, a mensagem só estará disponível para os agentes de destino no pulso $k+1$. A modalidade de ação $[send]$ garante que a informação representada pelo recebimento da mensagem só esteja disponível no sistema em seu próximo estado global. A função *send* é então adicionada dentro da guarda, seguida da expressão semântica (caso exista) que representa a fórmula contida no consequente da regra lógica.

3. As regras que tratam da evolução do conhecimento no sistema (a transformação de crença em conhecimento por parte dos agentes), são aplicadas recursivamente dentro do programa e devem conter a ação $[tick]$. O antecedente da regra forma uma guarda e o consequente adiciona uma expressão na base de conhecimento do agente, através da função *add*, dentro da guarda.

Além disso, são apresentadas algumas restrições necessárias para a construção do KBP em questão:

4. Devem ser indicadas as expressões semânticas relevantes ao problema que contêm em seu significado expressões lógicas equivalentes, por exemplo, no

problema estudado neste trabalho, as expressões 'YES' e 'NO' são interpretadas respectivamente como $[send]\ E_G K_i p_i$ e $[send]\ E_G \neg K_i p_i$, simbolizando o anúncio de cada criança/agente do seu conhecimento sobre a sua situação atual (no caso se ele sabe ou não que está com a testa suja). Estes tipos de expressão representam propriedades especiais relacionadas a um determinado problema e geralmente expressam a noção de conhecimento comum, como no caso do problema das crianças com lama na testa.

5. As expressões mencionadas acima devem ser utilizadas como padrão para a troca de mensagens entre os agentes, e portanto, devem ser inseridas nas condições de *input* que precedem as guardas.

6. A condição de parada é obtida através da regra que possui como consequente, $\bot$, com a adição da modalidade de ação $[tick]$.

A seguir é apresentado um exemplo de utilização deste método.

5.2.2 Exemplo de Utilização

Seguindo a mesma orientação do método anterior, será utilizado novamente o problema das crianças com lama na testa como exemplo.

Para se construir o KBP MC_i a partir de regras lógicas, é preciso inicialmente definir as expressões que correspondem a palavras-chave do problema, geralmente relacionadas às trocas de mensagens entre os agentes:

send 'YES': $K_i p_i \rightarrow [send]\ E_G K_i p_i$
send 'NO': $\neg K_i p_i \rightarrow [send]\ E_G \neg K_i p_i$

- Guardas:

São identificadas através das regras lógicas que contêm a ação $[send]$. As duas primeiras regras lógicas são mais simples e podem ser transformadas diretamente.

$K_i p_i \rightarrow [send]\ E_G K_i p_i \Rightarrow$ **if** $K_i p_i$ **then** *send* 'YES'
$\neg K_i p_i \rightarrow [send]\ E_G \neg K_i p_i \Rightarrow$ **if** $\neg K_i p_i$ **then** *send* 'NO'

A terceira regra lógica possui uma conjunção em seu antecedente, o que pode indicar a necessidade de se especificar o recebimento de uma mensagem na condição de *input* da guarda. Já que rec_j 'YES' resulta na obtenção da fórmula $E_G K_j p_j$ para o agente i, assim, por conseguinte é válido que $K_i K_j p_j$. Esta fórmula será expressa então através da condição de *input*, como será visto logo adiante.

$K_i K_j p_j \wedge \neg K_i p_i \rightarrow K_i \neg p_i \Rightarrow$ **if** $\neg K_i p_i$ **then** *add* $K_i \neg p_i$

A quarta regra lógica se refere ao tratamento das crenças do agente, ela é identificada através da ação [*tick*].

$B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge B_{i2}(\neg p_{i2} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...))) \rightarrow [tick]\ B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge B_{i2}(\neg p_{i2} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$

$\Rightarrow$

if $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge B_{i(k-1)}(\neg p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$

then

add $B_i(\neg p_i \wedge B_{i1}(\neg p_{i1} \wedge ... \wedge B_{i(k-2)}(\neg p_{i(k-2)} \wedge K_{i(k-1)}(p_{i(k-1)} \wedge K_{ik} p_{ik}))...)))$

- Condições de *input*:

São indicadas de acordo com a forma de comunicação dos agentes no sistema, para o caso do problema das crianças com lama na testa, as mensagens são enviadas em *broadcast*. Devem também ser utilizadas aqui as expressões definidas para a troca de mensagens entre os agentes. Desta forma, as condições de *input* são definidas de acordo com as guardas que foram criadas anteriormente.

Assim, considerando as guardas construídas aqui para o problema das crianças com lama na testa, nas duas primeiras é necessária apenas a condição de *input* básica:

input: *for all* $j \neq i$, rec_j nil

Para a terceira guarda, é incluída na condição de *input* o recebimento da mensagem que expressa a fórmula existente no antecedente da regra lógica.

input: *for all* $j \neq i$, rec_j 'YES'

Além disso, é necessário incluir uma verificação de resposta para todos os agentes no sistema, considerando o caso assíncrono. Isto é feito da seguinte forma:

input: *for all* $j \neq i$, rec_j 'YES' or rec_j 'NO'

Para a quarta guarda (caso assíncrono), que trata da evolução do conhecimento de cada agente, é utilizada uma condição de *input* para verificar que nenhum agente atingiu um estado de conhecimento completo. Para o exemplo considerado aqui, isto significa que a criança ainda não sabe que está com a própria testa suja.

input: *for all* $j \neq i$, rec_j 'NO'

- Condição de parada:

É obtida diretamente pela regra que contém o operador $\perp$. Neste caso, também pode ocorrer a substituição de fórmulas lógicas por expressões que correspondem a palavras-chave do problema, da mesma forma descrita para as condições de *input*, o objetivo desta substituição é a possibilidade de se oferecer um melhor entendimento para o programa.

$K_i p_i \vee K_i \neg p_i \rightarrow [tick] \perp \Rightarrow$ **until** $K_i p_i \vee K_i \neg p_i$

A demonstração desses exemplos de tradução deixa claro que existe um processo subjetivo de interpretação das guardas e condições de *input*, principalmente para o método de construção de KBP's. A utilização dos modelos que definem a forma de apresentação das informações de entrada e saída (regras lógicas e guardas de programa) evita que ocorra a geração de várias possíveis informações de saída a partir de uma mesma informação de entrada. Mas ainda assim, é necessário que exista interferência externa para que se possa completar o processo de forma satisfatória.

No próximo capítulo é apresentada uma proposta de implementação destes procedimentos com o objetivo de se propor a construção de uma ferramenta computacional capaz de executar os métodos demonstrados aqui. Para isso, são reapresentados modelos para a introdução de informações que restringem ainda mais a forma de apresentação para regras lógicas e guardas de programa, produzindo desta maneira métodos de tradução mais eficientes e independentes.

5.3 Funções de Tradução

A fim de oferecer um formalismo mais elevado para os métodos de tradução, são propostos vários pontos para a formação de funções que possam mapear os métodos de tradução apresentados. Para cada método de tradução, será criada uma função correspondente, Ψ_a para a extração da lógica e Ψ_b para a construção de um KBP. De forma que, dados um programa P e uma especificação S, tem-se que $\Psi_a : P \rightarrow S$ e $\Psi_b : S \rightarrow P$.

5.3.1 Extração da Lógica

Primeiramente, são definidos de forma breve os passos para tradução:

1. guarda do programa (**if ... then**): antecedente da regra;
2. conteúdo da guarda (*add*, *send*): consequente da regra (com a inserção da respectiva ação modal);
3. expressões contidas nas condições de input: antecedente da regra (forma indireta);
4. condição de parada: antecedente da regra que possui como consequente $[tick] \perp$.

Deve se lembrar ainda da substituição das expressões semânticas (utilizadas nas trocas de mensagens) pelas respectivas fórmulas equivalentes.

Desta forma, uma função de tradução Ψ_a pode ser construída através das seguintes transformações, definidas pelos passos de tradução anteriormente apresentados:

- ψ_{a1} - aplicada a uma guarda (declaração), extrai o antecedente de uma regra;
- ψ_{a2} - aplicada a uma guarda (conteúdo), extrai o consequente de uma regra;
- ψ_{a3} - aplicada a uma guarda (condição de *input*), extrai o antecedente de uma regra; *
- $\psi_{a\perp}$ - aplicada à condição de parada, extrai a regra lógica de término.

É importante notar que a transformação ψ_{a3} não pode ser definida de forma direta, o que dificulta a criação da sua transformação inversa para o processo de construção de um KBP, apresentado a seguir.

5.3.2 Construção de um KBP

Da mesma maneira feita na seção anterior, são definidos de forma breve os passos para tradução:

1. antecedente da regra: guarda do programa (**if ... then**);
2. consequente da regra: conteúdo da guarda (com a inserção da função correspondente);
3. condição de parada: antecedente da regra que possui como consequente $[tick]\ \bot$.

Aqui deve se notar que é introduzida a função *add* nas guardas do programa em substituição à ação modal $[tick]$ e as condições de *input* são formadas de acordo com a estrutura de comunicação utilizada no sistema. E ainda, deve-se definir as expressões semânticas a serem utilizadas nas trocas de mensagens.

Então, uma função de tradução Ψ_b pode ser construída da mesma forma feita para o procedimento anterior:

- ψ_{b1} - aplicada ao antecedente de uma regra, constrói a declaração da guarda;
- ψ_{b2} - aplicada ao consequente de uma regra, constrói o conteúdo da guarda;
- $\psi_{b\bot}$ - aplicada à regra lógica de término, constrói a condição de parada.

Pode-se perceber aqui, que devido ao fato de não existir uma transformação inversa correspondente à ψ_{a3}, também não é possível obter a correspondência $\Psi_b = {\Psi_a}^{-1}$ (e da mesma forma para o caso oposto). Este fato é contornado no próximo capítulo, através da introdução de um mecanismo nos métodos de tradução capaz de definir de forma mais direta as condições de *input* a serem criadas em um programa.

Ainda no próximo capítulo, são definidos os modelos para apresentação de um KBP e da lógica (especificação de um sistema). De maneira que todos os elementos necessários à fundação de um formalismo relacionado aos métodos de tradução estejam presentes neste trabalho.

5.4 Teoremas

Nesta seção, dois teoremas que apresentam resultados importantes sobre os métodos descritos neste capítulo são propostos e com a exposição de suas respectivas provas, é obtida uma validação para esses métodos e consequentemente um nível de formalização mais elevado também é alcançado.

O primeiro teorema diz respeito ao segundo método de tradução desenvolvido, para geração de um KBP a partir de regras lógicas:

Teorema 5.1 *Seja P um programa obtido pela tradução de uma especificação S gerada através de um conjunto de regras lógicas. Desta forma, toda execução de P satisfaz S.*

Prova 5.1 *Pode-se verificar que as regras de tradução para a geração de um KBP a partir das regras lógicas, transportam de forma direta as fórmulas contidas no antecedente e no consequente de cada expressão lógica. O mesmo acontecendo de acordo com a definição da função de tradução Ψ_b. Existindo somente como um fator externo a esse processo, a criação das condições de input, que são obtidas de acordo com a interpretação da forma de comunicação entre os agentes no sistema. Porém, estas condições definem apenas uma estrutura que tem por função organizar a troca de mensagens entre os agentes de forma a oferecer um melhor entendimento na leitura do programa.*

E o segundo teorema se refere ao método de tradução para geração de regras lógicas a partir de um KBP:

Teorema 5.2 *Seja S uma especificação gerada através das regras lógicas extraídas pela tradução de um determinado programa P. Desta forma, toda execução de P satisfaz S.*

Prova 5.2 *De maneira análoga à prova para o teorema anterior, pode-se verificar que como as regras lógicas que geram a especificação são obtidas diretamente das guardas do programa e das ações contidas nestas, então*

todo o comportamento dos agentes durante a execução dos programas estará previsto na especificação gerada pelo método de tradução. Isto também é verificado através da função de tradução Ψ_a. Mas deve-se observar novamente a questão das condições de input, que podem apenas aplicar alguma restrição para as execuções possíveis dos programas (isto é modelado através da transformação ψ_{a3}), não resultando em outras execuções que não possam satisfazer a especificação gerada.

O enunciado destes teoremas é importante pois espera-se que eles continuem sendo satisfeitos caso sejam introduzidas algumas modificações nos métodos de tradução. O que irá acontecer no capítulo seguinte, onde alguns novos artifícios serão incluídos visando facilitar a geração e análise de KBP's.

Capítulo 6

Implementação

Com o intuito de se oferecer a possibilidade de construção de um aplicativo capaz de implementar os métodos de tradução apresentados anteriormente, será definido um padrão para a introdução de dados pelo usuário, afim de que se possa garantir a execução para vários tipos de interpretações em problemas diferentes. Este padrão é discutido nas seções a seguir para os dois métodos de tradução existentes (regras lógicas para um KBP e vice-versa).

Como exemplo de utilização, será analisada uma situação bastante parecida com o jogo das três cartas apresentado no Capítulo 2. Mas desta vez, haverá uma maior complexidade. O número de jogadores será estendido para um valor n, porém a mecânica do jogo continuará a mesma. Cada jogador receberá uma carta, e uma carta ficará na mesa, com a face voltada para baixo (o que implica em uma quantidade $n + 1$ de cartas diferentes). As cartas são numeradas de 1 a n e os jogadores devem tentar então, descobrir qual é o valor da carta que está com a face voltada para baixo.

Ao início do jogo, cada jogador sabe somente o valor da sua própria carta. Com o andamento do jogo, a cada rodada um jogador tentará adivinhar qual é carta que está sobre a mesa através de um palpite. Caso o palpite seja falso, o jogador que possuir a carta que foi sugerida, deverá se manifestar e fazer um novo palpite. Assim, é fácil perceber que o jogo terminará em no máximo n rodadas.

Para a demonstração dos métodos de tradução revistos com as restrições de introdução de dados pelo usuário, é proposta uma inversão em relação ao capítulo anterior na ordem das seções a serem apresentadas. Inicialmente, serão obtidas as

guardas do programa a partir das regras lógicas introduzidas, o que parece ser mais natural que ocorra primeiro, já que se estará realizando uma modelagem para uma estrutura de mais alto nível. E então, estas guardas serão utilizadas na construção de um KBP, com algumas modificações necessárias, que será submetido ao método de extração das regras lógicas para que se possam ser feitas as devidas comparações entre os resultados obtidos por ambos os métodos.

6.1 Geração de um KBP a partir de Regras Lógicas

Neste passo, o usuário terá que entrar com as regras lógicas referentes ao problema a ser estudado, de acordo com o modelo que será apresentado. É necessário também a adição de alguns outros fatores, tais como a definição das expressões semânticas utilizadas nas mensagens trocadas entre os agentes e das mensagens a serem utilizadas nas condições de *input*. Este processo possui uma complexidade maior em relação ao método inverso (para extração de regras lógicas), necessitando portanto, de mais informações a serem fornecidas pelo usuário.

Os modelos para o formato das regras são construídos da maneira a seguir. É importante notar que as regras agora devem ser precedidas por uma expressão contendo uma ação modal que representa a mensagem a ser recebida pelo agente na condição de *input* necessária para a guarda em questão. Esta expressão será denominada como cabeçalho da regra. Voltando às funções de tradução apresentadas no capítulo anterior, este mecanismo implica na introdução de novas transformações que possam ser definidas de forma mais concreta. Sendo assim, é possível realizar uma correspondência entre o inverso dessas transformações em relação às funções de tradução para ambos os sentidos.

Também deve ser informado o conhecimento inicial (e a crença inicial, caso exista) que os agentes possuem, isto é, a informação que cada agente sabe antes do início da execução do programa.

Considerando m o número de agentes presentes no sistema, $i = 1, ..., m$:

- Conhecimento Inicial (inclui Crença Inicial): formado por fórmulas da lógica

de conhecimento (definidas no Capítulo 2) que devem conter obrigatoriamente os termos K_i ou B_i.

$\mathbf{A} ::= K_i\alpha_1 \mid B_i\alpha_2$, onde α_1, α_2 são fórmulas da lógica de conhecimento.

- Regras: possuem testes de conhecimento (ou testes padrões) formados por fórmulas da lógica de conhecimento.

  $[rec_j]$ 'msg(x)': $\mathbf{\Pi} \rightarrow [send]$ 'msg(y)'

  ou

  $[rec_j]$ 'msg(x)': $\mathbf{\Pi} \rightarrow K_i\beta$

  Onde β é uma fórmula da lógica de conhecimento.

  $\mathbf{\Pi} ::= \pi_1 \mid K_i\pi_2$, onde π_1 é uma fórmula da lógica proposicional (teste padrão) e π_2 é uma fórmula da lógica de conhecimento (teste de conhecimento).

- Fórmula de Incerteza (opcional): deve conter a ação modal $[tick]$ em seu consequente.

  $[rec_j]$ 'msg(z)': $B_i(\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_n) \rightarrow [tick]\ B_i(\varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_{n-1}) \wedge K_i\psi_n$

  Onde $n \leq m$ e φ_j e ψ_j são fórmulas da lógica de conhecimento, para $j = 1, ..., m$.

  Observação: As expressões 'msg(x)' simbolizam tanto a utilização de uma expressão semântica (por exemplo, 'YES'), bem como a presença de uma fórmula da lógica de conhecimento.

- Condição de Parada:

  $K_i\mathbf{\Phi} \rightarrow [tick]\ \bot$

  $\mathbf{\Phi}$ é o conjunto de fórmulas válidas da lógica de conhecimento.

A partir destas definições, é possível garantir a forma de introdução da lógica para desenvolver uma especificação (S) de um sistema. Com isso, pode-se revisar as funções de tradução Ψ_a e Ψ_b vistas anteriormente, através da inserção das seguintes transformações:

- ψ_{a3} - aplicada à uma guarda (condição de *input*), extrai o cabeçalho de uma regra.

- ψ_{b3} - aplicada ao cabeçalho da regra, constrói a condição de *input* da guarda.

Com essas novas regras de tradução, pode ser aplicado o método descrito no capítulo anterior para a geração de KBP's. Voltando ao exemplo sugerido, para o jogo de cartas, o modelo para apresentação das regras é o seguinte:

- Conhecimento inicial:

  $K_i c_i$

- Regras:

  $[rec_j]$ 'x': $K_i x \rightarrow [send]$ 'y' (a)

- Fórmula de Incerteza:

  $[rec_j]$ 'x': $B_i(K_0 c_0 \wedge K_1 c_1 \wedge K_2 c_2 \wedge ... \wedge K_m c_m) \rightarrow [tick]\ B_i(K_0 c_0 \wedge K_1 c_1 \wedge K_2 c_2 \wedge ... \wedge K_{m-1} c_{m-1}) \wedge K_i K_m c_m$ (b)

- Condição de Parada:

  $K_i K_0 c_0 \rightarrow [tick]\ \bot$ (c)

Também é preciso estipular o significado das mensagens enviadas pelos agentes. De acordo com as regras do jogo, o jogador deve fazer um palpite na sua vez de jogar, tentando adivinhar qual é a carta da mesa. Isto significa que caso ele acredite que a carta da mesa (considerada como sendo o agente 0) tem o valor 'x', então $B_i K_0 x$. Definindo de outra forma, *send* 'x': $B_i K_0 x \rightarrow [send]\ E_G B_i K_0 x$. A função do operador E_G é de especificar, como no caso do problema das crianças com lama na testa, que a mensagem enviada pelo agente é recebida por todos os outros agentes, ou seja, é do conhecimento de todos.

Estas regras determinam as ações a serem tomadas pelo agente de acordo com o valor da carta recebido. O processo de agora em diante para a geração do KBP segue diretamente do capítulo anterior. O que implica nas seguintes expressões para as guardas, condições de *input* e condição de parada:

- Condições de *input* e Guardas:

  (a) *input*: *there exists* j, rec_j 'x'

  if $K_i x$ **then**

  send 'y'

(b) *input: there exists* j, rec_j 'x'

if $B_i(K_0c_0 \wedge K_1c_1 \wedge K_2c_2 \wedge ... \wedge K_mc_m)$ *then*

add $B_i(K_0c_0 \wedge K_1c_1 \wedge K_2c_2 \wedge ... \wedge K_{m-1}c_{m-1}) \wedge K_iK_mc_m$

(c) *input: for all* j, rec_j nil

add $K_iK_0c_0$

- Condição de Parada:

  until $K_iK_0c_0$

O programa completo, com a inserção de algumas alterações necessárias, é apresentado na próxima seção, onde são feitas a extração das regras lógicas e a verificação dos resultados obtidos.

6.2 Geração da Lógica de Conhecimento a partir de um KBP

Para esta situação, deve ser fornecido um KBP que deverá seguir o modelo apresentado a seguir:

```
program Template_i
  add α                                          # α ∈ A (conhecimento inicial)
  repeat
    case
      input: there exists | for all j, rec_j 'msg(x)'
        if π then                                                  # π ∈ Π
          send 'msg(y)' / add K_iβ
      input: there exists | for all j, rec_j 'msg(z)'
        if B_i(φ_1 ∧ φ_2 ∧ ... ∧ φ_n) then
          add B_i(φ_1 ∧ φ_2 ∧ ... ∧ φ_{n-1}) ∧ K_iψ_n
          if K_i(ψ_1 ∧ ψ_2 ∧ ... ∧ ψ_n) then send msg
      ...
    end case
  until final                          # ou until φ, onde φ ∈ Φ (condição de parada)
end
```

Figura 6.1: Exemplo de programa para geração da lógica

Considerando o que foi feito no método anterior com a criação de um modelo para as regras lógicas capaz de definir uma especificação S para um sistema. Analo-

gamente, o modelo para KBP's aqui apresentado define a estrutura de um programa P a ser utilizado nos métodos de tradução.

Para o problema do jogo de cartas, é introduzido como exemplo o KBP baseado nas guardas obtidas na seção anterior. Deve-se notar que foram feitas algumas alterações nas guardas, mas apenas no sentido de corrigir detalhes relacionados à organização procedural do programa. Estas alterações são comentadas logo adiante e não influenciam diretamente a lógica incluída no programa.

program $Cards_i$
  add $K_i c_i$
  add $B_i(K_0 c_0 \wedge K_1 c_1 \wedge K_2 c_2 \wedge ... \wedge K_n c_n)$
  $m \leftarrow n$
  repeat
    case
      (a) *input: there exists* j, rec_j 'x'
        if $K_i x$ **then**
          send 'y'
          $y \in \{c_1, ..., c_m\}$
      (b) *input: there exists* j, rec_j 'x'
        if $B_i(K_0 c_0 \wedge K_1 c_1 \wedge K_2 c_2 \wedge ... \wedge K_m c_m)$ *then*
          add $K_i K_m x$
          $m \leftarrow m - 1$
      (c) *input: for all* j, rec_j nil
        add $K_i K_0 x$
        $c_0 \leftarrow x$
    end case
  until $K_i K_0 c_0$
end

Figura 6.2: Programa $Cards_i$

As alterações inseridas no programa em relação às guardas que foram obtidas na seção anterior, fazem respeito à estrutura procedural do programa. Isto é necessário para que o processamento ocorra de forma a não encontrar obstruções ou mesmo para melhorar a organização e o entendimento do programa.

Primeiramente, pode-se notar a introdução do comando $m \leftarrow n$ que simplesmente cria uma variável auxiliar (m) que possa ser modificada durante o processamento, já que o número de jogadores (n) não pode variar no jogo. Essa modificação é feita na guarda (b), explicada mais a frente.

Na guarda (a), é indicada através da linha $y \in \{c_1, ..., c_m\}$ a necessidade de algum procedimento para escolha do novo palpite ('y') a ser feito pelo jogador. Como não há uma definição específica sobre este procedimento (uma escolha aleatória entre os elementos do conjunto seria satisfatória) e ainda, não é prevista a utilização de operadores da lógica de conhecimento para esse mesmo procedimento, uma abordagem mais geral como a que foi feita já é suficiente.

Por simplicidade, na guarda (b) apenas a expressão que adiciona conhecimento ($K_i K_m x$) é incluída na base de conhecimento do agente, juntamente com um comando para atualizar o contador ($m \leftarrow m - 1$) que controla a quantidade de termos presentes na fórmula de incerteza do agente, de acordo com a evolução do seu conhecimento durante o andamento do sistema. É importante também ressaltar que a ordenação feita para os agentes na fórmula de incerteza não coincide com a numeração real dada aos jogadores. Pois essa ordenação é válida somente nos estados epistêmicos do agente em questão, e portanto, cada agente identifica os outros jogadores segundo a sua própria ordenação. Isto ocorre da mesma maneira já vista no Capítulo 4, para o problema das crianças com lama na testa.

E finalmente, na guarda (c) é feita somente uma associação de variáveis através do comando $c_0 \leftarrow x$ para que a condição de parada ($K_i K_0 c_0$) possa ser atendida.

Assim, para iniciar o processo de geração da lógica, primeiramente são definidas as expressões semânticas utilizadas no programa:

send 'x': $B_i K_0 c_0 \rightarrow [send]\ E_G B_i K_0 x$

final: $K_i K_0 c_0$

Então, as regras lógicas obtidas são:

- (a): $K_i B_j K_0 x \wedge K_i x \rightarrow [send]$ 'y'
- (b): $K_i B_j K_0 x \wedge K_i \neg x \rightarrow [tick]\ K_i K_m x$
- (c): Não gera regra.
- Condição de Parada: $K_i K_0 c_0 \rightarrow [tick]\ \bot$

Fazendo uma comparação entre os resultados obtidos, pode-se perceber que as regras lógicas que foram obtidas pelo método de extração de regras lógicas e as que

foram introduzidas no método de geração de KBP's possuem grande compatibilidade. As diferenças ocorrem principalmente quando se deseja construir um KBP a partir de regras lógicas, já que é realmente necessário que exista interferência externa neste processo. Pois como foi enfatizado anteriormente, um KBP é uma estrutura de mais alto nível para expressar as informações contidas nas regras lógicas. Assim, não se pode simplesmente realizar uma transformação direta a partir da inversão do método de tradução para extração de regras lógicas para se construir um KBP.

Contudo, tais diferenças encontradas nos conjuntos de regras lógicas utilizados em cada método, se resumem a algumas possíveis expressões lógicas decorrentes de alterações semelhantes àquelas feitas para o KBP estudado aqui. E estas alterações, como visto anteriormente, visam criar procedimentos auxiliares para facilitar ou organizar a execução do programa. As expressões lógicas produzidas dessa forma, portanto, não devem infringir a especificação inicial desejada para o programa, caso as inclusões feitas pelo usuário sejam corretas.

Por fim, através de um argumento semelhante ao que foi utilizado no parágrafo anterior, as alterações inseridas nos métodos de tradução neste capítulo não invalidam os teoremas descritos no capítulo anterior, sobre a relação entre os KBP's e as especificações baseadas em regras lógicas. Relação essa definida de forma a afirmar que todo KBP satisfaz a especificação que o gerou ou que foi obtida através dele.

Capítulo 7

Aplicações

Neste capítulo são apresentados mais dois exemplos de modelagem de sistemas multi-agentes envolvendo conhecimento, cada um deles oferece pontos de interesse para estudo sobre os KBP's e sua utilização. O primeiro exemplo trata o jogo *Cluedo*, discorrendo sobre as suas regras e a lógica inserida no problema. E o segundo exemplo estuda o algoritmo para PIF, principalmente sob o aspecto assíncrono do sistema. Para ambos os KBP's desenvolvidos, existem exemplos de execução para verificação do funcionamento dos programas.

7.1 Programa Cluedo

Agora será estudado um KBP desenvolvido para modelar um sistema baseado em um conhecido jogo de tabuleiro, chamado *Cluedo* (aqui conhecido como Detetive), onde os jogadores devem desvendar um assassinato acontecido em uma mansão. O vencedor do jogo é aquele que descobre primeiro as respostas para as três perguntas que elucidam o mistério (Quem é o assassino?, Como ele matou?, Onde ocorreu o crime?). A essência do problema envolvida no desenrolar do jogo, que basicamente consiste no que os jogadores sabem e o que eles supõe sobre os detalhes do crime, fornece uma aproximação interessante para a lógica de conhecimento e consequentemente também para os KBP's.

As questões relacionadas ao conhecimento e crença dos agentes (os jogadores) presentes neste jogo podem ser mapeadas de forma consistente em um KBP, além de adicionar mais alguns pontos de interesse para o estudo apresentado neste trabalho que não estão presentes no problema das crianças com lama na testa, como

por exemplo, o fato de que os agentes possuem uma incerteza sobre o estado dos outros agentes, enquanto que no problema das crianças, os agentes possuem conhecimento sobre o estado dos outros agentes e a incerteza está no seu próprio estado. Isto sugere um enfoque diferente na construção do KBP para essa situação, além de uma maior complexidade inserida no programa devido às próprias características dos fatores analisados no jogo, fornecendo um maior número de variáveis a serem consideradas.

7.1.1 Sobre o Jogo

O Dr. Black é encontrado morto na manhã de sábado. O corpo foi encontrado ao pé da escada que leva à adega, em um local marcado com um "X".

A causa da morte ainda precisa ser descoberta, mas existem vários objetos encontrados pelos vários cômodos da casa que poderiam ter sido utilizados na execução do crime:

- Armas:
  - *Dagger*
  - *Candlestick*
  - *Revolver*
  - *Rope*
  - *Lead pining*
  - *Spanner*
- Aposentos:
  - *Lounge*
  - *Dining room*
  - *Study*
  - *Ballroom*
  - *Conservatory*
  - *Kitchen*

São considerados suspeitos do assassinato os hóspedes que estiveram na casa durante o final de semana. São eles:

- Suspeitos:
  - *Col. Mustard (Yellow)*
  - *Prof. Plum (Purple)*
  - *Rev. Green (Green)*
  - *Mrs. Peacock (Blue)*
  - *Miss Scarlett (Red)*
  - *Mrs. White (White)*

Ao início do jogo, uma carta de suspeito, uma carta de arma e uma carta de aposento (as cartas do assassino) são colocadas em um envelope marcado com um "X". Para resolver o mistério, é preciso descobrir quem cometeu o crime, com que

arma e em qual aposento o crime foi cometido (as três cartas no envelope revelarão as respostas).

7.1.2 Modelagem do Sistema

Cada jogador receberá algumas cartas (que não podem ser vistas pelos outros jogadores), imediatamente o jogador pode eliminar estes personagens, armas e aposentos de sua investigação.

Durante o jogo, os jogadores terão várias chances de fazer uma suposição sobre as cartas do assassino, questionando os outros jogadores se eles possuem uma das cartas perguntadas. A jogada termina quando uma das cartas perguntadas é apresentada por um dos jogadores (somente ao jogador que fez a suposição). Por hábil dedução, um dos jogadores chegará primeiro à conclusão sobre as cartas do assassino.

Modelando o comportamento dos jogadores durante o andamento do jogo, as ações essenciais realizadas pelos agentes serão interpretadas no KBP da seguinte forma:

- $ask(i, (x, y, z))$

  O jogador i pergunta pelas cartas (x, y, z).

- $show(i, x)$

  A carta x é mostrada ao jogador i.

- $nonshow(i)$

  Nenhuma carta é mostrada ao jogador i.

- $accuse(i, (x, y, z))$

  O jogador i afirma que (x, y, z) são as cartas do assassino.

- *success*

  A acusação foi correta.

As ações *accuse* e *success* são irrelevantes para o andamento do jogo pois ocorrem somente ao final deste. Um ponto importante a frisar para esta modelagem do jogo, é que não se considera possível que os agentes/jogadores façam uma acusação

falsa, ou seja, somente quando o agente atinge o conhecimento completo sobre as cartas do assassino é que a ação *accuse* é feita.

Somente as combinações de *ask* + *show* e *ask* + *nonshow* possibilitam um movimento no jogo. As ações correspondentes a estas combinações são:

- $show(i, (x, y, z); j, x)$
- $show(i, (x, y, z); j, y)$
- $show(i, (x, y, z); j, z)$

  O jogador j mostra uma das cartas que ele possua dentre (x, y, z) ao jogador i.

- $nonshow(i, (x, y, z))$

  Nenhuma carta é mostrada ao jogador i.

O incremento na informação dos agentes feito a cada movimento do jogo é dado por:

1. No caso de uma resposta *show*:

   - pré-condição: $K_j x \vee K_j y \vee K_j z$

     O jogador j possui pelo menos uma das cartas perguntadas pelo jogador i.

   - programa: $show(i, (x, y, z); j, x)$

     O jogador j mostra somente uma carta dentre aquelas perguntadas ao jogador i (no caso, x).

   - pós-condição: $K_i K_j x$

     O jogador i sabe que o jogador j possui a carta que foi mostrada (x).

2. No caso de uma resposta *nonshow*:

   - pré-condição: $K_j \neg x \wedge K_j \neg y \wedge K_j \neg z$

     O jogador j não possui nenhuma das cartas perguntadas pelo jogador i.

   - programa: $nonshow(i, (x, y, z))$

     Nenhuma carta é mostrada ao jogador i.

- pós-condição: $K_iK_j\neg x \wedge K_iK_j\neg y \wedge K_iK_j\neg z$

  O jogador i sabe que o jogador j não possui nenhuma das cartas (x, y, z) perguntadas.

As principais regras lógicas obtidas através desta modelagem, considerando o método de tradução revisado (no Capítulo 6), são então as seguintes:

- $[rec_j\ ask(j, (susp, weap, room))]$: $K_i susp \rightarrow [send_j]\ show(j, susp)$ ou $K_i weap \rightarrow [send_j]\ show(j, weap)$ ou $K_i room \rightarrow [send_j]\ show(j, room)$
- $[rec_j\ ask(j, (susp, weap, room))]$: $K_i\neg susp \wedge K_i\neg weap \wedge K_i\neg room \rightarrow [send_j]\ nonshow(j)$
- $[rec_j\ show(j, susp)]$: $K_iK_j susp$
- $[rec_j\ nonshow(j)]$: $K_iK_j\neg susp \wedge K_iK_j\neg weap \wedge K_iK_j\neg room$

As cartas do assassino serão consideradas como pertencendo ao agente 0, portanto para que a condição de parada seja atingida, a ação *success* deve ser executada através de uma verificação. Para um agente i, a acusação será correta quando $K_iK_0x \wedge K_iK_0y \wedge K_iK_0z$ for verdade, sendo (x, y, z) as cartas do assassino (suspeito, arma, aposento).

7.1.3 Apresentação do Programa

Para simplificação das fórmulas envolvendo os vários tipos de cartas existentes no jogo será introduzida a expressão $cards_i$ que representa as cartas que o jogador i possui, ou seja, $cards_i \leftrightarrow (susp_i \wedge weap_i \wedge room_i)$.

program $Cluedo_i$
  add $K_i cards_i$
  add $B_i(K_0 cards_0 \wedge K_1 cards_1 \wedge K_2 cards_2 \wedge ... \wedge K_{n-1} cards_{n-1})$
  $m \leftarrow n - 1$
  repeat
    case of
    (a) *input*: *for all* $j \neq i$, rec_j nil
      if $K_i K_j cards_j$ **then**
        add $K_i K_0 cards_0$
        send $accuse(i, (susp_0, weap_0, room_0))$
    (b) *input*: *for all* $j \neq i$, rec_j nil
      if $\neg K_i K_j cards_j$ **then**
        send $ask(i, (susp_x, weap_x, room_x))$, $x < m$
    (c) *input*: *there exists* $j \neq i$, rec_j $ask(j, (susp_y, weap_y, room_y))$
      if $K_i susp_y \vee K_i weap_y \vee K_i room_y$ **then**
        send $show(j, susp_i)$ ou $show(j, weap_i)$ ou $show(j, room_i)$
                # $susp_y = susp_i$ ou $weap_y = weap_i$ ou $room_y = room_i$
      else *send* $nonshow(j)$
    (d) *input*: *there exists* $j \neq i$, rec_j $show(j, susp_x)$ ou $show(j, weap_x)$ ou $show(j, room_x)$
      add $K_i K_j susp_j$ ou *add* $K_i K_j weap_j$ ou *add* $K_i K_j room_j$
                # $susp_x = susp_i$ ou $weap_x = weap_i$ ou $room_x = room_i$
      if $K_i K_j cards_j$ **then**
        $m \leftarrow m - 1$
        add $B_i(K_0 cards_0 \wedge K_1 cards_1 \wedge K_2 cards_2 \wedge ... \wedge K_m cards_m)$
    (e) *input*: *there exists* $j \neq i$, rec_j $nonshow(i)$
      add $K_i K_j \neg susp_x \wedge K_i K_j \neg weap_x \wedge K_i K_j \neg room_x$
      if $K_i K_j cards_j$ **then**
        $m \leftarrow m - 1$
        add $B_i(K_0 cards_0 \wedge K_1 cards_1 \wedge K_2 cards_2 \wedge ... \wedge K_m cards_m)$
    end case
  until *success*
end

Figura 7.1: Programa $Cluedo_i$

E o programa que verifica quando a condição *success* é atingida, que é executado por um agente considerado especial, já que ele não toma parte como jogador, pois conhece desde o início do jogo as cartas do assassino.

```
program Cluedo_0                                                    # murderer
  add K_0cards_0
  repeat
    input: there exists k, rec_k accuse(k, (susp_z, weap_z, room_z))
    if K_0susp_z ∧ K_0weap_z ∧ K_0room_z then
      success                                  # susp_z = susp_0, weap_z = weap_0, room_z = room_0
  until success
end
```

Figura 7.2: Programa $Cluedo_0$

Pode-se perceber várias semelhanças entre este programa e o que foi apresentado no Capítulo 6, em especial em relação à fórmula de incerteza. Isto ocorre devido à natureza da situação em si, que trata de um jogo de cartas, onde os jogadores geralmente tentam adivinhar quais são as cartas em poder dos outros jogadores, ou seja, os agentes tomam decisões de acordo com a crença que eles possuem sobre o estado de conhecimento dos outros agentes.

E ainda deve se notar a semelhança nos comandos inseridos no programa, tais como $m \leftarrow n-1$, em relação ao que foi feito no capítulo anterior. Citando em especial a guarda (b), onde uma ação *ask* é feita. Somente uma escolha simples de cartas a partir de um conjunto que o agente considera possível ($susp_x \in \{susp_1, ..., susp_m\}$, por exemplo) já é o bastante. Porém, é possível criar um procedimento (contendo inclusive operadores da lógica de conhecimento) capaz de fazer escolhas mais vantajosas para os jogadores. Isto não foi realizado aqui por motivos de simplificação do programa, já que a inclusão de um procedimento deste tipo acarretaria em uma complexidade desnecessária para a demonstração de um exemplo de execução, a ser realizado na próxima seção.

7.1.4 Exemplo de Execução

Para testar o programa, é utilizado um exemplo com as seguintes simplificações:

- 3 jogadores (1, 2, 3)
- 4 cartas de suspeitos (yellow, purple, green, blue)

- 4 cartas de armas (dagger, candlestick, revolver, rope)
- 4 cartas de aposentos (lounge, dining room, study, ballroom)

Uma diferença existente neste caso em relação ao problema das crianças é que os turnos entre os jogadores não se distribuem igualmente, um jogador pode ter direito a mais jogadas do que outros. Esse fator não está sendo levado em consideração na construção do KBP, pois ele ocorre justamente na chamada ao programa $Cluedo_i$ e portanto, pode ser considerado como um fator externo ao sistema. O exemplo a seguir ilustra uma possível partida do jogo:

Ordem dos turnos: 1, 2, 3, 1, 1
Cartas do assassino: *blue*, *rope*, *ballroom*

$Cluedo_1$	$Cluedo_2$	$Cluedo_3$
Conhecimento inicial: $K_1 yellow \wedge K_1 dagger \wedge K_1 lounge$	Conhecimento inicial: $K_2 purple \wedge K_2 candles. \wedge K_2 dining$	Conhecimento inicial: $K_3 green \wedge K_3 revolver \wedge K_3 study$
Início: b) $ask(1, (purple, revolver, ballr.))$	Início:	Início:
	c) $show(1, purple)$	c) show$(1, revolver)$
d) $K_1 K_2 purple$ d) $K_1 K_3 revolver$		
	b) $ask(2, (green, rope, lounge))$	
c) $show(2, lounge)$		c) show$(2, green)$
	d) $K_2 K_3 green$ d) $K_2 K_1 lounge$	
		b) $ask(3, (blue, dagger, dining))$
c) $show(3, dagger)$	c) $show(3, dining)$	
		d) $K_3 K_1 dagger$ d) $K_3 K_2 dining$
b) $ask(1, (green, rope, ballr.))$		
	c) $nonshow(1)$	c) $show(1, green)$
e) $K_1 K_2 \neg rope \wedge K_1 K_2 \neg ballr.$ d) $K_1 K_3 green$ b) $ask(1, (yellow, dagger, study))$		
	c) $nonshow(1)$	c) $show(1, study)$
e) $K_1 K_2 \neg study$ d) $K_1 K_3 study$ a) $accuse(1, (blue, rope, ballr.))$ ■	■	■

Tabela 7.1: Exemplo de execução para o programa $Cluedo_i$

As combinações para ordem dos turnos podem ser muitas, e a execução do programa pode se estender por muitas fases, considerando que se pode aumentar também a quantidade de cartas no jogo (suspeitos, armas e aposentos), mas pode-se garantir que o programa sempre termina, usando um argumento semelhante ao que foi visto para o caso do problema das crianças com lama na testa.

7.2 Programa PIF

O algoritmo para PIF (*Propagation of Information with Feedback*), apresentado no Capítulo 3 é agora revisitado com o objetivo de se modelar o seu funcionamento através de KBP's. São demonstradas duas versões para os programas, assim como no caso do problema das crianças com lama na testa visto no Capítulo 4. Uma versão mais simples, considerando o modelo síncrono, e outra que estuda a evolução do conhecimento para os agentes de forma mais abrangente, utilizando o modelo assíncrono.

7.2.1 Caso Síncrono

Para o caso síncrono, é proposto um programa mais elementar onde apenas é necessário que cada agente receba uma única resposta de todos os outros agentes. Este programa pode ser estendido para um número de interações cada vez maior, até que se atinja o conhecimento comum da mensagem enviada para todos os agentes. Isto ocorre, porém, apenas no caso síncrono, onde cada agente pode "prever" o conhecimento dos outros agentes através do sinal de relógio do sistema. Mas para isso, deve ser considerado também que as propriedades de *safety*, *fairness* e *liveness* são respeitadas para os programas.

Isto significa que o meio não possui falhas, ou seja, as mensagens são sempre entregues aos seus destinatários de forma intacta (*safety*) e em um tempo finito (*liveness*). E ainda, todos os agentes devem executar seus programas de maneira uniforme, não havendo restrições em seu processamento que não estejam definidas no próprio programa (*fairness*). Estas propriedades são vistas de forma mais detalhada no Apêndice B.

O programa para o agente iniciador, definido como sendo o agente 1, possui algumas diferenças em relação aos outros programas para os todos os outros agentes restantes no sistema. Pois é necessário que exista uma guarda especial para iniciar o envio das mensagens, e para a situação considerada no caso síncrono, o agente iniciador também não precisa reenviar as mensagens recebidas por ele.

```
program PIF_1
  add K_1 x
  repeat
    case of
    (a) input: for all j ≠ 1, rec_j nil
      if initial then send 'x'
    (b) input: for all j ≠ 1, rec_j 'x'
      add K_1 K_j x
    end case
  until K_1(K_2 x ∧ K_3 x ∧ ... ∧ K_n x)
end
```

Figura 7.3: Programa PIF_1 (iniciador)

As regras lógicas obtidas são:

- $K_1 x \rightarrow [send]$ 'x'
- $[tick]\ K_1 K_j x$
- $K_1(K_2 x \wedge K_3 x \wedge ... \wedge K_n x) \rightarrow [tick]\ \bot$

O restante dos agentes executam o seguinte programa:

```
program PIF_i
  repeat
    case of
    (a) input: for all j ≠ i, rec_j 'x'
      add K_i K_j x
      if ¬K_i K_k x then send_k 'x'
    end case
  until K_i(K_1 x ∧ K_2 x ∧ ... ∧ K_n x)
end
```

Figura 7.4: Programa PIF_i (síncrono)

As regras lógicas obtidas são:

- $[tick]\ K_i K_j x$
- $\neg K_i K_k x \rightarrow [send_k]$ 'x'

- $K_i(K_1x \wedge K_2x \wedge ... \wedge K_nx) \rightarrow [tick] \perp$

Devido à presença do sinal de relógio do sistema, os agentes podem simplesmente reenviar as mensagens recebidas a cada passo para todos os agentes exceto àquele que enviou a mensagem, até que o nível de conhecimento definido na condição de parada seja atingido.

7.2.2 Caso Assíncrono

Para o caso assíncrono, é construído um KBP que propõe uma tentativa de obtenção de conhecimento comum para os agentes do sistema. Porém, devido à ausência de simultaneidade nos modelos assíncronos, não é possível se obter conhecimento comum para este tipo de sistemas, (PANANGADEN, TAYLOR, 1992). Como alternativa a ser utilizada na construção dos KBP's, é proposta a obtenção da expressão de conhecimento $E_G \wedge {E_G}^2 \wedge {E_G}^3 \wedge ... \wedge {E_G}^n$, onde n é o número de agentes no sistema. Para o caso $n = 3$, isso equivaleria a seguinte fórmula:

$$K_1x \wedge K_2x \wedge K_3x \wedge K_1K_2x \wedge K_1K_3x \wedge K_2K_1x \wedge K_2K_3x \wedge K_3K_1x \wedge K_3K_2x \wedge$$
$$K_1K_2K_3x \wedge K_1K_3K_2x \wedge K_2K_1K_3x \wedge K_2K_3K_1x \wedge K_3K_1K_2x \wedge K_3K_2K_1x$$

Por simplicidade, será utilizada apenas a expressão ${E_G}^nx$ como condição de parada nos programas, pois a evolução do conhecimento de cada agente ocorre de forma progressiva. Portanto, basta que seja obtida apenas a última cadeia de expressões, equivalente a ${E_G}^nx$, para satisfazer a especificação desejada para o sistema.

A fórmula $\mathbf{K}x$ existente nos programas a seguir, representa uma sequência de operadores $K_aK_bK_cx$ que definem uma determinada ordem de envio de mensagens entre os agentes. Esta fórmula tem por finalidade apenas simplificar o entendimento do programa.

```
program PIF_1 Assync
  add K_1 x
  repeat
    case of
    (a) input: for all j ≠ 1, rec_j nil
      if initial then send 'x'
    (b) input: there exists j ≠ 1, rec_j Kx
      add K_1 K_j Kx
      send_{k≠j} K_j Kx
    end case
  until E_G^n x
end
```

Figura 7.5: Programa PIF_1 *Assync* (iniciador)

As regras lógicas obtidas são:

- $K_1 x \rightarrow [send]$ 'x'
- $\mathbf{K}x \rightarrow [send_{k \neq j}]\ K_j \mathbf{K}x$
- ${E_G}^n x \rightarrow [tick]\ \bot$

O restante dos agentes executam o seguinte programa:

```
program PIF_i Assync
  add K_1 x → x
  repeat
    case of
    (a) input: there exists j ≠ i, rec_j Kx
      add K_i K_j Kx
      send_{k≠j} K_j Kx
    end case
  until E_G^n x
end
```

Figura 7.6: Programa PIF_i *Assync*

As regras lógicas obtidas são:

- $\mathbf{K}x \rightarrow [send_{k \neq j}]\ K_j \mathbf{K}x$

- $E_G{}^n x \rightarrow [tick] \perp$

É importante observar a existência da fórmula $K_1 x \rightarrow x$, válida para todos os agentes, no caso assíncrono. Isto ocorre devido à mudança provocada no algoritmo com o uso dos operadores K_i nas trocas de mensagens entre os agentes. A inclusão dessa fórmula é necessária para que todos os agentes no sistema, além do iniciador, atinjam a condição de parada. E essa inclusão pode ser justificada através da adição na especificação de propriedades do sistema de que todos os agentes sabem que o agente 1 é o iniciador, portanto um agente i obtém conhecimento sobre a mensagem inicial 'x' através da expressão $K_i K_1 x$. A utilização deste mecanismo será demonstrada com a verificação de uma possível execução dos programas, na próxima seção.

7.2.3 Exemplo de Execução

A tabela a seguir, exemplifica uma possível execução dos KBP's gerados para o algoritmo para PIF, considerando o caso assíncrono e para um sistema com três agentes (incluindo o agente iniciador).

	PIF_1	PIF_2	PIF_3
	Início:	Início:	Início:
e_0	$K_1 x$, *send* 'x'		
e_{1a}		$K_2 K_1 x$, $send_3 K_1 x$	
e_{2a}			$K_3 K_2 K_1 x$, $send_1 K_2 x$*
e_{1b}			$K_3 K_1 x$, $send_2 K_1 x$
e_{2b}		$K_2 K_3 K_1 x$, $send_1 K_3 x$*	
e_{3b}	$K_1 K_2 K_3 x$, $send_3 K_2 K_3 x$		
e_{3a}	$K_1 K_3 K_2 x$, $send_2 K_3 K_2 x$		
e_{4b}			$K_3 K_1 K_2 K_3 x$, $send_2 K_2 K_3 x$**
e_{5b}		$K_2 K_3 K_2 K_3 x$, $send_1 K_3 K_2 K_3 x$	
e_{4a}		$K_2 K_1 K_3 K_2 x$, $send_3 K_3 K_2 x$**	
e_{5a}			$K_3 K_2 K_3 K_2 x$, $send_1 K_2 K_3 K_2 x$

Tabela 7.2: Exemplo de execução para o algoritmo PIF (assíncrono)

A coluna à esquerda da tabela informa qual o evento associado à cada guarda executada nos programas. A partir do evento e_0 onde o agente iniciador (1) envia a mensagem inicial para os agentes restantes no sistema (2 e 3), seguem-se então duas linhas possíveis de trocas de mensagens (a e b) entre os agentes. Deve-se notar que nas ações marcadas com * (e_{2a} e e_{2b}), é aplicada a expressão $K_1 x \rightarrow x$ para simplificar a demonstração. E nas ações marcadas com ** (e_{4a} e e_{4b}), são obtidas as

expressões finais para os agentes 2 e 3, através da aplicação do axioma de conhecimento ($K_i\varphi \rightarrow \varphi$), que é válido para cada agente.

Em um sistema assíncrono, existem muitas execuções possíveis para este algoritmo, pode-se ter uma idéia disso através da Figura 3.6 que demonstra as execuções possíveis para um caso básico do algoritmo para PIF. Mas os programas sempre atingirão a condição de parada, pois qualquer execução possível sempre alcança os mesmos eventos, alterando apenas a ordem entre eles. Assim, em uma execução podem ser necessárias mais ou menos fases (equivalentes ao número de guardas executadas) do que em outras execuções possíveis para que a condição de parada seja atingida.

Capítulo 8

Conclusão

A apresentação das idéias contidas neste trabalho pretende demonstrar o poder que os KBP's e a lógica dinâmica de conhecimento possuem para auxiliar a compreensão e a solução de problemas que envolvem os sistemas multi-agentes, tratando a noção de conhecimento entre vários agentes que interagem entre si. Com o desenvolvimento de uma arquitetura baseada nos conceitos de sistemas distribuídos e a especificação de uma linguagem que define o formato da estrutura de um programa, é possível demonstrar, tanto através de exemplos quanto da apresentação de uma prova por indução, o correto funcionamento dos programas construídos para simular o raciocínio dos agentes presentes no problema das crianças com lama na testa, tanto na versão síncrona quanto na versão assíncrona.

Com a criação de um método de tradução capaz de extrair as regras lógicas a partir de um KBP que esteja de acordo com as especificações de linguagem e com as devidas restrições descritas na seção de tradução, é exibida a possibilidade de se fornecer a visualização mais formal de um sistema, facilitando assim, a verificação de determinadas propriedades (conhecimento comum, por exemplo) presentes na lógica do problema.

Com o estudo dos KBP's e dos conceitos relativos apresentados, também é proposta a construção de uma ferramenta computacional capaz de implementar a construção de KBP's, bem como a tradução destes para a lógica, em um sistema semelhante ao que se poderia chamar de um compilador KBP. É tratado um simples jogo de cartas como exemplo, para a verificação da utilização dos métodos de forma mais prática.

Ainda é feita a exemplificação de uma outra situação que analisa o conhecimento envolvido em um sistema multi-agentes modelado em um KBP, através do estudo do jogo *Cluedo*. Isto possibilita a abordagem de mais um ponto de vista sobre a questão do tratamento de sistemas desse tipo utilizando KBP's, já que é dado ao problema estudado para este jogo um enfoque diferente em relação ao tratamento do conhecimento, quando se faz uma comparação ao que foi visto para o caso do problema das crianças com lama na testa. Além do programa *Cluedo* demonstrar uma maior complexidade no seu desenvolvimento, devido ao maior número de variáveis relacionadas ao problema.

Também é realizada uma análise do algoritmo para PIF que evidencia mais um exemplo de utilização dos métodos de tradução criados, de forma bem simples e com a verificação de algumas observações importantes sobre a execução de sistemas multi-agentes nos modelos síncrono e assíncrono. Principalmente para o último caso, onde são vistos alguns fatos interessantes relacionados a conhecimento para os sistemas assíncronos.

Sobre os apêndices incluídos, no primeiro deles é indicada uma prova de corretude forte para o programa construído para tratar o problema das crianças com lama na testa, que pode ser estendida a vários outros problemas modelados em KBP's. E através da definição de conceitos ligados aos sistemas multi-agentes, pode ser apontada uma direção para o desenvolvimento de especificações formalmente amparadas para a modelagem desse tipo de sistemas, seguindo a estrutura apresentada para os KBP's neste trabalho. E no segundo apêndice, também é expressa a possibilidade de criação de especificações em função da declaração de propriedades gerais relacionadas a algoritmos, definidas de forma simples e direta de acordo com as características do problema a ser modelado.

Um objetivo desejável a ser alcançado é a aplicação do estudo aqui feito no sentido de se obter a realização de alguma tarefa prática relacionada aos sistemas multi-agentes envolvendo conhecimento, como por exemplo a implementação de protocolos de comunicação ou de jogos entre outras opções, com o uso de uma ferramenta computacional. O que forneceria uma outra forma de construção de aplicativos, através da utilização dos KBP's.

Por fim, pode ser proposto como trabalho futuro, um aprofundamento dos métodos de tradução para extração da lógica e construção de KBP's. Tanto para casos mais específicos, através do estabelecimento de regras de tradução mais rígidas, quanto para casos mais gerais, com a introdução de heurísticas para a escolha de regras mais apropriadas a cada situação. E ainda, considerando uma abordagem mais profunda, existe a possibilidade de se incluir operadores referentes à lógica temporal, a fim de se verificar especificações mais complexas relacionadas à propriedades temporais envolvidas nos sistemas multi-agentes.

Apêndice A

Corretude Forte

Quando se está construindo ou analisando um sistema multi-agentes, geralmente existem algumas propriedades que se deseja que o sistema satisfaça. Várias vezes, se planeja construir um protocolo a partir destas propriedades desejadas para o sistema. Estas propriedades podem ser definidas como a especificação do sistema. De acordo com o formalismo existente para os sistemas multi-agentes, uma especificação pode ser identificada como uma classe de sistemas interpretados. Assim, um sistema interpretado I satisfaz uma especificação σ se ele faz parte desta classe, ou seja, $I \in \sigma$, (FAGIN, HALPERN, et al., 1995).

Como já visto no Capítulo 2, define-se um protocolo P_i para um agente i como sendo uma função de um conjunto L_i dos estados locais do agente para conjuntos não-vazios de ações de um conjunto ACT_i das ações possíveis para o agente i. Define-se também um protocolo conjunto P como sendo uma tupla $(P_1, ..., P_m)$ que consiste de protocolos P_i, para cada um dos agentes $i = 1, ..., m$.

Então, já que o protocolo conjunto P descreve o comportamento dos agentes no sistema, é desejável que possam ser geradas as execuções deste protocolo. Para isso, utiliza-se a variável chamada contexto, onde P está sendo executado. Formalmente, um contexto γ é uma tupla $(P_e, \mathcal{G}_0, \tau, \Psi)$, onde P_e é o protocolo do ambiente, $\mathcal{G}_0$ é o conjunto dos estados iniciais, τ é uma função de transição, e Ψ um conjunto de execuções admissíveis.

Com o protocolo conjunto P e o protocolo do ambiente P_e, pode-se saber como os agentes do sistema irão agir em qualquer ponto, mas é preciso saber também como as suas ações irão afetar o sistema. A função de transição τ caracteriza isso. De-

finindo uma ação comum como sendo a tupla da forma $(a_e, a_1, ..., a_m)$, onde a_e é a ação executada pelo ambiente e a_i a ação executada pelo agente i. Cada ação comum é associada a um estado global através da função de transição τ. Assim, $\tau(a_e, a_1, ..., a_m)g = g'$, significa que g' é o resultado da execução da ação conjunta $(a_e, a_1, ..., a_m)$ no estado g.

É dito que r é uma execução de um protocolo conjunto $P = (P_1, ..., P_m)$ no contexto $\gamma = (P_e, \mathcal{G}_0, \tau, \Psi)$ se $r(0) \in \mathcal{G}_0$. E $r(n+1)$ segue de acordo com τ a partir de $r(n)$ por uma ação comum determinada por (P_e, P) em $r(n)$, para $m \geqslant 0$ e $r \in \Psi$. Define-se então $\mathbf{R}(P, \gamma)$ como sendo o sistema que consiste de todas as execuções de P no contexto γ, chamado sistema que representa o protocolo conjunto P no contexto γ.

Em vários casos, existe uma coleção Φ de primitivas e uma interpretação particular π para as proposições em Φ, quando se define um contexto. Assim, é chamado de contexto interpretado o par (γ, π) que consiste de um contexto γ e uma interpretação π. Dado o protocolo conjunto P, diz-se que $\mathbf{I}(P, \gamma, \pi) = (\mathbf{R}(P, \gamma), \pi)$ é o sistema interpretado que representa P no contexto interpretado (γ, π).

Desta forma, existe um grande interesse em se provar a corretude de protocolos e programas com relação a todos os sub-contextos de algum contexto γ. Assim, é definida uma noção de corretude forte (para programas *standard*) a seguir. Um programa **Pg** satisfaz fortemente uma especificação σ (ou é fortemente correto com relação à σ) em um contexto interpretado (γ, π) se todo sistema interpretado consistente com **Pg** em (γ, π) satisfaz σ. De acordo com o seguinte lema:

Lema A.1 *R é consistente com P no contexto γ se e somente se R representa P em algum sub-contexto $\gamma' \sqsubseteq \gamma$.*

Prova A.1 *Se R representa P no sub-contexto $\gamma' \sqsubseteq \gamma$, então é fácil ver que toda execução de R precisa ser consistente com P no contexto γ. Assim, R é consistente com P no contexto γ. Inversamente, supondo que R é consistente com P no contexto $\gamma = (P_e, \mathcal{G}_0, \tau, \Psi)$. Seja $\gamma_R = (P_e, \mathcal{G}_0, \tau, R)$. Como R é um sub-conjunto de Ψ, então $\gamma_R \sqsubseteq \gamma$. É fácil ver que R representa P em γ_R.*

Isto significa que todo sistema interpretado que representa **Pg** em um sub-contexto $\gamma' \sqsubseteq \gamma$ satisfaz σ.

No caso de um programa *standard* **Pg**, existe uma relação entre o sistema (único) que representa **Pg** em um dado contexto interpretado (γ, π), e os sistemas consistentes com **Pg** em (γ, π). Um sistema consistente com **Pg** precisa ser um sub-conjunto dos sistemas que representam **Pg**, porque os sistemas que representam **Pg** consistem de todas as execuções que são consistentes com **Pg** em (γ, π). Esta relação não existe nos programas baseados em lógica, mesmo em situações onde existe um sistema interpretado único que representa o programa. Na verdade, não existe uma relação simples entre os sistemas que são consistentes com **Pg** e os sistemas que representam **Pg** em (γ, π), exceto pelo fato de que um sistema que representa **Pg** em (γ, π) é obviamente também consistente com **Pg** em (γ, π). No entanto, pelo lema A.1, a conexão existente entre os sistemas que são consistentes com **Pg** em (γ, π) e os sistemas que representam **Pg** nos sub-contextos de (γ, π) se mantém:

Lema A.2 *I é consistente com o programa baseado em conhecimento* **Pg** *no contexto interpretado (γ, π) se e somente se I representa* **Pg** *em algum sub-contexto $\gamma' \sqsubseteq \gamma$.*

Prova A.2 *Semelhante ao lema anterior.*

Em comparação com as definições apresentadas para os programas *standard*, é dito que **Pg** satisfaz fortemente σ ou é fortemente correto com relação à σ no contexto interpretado (γ, π) se todo sistema interpretado consistente com **Pg** em (γ, π) satisfaz σ. A motivação para considerar a corretude forte aqui é a mesma do caso para os programas *standard*, pelo lema anterior, A.2, através da prova de corretude forte com relação a um contexto interpretado, prova-se a corretude com relação a todos sub-contextos do sistema, como será feito a seguir para o programa das crianças com lama na testa.

Primeiramente, descreve-se o contexto (γ^{mc}, π^{mc}) correspondente a uma descrição intuitiva do problema das crianças com lama na testa (*muddy children*). Aqui os agentes são o pai e as crianças. Pode-se ver $\gamma^{mc} = (P_e^{mc}, \mathcal{G}_0, \tau, true)$ como um

contexto que registra a passagem de mensagens, onde um agente qualquer (o pai ou uma das crianças) diz uma mensagem, em um determinado passo, que é transmitida para todos os outros agentes no mesmo passo. Os estados iniciais das crianças e do pai descrevem o que eles vêem, os estados posteriores descrevem o que eles escutam. Assim, $\mathcal{G}_0$ consiste de todas as 2^m tuplas da forma $(\langle\,\rangle, X^{-1}, ..., X^{-m}, X)$, onde $X = (x_1, ..., x_m)$ é uma tupla de 0's e 1's, com $x_i = 0$ representando que a criança i está limpa, e $x_i = 1$ representando que a criança i tem a testa suja, e $X^{-i} = (x_1, ..., x_{i-1}, *, x_{i+1}, ..., x_m)$, isto é, X^{-i} difere de X somente na posição i, que contém $*$. Intuitivamente, X^{-i} descreve o que a criança i vê e X descreve a situação real, onde $*$ significa "sem informação". Somente o pai vê todas as crianças, então o seu estado local inicial é X. O estado local inicial do ambiente é vazio, $\langle\,\rangle$. As únicas ações executadas pelas crianças e pelo pai são o envio de mensagens, e estas ações têm o resultado óbvio de mudar os seus estados locais e o estado local do ambiente. O protocolo de ambiente P_e^{mc} existe simplesmente para garantir a entrega de mensagens no mesmo passo para todos os agentes.

As crianças executam o programa baseado em conhecimento apresentado anteriormente. O pai executa o seguinte programa (*standard*):

```
program MC0                                                    # dad
  repeat
    case of
    (a) input: for all i, rec_i nil
      if initial ∧ ⋁_i p_i then
        say "Pelo menos um de vocês está com a testa suja."
        say "Alguém sabe que está com a testa suja?"
    (b) input: for all i, rec_i 'NO'
      say "Alguém sabe que está com a testa suja?"
    (c) input: there exists i, rec_i 'YES'
      final
    end case
  until final
end
```

Figura A.1: Programa MC_0

Aqui, initial é uma proposição primitiva que é verdadeira no estado inicial e a condição, *input*: *there exists i* rec_i 'YES', significa que o pai recebeu uma resposta

'Sim' da criança j no passo anterior (o que satisfaz a condição de término, final, para o programa). Assim, a interpretação π^{mc} representa as proposições p_i (a criança i está suja) e initial de forma óbvia. É simples verificar que no contexto interpretado (γ^{mc}, π^{mc}), o programa baseado em conhecimento MC_i satisfaz a especificação baseada em conhecimento σ^{mc}. Uma criança diz 'Sim' se ela sabe que tem a testa suja ou 'Não', caso contrário.

Uma prova de corretude completa para um caso similar a este pode ser vista em (HALPERN, ZUCK, 1992).

Apêndice B

Propriedades de Algoritmos

Para introduzir as propriedades básicas relacionadas a algoritmos, serão considerados os seguintes protocolos para os processos *Sender* (S) e *Receiver* (R), já vistos no estudo de um problema simples de transmissão de um bit, no Capítulo 2.

- Protocolo S:

initial: $i \leftarrow 0$, *read* y
repeat
  if $\neg K_S K_R x_j$ **then** *send* (i, y)
  else $i \leftarrow i + 1$, *read* y
until final

- Protocolo R:

initial: $i \leftarrow 0$, *read* y
repeat
  if $\neg K_R x_j$ **then** *send* $\neg K_R x_j$
  else
    if $K_R 0$ **then** *write* 0, $j \leftarrow j + 1$
    else *write* 1, $j \leftarrow j + 1$
until final

Figura B.1: Protocolos *Sender* e *Receiver*

Um teorema especial baseado nestes protocolos que fornece um resultado importante para a implementação de KBP's em geral e que pode ser provado através da semântica de sistemas interpretados é o seguinte, (HALPERN, ZUCK, 1992):

Teorema B.1 *Seja I é um sistema interpretado consistente com os protocolos S e R. Então toda execução de I tem a propriedade safety. E*

toda execução de I que satisfaz a propriedade fairness, também possui a propriedade liveness.

As definições das propriedades de *safety, fairness* e *liveness* possuem uma forma geral cada uma. As propriedades de *safety* basicamente afirmam que "uma situação negativa nunca acontece", as propriedades de *fairness* estabelecem condições para que "nenhum participante do sistema seja penalizado na distribuição de recursos do sistema" e as propriedades de *liveness* essencialmente dizem que "um resultado positivo acontecerá em algum instante finito". No programa construído para modelar o problema das crianças com lama na testa, essas propriedades são definidas a seguir:

safety: uma mensagem após ser enviada por um agente não sofre alterações quando é recebida por um outro agente (o meio de transmissão não possui falhas);

fairness: todos os agentes executam os seus respectivos programas em seus turnos (após cada pergunta feita pelo pai das crianças);

liveness: as mensagens enviadas pelos agentes sempre são recebidas em um tempo finito pelos outros agentes (ausência de *deadlocks*).

Estas simples definições para esse tipo de propriedades de um determinado problema podem também ser consideradas como uma especificação para a modelagem do sistema em um KBP. O que pode ser uma vantagem, pois grande parte das vezes é uma tarefa difícil construir uma especificação através da descrição de conceitos formais que devem ter por objetivo definir o comportamento desejado para o sistema.

Referências Bibliográficas

BARBOSA, V. C., 1996, *An Introduction to Distributed Algorithms*, 1st ed., MIT Press, Cambridge, Massachussetts.

COSTA, M. M. C., 1992, *Introdução à Lógica Modal Aplicada à Computação*, 1ª ed., IC, UFRGS, Porto Alegre, Brasil.

DELGADO, C. A. D. M., 2001, *Lógica de Conhecimento e Eventos em Sistemas Assíncronos*, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil, Dezembro.

DELGADO, C. A. D. M., BENEVIDES, M. R. F., 2001, "Dynamic Knowledge Logics", In:*Anais do XXI Congresso da Sociedade Brasileira de Computação*, volume 1, pp. 187–197, Fortaleza, CE, Julho.

DOLEV, D., MOSES, Y., HALPERN, J. Y., 1986, "Cheating husbands and other stories: a case study of knowledge", *Distributed Computing*, v. 1, n. 3, pp. 167–176.

ENGELHARDT, K., van der MEYDEN, R., MOSES, Y., 1998, "Knowledge and the Logic of Local Propositions", In:*Proceedings of the 1998 Conference on Theoretical Aspects of Reasoning about Knowledge (TARK'98)*, pp. 29–41, July.

FAGIN, R., GEANAKOPLOS, J., HALPERN, J. Y., et al., 1999, "The Hierarchical Approach to Modelling Knowledge and Common Knowledge", *International Journal of Game Theory*, v. 28, n. 3, pp. 331–365.

FAGIN, R., HALPERN, J. Y., 1988a, "Belief, Awareness, and Limited Reasoning", *Artificial Intelligence*, v. 34, pp. 39–76.

FAGIN, R., HALPERN, J. Y., 1988b, "I'm OK if you are OK: on the notion of trusting communication", *Journal of Philosophical Logic*, v. 17, n. 4, pp. 329–354.

FAGIN, R., HALPERN, J. Y., MOSES, Y., et al., 1995, *Reasoning About Knowledge*, 1st ed., MIT Press, Cambridge, Massachussetts.

FAGIN, R., HALPERN, J. Y., MOSES, Y., et al., 1996, "Common Knowledge: now you have it, now you don't", In:*Proceedings of the 1996 Int'l Multidisciplinary Conference*, volume 1, pp. 177–183.

FAGIN, R., HALPERN, J. Y., MOSES, Y., et al., 1997, "Knowledge-Based Programs", *Distributed Computing*, v. 10, n. 4, pp. 199–225.

FAGIN, R., HALPERN, J. Y., MOSES, Y., et al., 1999, "Common Knowledge Revisited", *Annals of Pure and Applied Logic*, v. 96, n. 1, pp. 89–105.

FAGIN, R., HALPERN, J. Y., VARDI, M. Y., 1992, "What can machines know? on the properties of knowledge in distributed systems", *Journal of the ACM*, v. 39, n. 2, pp. 328–376.

FAGIN, R., VARDI, M. Y., 1986, "Knowledge and Implicit Knowledge in a Distributed Environment", In:*Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge (TARK'86)*, pp. 187–206.

HALPERN, J. Y., 2000, "A Note on Knowledge-Based Programs and Specifications", *Distributed Computing*, v. 13, n. 3, pp. 145–153.

HALPERN, J. Y., FAGIN, R., 1989, "Modelling Knowledge and Action in Distributed Systems", *Distributed Computing*, v. 3, n. 4, pp. 159–179.

HALPERN, J. Y., MOSES, Y., 1990, "Knowledge and Common Knowledge in a Distributed Environment", *Distributed Computing*, v. 37, n. 3, pp. 549–587.

HALPERN, J. Y., VARDI, M. Y., 1991, "Model Checking vs. Theorem Proving: a Manifesto", In:*Artificial Intelligence and Mathematical Theory of Computation*, pp. 151–176, San Diego, California.

HALPERN, J. Y., ZUCK, L. D., 1992, "A little knowledge goes a long way: simple knowledge-based derivations and correctness proofs for a family of protocols", *Journal of the ACM*, v. 39, n. 3, pp. 449–478.

HINTIKKA, J., 1962, "Knowledge and Belief", *Cornell University Press.*

KRIPKE, S. A., 1963, "Semantical Considerations on Modal Logic", In:*Proceedings of a Colloquium: Modal and Many Valued Logics*, volume 16, pp. 83–94.

LEHMANN, D. J., 1984, "Knowledge, Common Knowledge, and Related Puzzles", In:*Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 62–67.

LOMUSCIO, A. R., van der MEYDEN, R., RYAN, M. D., 2000, "Knowledge in Multi-Agent Systems: Initial Configurations and Broadcast", *Computational Logic*, v. 1, n. 2, pp. 247–284.

PANANGADEN, P., TAYLOR, K., 1992, "Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems", *Distributed Computing*, v. 6, n. 2, pp. 73–93.

STULP, F., VERBRUGGE, R., 2000, "A Knowledge-Based Algorithm for the Internet Protocol TCP", In:*Proceedings of the Fourth Conference on Logic and the Foundations of Game and Decision Theory (LOFT 4)*, Torino, Italy, June.

van der MEYDEN, R., 1996, "Finite State Implementations of Knowledge-Based Programs", In:*Proceedings of the 1996 Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 262–273, Hyderabad, India, December.

van der MEYDEN, R., 1998, "Common Knowledge and Update in Finite Environments", *Information and Computation*, v. 140, n. 2, pp. 115–157.

van der MEYDEN, R., VARDI, M. Y., 1998, "Synthesis from Knowledge-Based Specifications", In:*Proceedings of the Nineth International Conference on Concurrency Theory (CONCUR'98)*, pp. 34–49.

VARDI, M. Y., 1996, "Implementing Knowledge-Based Programs", In:*Proceedings of the 1996 Conference on Theoretical Aspects of Reasoning about Knowledge (TARK'96)*.