UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

SISTEMA MULTIAGENTES UTILIZANDO A LINGUAGEM AGENTSPEAK(L) PARA CRIAR ESTRATÉGIAS DE ARMADILHA E COOPERAÇÃO EM UM JOGO TIPO PACMAN

ALISSON RAFAEL APPIO

BLUMENAU
2004

2004/II-04

ALISSON RAFAEL APPIO

SISTEMA MULTIAGENTES UTILIZANDO A LINGUAGEM AGENTSPEAK(L) PARA CRIAR ESTRATÉGIAS DE ARMADILHA E COOPERAÇÃO EM UM JOGO TIPO PACMAN

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação – Bacharelado.

Prof. Jomi Fred Hübner – Orientador

BLUMENAU
2004

2004/II-04

SISTEMA MULTIAGENTES UTILIZANDO A LINGUAGEM AGENTSPEAK(L) PARA CRIAR ESTRATÉGIAS DE ARMADILHA E COOPERAÇÃO EM UM JOGO TIPO PACMAN

Por

ALISSON RAFAEL APPIO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Jomi Fred Hübner – Orientador, FURB
Membro:	Prof. Paulo César Rodacki Gomes, FURB
Membro:	Prof. Dalton Solano dos Reis, FURB

BLUMENAU
2004

AGRADECIMENTOS

Muitas pessoas contribuíram para a realização deste trabalho. Agradeço aos meus pais e professores que apoiaram-me no decorrer desses anos de estudo.

Aos meus colegas e amigos, pelos enpurrões e cobranças.

À Deus, pelo seu imenso amor e graça.

À minha família, que mesmo longe, sempre esteve presente em meu coração.

Ao meu orientador e amigo Jomi, de quem pude aprender muito, por ter acreditado e dedicado seu tempo em discussões sobre este trabalho.

Ao Rafael H. Bordini e Jomi pela ajuda e discussões sobre a ferramenta ***Jason***.

Especialmente a Mariele, pelo apoio nas horas difícies, compreensão e dedicação com minha pessoa.

RESUMO

Este trabalho apresenta um Sistema Multiagentes (SMA) atuando sobre um jogo tipo PacMan. Os personagens fantasmas são concebidos como agentes. A arquitetura e a linguagem utilizadas são *Belief-Desires-Intentions* (BDI) e *AgentSpeak(L)*, respectivamente. Os agentes têm como objetivo criar estratégias de armadilha e cooperar na execução das armadilhas criadas, dificultando a vitória do personagem come-come, que é controlado por um usuário. Também é apresentado o código *AgentSpeak(L)* dos agentes e a ferramenta ***Jason***, que interpreta esta linguagem.

Palavras Chave: Cooperação em Sistemas Multiagentes; Jogos; Estratégias de Armadilha em Jogos; Inteligência Artificial (IA); Arquitetura BDI; *AgentSpeak(L)*.

ABSTRACT

This work describes a Multi-Agent Systems that controls the ghosts of a PacMan game. The ghosts characters are conceived as agents. The architecture and the language used are *Belief-Desires-Intentions* and *AgentSpeak(L)*, respectively. The agents have as objective to create trap strategies and to cooperate in the execution of the created traps, hindering the victory pacman character's, that is controlled by an user. The AgentSpeak(L) specification of the agents is describe and the ***Jason*** interpreter of this language is used to implement the game.

Key-Words: Cooperate in Multi-Agent Systems; Games; Strategics of Snare in Games; Artificial Inteligence; Arquiteture BDI; *AgentSpeak(L)*.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Os jogos de computadores cada vez mais possuem um mercado atrativo, recebendo a atenção dos cientistas no desenvolvimento de técnicas computacionais sofisticadas com o uso de várias mídias, animações com gráficos 2D e 3D, vídeos, som, etc. A construção de jogos é uma das tarefas mais difíceis dentro da computação.

Um jogo é composto por três partes básicas: enredo, motor (*engine*) e interface. Em síntese, o enredo define qual o tema, a trama (histórias individuais, interação entre personagens, influências causadas pelas interações e evolução dos personagens) e os objetivos do jogo. O motor é o mecanismo que controla a reação do jogo / personagem em função de uma ação do usuário (parte técnica do jogo). A interface controla a comunicação entre o motor e o usuário, reportando graficamente um novo estado do jogo. Na parte da interface estão envolvidos aspectos artísticos, cognitivos, técnicos e pedagógicos (BATTAIOLA, 2000).

Um dos jogos mais populares é o PacMan. Neste jogo existem dois tipos de personagens, sendo eles: os fantasmas e o come-come. O personagem come-come é controlado por um usuário e tem como objetivo comer todas as bolinhas que estão situadas no cenário. Quando terminar de comer as bolinhas, o usuário vence. A fig. 1.1 a) mostra o formato típico do personagem come-come. Os personagens fantasmas são controlados pelo computador e têm como objetivo evitar que o come-come vença o jogo. Para isso, os fantasmas tentam matar o personagem come-come. O formato típico do personagem fantasma é mostrado na fig. 1.1 b). A morte do personagem come-come acontece quando algum fantasma consegue enconstar-se a ele (i.e., o fantasma está na mesma posição que o come-come). Um exemplo de cenário do jogo pode ser visto na fig. 1.2.

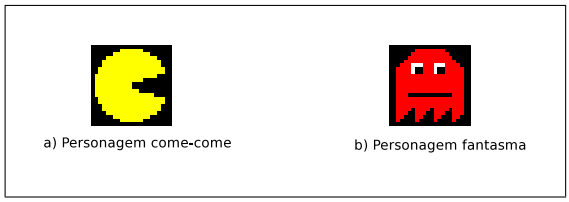


Figura 1.1 – Personagens.

Uma das formas dos fantasmas impedirem que o usuário vença o jogo é criando estratégias de armadilha, prendendo o come-come em algum lugar do mundo para con-

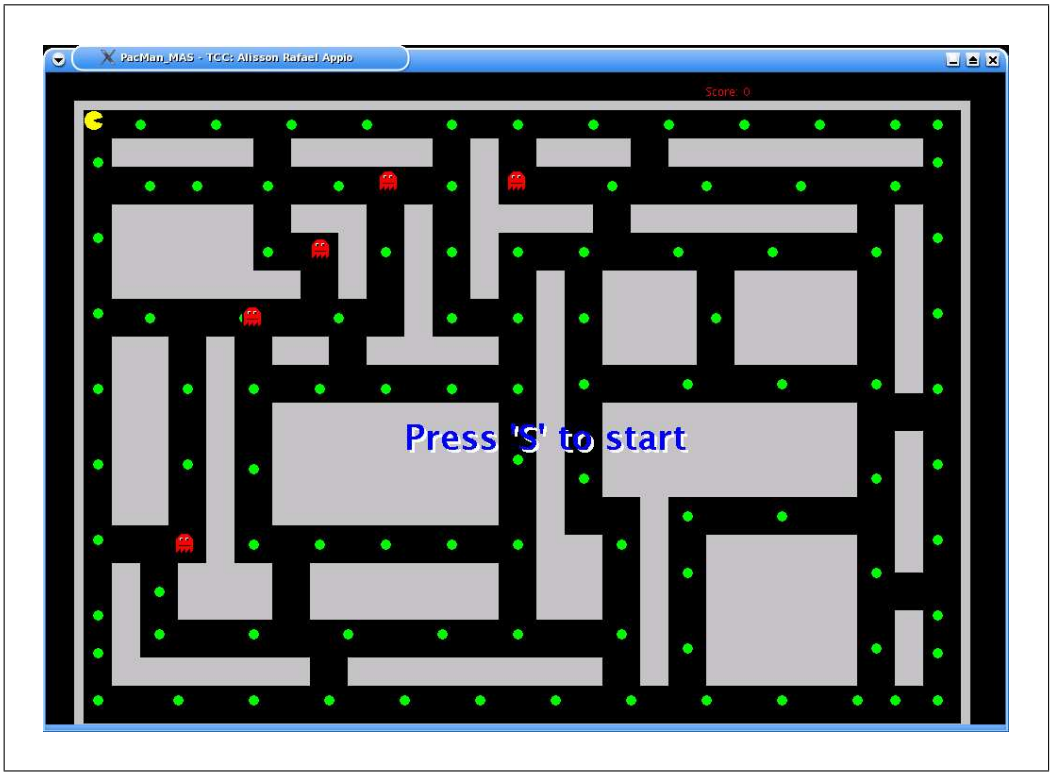


Figura 1.2 – Demonstração de um jogo tipo PacMan.

seguir matá-lo e conseqüentemente não deixando que o jogador vença. Neste trabalho, pretende-se construir um jogo tipo PacMan aplicando técnicas de cooperação em Sistema Multiagentes (SMA) nos personagens fantasmas para eles criarem e executarem estratégias de armadilha.

1.1 OBJETIVO DO TRABALHO

O objetivo deste trabalho é verificar a viabilidade do uso de técnicas de cooperação em SMA para resolver o problema de criar estratégias de armadilhas em um jogo do tipo PacMan.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 faz a fundamentação teórica, apresentando os tópicos mais importantes para o desenvolvimento deste trabalho. A seção 2.1 trata dos conceitos de SMA e aspectos importantes que um agente cognitivo (inserido em um SMA) deve possuir. Na seção 2.1.1 é abordada a necessidade de coordenação em SMA, visto que, os agentes precisam coordenar suas ações para poder vencer o jogo. Na seção 2.1.2 são abordados os temas cooperação e competição como formas de coordenação das interações entre os agentes, ainda nesta seção

é feita uma breve descrição de negociação. Na próxima seção (seção 2.2) são mostrados os conceitos da arquitetura BDI, para o desenvolvimento de agentes cognitivos. A seção 2.3 e a seção 2.3.1 mostram a linguagem *AgentSpeak(L)*, linguagem utilizada na construção dos agentes. A ferramenta ***Jason*** (seção 2.4) é utilizada para criar o SMA, sendo que esta ferramenta interpreta códigos escritos na linguagem *AgentSpeak(L)*.

No capítulo 3 é apresentada a especificação e a implementação do jogo desenvolvido. Na seção 3.1 são descritos os principais requisitos (requisitos funcionais e requisitos não funcionais) deste jogo. Na seção 3.5.3 é apresenta a estratégia de armadilha que os agentes podem executar. Na seção 3.6 é detalhado como foi implementado o jogo. A seção 3.8 apresenta as dificuldades encontradas na realização deste trabalho.

No capítulo 4 são apresentadas as conclusões deste trabalho, bem como possíveis extensões que podem ser desenvolvidas (seção 4.1).

2 FUNDAMENTAÇÃO TEÓRICA

Para a realização deste trabalho, foi feita uma revisão bibliográfica dos seguintes temas: SMA, arquitetura BDI e linguagem *AgentSpeak(L)*.

2.1 SISTEMA MULTIAGENTES (SMA)

A área de SMA estuda o comportamento de um grupo de agentes (aplicando as técnicas de IA clássica) que cooperam para resolver um problema que normalmente um único agente não seria capaz de resolver. Ocupa-se da construção de sistemas computacionais a partir da criação de entidades de software autônomas que interagem através de um ambiente compartilhado por outros agentes de uma sociedade e atuam sobre esses ambientes, alterando seu estado. Definições mais detalhadas de SMA, seus problemas e aplicações podem ser encontradas nas seguintes referências: (ALVARES; SICHMAN, 1997; BORDINI; VIEIRA; MOREIRA, 2001; DEMAZEAU; MÜLLER, 1990; JENNINGS; WOOLDRIDGE, 1998; WEISS, 2000; WOOLDRIDGE, 2002; BORDINI; VIEIRA, 2003).

Vários autores apresentam definições de agentes, e entre as definições mais aceitas (do ponto de vista do autor) destacam-se Alvares e Sichman (1997), Barone et al. (2004), Bordini e Vieira (2003). Segundo estes autores, um agente é uma entidade real ou virtual, que está inserida em um ambiente, podendo perceber, agir, deliberar e comunicar-se com outros agentes e possui comportamentos autônomos. Os agentes podem ser divididos em duas categorias: reativos e cognitivos. Agentes reativos possuem comportamentos simples, não possuindo nenhum modelo do mundo onde estão atuando e possuem comportamento estímulo-resposta. Agentes cognitivos possuem comportamentos complexos onde eles deliberam e negociam suas ações com os outros agentes. Na construção de agentes cognitivos em SMA é importante mencionar alguns aspectos, como:

a) percepção: um agente deve ser capaz de perceber o ambiente onde ele está agindo;
b) ação: um agente deve ser capaz de executar alguma ação no ambiente (alterar o ambiente);
c) comunicação: um agente deve ser capaz de comunicar-se com outros agentes;
d) representação: um agente deve ser capaz de representar aquilo que ele acredita ser verdade no ambiente;

e) motivação: um agente deve ser capaz de almejar desejos e objetivos (i.e., aspectos motivacionais). Em termos práticos, isto significa ter uma representação de estados do ambiente que o agente almeja alcançar;
f) deliberação: um agente deve ser capaz de decidir qual estado do ambiente tem maior utilidade no futuro;
g) raciocínio e aprendizagem: técnicas de inteligência artificial clássica para, por exemplo, raciocínio e aprendizagem poderem ser estendidas para múltiplos agentes, aumentando significativamente o desempenho desses, por exemplo no aspecto de deliberação. Os mecanismos de aprendizagem em SMA são uma área de pesquisa e investigação.

Em SMA, existe a necessidade de coordenar as interações que ocorrem entre os agentes. Vários autores apresentam o tema coordenação, entre os quais cita-se Weiß (2000), Wooldridge (2002), Oliveira (2001), Russel e Norvig (2003). Os tipos de coordenação são descritos nas seções seguintes.

2.1.1 COORDENAÇÃO

Pode-se dizer que coordenação em SMA é um processo no qual um agente raciocina sobre suas ações locais e ações de outros agentes com o objetivo de garantir que a comunidade funcione de maneira coerente (JENNINGS, 1996). É um ato de trabalhar em conjunto no sentido de atingir um acordo com objetivo(s) comum(ns) de forma harmoniosa.

A coordenação é um fator vital em SMA. Jennings e Wooldridge (1998) dizem que a necessidade de coordenação surge do fato da existência de dependências entre as ações dos agentes e da impossibilidade de resolução de um problema por um único agente, seja pela insuficiência de recursos, informações ou capacidade dos agentes.

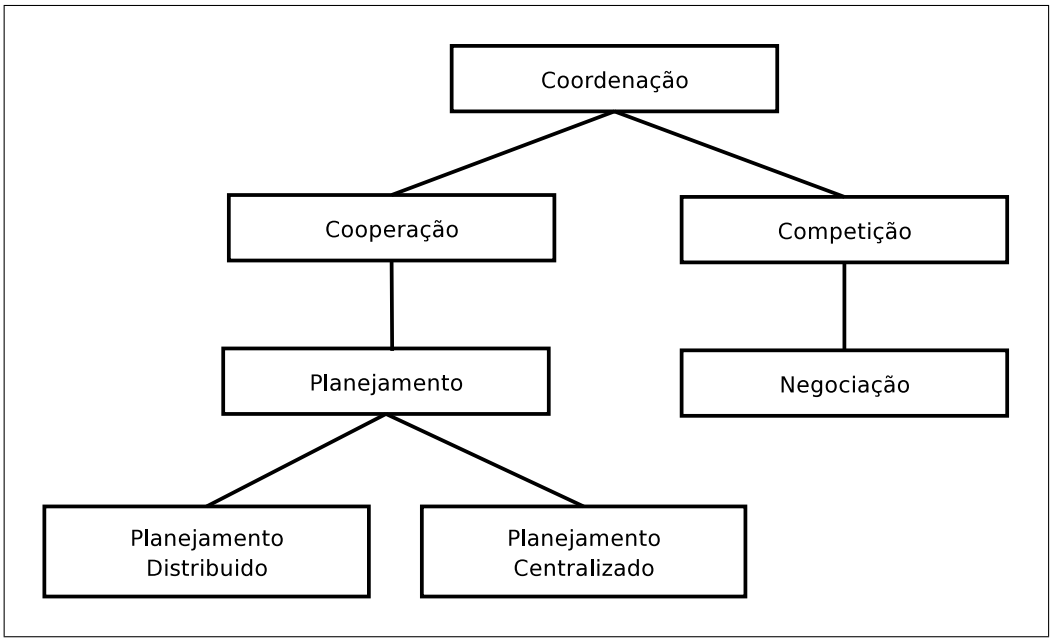
Se a coordenação falhar a comunidade pode ter comportamentos incoerentes. Por esta razão surgiram vários mecanismos de coordenação, sendo divididos em três categorias: organização estrutural, meta-nível de troca de informações e planejamento multiagente. A escolha de qual paradigma usar é uma questão de projeto (JENNINGS, 1994).

Ainda segundo Jennings (1994), no contexto de SMA, uma organização estrutural pode ser visualizada como um padrão de informações e controle dos relacionamentos entre os indivíduos da comunidade. Meta-nível de troca de mensagens envolve o envio de mensagens para controlar a prioridade das ações do agente. Planejamento multiagente acontece quando cada agente efetua um planejamento de todas as suas ações e interações

com um respectivo objetivo, evitando inconsistências ou ações conflitantes.

Um exemplo de coordenação entre agentes acontece quando dois ou mais robôs precisam carregar uma mesa. Eles devem decidir quem vai pegar uma determinada parte da mesa através de uma negociação entre eles e também coordenar suas ações para conseguirem carregar a mesa, pois se eles não coordenarem suas ações um robô poderá ficar parado enquanto o outro tentará carregá-la, conseqüentemente derrubando a mesa. A cooperação acontece quando os robôs trabalham de forma a resolver o problema, ou seja, mover a mesa para outro lugar.

Normalmente a coordenação é alcançada por meio da comunicação entre os agentes. A fig. 2.1 mostra os tipos de coordenação que podem surgir.



Fonte: adaptado de Weiß (2000).

Figura 2.1 – Tipos de coordenação entre agentes.

Na maioria dos aplicações de SMA é necessário utilizar o conceito de coordenação. Alguns exemplos de aplicações que necessitam deste conceito são apresentado a seguir:

a) controle de tráfego áereo;
b) time de futebol;
c) trânsito de automóveis em uma cidade;
d) operação militar.

Os tipos de coordenação são apresentadas nas seções seguintes.

2.1.2 COOPERAÇÃO X COMPETIÇÃO

Cooperação é um tipo de coordenação entre os agentes que exercem ações com objetivo de atingir um bem social[1], *i.e.*, estão preocupados em atingir objetivos partilhados com outros agentes. Para que os agentes consigam uma cooperação satisfatória cada agente deve manter um modelo dos outros agentes e também desenvolver um modelo das futuras interações (WEISS, 2000).

Quando os agentes estão trabalhando cooperativamente, pode-se dizer que estão trabalhando como equipes e comportam-se de forma a incrementar a utilidade global do sistema e não sua utilidade individual. Exemplos:

a) cooperação entre agentes acontece em um ambiente de direção de um táxi, onde o agente deve evitar colisões para maximizar a medida de desempenho de todos os agentes (RUSSEL; NORVIG, 2003);
b) o planejamento de equipes em tênis de duplas. Dois agentes que jogam em uma equipe de tênis de duplas têm como objetivo comum vencer a partida, dando origem a vários sub-objetivos. Um dos sub-objetivos gerado é que eles tenham que devolver a bola que foi lançada para eles e assegurar que pelo menos um deles estará cobrindo a rede, pois essa é uma boa estratégia de jogo (RUSSEL; NORVIG, 2003).

Muitas vezes são adotadas "convenções" ou "leis sociais" para os planejamentos em SMA. Por exemplo, em jogos de tênis de dupla, a bola pode estar aproximadamente eqüidistante dos dois parceiros. Para resolver esse impasse, um dos agentes poderia gritar "é minha!" ou "é sua!", através da comunicação estabelecida entre eles.

A resolução de uma determinada tarefa pode necessitar de mais de um agente para finalizá-la, desta forma os agentes devem comunicar-se para atingir sua meta (resolução do problema). Um agente "A" pode ter como desejo executar uma tarefa "X". A tarefa "X" necessita da interação ou ajuda de dois agentes para ser realizada, imagine que exista um agente "B" que é capaz de realizar parte da tarefa "X". O agente "A" deve então enviar uma mensagem para o agente "B" cooperar na execução da tarefa "X". O agente "B" concorda em resolver a tarefa que lhe é imposta (quando os agentes são autônomos, eles podem se negar a cooperar em determinada tarefa, isso pode acontecer quando ele percebe que a tarefa pode entrar em conflito com seus futuros desejos).

[1]Bem social, no sentido onde os agentes não entrem em discussão sobre seus objetivos, quando estes objetivos são de interesse da sociedade como um todo

Normalmente um agente precisa de ajuda ou trocar informações para realizar certas tarefas. Um agente pode assumir dois papéis na cooperação, sendo eles: organizador da cooperação, sendo aquele que solicita ajuda em determinada tarefa, pode-se dizer que o agente "A" assumiu o papel de organizador da cooperação. Quem atende a solicitação de cooperação assume o papel de respondente. O agente "B" assumiu o papel de respondente em relação ao agente "A".

Competição é um tipo de coordenação entre os agentes, acontecendo quando os recursos são os mesmos para diferentes agentes, *i.e.*, cada agente busca um determinado estado do mundo que melhor lhe agrade, visando apenas seus próprios objetivos e não o bem social. Um exemplo de competição entre agentes acontece em um jogo de xadrez (com dois agentes), onde cada agente tenta maximizar a sua medida de desempenho, conseqüentemente diminuindo o desempenho do outro agente; neste contexto o jogo de xadrez é um ambiente multiagente competitivo (RUSSEL; NORVIG, 2003).

Ainda segundo Russel e Norvig (2003), um agente em um ambiente competitivo deve:

a) reconhecer que existem outros agentes;
b) calcular alguns dos planos possíveis do(s) outro(s) agente(s);
c) calcular como os planos do(s) outro(s) agente(s) interagem com seus próprios planos;
d) decidir sobre a melhor ação em face dessas interações.

Assim, a competição, como a cooperação, exige um modelo dos planos dos outros agentes. Por outro lado, não existe nenhum compromisso com um plano conjunto em um ambiente competitivo.

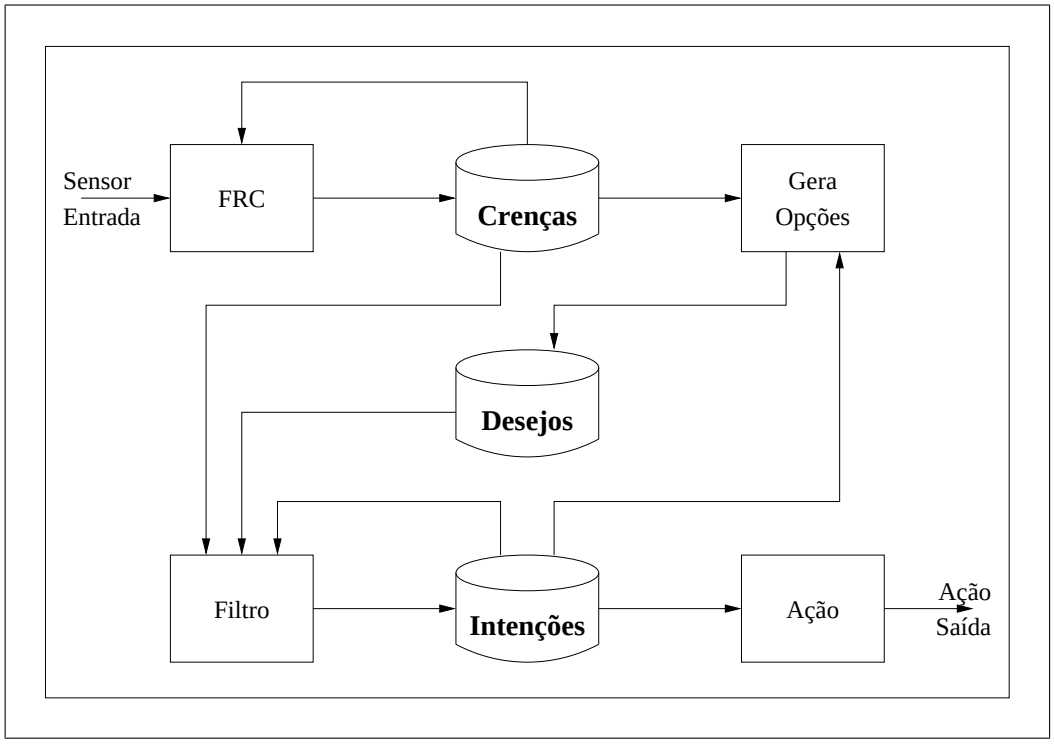
Uma das formas dos agentes coordenarem suas ações é formarem acordos entre eles através de uma negociação. Uma negociação entre agentes é a troca de mensagens usando um protocolo e uma linguagem (normalmente é utilizado a linguagem *Knowledge Query Markup Language* (KQML) (LABROU; FININ, 1997)) visando um acordo sobre uma determinada cooperação. A negociação entre os agentes consiste em determinar qual tarefa cada agente deve executar em um dado momento. Este processo de decisão sobre qual ação deve ser executada implica em uma comunicação entre os agentes, com vista à coordenação de suas atividades.

Existem diversas arquiteturas para os agentes. As arquiteturas mais conhecidas na literatura são: Reativa, *Subsumption*, BDI, Deliberativa e em Camadas. Uma arquitetura

para o modelo cognitivo é a arquitetura BDI (explicada na seção seguinte). As demais arquiteturas não são abordadas neste trabalho.

2.2 ARQUITETURA BDI

BDI é uma arquitetura caracterizada por três atitudes mentais que são as crenças, os desejos e as intenções. Os principais criadores da arquitetura BDI foram Georgeff e Rao (BORDINI; VIEIRA, 2003). A fundamentação filosófica para esta concepção de agentes vem do trabalho de Dennett (1987) sobre sistemas intencionais e de Bratman (1987) sobre raciocínio prático. Uma arquitetura BDI genérica é mostrada na fig. 2.2.



Fonte: adaptado de Wooldridge (1999).

Figura 2.2 – Arquitetura BDI genérica.

As crenças representam tudo aquilo que o agente sabe sobre o ambiente e sobre os agentes daquele ambiente (inclusive sobre si mesmo). Os desejos representam os estados do mundo que o agente quer atingir. As intenções representam a seqüência de ações que um agente compromete-se a executar para atingir sua meta.

A função FRC (Função de Revisão de Crenças) recebe as informações do ambiente, podendo ler e atualizar a base de crenças do agente. Com as alterações do estado do ambiente, podem ser gerados novos objetivos. A função Gera Opções verifica quais estados devem ser atingidos de acordo com o estado atual e as intenções com que o agente está

comprometido. A função filtro serve para atualizar o conjunto de intenções do agente com base nas crenças e desejos que ele possui. A função de Ação representa a escolha de uma determina ação para ser executada.

Uma das linguagens que considera os conceitos da arquitetura BDI é a linguagem *AgentSpeak(L)* descrita na próxima seção.

2.3 LINGUAGEM AGENTSPEAK(L)

Agentes cognitivos podem ser especificados através da linguagem *AgentSpeak(L)*. Um programa *AgentSpeak(L)* é especificado por um conjunto de crenças, planos, eventos ativadores e um conjunto de ações básicas que o agente executa no ambiente. Os programas feitos em *AgentSpeak(L)* são interpretados de maneira similar à programas escritos em lógica (como por exemplo, programas escritos em Prolog).

Uma crença é um predicado de primeira ordem na notação lógica usual (ou fatos, no sentido de programação lógica) e literais de crenças são átomos de crenças ou suas negações que formarão a base de crenças do agente.

Planos fazem referência a ações básicas que um agente é capaz de executar em seu ambiente, sendo composto por um evento ativador, contexto e corpo. Os planos são sensíveis ao contexto, i.e., necessitam que certas condições sejam satisfeitas para serem executados, sendo que o contexto deve ser uma conseqüência lógica da base de crenças do agente no momento em que o evento é selecionado pelo agente para o plano ser considerável aplicável. O corpo do plano é uma seqüência de ações básicas ou subobjetivos que o agente deve atingir ou testar quando uma instância do plano é selecionada para execução.

A linguagem *AgentSpeak(L)* distingue dois tipos de objetivos: objetivos de realização e objetivos de teste. Objetivos de realização e teste são predicados, tais como crenças, porém com operadores prefixados "!" e "?" respectivamente. Objetivo de realização expressão os desejos do agente e objetivos de teste retornam a unificação do predicado de teste com uma crença do agente ou pode falhar quando não existir nenhuma crença que seja satisfeita (BORDINI; VIEIRA, 2003).

Quando o agente percebe informações sobre o ambiente, é gerado um evento com esta percepção, sendo adicionado a sua base de crenças. É gerada uma lista com os planos que podem ser executados com essa percepção (i.e. que tenham o predicado do evento gerado pelo ambiente) e testado o contexto do plano para verificar quais planos podem ser aplicados. Por fim é selecionado um plano da lista de planos e o agente executa o

plano selecionado.

2.3.1 SINTAXE ABSTRATA DA LINGUAGEM

Algumas definições da Linguagem *AgentSpeak(L)* segundo Rao (1996):

a) se `b` é um símbolo de predicado e t_1, ..., t_n são termos, então `b(`t_1`, ..., `t_n`)` ou `b(t)` é um átomo de crença, not `b(t)` ou `b(t)` & `c(s)` são crenças;

b) se `g` é um símbolo de predicado e t_1, ..., t_n são termos, então `!g(`t_1`, ...,` t_n`)` ou `!g(t)` e `?g(`t_1`, ...,` t_n`)` ou `?g(t)` são metas que o agente pretende atingir;

c) se `b(t)` é um átomo de crença e `!g(t)` e `?g(t)` são metas, então `+b(t)`, `-b(t)`, `+!g(t)`, `+?g(t)`, `-!g(t)` e `-?g(t)` são eventos de ativação (*triggering events*);

d) se `a` é um símbolo de ação e t_1, ..., t_n são termos de primeira ordem, então `a(`t_1`, ...,` t_n`)` ou `a(t)` são ações que o agente executa no ambiente;

e) se `e` é um evento de ativação (*triggering event*), b_1, ..., b_n, são crenças literais e h_1, ..., h_n são metas ou ações no ambiente, então `e :` b_1 `&` ... `&` b_m `<-` h_1`;` ...`;` h_n é um plano.

Um plano, possui um evento ativador (*triggering event*), contexto do plano (i.e. a parte que é testada para verificar se o plano é valido) e uma seqüência de ações básicas, objetivos ou atualizações de crenças. As ações básicas do agente podem ser classificadas de duas maneiras. sendo elas:

a) ações internas são prefixadas por um ponto (".") e não alteram o ambiente. Exemplo: `.plus(1,2,Soma)`; esta ação efetua a soma dois primeiros argumentos e retorna o resultado na variável `Soma`;

b) ações externas são ações que o agente efetua no ambiente. Exemplo: `moverRobo(1, 1)` esta ação move o robô para uma coordenada (x, y) passada como argumento.

Um evento ativador pode ser interno ou externo. Eventos internos correspondem a adição ou remoção de crenças ou objetivos (`+b, -b, +!g, -!g, +?g,-?g`) do agente. Eventos externos são originados pela percepção do ambiente ou das ações dos outros agentes.

O quadro 2.1 mostra um exemplo de planos de um agente que pretende ir a um concerto, implementados na linguagem *AgentSpeak(L)*. O primeiro plano é ativado quando o anúncio de um concerto com um artista A num local `V` (correspondendo à adição de

```
+concert(A,V) : likes(A)
        ← !book_tickets(A,V).
+!book_tickets(A,V) : ¬busy(phone)
        ← call(V);
        ...;
       !choose_seats(A,V).
```

Fonte: Bordini, Hübner et al. (2004).

Quadro 2.1 – Exemplos de planos *AgentSpeak(L)*.

uma crença `concert(A, V)` como conseqüência da percepção do ambiente). Se o agente gostar do artista A, ele terá como objetivo a reserva dos ingressos para esse concerto. O segundo plano é executado quando o agente gostar do artista A. Se a linha telefônica não estiver ocupada ele executa a ação básica de fazer contato telefônico e termina com um subplano de escolha de assentos.

O estado atual do agente é representado pelas crenças sobre os outros agentes, percepções do ambientes e as crenças sobre si mesmo naquele momento.

Os estados que o agente pretende alcançar baseados em seus estímulos internos ou externos (ambiente, outros agentes) podem ser vistos como desejos que ele almeja atingir. As intenções são as ações que o agente se compromete a executar para realizar um desejo.

2.4 FERRAMENTA *JASON*

A ferramenta ***Jason***[2] (do inglês: *A Java-based AgentSpeak Interpreter Used with Saci For Multi-Agent Distribution Over the Net*) proporciona a interpretação e execução de programas escritos na linguagem *AgentSpeak(L)*. SMA são facilmente configurados nesta ferramenta, podendo ser executado em vários computadores através do SACI[3] (HÜBNER; SICHMAN, 2000). ***Jason*** é implementado na linguagem Java (executada em múltiplas plataformas), sendo *Open Source* e distribuído sob a licença GNU LGPL.

Um SMA desenvolvido na ferramenta ***Jason***, possui um ambiente onde os agentes estão situados e um conjunto de instâncias de agentes *AgentSpeak(L)*. O ambiente dos agentes, deve ser desenvolvido na linguagem Java. ***Jason*** possui os seguintes recursos (BORDINI; HÜBNER et al., 2004):

[2]Disponível em `http://jason.sourceforge.net`
[3]Disponível em `http://www.lti.pcs.usp.br/saci/`

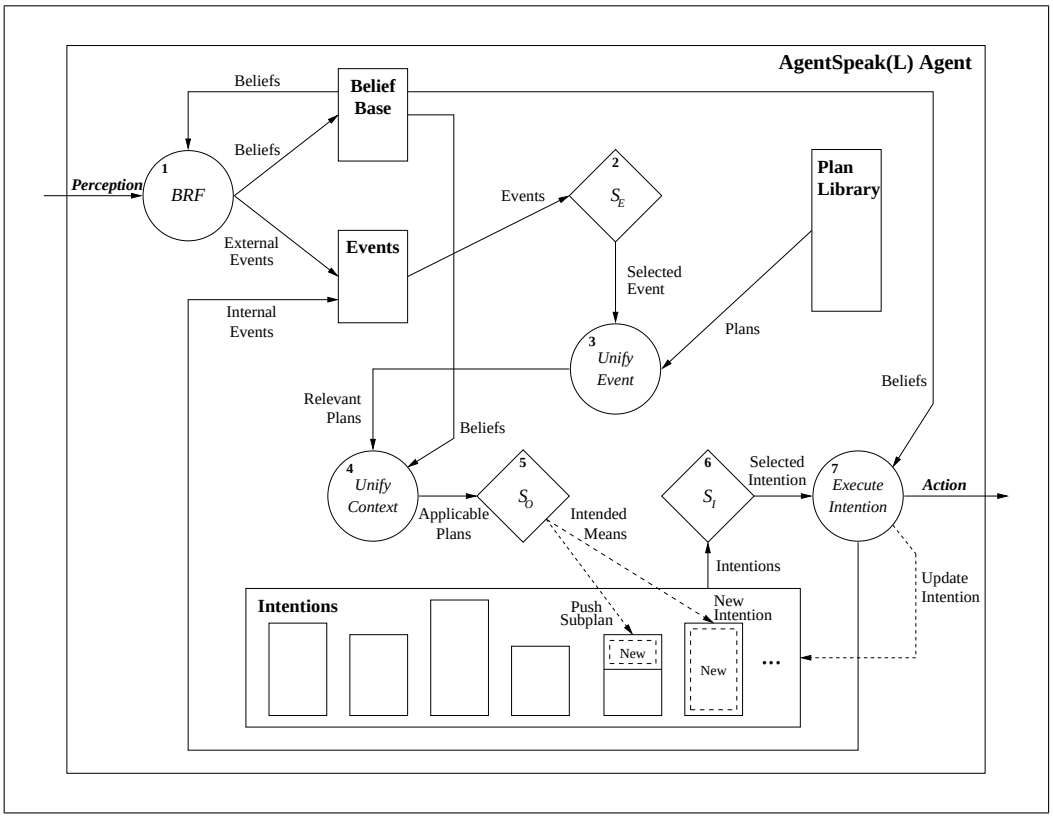
a) negação forte (*strong negation*), portanto é possível construir sistemas que consideram mundo-fechado (*closed-world*) e mundo-aberto (*open-world*);
b) tratamento de falhas em planos;
c) comunicação baseada em atos de fala (incluindo informações de fontes como anotações de crenças);
d) anotações em identificadores de planos, que podem ser utilizados na elaboração de funções personalizadas para seleção de planos;
e) suporte para desenvolvimento de ambientes (que normalmente não é programado em *AgentSpeak(L)*);
f) possibilidade de executar o SMA distribuidamente em uma rede (usando o SACI);
g) possibilidade de especificar (em Java) as funções de seleção de planos, as funções de confiança e toda a arquitetura do agente (percepção, revisão de crenças, comunicação e atuação);
h) possui uma biblioteca básica de "ações internas";
i) possibilitar a extensão da biblioteca de ações internas.

A configuração do SMA é feita em arquivos com extensão .mas2j, basicamente é informado qual a arquitetura do SMA, "Centralized" ou "Saci" (centralizado ou distribuída), qual o ambiente onde os agentes estão situados e os agentes. A programação dos agentes *AgentSpeak(L)* é feita em arquivos com extensão .asl. Para ver a BNF da linguagem *AgentSpeak(L)* e como configurar um SMA na ferramenta ***Jason*** consulte Bordini, Hübner et al. (2004).

A forma que os programas *AgentSpeak(L)* são interpretados é detalhada nos parágrafos subseqüentes com auxilio da fig. 2.3 reproduzida de Machado e Bordini (2002).

O conjunto de crenças, eventos, planos e intenções são representados por retângulos. Losangos representam a seleção de um elemento de um conjunto. Círculos representam alguns dos processos envolvidos na interpretação de programas *AgentSpeak(L)*.

A função de revisão de crenças (BRF, com o rótulo 1) recebe as percepções do ambiente, adicionando um evento na lista de eventos e uma crença a cada percepção recebida. O conjunto de eventos é atualizado através das percepções do ambiente ou através da execução de novas intenções (i.e., quando subobjetivos são especificados no corpo do plano). A cada adição ou remoção de crença é gerado um evento. Eventos também podem ser gerados quando o agente se compromete a realizar um objetivo.



Fonte: Machado e Bordini (2002).

Figura 2.3 – Ciclo do interpretador *AgentSpeak(L)*.

Após ter selecionado um evento S_E (rótulo 2), o interpretador deve unificar o evento selecionado com eventos ativadores das cabeças dos planos, para isso é necessário verificar a biblioteca de planos do agente (rótulo 3). Depois é gerado um conjunto de todos os planos relevantes para o evento escolhido, através da unificação do contexto dos planos. É verificado o contexto de cada plano para gerar um conjunto de planos aplicáveis (planos que podem ser aplicados no estado atual do agente) e descartasse os planos que não podem ser aplicados, *i.e.*, que o contexto falha (rótulo 4). Depois S_O (rótulo 5) escolhe um único plano do conjunto de planos aplicáveis e coloca o plano no topo de uma intenção existente (se o evento for interno) ou cria uma nova intenção no conjunto de intenções (se o evento for externo, i.e. gerado pela percepção do ambiente) (BORDINI; HÜBNER et al., 2004).

Ainda segundo Bordini, Hübner et al. (2004), a função S_I (rótulo 6) seleciona uma intenção do conjunto de intenções do agente (as intenções do agente são instâncias de planos). E então é executado o plano (rótulo 7) selecionado pela S_I. Isso implica em uma ação básica a ser executada no ambiente, na geração de um evento (em caso da fórmula ser um objetivo de realização) ou na execução de um objetivo de teste (verificando a base de crenças).

Se a intenção for uma ação básica ou um objetivo de teste, o conjunto de intenções precisa ser atualizado. No caso de objetivo de teste, é percorrida a base de crenças, procurando por um átomo de crença que unifique com o predicado de objetivo de teste. Se for possível a unificação do predicado, podem ocorrer instanciações das variáveis nos planos parcialmente instanciados que contenham objetivo de teste e deve ser removido o objetivo do conjunto de intenções, pois ele já foi resolvido. Nos casos que as ações básicas serem selecionadas, o interpretador avisa o ambiente para executar a ação e remove a ação do conjunto de intenções.

Quando todas as fórmulas do corpo do plano forem removidas (isto é, forem executadas) o plano é removido da intenção, assim como o objetivo de realização que o gerou, se esse for o caso. O ciclo de interpretação termina e *AgentSpeak(L)* começa um novo ciclo, verificando as percepções do ambiente, gerando os eventos necessários e continuando o ciclo de raciocínio do agente como descrito acima (BORDINI; VIEIRA, 2003).

Nos parágrafos a seguir é mostrado um exemplo do uso da ferramenta ***Jason***. O código implementado se refere a um robô aspirador de pó, ou seja, quando o robô perceber sujeira no ambiente (o ambiente é representado por coordenadas X e Y, como se fosse um *grid*), ele tem como objetivo parar para poder executar a ação básica no ambiente de aspirar a sujeira. O objetivo da implementação é a demonstração da sintaxe da linguagem *AgentSpeak(L)*, como criar o ambiente e configurar agentes no ***Jason***.

O arquivo de configuração do projeto (arquivo com extensão .mas2j) de SMA utilizado na ferramenta ***Jason*** é mostrada no quadro 2.2. Deve ser informado qual o tipo da arquitetura (no caso, arquitetura centralizada, *i.e.*, execução em uma única máquina), qual é a classe java que simula o ambiente (a classe Mundo) dos agentes e quais são os agentes (o agente robô que tem seu código no arquivo robo.asl). Para ver mais detalhe sobre a BNF veja (BORDINI; HÜBNER et al., 2004).

O ambiente onde o robô está atuando é apresentado no quadro 2.3. É simulado uma sala $2X2$ contendo sujeira em dois lugares. O robô consegue perceber sujeira através dos seus sensores. O método `executeAction` simula os pedidos de atuação do robô, no caso do quadro 2.3, estas ações podem ser andar (move o agente para uma nova posição x, y) ou aspirar (remove o lixo do ambiente). Quando o robô faz percepção o simulador do ambiente lhe envia onde tem sujeira e a sua posição no ambiente, através do método `getPercepts`.

As percepções do ambiente do agente robô são: a posição que ele se encontra no

```
MAS robotAspirador {
      architecture:
                    Centralised
      environment:
                    Mundo
      agents:  robo robo.asl;
}
```

Quadro 2.2 – Configuração do arquivo de projeto na ferramenta ***Jason***.

ambiente através do predicado `pos(X,Y)` (ver marca (3) no quadro 2.3) e a crença que tem lixo onde ele se encontra, representada pelo predicado `temLixo` (ver marca (4)). As ações que o agente executa no ambiente são: a ação de aspirar o lixo (ver marca (2)) na coordenada onde ele se encontra e mover (ver marca (1)) para uma outra posição no mundo representadas pelos predicados `aspirarLixo` e `andar(posLivre)` respectivamente.

No quadro 2.4 é apresentado o código *AgentSpeak(L)* do robô aspirador de pó. Os planos são denotados por um label (Pn) para que se possa referir ao plano no texto que segue.

O plano `P1` é executado sempre que o agente recebe do ambiente uma percepção `pos(X, Y)`, ou seja, quando ele perceber uma nova posição, estiver procurando por lixo e não tem lixo na posição que ele se encontra, então ele tem como intenção executar o corpo do plano que é apenas executar a ação no ambiente `andar(posLivre)` movendo o robô para a próxima posição no ambiente.

O plano `P2` é selecionado para execução quando o agente receber do ambiente o predicado `temLixo` e o robô estiver procurando por lixo, as intenções do agente são: parar de procurar por lixo (pois já encontrou), aspirar o lixo e continuar a procurar por lixo. Sendo que o corpo do plano é constituído por objetivos de realização, ou seja, o robô tem como primeiro objetivo executar o plano `P3` que remove a crença `procurandoLixo`, depois o agente tem como objetivo executar o plano `P4` que apenas executa a ação no ambiente de aspirar o lixo e por fim executa o plano `P5` que adiciona a crença `procurandoLixo` e move o robô para uma nova posição. Ficando neste ciclo até que o robô percorra todo o ambiente e não encontre mais lixo.

```
import jason.*;
public class Mundo extends Environment {
      private boolean[][] ambiente = new boolean[2][2];
     ...
     private int x = 0, int y = 0
     Term posAtual = null;
     public Mundo() {
       for(int i=0; i < 2; i++)
         for(int j=0; j < 2; j++)
           ambiente[i][j] = false;
       //sugeira
       ambiente[0][1] = true;
       ambiente[1][0] = true;
       posAtual = Term.parse("pos("+x+","+y+")");
       getPercepts().add(posAtual);
     }
     public boolean executeAction(String ag, Term action) {
       if (action.equals(andar)) { //(1)
         y++;
         if (y == 2) {
           y = 0;
           x++;
         }
         if (x == 2) //acabou grid
           return true;
         System.out.println("robo x="+x+"; y="+y);
       } else if (action.equals(aspirar)) { //(2)
         if ( ambiente[x][y] ) {
           ambiente[x][y] = false;
           getPercepts().remove(temLixo);
           ambiente[x][y] = false;
         }
       }
       getPercepts().remove(posAtual);
       posAtual = Term.parse("pos("+x+","+y+")");
       getPercepts().add(posAtual); //(3)
       if (ambiente[x][y])
         getPercepts().add(temLixo); //(4)
       return true;
     }
}
```

Quadro 2.3 – Ambiente do agente robô aspirador de pó.

```
procurandoLixo.
+ pos(X, Y) : procurandoLixo & not (temLixo) (P1)
     ← andar(posLivre).
+ temLixo :  procurandoLixo                  (P2)
    ← !parar;
    !executeAcaoAsp;
    !continar.
+!  parar :  true                            (P3)
    ← - procurandoLixo.
+!executeAcaoAsp :  true                     (P4)
    ← aspirarLixo.
+!continar :  true                           (P5)
    ← + procurandoLixo;
    andar(posLivre).
```

Quadro 2.4 – Planos *AgentSpeak(L)* do agente robô aspirador.

2.5 TRABALHOS CORRELATOS

Torres, Nedel e Bordini (2003), desenvolveram um trabalho usando a arquitetura BDI para construir personagens virtuais articulados e autonomos em um ambiente virtual. Para construção dos personagens (parte onde eles efetuam um raciocínio) foi utilizado a linguagem *AgentSpeak(L)*. A interface recebe um evento do ambiente e faz um mapeamento deste para uma percepção, que será repassada ao interpretador da linguagem *AgentSpeak(L)*. O interpretador verifica a percepção recebida, com base nas crenças e desejos do personagem e devolve uma ação que será executada no ambiente virtual.

No recente trabalho de Torres, Nedel e Bordini (2004), os personagens podem planejar seu comportamento sob multiplos focos de atenção, *i.e.*, o agente pode desviar sua atenção em função de uma nova percpção do ambiente.

Algumas técnicas e estratégias para construir um jogo PacMan podem ser encontradas em BONET e STAUFFER (2004?). Um jogo PacMan deve ser projetado, tendo paredes, pontos (bolinhas) e pílulas poderosas que tornam o come-come invencível por alguns instantes. Os níveis de dificuldade do jogo são decorrente a mudança das posições dos objetos: paredes, bolinhas e pílulas.

3 DESENVOLVIMENTO

Com base nas definições apresentadas no capítulo 2, neste capítulo é apresentado como foi implementado o jogo tipo PacMan. Jogo denominado PacMan_MAS (PacMan-*MultiAgent Systems*).

3.1 REQUISITOS PRINCIPAIS DO JOGO

Os principais requisitos do jogo são:

a) o cenário do jogo (ambiente) deve ser desenvolvido na linguagem Java, pois a ferramenta ***Jason*** pressupõe que o ambiente (onde os agentes estão atuando) seja construído nesta linguagem;
b) utilizar o paradigma de programação orientada a agentes com base em uma linguagem particular chamada *AgentSpeak(L)*;
c) desenvolver agentes BDI que devem criar e executar estratégias de armadilhas para pegar o come-come.

3.2 ESPECIFICAÇÃO

Para desenvolver este trabalho, inicialmente foi construído o jogo (ambiente). Após esta etapa, foi dado ênfase na implementação dos agentes fantasmas, visando que a implementação ficasse o mais próxima possível da arquitetura BDI. O *software* desenvolvido está separado em camadas (fig. 3.1): camada de persistência; camada de lógica do jogo e a camada de SMA (agentes).

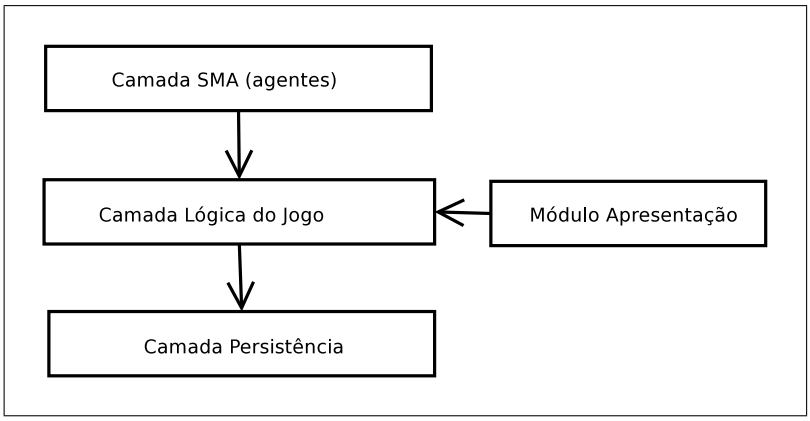


Figura 3.1 – Camadas do jogo PacMan_MAS.

O *software* foi desenvolvido em camadas para facilitar a manutenção e ajudar na organização / separação da lógica, procurando deixar uma camada independente das camadas que estão acima dela. Cada uma das camadas será detalhada nas seções seguintes.

3.3 CAMADA DA LÓGICA DO JOGO

Esta camada contém a lógica de um jogo tipo PacMan. Na fig. 3.2 é apresentado o diagrama de classes para construir o jogo PacMan_MAS. Também é apresentada neste diagrama a classe que desenha o jogo (classe da camada que interage com o usuário, denominada `GameCanvas`). Uma breve descrição do diagrama de classes é apresentado nos parágrafos a seguir.

A classe `GameObject` (objetos do jogo) é uma classe abstrata, contém os atributos que representam as coordenadas de um objeto no jogo (`x` e `y`) e possui dois métodos abstratos `int getHeight()` e `int getWidth()`, qualquer classe que desejar ser um objeto do jogo, deve implementar estes métodos para retornar a altura e largura do objeto. Esta classe também consegue verificar se um objeto do jogo está colidindo com outro objeto, através do método `collision(obj GameObject)`.

A classe `Player` (Jogador) é uma classe abstrata, possui como super classe `GameObject`. É classe base para `PacMan` (Come-come) e `Ghost` (Fantasma) e tem como principal método, `paint(Graphics g, Component c)` que desenha a figura do jogador (come-come ou fantasma) no componente passado por parâmetro.

A classe `PacManWorld` (mundo do pacMan) representa a principal classe do jogo, possuindo listas de `Ghost` (Fantasma), `Ball` (Bola), `Wall` (Parede) e um objeto `PacMan` (Come-come). Os principais métodos dessa classe são: `simulate(); lookPacMan(String ghostName, char direction); winner(); collideWall(Player player); collideBall() e collideGhost()`.

A classe `Ball` representa uma bolinha do jogo. Herda da classe `GameObject` para ser um objeto do jogo. Possui como atributos uma imagem de uma bolinha e seu estado (visível ou invisível), método para verificar se a bola está visível (`isVisible`) e método para desenhar a bolinha em um componente (`paint`).

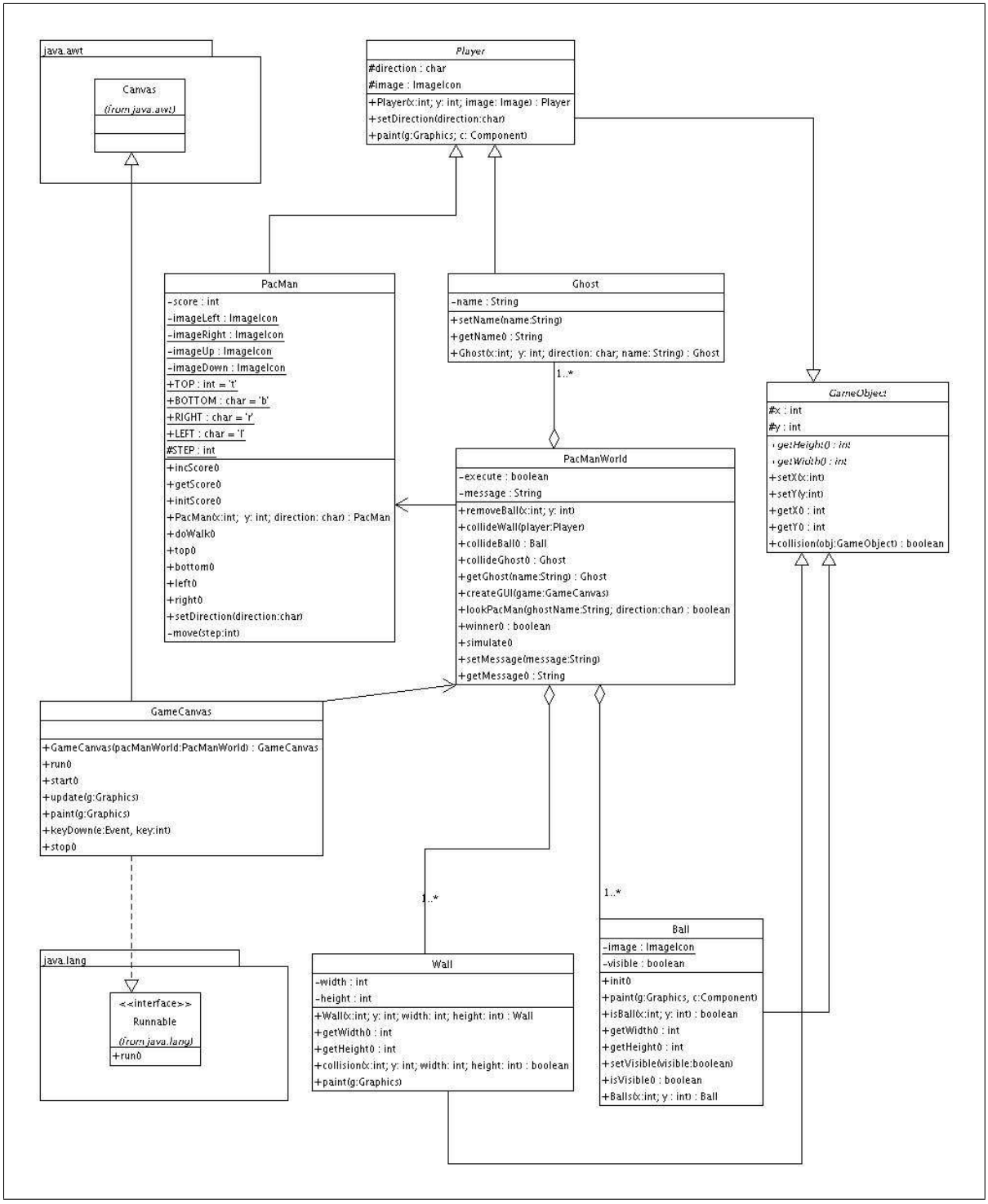


Figura 3.2 – Diagrama de Classes do jogo - pacote pacman.

3.3.1 MÓDULO DE APRESENTAÇÃO

A camada de apresentação, desenha o jogo PacMan_MAS. Foi utilizado a estratégia de *double buffer*[1] para desenhar as bolinhas, paredes, come-come e os fantasmas.

O usuário controla a direção do come-come, precionado as teclas *UP, DOWN, LEFT, RIGHT* que move o fantasma para cima, baixo, esquerda e direita, respectivamente. Para iniciar o jogo deve ser precionado a tecla S. Na fig. 1.2 é apresentado a interface do jogo desenvolvido.

3.4 CAMADA DE PERSISTÊNCIA

A camada de persistência faz todo o acesso de leitura de arquivos texto contendo as coordenadas dos objetos do jogo (paredes, bolinhas, etc).

No arquivo que contém as coordenadas das bolinhas (pílulas), cada linha representa uma instância de uma bolinha no jogo. Uma linha deste arquivo contém dois valores numéricos, o primeiro valor representa a coordenada x e o segundo valor representa a coordenada y da bolinha.

Os fantasmas são movimentados através de um grafo não dirigido. Para criar o grafo é necessário ler os dados de um arquivo de vértices e outro de arestas. Cada linha do arquivo de vértices contém três valores numéricos, o primeiro valor representa a coordenada x, o segundo valor representa a coordenada y e o terceiro valor representa um identificador (único) de cada vértice. Cada linha do arquivo de arestas possui dois valores numéricos, sendo que, cada número representa um identificador de um vértice e uma linha (contém dois números) representa a ligação entre os dois vértices. A fig. 3.3 apresenta o grafo do jogo, cada vértice é representado por um ponto e as arestas são representadas pelas ligações (linhas) entre os vértices.

Cada linha do arquivo de paredes, representa uma instância de uma parede. Uma linha deste arquivo contém quatro valores numéricos, o primeiro valor representa a coordenada x, o segundo valor representa a coordenada y, o terceiro valor representa a largura e o quarto valor representa a altura da parede.

[1] É criado uma imagem, feito todo o desenho do jogo nesta imagem (desenhando em segundo plano), para depois desenhar a imagem final na tela.

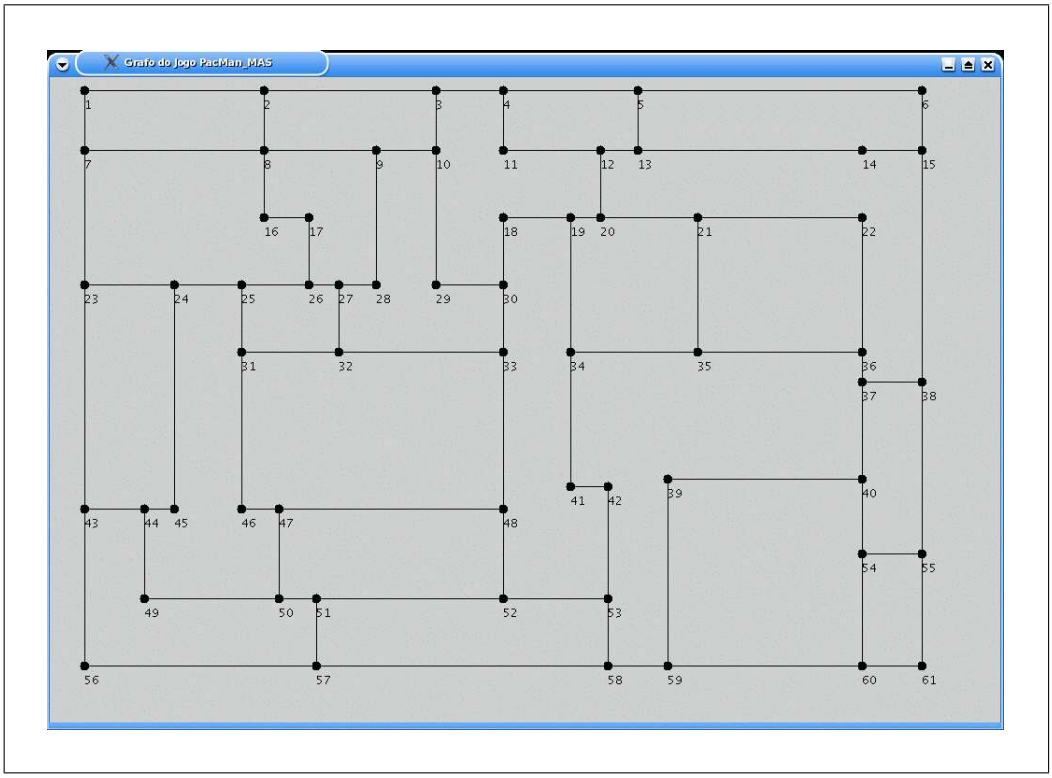


Figura 3.3 – Grafo do Jogo PacMan_MAS.

3.5 CAMADA SMA

Nesta camada estão implementados as classes: `EnvironmentPacMan` para efetuar uma ponte entre o jogo (ambiente) e o interpretador ***Jason***; a classe `AgentGhost` que customiza a escolha de planos do agente; e a classe `EstadoJogo` (fig. 3.7) utilizada para gerar os estados sucessores (do algoritmo de busca) que é utiliza pelo *framework*[2] de busca.

3.5.1 AMBIENTE DO SMA

O diagrama de classes que efetua a ligação do ambiente com a ferramenta ***Jason*** é apresentado na fig. 3.4. A classe `EnvironmentPacMan` herda da classe `Environment` (classe que a ferramenta ***Jason*** especifica) para se tornar um ambiente para o SMA, assim, efetuando uma ligação entre o ambiente e o ***Jason***.

Os principais métodos da classe `EnvironmentPacMan` são:

a) `executeAction(String ag, Term action)`, este método executa uma ação no ambiente. É passado por parâmetro o nome do agente e a ação que deve ser

[2]O prof. Jomi implementou e disponibilizou (`http://www.inf.furb.br/~jomi/ia`), algoritmos de busca baseados em Russel e Norvig (2003).

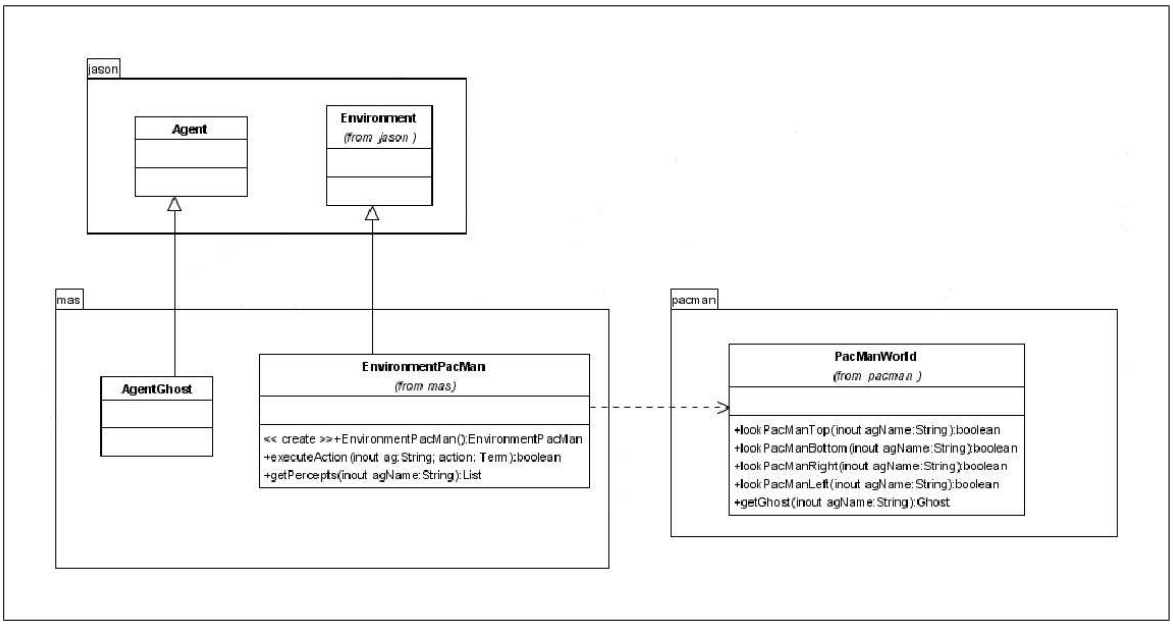


Figura 3.4 – Diagrama de Classes da ligação entre ***Jason*** e o ambiente.

executada, e retorna *true* se conseguiu executar a ação ou *false* se falhar;

b) `getPercepts(String agName)`, este método é chamado por cada agente do SMA, sendo passado como parâmetro o nome do agente que está solicitando as percepções do ambiente. É retornada uma lista com as percepções (naquele instante) do agente.

Constantemente o agente solicita as suas pecepções ao ambiente e verifica se recebeu alguma mensagem de outro agente (fig. 3.5). O agente recebe do ambiente a sua posição (coordenada `x,y`), através da adição da crença `pos(X, Y)`. Ele também consegue perceber o come-come (coordenada `x,y`), através da adição da crença `pm(X, Y)`, desde que o come-come esteja em seu campo de visão, *i.e.*, não deve existir nenhuma parede entre o come-come e o agente nas quatro direções: norte; sul; leste e oeste.

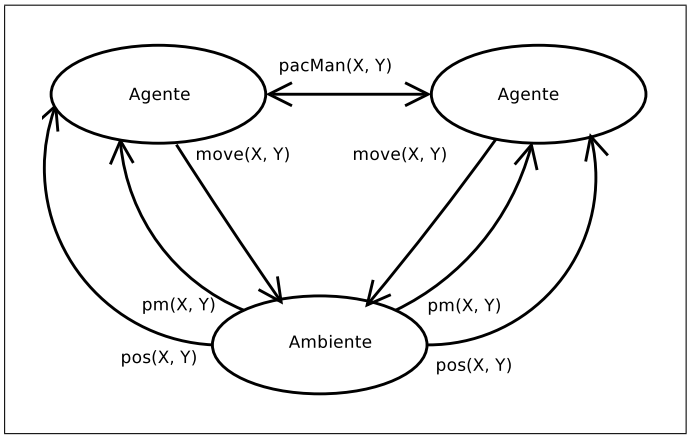


Figura 3.5 – Comunicação entre agentes e ambiente.

```
+ pos(X, Y) : not(moving) & not(cooperate) ←
    .nodoMaisProximo(X, Y, null, null, Xresult, Yresult, CornerGoal);
         + moving;
         !go(Xresult, Yresult, CornerGoal).
```

Quadro 3.1 – Plano `pos` - *AgentSpeak(L)*.

3.5.2 ESPECIFICAÇÃO DOS PLANOS DE MOVIMENTAÇÃO DO AGENTE

Quando o agente recebe a percepção de sua posição no ambiente, por meio do evento `+pos(X, Y)`, o plano do quadro 3.1 é aplicável. Caso o agente não esteja se movendo e não está em modo de cooperação, o plano é selecionado e vira uma intenção do agente.

O código do plano apresentado no quadro 3.1 faz o agente andar aleatoriamente. O corpo do plano, o agente executa a ação interna `.nodoMaisProximo(...)` para buscar a esquina mais próxima a ele naquele momento. É adicionado uma crença `moving` e por fim é executado o plano `go(...)` (objetivo de realização).

Para o agente descobrir uma esquina, ele executa a ação interna `.nodoMaisProximo(Xg, Yg, Xpm, Ypm, Xresult, Yresult, CornerGoal)`. Esta ação retorna as coordenadas x, y e o identificador da esquina nos parâmetros `Xresult`, `Yresult`, `CornerGoal`. Esta ação pode ser executada de duas maneiras: quando o agente está andando aleatoriamente ou quando entra em modo de cooperação (modo de cooperação é explicado a frente). Para mover o agente de uma esquina para outra aleatoriamente (os parâmetros `Xpm` e `Ypm` não estão instanciados), é necessário efetuar uma consulta ao grafo, descobrindo quais são as esquinas vizinhas. Após descobrir quais são as esquinas vizinhas, é sorteada uma esquina aleatoriamente para simular que o agente está se movendo de uma esquina para outra ao acaso.

Para o agente mover-se de uma esquina para outra, ele executa o plano `go(X, Y, CornerGoal)` recursivamente. O primeiro plano `go(X, Y, CornerGoal)` é executado quando o agente conseguiu chegar em uma esquina passada como parâmetro, quando o contexto deste plano falha é executado o segundo plano, que possui contexto igual a *true*. O quadro 3.2 apresenta o código *AgentSpeak(L)* que implementa estes conceitos.

A ação interna `.nextPosition(...)` retorna a próxima coordenada x, y (nos

```
+!  go(X, Y, CornerGoal):  pos(X, Y) ←
        - moving;
        + pos(X, Y). // para o plano de posição ficar ativo
+!  go(X, Y, CornerGoal):  true ←
        ?  pos(Xpos, Ypos);
        .nextPosition(Xpos, Ypos, CornerGoal, Xresult, Yresult);
        move(Xresult, Yresult);
        !go(X, Y, CornerGoal).
```

Quadro 3.2 – Plano `go` - *AgentSpeak(L)*.

parâmetros `Xresult, Yresult`) em direção a uma esquina que ele pretende ir. A esquina é representada pelo parâmetro `CornerGoal`, os parâmetros `Xpos, Ypos` representam a ultima posição do agente no ambiente.

O agente modifica o ambiente (alterar o estado do jogo) através da ação representada pelo literal `move(X, Y)`, que atualiza as coordenadas x, y do agente que executou esta ação.

3.5.3 ESTRATÉGIA DE ARMADILHA NO JOGO

Quando um fantasma percebe o come-come, ele gera dois objetivos: mover-se para as coordenadas x e y do come-come (para matar o come-come) e o outro objetivo gerado é de enviar uma mensagem para os outros agentes do SMA, informando a posição do come-come. Quando um fantasma recebe uma mensagem de um outro fantasma sobre a posição do come-come, ele entra em modo de cooperação e executa uma busca heurística para escolha de uma esquina.

A fig. 3.6, representa a criação de uma estratégia de armadilha no jogo. Um dos fantasmas percebeu o come-come à sua esquerda, o objetivo deste agente neste momento é mover-se para as coordenadas x e y do come-come (a linha representa o caminho que o fantasma deve percorrer, a seta indica a meta onde o agente deve chegar) e avisar os outros agentes da posição do come-come. Quando um fantasma recebe uma mensagem, contendo a informação que o come-come está em uma coordenada qualquer, ele entra em modo de cooperação.

Ao entrar em modo de cooperação, a ação interna `.nodoMaisProximo(...)` efetua

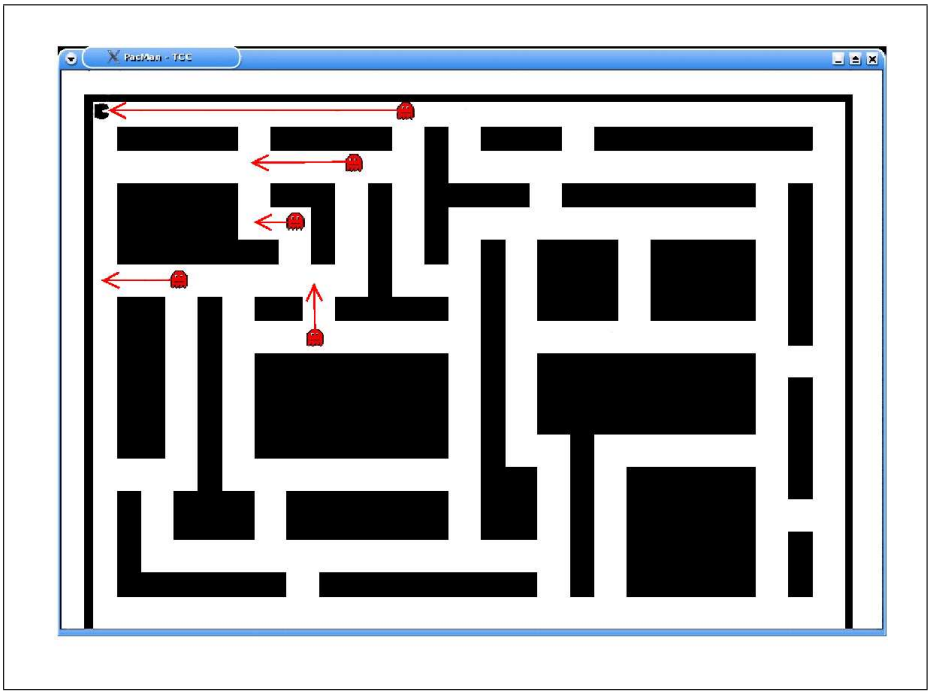


Figura 3.6 – Estratégia do Jogo PacMan_MAS.

uma busca, utilizando o algoritmo A* para saber qual é a esquina mais próxima a ele e ao come-come. É feita uma busca com heurística da menor distância em linha reta entre dois pontos. A fig. 3.7 apresenta o diagrama de classes para efetuar a busca. Neste diagrama é apresentada a classe *EstadoJogo*, que é utilizada para gerar os estados sucessores que serão utilizados pelo algoritmo de busca. Os estados sucessores possuem como *custo* a distância de uma esquina até outra. O g é o custo acumulado de todo o caminho (soma do *custo*). O h é a distância do vértice corrente até o vértice meta. O f representa a soma do $g + h$. As demais classes desse diagrama fazem parte do *framework* de busca.

O plano do quadro 3.3 é selecionado quando um agente está vendo o come-come, este plano torna-se uma intenção quando é percebido o evento `+pm(X, Y)`. O agente deve comunicar-se com os outros agentes para avisar onde o come-come esta (a posição x, y do come-come) e mover-se em direção ao come-come. Para o agente se mover em direção ao come-come, ele executa a ação interna `.nodoMaisLonge(...)`, para buscar a esquina mais longe em linha reta em relação ao come-come e o agente, *i.e*, fazendo o agente matar o come-come. Para o agente comunicar-se com os outros agentes ele executa o plano `sendCoordinatePacMan(X, Y, ghost, 1)`, os parâmetros x, y representam a posição do come-come, o terceiro parâmetro (`ghost`) será concatenado com um número que estiver no quarto parâmetro (o número começa com 1 e termina com 5), representando os nomes dos agentes (ghost1, ghost2, ..., ghost5).

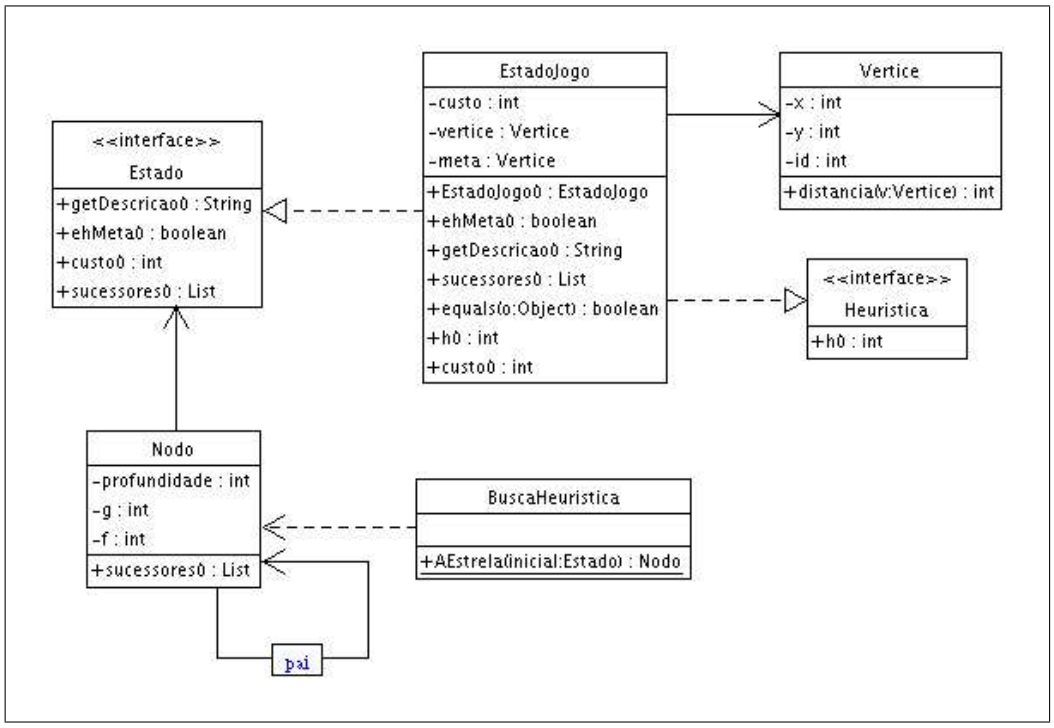


Figura 3.7 – Diagrama de classes da geração dos estados sucessores.

```
+ pm(X, Y) : //...
← //...
  ? pos(Xg, Yg);
  .nodoMaisLonge(Xg, Yg, X, Y, Xresult, Yresult, GoalEsquina);
  ! sendCoordinatePacMan(X, Y, ghost, 1);
  + moving;
  ! go(Xresult, Yresult, GoalEsquina).
```

Quadro 3.3 – Plano `pm` - *AgentSpeak(L)* - Agente recebe a percepção `pm`.

```
// notice that the pacMan is in position X an Y
+!  sendCoordinatePacMan(X, Y, Name, Number):  .name(N) &
  numberAg(Num) & .concat(Name, Number, NewName) &
  N != NewName & Number < Num ←
         .send(NewName, tell, pacMan(X,Y));
         .plus(Number, 1, NewNumber);
         !  sendCoordinatePacMan(X, Y, Name, NewNumber).
+!  sendCoordinatePacMan(X, Y, Name, Number):
  numberAg(N) & Number < N ←
         .plus(Number, 1, NewNumber);
         !  sendCoordinatePacMan(X, Y, Name, NewNumber).
+!  sendCoordinatePacMan(X, Y, Name, Number):  true ← true.
```

Quadro 3.4 – Planos `sendCoordinatePacMan` - *AgentSpeak(L).*

A comunicação entre os agentes é feita através do envio de uma mensagem. Para isso, ele executa a ação interna `.send(ghost, tell, literal)`, o primeiro parâmetro reprensenta quem vai receber a mensagem, o segundo parâmetro representa que a mensagem é afirmativa (LABROU; FININ, 1997), o último parâmetro representa um literal que será enviado. Ao perceber o come-come, o agente tem como objetivo avisar os outros agentes da posição do come-come. Para isso, ele executa o plano `sendCoordinatePacMan(...)` (quadro 3.4). No corpo deste plano é executado a ação interna `.send(...)` que envia a mensagem com o literal `pacMan(X,Y)`, para um determinado agente. Para evitar que o agente mande uma mensagem para ele mesmo, ele executa a ação interna `.name(MyName)`, no contexto do plano `sendCoordinatePacMan`, esta ação retorna o nome do agente (que executou esta ação) no parâmetro `MyName`.

Ao receber a mensagem contendo o literal `pacMan(X,Y)`, é ativado o plano do quadro 3.5, deixando o agente em modo de cooperação. O agente sabe que um outro agente está vendo o come-come na coordenada x, y, então ele executa uma busca para mover-se na direção do come-come, criando uma estratégia de armadilha.

Os planos podem ser executados em paralelo. Para a estratégia de armadilha dar certo o agente deve preferir pelos planos de movimentação (`go(...)`) e pelos planos de percepção do come-come (`pm(...)`), para fazer isto, é necessário implementar a classe `AgentGhost` que customiza a seleção de planos do agente. Caso não seja implementado uma seleção de planos, a busca heurística pode ser executado com uma visão parcial do

```
+ pacMan(X,Y) : true
          ← + cooperate.
```

Quadro 3.5 – Plano `pacMan` - *AgentSpeak(L)* - Agente entra em modo de cooperação.

mundo.

3.6 IMPLEMENTAÇÃO DA CAMADA DE LÓGICA DO JOGO

A classe `GameCanvas` faz parte da camada de apresentação, herda as características de um `Canvas` e implementa `Runnable` (para se comportar como uma `Thread`). Possui uma referência para um objeto de `PacManWorld`, executando constantemente (enquanto não for fim de jogo) o método `simulate` e desenhando o jogo com as novas atualizações.

O método `executeAction(String ag, Term action)` é apresentado no algoritmo 3.1. O método recebe dois parâmetros de entrada. O primeiro parâmetro representa o nome do agente que está executando uma ação no ambiente. A ação a ser executada é representada pelo segundo parâmetro. É descoberto qual agente está executando o método, através do nome. Após isto, é verificado qual a ação que o agente pretende executar, se ação for igual a se mover é buscado a coordenada x e y (através do segundo parâmetro) e é atualizado os atributos x e y do fantasma, movendo o agente para a nova posição no ambiente.

Entrada: agente e ação
Saída: verdadeiro ou falso
1 pega agente que chamou o método;
2 **if** *ação igual move* **then**
3 | pegar valor x e y da ação;
4 | atualizar a nova posição x e y do agente;
5 **return** *verdadeiro*

Algoritmo 3.1: Algoritmo do método *executeAction*

O algoritmo do método `getPercepts(String agName)` é apresentado no algoritmo 3.2. O método recebe como parâmetro o nome do agente que deseja receber suas percepções. É criada uma lista para retornar as percepções do agente. Se o agente estiver vendo o come-come, em qualquer uma das direção, acima, abaixo, à esquerda ou à direita (o agente não consegue ver o come-come se ele estiver atrás de uma parede) é criada uma percepção `pm(x,y)` representando que o agente está vendo o come-come nas coordenadas

x, y. Além disso, é gerado uma percepção `pos(x,y)`, representando a posição do agente no ambiente. Por fim é retornado a lista de percepções.

```
   Entrada: agente
   Saída: lista de percepções
 1 criar lista;
 2 if agente está vendo o come-come a cima, a baixo, esquerda, direita then
 3 |  pegar valor x e y da posição do come-come;
 4 |  criar percepção pm(x, y);
 5 |  adicionar percepção na lista;
 6 pegar fantasma que chamou este método;
 7 pegar valor x e y da posição do fantasma;
 8 criar percepção pos(x,y);
 9 adicionar percepção na lista;
10 return lista
```

Algoritmo 3.2: Algoritmo do método *getPercepts*

Uma breve descrição da classe `PacMan`, é apresentada a seguir. O come-come, pode mover-se nas quatro direções: cima, baixo, esquerda e direita (`top(), bottom(), left(), right()`). O método `move(int step)`, é um método privado, atualiza o atributo `x` ou `y` conforme a direção do come-come. O parâmetro, indica quantos passos (*pixels*) o come-come deve mover-se em uma direção. Este método é chamado internamente, pelos métodos: `doWalk(); back()`.

Os métodos `top(), bottom(), left()` e `right()`, atualizam a direção e imagem do come-come (virando-o para a direção desejada). Estes métodos são chamados pela classe `GameCanvas` quando o usuário deseja mudar a direção do come-come. O método `back()`, faz o come-come voltar para última posição visitada, para isso, é feita uma chamada ao método `move(-STEP)` com o argumento negativo.

Os principais métodos da classe `Ghost` (Fantasma) são: `getName()`, retorna o nome do agente; e os métodos herdados `setX(int x), setY(int y), getX(), getY()`, para alterar e recuperar o valor do atributo `x` e `y`.

O algoritmo do método `simulate` (*loop* principal do jogo, classe `PacManWorld`) é apresentado no algoritmo 3.3. Em síntese, é enviada a mensagem para o personagem come-come andar, é verificado se o come-come está colidindo com alguma bola, se for o caso, a bola que colidiu não é mais desenhada (fica invísivel) e também são incrementados os pontos do come-come. Se o come-come estiver colidindo com uma parede, é enviada uma mensagem para ele voltar uma posição. O jogo termina ou quando o come-come colidiu com um fantasma ou quando é fim de jogo, isto é, todas as bolinhas já foram

comidas.

```
1  while não é fim de jogo do
2      manda pacman andar;
3      if pacman colidir com uma bola then
4          muda estado da bola para invisível;
5          incrementa pontos pacman;
6      if todas as bolinhas estão invisíveis then
7          pacman venceu o jogo
8      if pacman colidir com parede then
9          manda pacman voltar;
10
11     else if pacman colidir com fantasma then
12         pacman perdeu o jogo;
```

Algoritmo 3.3: Algoritmo do *Loop* principal do jogo

Para perceber a presença do come-come o agente chama o método `lookPacMan(String ghostName, char direction)`. Este método retorna verdadeiro se o fantasma estiver vendo o come-come. O primeiro parâmetro `ghostName` representa o nome do fantasma que está chamando este método. O segundo parâmetro representa a direção (acima, abaixo, à esquerda ou à direita) que o fantasma está olhando. O método `winner` retorna verdadeiro se todas as bolinhas estiverem invisíveis, em outros casos retorna falso. O método `collideWall(Player player)` retorna verdadeiro se o come-come ou fantasma estiver colidindo com uma parede. O método `collideBall()` retorna verdadeiro se o come-come estiver colidindo com uma bolinha, em outros casos retorna falso. O método `collideGhost()` verifica se algum fantasma está colidindo com o come-come.

3.7 RESULTADOS

O principal resultado é que a estratégia de armadilha proposta neste trabalho é criada pelos agentes. Porém, quando os fantasmas estão muito longe do come-come a estrátegia não tem muito sucesso, visto que, a estratégia adotada é de se mover para uma esquina em relação ao come-come para prendê-lo em algum lugar do mundo, fica difícil para eles conseguirem prender (matar) o come-come. Outro fator que pode causar a falha na estratégia de armadilha é quando o come-come sai fora do campo de visão de um dos fantasmas os outros fantasmas sairão do modo de cooperação (ver seção 3.5.3).

Uma limitação do *software* desenvolvido é que, não foi implementada uma negociação entre os agentes, evitando que eles tenham objetivos de ir para mesma esquina, isso pode acontecer quando a ação interna de `nodoMaisProximo` retornar a mesma esquina

(naquele momento) para os agentes. Uma forma de negociação para resolver este problema poderia ser o agente que estiver mais próximo da esquina se mover em relação a esta esquina e o outro agente deveria escolher uma outra esquina. Quando a distância para chegar na esquina for a mesma para os agentes, poderia ser feita uma escolha aleatória de duas esquinas, uma para cada agente.

Este trabalho representa uma das maiores implementações (utilizando grafo, busca heurística, cooperação, representação gráfica dos agentes *AgentSpeak(L)*, etc) que se conhece, usando a ferramenta ***Jason***. A ferramenta demonstrou-se adequada na execução de códigos *AgentSpeak(L)*. A comunicação entre os agentes é feita de maneira transparente e de alto nível para o programador, pois o ***Jason*** garante o envio e o recebimento das mensagens através da ferramenta SACI (HÜBNER; SICHMAN, 2000). O SACI permite trocar mensagens entre agentes, que podem estar distribuídos em uma rede, através do padrão KQML.

O objetivo deste trabalho não era testar a ferramenta ***Jason***, mas como ela está em desenvolvimento, foram feitos vários testes (com o desenvolvimento deste trabalho), conseqüêntemente encontrando alguns *bugs* (um dos *bugs* encontrados, o código do ambiente não podia estar dentro de pacotes) que foram comunicados aos desenvolvedores da ferramenta e corrigidos.

O jogo ficou um pouco lento, em conseqüência dos fantasmas se moverem de dois em dois *pixels*. Contudo, como o *software* está separado em camadas, fazer alterações para os fantasmas se moverem mais rápido, envolve alterar a ação interna `nextPosition` e o cógido *AgentSpeak(L)*. Para desenvolver novas técnicas de cooperação para o jogo, basta apenas alterar o código *AgentSpeak(L)* e ações internas dos agentes, alterando apenas a camada SMA (agentes).

Fazer os agentes cooperarem não é uma tarefa trivial, pois os agentes devem fazer um planejamento, este planejamento pode ser centralizado ou distribuído, no caso deste trabalho é feito um planejamento distribuído (fig. 2.1).

3.8 DIFICULDADES ENCONTRADAS

No desenvolvimento deste trabalho, uma das dificuldades encontradas foi que, por se tratar de um SMA atuadando sobre um jogo (tempo real) o "estado mental" de cada agente e o ambiente estão mudando constantemente, dificultando a depuração dos agentes. A ferramenta ***Jason*** permite acompanhar as mudanças nas crenças, desejos e intenções

de cada agente (através do comando `[verbose = 3]`, que deve ser colocado no arquivo de projeto antes de terminar a definição do agente, mais detalhes (BORDINI; HÜBNER et al., 2004)), para isso, a ferramenta gera um *log* do "estado mental de cada agente".

Outra dificuldade, está relacionada com a mudança de paradigma de programação Orientação a Objetos (OO) para orientada a agentes.

Foram encontradas poucas bibliografias sobre a linguagem *AgentSpeak(L)* (linguagem especificada a poucos anos (RAO, 1996)), existindo poucas aplicações com esta linguagem. Contudo, Os poucos códigos que estão disponíveis na ferramenta ***Jason*** foram de grande utilidade para compreensão da linguagem e construção deste trabalho. Alguns planos podem ser executados em paralelo, causando inconsistências nas técnicas de cooperação utilizadas caso não seja implementado uma seleção de intenção.

4 CONCLUSÕES

A proposta deste trabalho, usar a arquitetura BDI e cooperação para criar armadilha no tipo jogo PacMan, mostrou-se, de forma geral bastante adequada. A arquitetura BDI proporciona construir jogos onde os personagens devem ter um certo nível de inteligência, as atitudes dos personagens estão mudando constantemente de acordo com o estado corrente do jogo e de sua "mente".

Uma das vantagens da abordagem adotada, é que o comportamento de cada agente é muito simples, não existindo nenhum plano explícito para criar armadilhas, os planos são apenas de movimentação de uma esquina para outra. A partir da interação dos agentes surge (emerge) o comportamento de criar uma armadilha (seção 3.5.3).

A linguagem *AgentSpeak(L)* é muito poderosa para desenvolver aplicações onde o ambiente dos agentes muda constantemente. Os planos são facilmente construídos e ampliados. Por ser uma linguagem declarativa, a programação em *AgentSpeak(L)* se torna mais elegante, proporcionando um alto nível de abstração na especificação (crenças, desejos e intenções) do agente.

Mesmo os jogos mais sofisticados (que usam busca, possuindo a localização global da posição do come-come) de PacMan, não garantem que o computador vença o jogo, em muitos casos, os fantasmas ficam correndo atrás do personagem come-come até o fim do jogo. Neste trabalho, quando os agentes estão perto das esquinas próximas ao come-come e algum fantasma está vendo o come-come, é quase impossível o come-come fugir da estratégia de armadilha, pois a armadilha vai fechar todas as passagens pelas esquinas próxima do come-come.

4.1 EXTENSÕES

Propõe-se como extensões sobre este trabalho:

a) desenvolver outras técnicas de armadilhas para este jogo, como, dar papéis aos agentes, onde o agente pode ser o organizador de uma estratégia e os outros agentes apenas executarem as tarefas recebidas do agente organizador. Desenvolver mecanismos para evitar que os agentes se movam para a mesma esquina. Quando os agentes estiverem muito longe do come-come a estratégia poderia ser

de eles se moverem para a segunda ou terceira esquina em relação ao come-come. Outra técnica de armadilha poderia ser quando o come-come sair do campo de visão do agente, ele deve informar qual a ultima posição do come-come para os agentes tentarem imaginar qual será o caminho que o come-come vai percorrer;

b) desenvolver jogos mais sofistivados (com animações em 3D, realidade virtual, etc) que o PacMan, utilizando a linguagem *AgentSpeak(L)* para dar inteligência ao jogo;

c) desenvolver personagens autônomos em um ambiente virtual (TORRES; NEDEL; BORDINI, 2003);

d) desenvolver sistemas de simulação (sociedade de seres humanos interagindo uns com os outros). Para isso, poderiam ser utilizadas as técnicas de computação gráfica para fazer a simulação gráfica dos personagens humanos e a arquitetura BDI para construir a parte responsável por interagir com outro humano da sociedade.

REFERÊNCIAS BIBLIOGRÁFICAS

ALVARES, Luiz Otavio; SICHMAN, Jaime Simão. Introdução aos sistemas multiagentes. In: MEDEIROS, Cláudia Maria Bauzer (Ed.). **Jornada de Atualização em Informática (JAI'97)**. Brasília: UnB, 1997. cap. 1, p. 1–38.

BARONE, Dante Augusto Couto et al. **Sociedades artificiais**: a nova fronteira da inteligência nas máquinas. Porto Alegre: Bookman, 2004.

BATTAIOLA, André Luiz. Jogos por computador - histórico, relevância tecnológica e mercadológica, tendências e técnicas de implementação. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (SBC2000), XIX JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA (JAI), 19. **Anais...** Curitiba-PA, Brasil: Sociedade Brasileira de Computação, 2000.

BONET, Jeremy S.; STAUFFER, Chris P. **Learning to play PacMan using incremental reinforcement learning**. Apresenta estratégias para jogo pacman. [S.l.], 2004? Disponível em: <http://www.debonet.com/Research/Learning/PacMan>. Acesso em: 10 maio 2004.

BORDINI, Rafael H.; HÜBNER, Jomi F. et al. ***Jason*: a java-based agentspeak interpreter used with SACI for multi-agent distribution over the net**. Manual, first release. [S.l.], Jan 2004. Disponível em: <http://jason.sourceforge.net>. Acesso em: 20 set. 2004.

BORDINI, Rafael H.; VIEIRA, Renata. Linguagens de programação orientadas a agentes: uma introdução baseada em AgentSpeak(L). **Revista de Informática Teórica e Aplicada**, v. 10, p. 7–38, Agosto 2003. Instituto de Informática da UFRGS, Brazil.

BORDINI, Rafael Heitor; VIEIRA, Renata; MOREIRA, Álvaro Freitas. Fundamentos de sistemas multiagentes. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (SBC2001), XX JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA (JAI), 21. **Anais...** Fortaleza-CE, Brasil: Sociedade Brasileira de Computação, 2001. p. 3–41.

BRATMAN, Michael E. **Intentions, plans and practical reason**. Cambridge, MA: Harvard University Press, 1987.

DEMAZEAU, Yves; MÜLLER, Jean Pierre. **Decentralized artificial inteligence 1**. North-Holland: Elsevier Science Publishers, 1990.

DENNETT, Daniel C. **The intentional stance**. Cambridge, MA: The MIT Press, 1987.

HÜBNER, Jomi Fred; SICHMAN, Jaime Simão. SACI: Uma ferramenta para implementação e monitoração da comunicação entre agentes. In: MONARD,

Maria Carolina; SICHMAN, Jaime Simão (Ed.). International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (IBERAMIA/SBIA 2000), 7/15, Atibaia, São Paulo, Brazil, 2000**. Proceedings (Open Discussion Track)**. São Carlos: ICMC/USP, 2000. p. 47–56. Disponível em: <http://www.inf.furb-.br/~jomi/pubs/2000/Hubner-iberamia2000.pd>. Acesso em: 15 mar. 2004.

JENNINGS, N. R. **Cooperation in industrial multi-agent systems**. Singapore: World Scientific Publishing Co. Pte. Ltd, 1994. ISBN 981-02-1652-1.

JENNINGS, N. R. Coordination techniques for distributed artificial intelligence. In: (EDS), O'Hare G.M.P Jennings N.R. (Ed.). **Foundations of distributed Artificial Intelligence**. [S.l.]: John Wiley & Sons, Inc, 1996.

JENNINGS, Nicholas R.; WOOLDRIDGE, Michael J. (Ed.). **Agent technology**: foundations, applications, and markets. Berlin: Springer-Verlag, 1998.

LABROU, Yannis; FININ, Tim. **A proposal for a new KQML specification**. Baltimore, 1997.

MACHADO, Rodrigo; BORDINI, Rafael H. Running AgentSpeak(L) agents on SIM_AGENT. In: INTERNATIONAL WORKSHOP ON AGENT THEORIES, ARCHITECTURES, AND LANGUAGES (ATAL-2001), INTELLIGENT AGENTS VIII, 8., 2001, Seattle. **Proceedings...** Berlin: Springer-Verlag, 2002. (Lecture Notes in Artificial Intelligence), p. 158–174.

OLIVEIRA, Eugênio. Agents advanced features for negotiation and coordination. In: LUCK, Michael et al. (Ed.). **Multi-agent systems and applicatons**. Prague: Springer, 2001. p. 173–186.

RAO, Anand S. AgentSpeak(L): BDI agents speak out in a logical computable language. In: WORKSHOP ON MODELLING AUTONOMOUS AGENTS IN A MULTI-AGENT WORLD (MAAMAW'96), 7., 1996, Eindhoven, The Netherlands. **Proceedings...** London: Springer-Verlag, 1996. (Lecture Notes in Artificial Intelligence), p. 42–55.

RUSSEL, Stuard; NORVIG, Peter. **Artificial intelligence**: a modern approach. 2^o. ed. New Jersey: Prentice Hall, 2003. ISBN 0-13-790395-2.

TORRES, Jorge A. R.; NEDEL, Luciana P.; BORDINI, Rafael H. Using the BDI architecture to produce autonomous characters in virtual worlds. In: RIST, Thomas et al. (Ed.). **Proceedings...** Heidelberg: Springer-Verlag, 2003. (Lecture Notes in Artificial Intelligence, 2792), p. 197–201. Short paper.

TORRES, Jorge A. R.; NEDEL, Luciana P.; BORDINI, Rafael H. Autonomous agents with multiple foci of attention in virtual environments. In: INTERNATIONAL CONFERENCE ON COMPUTER ANIMATION AND SOCIAL AGENTS (CASA 2004), 17. **Proceedings...** Geneva: Switzerland, 2004. p. 189–196.

WEISS, Gerhard. **Multiagent systems**: a modern approach to distributed artificial intelligence. Cambridge, MA: MIT Press, 2000.

WOOLDRIDGE, Michael. Intelligent agents. In: WEISS, Gerhard (Ed.). **Multiagent Systems–A Modern Approach to Distributed Artificial Intelligence**. Cambridge, MA: MIT Press, 1999. cap. 1, p. 27–77.

WOOLDRIDGE, Michael. **An introduction to multiagent systems**. New York: John Wiley & Sons, 2002.