UNIVERSIDADE FEDERAL DE SANTA CATARINA

SISTEMAS NEBULOSOS HIERÁRQUICOS PARA IMPLEMENTAÇÃO DE COMPORTAMENTOS EM ROBÔS MÓVEIS AUTÔNOMOS

Alexandre Vidal Riso

Florianópolis, SC

Novembro de 2004

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

SISTEMAS NEBULOSOS HIERÁRQUICOS PARA IMPLEMENTAÇÃO DE COMPORTAMENTOS EM ROBÔS MÓVEIS AUTÔNOMOS

Alexandre Vidal Riso

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Florianópolis, SC
Novembro de 2004

Alexandre Vidal Riso

SISTEMAS NEBULOSOS HIERÁRQUICOS PARA IMPLEMENTAÇÃO DE COMPORTAMENTOS EM ROBÔS MÓVEIS AUTÔNOMOS

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Orientador: Mauro Roisenberg, Dr.

Banca examinadora:

Jorge Muniz Barreto, Dr.
Jovelino Falqueto, Dr.

"Those who do not want to imitate anything, produce nothing."

Salvador Dalí

"No computer has ever been designed that is ever aware of what it's doing; but most of the time, we aren't either."

Marvin Minsky

"No, no, you're not thinking, you're just being logical."

Niels Bohr

Às minhas irmãs Fernanda e Vanessa e à minha namorada Nina.

Agradecimentos

Aos meus pais, pela imensa ajuda.

Ao meu orientador Mauro Roisenberg e ao professor Jorge M. Barreto por me apresentaram um tema tão empolgante, que é a inteligência artificial, me motivando para a realização desse trabalho.

Aos meus avós e minhas tias Aparecida e Maristela, por fazerem bolos tão saborosos e pela simpatia e muita ajuda que forneceram durante os anos universitários.

A todos os meus familiares.

Ao Amon e Jairo pelo apoio e ensinamentos que venho recebendo.

Aos desenvolvedores de extraordinários programas de domínio público, como o LaTeX, Debian GNU/Linux, KDE, Mozilla Firefox e Xfig.

Sumário

Lista de Figuras

Resumo

A navegação de robôs móveis autônomos em ambientes reais não estruturados é uma tarefa desafiadora, pois o ambiente é caracterizado por um grande número de incertezas, como presença de objetos que se movimentam, lugares bloqueados, superfícies escorregadias. Aliado às restrições sensoriais e computacionais dos robôs, o projeto de um sistema de navegação pode ser inviabilizado diante de tantas complexidades e restrições. Técnicas de inteligência artificial são ideais para esses tipos de problemas, tanto que redes neurais, computação evolucionária e outras tem sido aplicadas com sucesso. Assim, este trabalho avalia e relata a implementação de um sistema nebuloso hierárquico aplicado ao problema da navegação. Modelos de sistemas nebulosos hierárquicos, tendo como base módulos comportamentais, apresentaram grande robustez e simplicidade na busca de soluções para esse problema.

Palavras-chave: sistemas nebulosos hierárquicos, robótica, comportamentos robóticos, inteligência artificial.

Abstract

The navigation of an autonomous mobile robot in a real-world unstructured environment is a challenging task; because the environment is characterized by the large amount of uncertainties, as objects movement, blocked places and slippery surfaces. With the sensorial and computational restrictions of the robots together, the navigation system project can be impracticable with so many complexities and restrictions. Techniques of artificial intelligence are ideal for these types of problems, in fact, neural nets, evolutionary computation and others have been applied successfully. Thus, this work evaluates and explains the implementation of a hierarchical fuzzy system applied to the problem of the navigation. Models of hierarchical fuzzy systems, based on behaviors modules, had presented great robustness and simplicity in the solution for this problem.

Keywords: hierarchical fuzzy systems, robotics, robotic behaviors, artificial intelligence.

Capítulo 1

Introdução

As capacidades de locomoção, auto-localização, exploração de ambientes e relacionamento com membros da mesma espécie e de outras são vitais para a maioria dos animais, pois através delas pode-se obter autonomia, segurança, perpetuação da espécie, alimentos, ou seja, a sobrevivência. Todavia, o domínio dessas habilidades é desafiador, porquanto requerem treino, reconhecimento do meio em que se encontra, capacidade de adaptação a novas situações e imprevisibilidades, controle motor, domínio de linguagens.

Entretanto, mais desafiador ainda é dotar robôs móveis autônomos com essas mesmas capacidades, devido a grande complexidade envolvida, conseqüente principalmente das incertezas que caracterizam um ambiente real, como posição e movimentação de objetos, lugares ou passagens bloqueadas(9).

No entanto, o desenvolvimento de sistemas de navegação para robôs tem sido bastante auxiliado, inspirado pela compreensão da navegação animal, que desempenham com maestria essa habilidade. Muitos insetos tem sido a fonte inspiração para pesquisas nessa área, como formigas e abelhas.

A navegação pode ser considerada como uma das mais importantes capacidades, indispensável a um robô móvel autônomo, pois será através dela que ele obterá seus objetivos; permitindo evitar colisões, atingir um local específico, evitar situações perigosas(10).

Técnicas de inteligência artificial tem sido empregadas com sucesso nos últimos anos para resolver esse tipo de problema(11), como redes neurais artificiais(12), lógica nebulosa(8), algoritmos evolutivos(13), autômatos celulares(14) e combinações entre elas(11).

1.1 Breve Introdução às Abordagens ao Problema da Navegação

Além da grande quantidade de incertezas que caracterizam um mundo real, outros problemas podem dificultar ainda mais a tarefa de navegação: dados com ruídos e imprecisos providos dos sensores, dificuldades para a interpretação dessas informações, limitação do campo de visão, problemas de interação do robô com o ambiente, como escorregamento das rodas, bloqueio dos movimentos ou danos no robô causados por obstáculos não percebidos(9)(8).

Diante desse desafio, foram desenvolvidas basicamente 3 abordagens para o tratamento desse problema(15):

- Abordagem Planejada ou Deliberativa;
- Abordagem Reativa;
- Paradigma Cognitivo ou Híbrido.

1.1.1 Abordagem Planejada

Modelo tradicional para controle robótico, também conhecida como SMPA (*Sense-Model-Plan-Act, Sentir-Modelar-Planejar-Atuar*), visto na figura 1.1, que decompõem o problema em unidades funcionais e se caracteriza por utilizar um modelo estático do mundo, construído *a priori*(15)(16).

Possuem limitações que podem inviabilizar sua implementação quando aplicados em ambientes reais(8). Seus principais problemas são(8)(15):

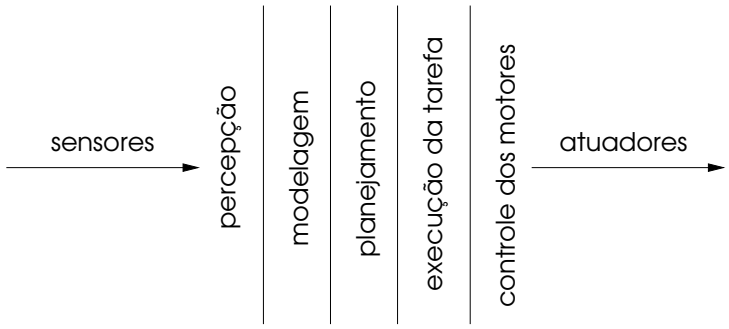

Figura 1.1: Decomposição SMPA(1)

- O modelo adquirido pelo robô é necessariamente incompleto e impreciso, devidos às incertezas na percepção;
- Não consegue tratar ambientes dinâmicos;
- A complexidade do ciclo SMPA pode levar o tempo de computação na ordem de segundos, excessivo em ambientes dinâmicos.

1.1.2 Abordagem Reativa

Busca resolver as limitações da abordagem planejada, possibilitando ao robô executar as ações como resultado direto dos estímulos recebidos pelos sensores. Por não gerar ou utilizar nenhum modelo do ambiente, um dos maiores desafios para a navegação e que geralmente necessita de sensores precisos e bastante recursos computacionais, permite ao robô responder rapidamente às variações das informações recebidas em um mundo dinâmico, desconhecido e imprevisível.

Nessa abordagem, a camada de execução é dividida em pequenos e independentes processos, chamados de *comportamentos*(8), que executam uma tarefa específica, como seguir paredes, desviar de obstáculos, fugir de regiões perigosas, procurar por fontes de luz.

Com essa divisão *"os comportamentos operam assincronamente e em paralelo, baseando-se unicamente nos dados sensoriais relevantes para a tomada de decisão, assim o processamento da percepção do robô fica distribuído entre*

os diversos comportamentos"(16).

Rodney Brooks foi pioneiro nesse tipo de abordagem(1). Ele elaborou uma arquitetura chamada de *Subsumption (Subsunção)*, onde o controlador é decomposto em camadas verticais e concorrentes, sendo cada camada responsável por uma *competência* que especifica um comportamento, como visto na figura 1.2(16).

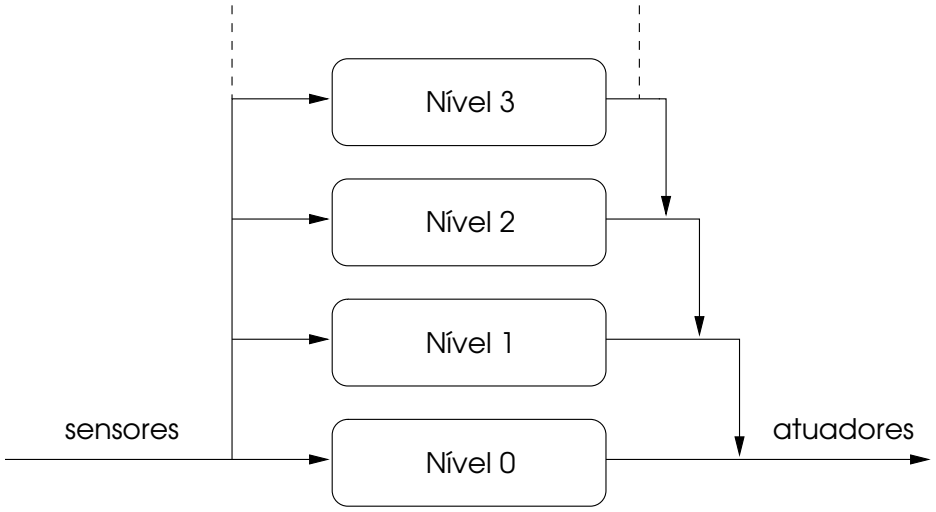

Figura 1.2: Relação entre as camadas de comportamentos na *Arquitetura de Subsunção*(1)

Essa arquitetura foi um marco para a navegação robótica, gerando grandes avanços na construção de robôs *inteligentes* ou que exibem um comportamento *inteligente* ao observador.

Uma característica interessante na arquitetura de Subsunção é a *emergência de comportamentos* originadas da interação entre as camadas de ação simples e o ambiente(12).

Essa arquitetura possui inúmeras vantagens que se destacam em relação à abordagem planejada:

- Tempo de resposta rápido aos estímulos, tornando-a adequada ao desenvolvimento de sistemas de controle em tempo-real;

- As camadas são independentes, possibilitando sua reutilização em outros projetos baseados nessa arquitetura;
- Falha em qualquer uma das camadas não causa necessariamente o colapso total do sistema (12);
- Possibilita processamento paralelo, devido à independência entre as camadas;
- Utiliza a estratégia *dividir para conquistar*, reduzindo a complexidade da modelagem(8);
- Comportamentos mais complexos podem emergir da interação de comportamentos mais simples.

Entretanto, nesse sistema não há nenhum objetivo, plano ou modelo do mundo explícito: *"ele simplesmente reage à situação que tem em mãos"*(12).

1.1.3 Abordagem Híbrida

Ambas abordagens citadas anteriormente possuem vantagens e desvantagens. Esse paradigma, também conhecido como *Paradigma Cognitivo* procura capturar as principais vantagens dessas abordagens, como o a resposta rápida e modelagem simplificada dos sistemas reativos com o conhecimento do mundo e a obtenção de planos e estratégias utilizados na abordagem planejada(15)(16).

Geralmente a integração desses modelos, apresentado na figura 1.3, é realizado de 2 formas. Através do uso de um mapa do ambiente construído *a priori*, podendo assim o robô saber sua localização e planejar sua estratégia, e com o uso de comportamentos para desvio de obstáculos não presentes no mapa. Ou através da construção dinâmica do mapa baseado na leitura dos sensores(16).

Seu grande desafio é obter durante sua modelagem um equilíbrio entre o planejamento e a reatividade(16).

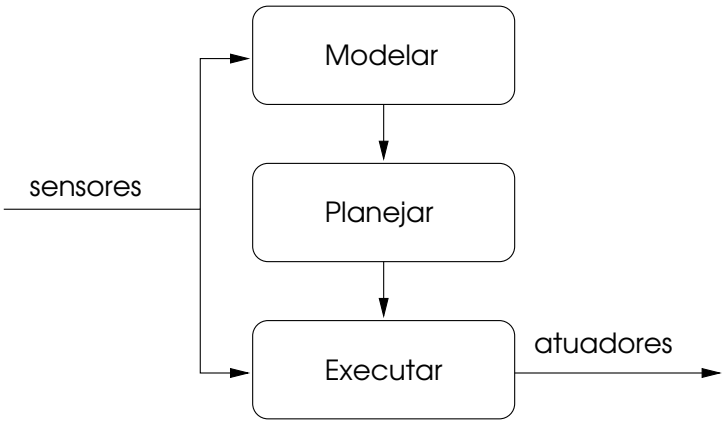

Figura 1.3: Modelo Hierárquico para controle robótico

Aprimoramentos nesse e nos outros modelos estão advindo de estudos nas áreas de neurociência, psicologia, fisiologia, buscando compreender mais detalhadamente as capacidades instintivas, reativas e cognitivas de animais para serem aplicados e se obter melhores resultados(16)(11)(12).

1.2 Motivação

Sistemas de navegação baseados na abordagem reativa, principalmente na arquitetura de subsunção, devem responder as seguintes questões durante sua fase de modelagem(8):

- Como projetar módulos comportamentais robustos e eficientes;
- Como combinar inúmeros comportamentos, podendo ser em algum momento contraditórios;
- Como usar as informações provindas dos sensores, geralmente imprecisas e ruidosas na modelagem e execução dos módulos;
- Como integrar os processos e representações que pertencem a diferentes camadas.

Além desses questionamentos, praticamente qualquer sistema de controle robótico deve possuir características como(1):

Múltiplos Objetivos Robôs móveis, ao contrário de robôs fixos, que normalmente realizam tarefas repetitivas, podem possuir vários objetivos, como evitar obstáculos, utilizar o mínimo consumo de energia na execução de uma tarefa, atingir um ponto específico;

Múltiplos Sensores Para a realização dos seus objetivos, um robô pode precisar de vários sensores, como sensores infravermelhos, câmera de vídeo, ultra-som;

Robustez Deve suportar às imprecisões de sensores e surpresas que o mundo real pode oferecer;

Expansibilidade Pode necessitar que mais sensores, recursos computacionais, atuadores sejam adicionados para que ele execute outras tarefas, ou que satisfaça novas restrições;

Reusabilidade Necessidade de projetar novos objetivos ou até novos robôs reutilizando o que já foi feito;

Baixo Acoplamento Módulos responsáveis por determinada tarefa podem ser reutilizados, portanto é adequado que ele tenha pouca dependência de outras partes do sistema.

Por isso, o projeto de controladores robóticos não é uma atividade trivial, e necessita de modelos conceituais claros para sua implementação(11), senão esta tarefa pode se tornar inviável diante das complexidades do mundo real e das limitações sensoriais, computacionais e de atuação de um robô.

1.3 Objetivos

1.3.1 Objetivo Geral

Utilização e avaliação de sistemas nebulosos hierárquicos para implementação de comportamentos em robôs móveis.

1.3.2 Objetivos Específicos

Implementar e testar um controlador nebuloso hierárquico, apresentado por (8), que implemente vários comportamentos, e que seja capaz de resolver o problema da atuação de comportamentos contraditórios ativados no mesmo instante.

Analisar a capacidade de satisfazer as características e responder as questões citadas na seção 1.2.

1.4 Organização

Este trabalho está dividido da seguinte maneira:

Capítulo 2 Apresenta a área da robótica, um pouco da sua história, conceitos e tipos de robôs, e sua situação nos dias atuais.

Capítulo 3 Aborda a teoria do conjuntos nebulosos e modelos para a implementação de sistemas nebulosos tradicionais e hierárquicos. Discutindo também modelos hierárquicos para implementação de comportamentos em robôs móveis.

Capítulo 4 Descreve a implementação de um sistema de navegação e seus detalhes utilizando o modelo nebuloso hierárquico *Context-dependent Blending*. Também é discutido os testes e resultados obtidos com ele.

Capítulo 5 Dedica-se às conclusões e perspectivas de trabalhos futuros.

Capítulo 2

Robótica

2.1 Introdução

A construção de artefatos autônomos tem instigado a imaginação humana desde as mais remotas eras. Esses artefatos, utilizados para auxiliar na realização de diversas tarefas, como o distribuidor de água de Delfos, na Grécia antiga(11), moinhos de vento, sofisticados robôs utilizados para a coleta de material e exploração do planeta Marte, têm revolucionado a sociedade e nossa percepção da realidade.

2.2 Breve Histórico

Entre os primeiros artefatos autônomos desenvolvidos, que se destacou por sua engenhosidade, foi o relógio de água, ou clepsidra, figura 2.1, projetado e aprimorado pelo físico e inventor grego Ctesibius no segundo século A.C.(17). Seu relógio foi o mais preciso por mais de um milênio, tendo sido superado apenas no século XVII, pelo relógio regulado a pêndulo, desenvolvido pelo holandês Christiaan Huygens(18).

Um outro dispositivo autônomo, famoso até os dias atuais, foi o notável *"The Duck"*, pato artificial construído pelo francês Jacques de Vaucanson, figura 2.2. Criado em 1739, ele era capaz de além de imitar os movimentos de

locomoção de um pato, também os movimentos para beber, comer, respirar e digerir. Ele possuía mais de 400 partes móveis. Infelizmente o original está desaparecido(17).

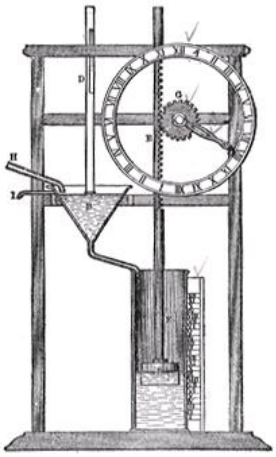

Figura 2.1: Clepsidra(2)

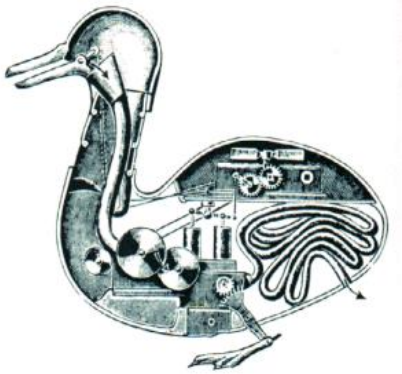

Figura 2.2: The Duck(3)

Atualmente os dispositivos autônomos estão tão imersos em nosso cotidiano, que suas utilidades vão desde simples bichinhos virtuais para divertimento *(Tamagochis)*, até corações artificiais. Entretanto, os robôs autônomos são um dos dispositivos mais explorados pelos meios de comunicação, como pode ser visto no recente filme A.I. (*Artificial Intelligence*), de Steven Spielberg e Stanley Kubrick. Robôs com aparência e comportamentos semelhantes aos humanos, como puderam ser vistos nesse filme, têm sido um sonho e que se está buscando realizar.

Em aproximadamente 500 anos, conseguiu-se uma evolução surpreendente na direção a esse objetivo. A figura 2.3 mostra os esboços de Leonardo da Vinci para construção de um possível autômato projetado para sentar-se, dobrar as pernas e os braços, abrir a boca e provavelmente realizar algum tipo de barulho. Ele era vestido com uma armadura(4). A figura 2.4 é a versão digital do modelo para se certificar de suas habilidades.

Os extensivos estudos de Da Vinci sobre a anatomia humana e seus desenhos em perspectiva, transparentes e detalhados, além de muito úteis para medicina e engenharia, foram inspiradores para o desenvolvimento de arti-

culações mecânicas, indispensáveis ao robôs.

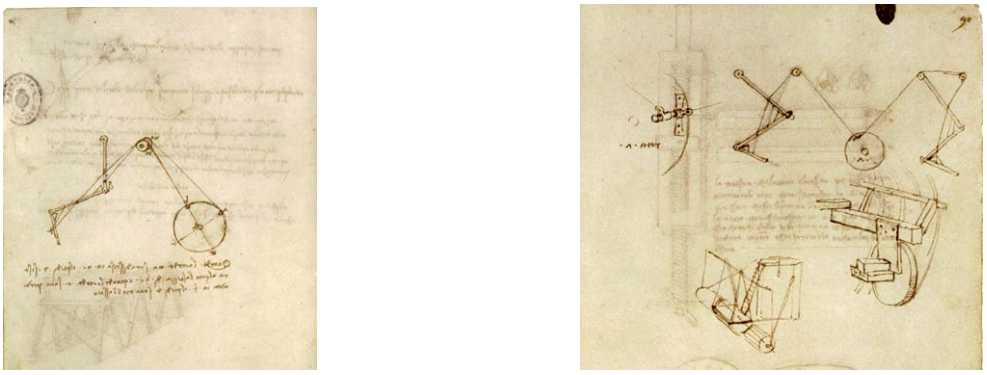

Figura 2.3: Esboços de Leonardo Da Vinci(4)

Já a figura 2.5 mostra o robô *ASIMO* (*Advanced Step in Innovative Mobility*), desenvolvido pela Honda, capaz acender e desligar lâmpadas, abrir e fechar portas, mover objetos, dançar e muitas outras ações(5). É considerado um dos *humanóides* mais desenvolvidos até o momento.

Figura 2.4: Modelo Digital do autômato de Da Vinci(4)

Figura 2.5: ASIMO(5)

2.3 O que é um robô?

De acordo com o R.I.A. (*Robotics Institute of America*), um robô é: "*Um manipulador programável multi-funcional capaz de mover materiais, partes, ferramentas ou dispositivos especializados através de movimentos variáveis*

programados para realizar uma variedade de tarefas."(11) Porém pode-se complementar essa definição, uma vez que ela não faz referência a forma humanóide, que a maioria robôs são dedicados a realizarem tarefas puramente seqüenciais e que alguns podem ter o poder de tomar decisões, adaptando-se a diferentes condições(11).

A palavra *robô* teve origem na peça teatral de 1921 chamada "R.U.R." (*Russum's Universal Robots*) do dramaturgo tcheco Karel Capek. Ela deriva da palavra tcheca *robota*, que significa trabalho forçado, escravo. E a palavra *robótica* foi usada pela primeira vez pelo escritor Isaac Asimov em seu romance *Runaround*, em 1942(10).

2.3.1 Leis da Robótica

Isaac Asimov em seu livro *I, Robot* (Eu, Robô) definiu e popularizou as 3 fundamentais leis da robótica, acrescentando mais tarde a Lei Zero(19):

Lei Zero Um robô não pode causar mal à humanidade ou, por omissão, permitir que a humanidade sofra algum mal, nem permitir que ela própria o faça.

Lei 1 Um robô não pode ferir um ser humano ou, por omissão, permitir que um ser humano sofra algum mal.

Lei 2 Um robô deve obedecer às ordens que lhe sejam dadas por seres humanos, exceto nos casos que em tais ordens contrariem a Primeira Lei.

Lei 3 Um robô deve proteger sua própria existência, desde que tal proteção não entre em conflito com a Primeira e a Segunda Leis.

Recentemente, seu livro Eu, Robô foi adaptado em um filme de mesmo nome e dirigido por Alex Proyas, estreando nos cinemas em agosto de 2004 (20).

2.4 Tipos de Robôs

Conforme o JIRA (*Japanese Industrial Robot Association*), os robôs podem ser divididos em 6 classes(10), de acordo com sua autonomia e flexibilidade:

Classe 1 Dispositivo operado manualmente: um dispositivo com muitos graus de liberdade, manipulado por um operador;

Classe 2 Robô de seqüências fixas: dispositivo operado que executa os sucessivos estágios de uma tarefa conforme um método pré-determinado e invariável, que são difíceis de se modificarem;

Classe 3 Robô de de seqüências variáveis: o mesmo que dispositivo da classe 2, porém os estágios podem ser facilmente modificados;

Classe 4 Robô repetidor: um operador humano executa a tarefa manualmente, conduzindo ou controlando o robô, que grava as trajetórias. Possibilitando posteriormente que ele execute a tarefa de modo automático.

Classe 5 Robô com controle numérico: o operador humano fornece ao robô um conjunto de movimentos que ele deverá executar, ao invés de ensiná-lo;

Classe 6 Robôs inteligentes: um robô capaz de compreender o seu ambiente e possui a habilidade de completar a tarefa, adaptando-se a mudanças de condições.

Além dessas, pode-se incluir ainda mais duas classes, correspondentes aos robôs móveis:

Robôs móveis Robô dotado de dispositivos para movimentação e equipado com sensores e atuadores. Possui movimentos de rotação e translação e responde a um sistema computacional(16).

Robôs móveis autônomos Robô móvel com o poder de se auto-governar. Possui a habilidade de se locomover em um ambiente para a execução de inúmeras tarefas; podendo ser capaz de se adaptar a mudanças, aprender através da experiência, construir uma representação interna do mundo, e mudar seu comportamento dependendo do seu estado(10).

2.5 Sensores

Sensores são dispositivos transdutores que podem medir alguma grandeza física, como luminosidade, temperatura, peso, campo magnético. Eles são os órgãos dos sentidos dos robôs, fornecendo informações necessárias para que ele perceba o mundo a sua volta e possa tomar uma atitude.

Todos os sensores são caracterizados por um grande número de propriedades. As mais importantes são(10)(21):

Sensibilidade Mínimo valor do parâmetro físico que irá produzir uma mudança detectável na saída. Em alguns sensores, a sensibilidade é definida como variação na entrada necessária para produzir uma mudança na saída.

Linearidade Função que descreve a relação entre a entrada e a saída até o limite máximo de medida;

Limites de medida Valores máximos e mínimos que podem ser medidos;

Tempo de resposta Tempo de propagação para que uma alteração detectável na entrada reflita na saída;

Precisão Diferença entre o valor de saída com o valor real;

Resolução Menor valor de mudança detectável na entrada que pode ser detectado na saída.

Existe uma variedade de sensores utilizados nos robôs, que geralmente são escolhidos conforme o ambiente onde irá atuar e a atividade que irá desempenhar. Entre os principais sensores, pode-se citar:

Infravermelho Um dos sensores mais utilizados, devido a sua simplicidade e baixo custo. Geralmente usado para detectar obstáculos. Ele funciona através da transmissão de ondas infravermelhas que refletem ao atingir um objeto e são capturadas por um receptor;

Sonar Sensor inspirado nos morcegos, também possui seu funcionamento baseado na transmissão e recepção de ondas, porém ondas sonoras ultrasônicas, que refletem ao atingir um objeto. Ele possibilita o cálculo da distância do objeto refletido, através tempo decorrido entre transmissão e recepção do sinal;

Laser Funciona de forma semelhante ao sonar, diferenciando a natureza da onda emitida, cujo comprimento de onda é próxima ao infravermelho. Ele permite calcular a distância, velocidade e aceleração de objetos detectados;

Câmera CCD Câmera CCD (*Charge Coupled Device*) permite capturar a representação de uma imagem codificada na forma de uma matriz numérica, onde cada elemento (*pixel*) corresponde a um nível em tons de cinza ou em tons coloridos. A principal dificuldade no uso desse sensor é a complexidade envolvida no processamento da imagem.

Contato É o mais simples dos sensores, que é acionado quando o robô colide em algum obstáculo, provocando o fechamento de uma micro-chave nos modelos mais comuns.

Odômetro Sensor de posicionamento que permite calcular a distância percorrida pelo robô.

2.5.1 Fusão Sensorial

De forma semelhante aos animais que usam diversos tipos de sensores, como tátil, visual, olfativo para reconhecimento de um ambiente, os robôs também podem necessitar para a realização de alguma tarefa. A fusão sensorial descreve o processo de utilizar as informações de diversos sensores para a construção de uma imagem do ambiente ou situação no qual o robô está inserido.

Há várias técnicas para a realização dessa integração, como redes neurais artificiais, lógica nebulosa, filtros de Kalman e fusão bayesiana(16).

2.6 Atuadores

São dispositivos que permitem o robô atuar no ambiente, possibilitando sua locomoção, movimentação de membros, orientação de sensores, manipulação de objetos.

Os mais comuns são os motores elétricos de corrente contínua ou alternada, motores de passo e atuadores pneumáticos ou hidráulicos.

2.7 Robótica Hoje

A utilização de robôs ou dispositivos autônomos em nossa sociedade, vem crescendo aceleradamente desde a Revolução Industrial, substituindo o homem nas tarefas mais perigosas, enfadonhas ou ainda para auxiliar na sua realização.

Pode-se citar inúmeras tarefas que ele pode cumprir: detecção de minas, exploração de ambientes perigosos, como vulcões, regiões aquáticas ou contaminadas com radioatividade, montagem de produtos industriais, tarefas médicas, como cirurgias e diagnósticos. E também em trabalhos domésticos, como aspiradores de pó, limpadores de piscina e cortadores de grama, e que segundo previsões da Comissão Econômica para a Europa das Nações Unida (UNECE) até 2007 sua comercialização terá crescido de aproximadamente

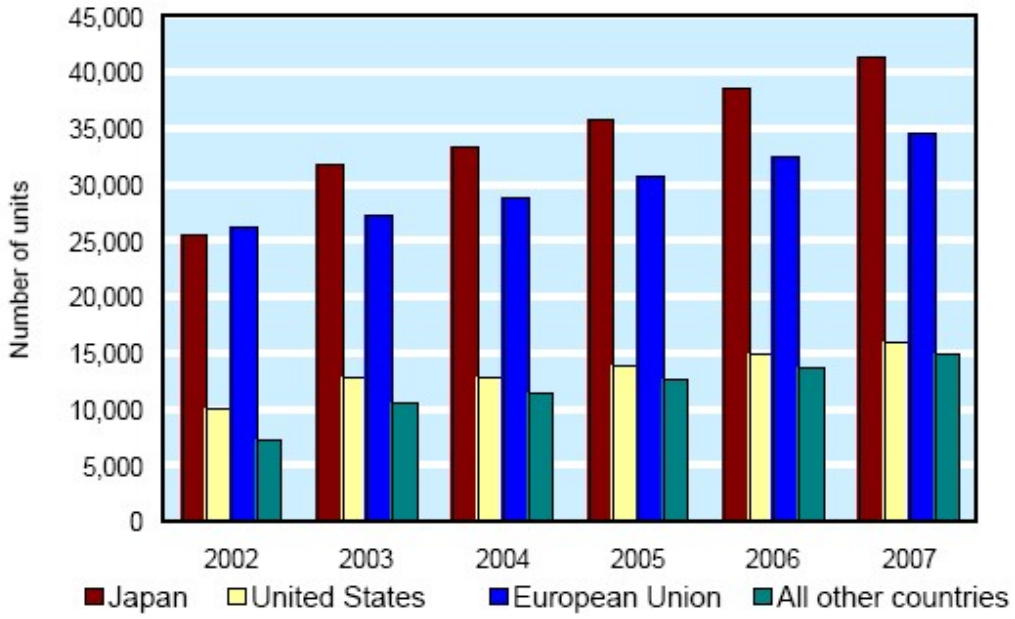

Figura 2.6: Instalações anuais de robôs industriais entre 2003-2004 e previsão para 2004-2007(6)

610 mil para mais de 4 milhões de unidades(6).

O Japão, União Européia e Estados Unidos são os lugares mais robotizados, e como mostra o gráfico 2.6 a tendência é aumentar ainda mais, favorecido pela queda dos preços e aumento de desempenho(6).

Capítulo 3

Sistemas Nebulosos

3.1 Introdução

A teoria dos *conjuntos nebulosos* foi apresentada à comunidade científica em 1965 no artigo intitulado *Fuzzy Sets*, do pesquisador Lotfi A. Zadeh. Desde então tem sido um dos assuntos mais abordados na literatura e aplicados em realizações concretas nos mais variados problemas de controle e automação, sistemas de apoio à decisão, representação do conhecimento, processamento de sinais e sistemas especialistas(22)(11).

No Brasil o termo *Fuzzy Sets* ficou conhecido como *conjuntos nebulosos* ou *conjuntos difusos*, que conforme o professor Jorge M. Barreto(11) a última denominação pode ter surgido a partir da sugestão de um oficial da Aeronáutica que não apreciou o termo nebuloso. Porém Zadeh em uma entrevista fez um comentário interessante sobre o termo: *"... the name produced a certain curiosity: "Fuzzy logic?", people say, "what is this thing?" If I used some blunt name like "continuous logic", people would not notice it"*(23).

3.2 Conjuntos Nebulosos

Na teoria clássica dos Conjuntos, um elemento *pertence* ou não a um determinado conjunto. Por exemplo, dado o conjunto universo $\mathbb{U}$, A um subconjunto

e os elementos a e z:

$$a \in A, \quad z \notin A$$

Uma outra maneira de se representar o conceito de pertinência seria através da definição da *função característica* de um conjunto:

$$\mu_A(x) = \begin{cases} 1 \text{ se } x \in A, \\ 0 \text{ se } x \notin A. \end{cases} \tag{3.1}$$

Assim essa função mapeia os elementos do conjunto universo $\mathbb{U}$ para o conjunto $\{1, 0\}$:

$$\mu_A : \mathbb{U} \to \{0, 1\} \tag{3.2}$$

Um conjunto nebuloso é obtido pela modificação do conjunto de valores da função característica, que ao invés de mapear para o conjunto $\{0, 1\}$, mapearia para um conjunto representado por um intervalo. Geralmente esse intervalo é definido entre $[0, 1]$, como descrito no trabalho de Zadeh(11). Então:

$$\mu_A : \mathbb{U} \to [0, 1] \tag{3.3}$$

A esta nova função que caracteriza o conjuntos nebuloso chama-se *Função de Pertinência*(11), que permite definir o quanto é possível, ou a que *grau de possibilidade* um elemento x pertença ao conjunto nebuloso A.

3.2.1 Exemplos de Conjuntos Nebulosos

No cotidiano costuma-se usar conjuntos nebulosos, muitas vezes sem seu conhecimento, para a representação de conjuntos que não se sabe precisamente, ou seja, com um grau de *imprecisão* os seus limites.

Por exemplo, o conjunto das pessoas jovens. Não existe um definição clara ou precisa a partir de qual idade uma pessoa é considerada jovem. Pode-se considerar que alguém com 15 anos é jovem, como também com 18 anos, porém menos. A medida que a idade aumenta, o valor da função de

pertinência decresce, representando que a pessoa pertence menos ao conjunto das pessoas jovens e mais ao conjuntos das pessoas maduras.

O gráfico abaixo, 3.1, mostra as funções de pertinência para os conjuntos nebulosos jovem, maduro e idoso.

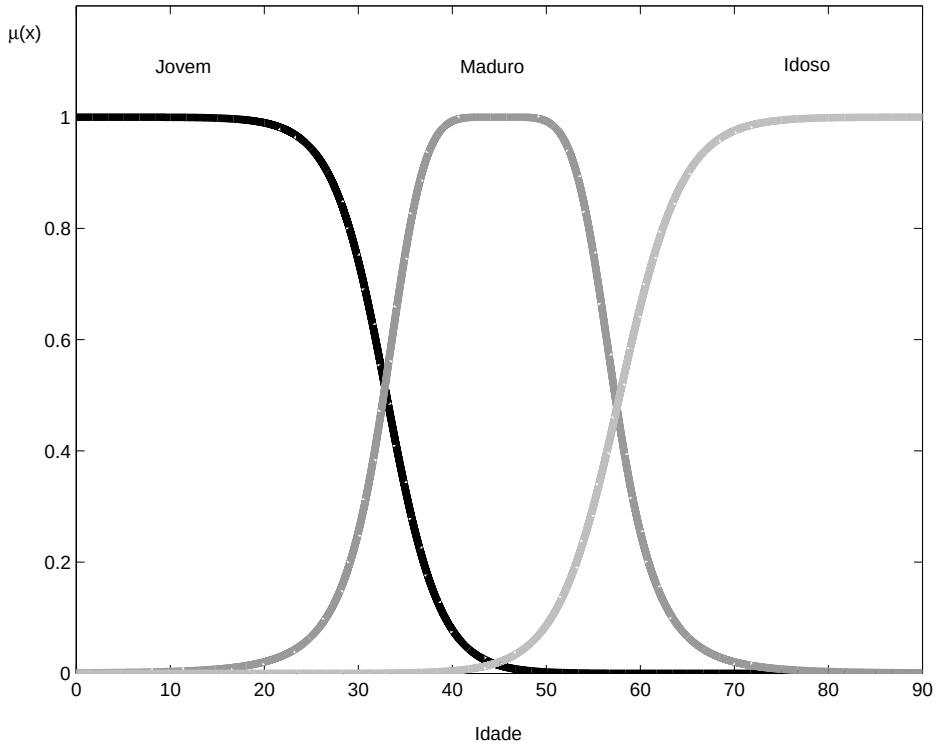

Figura 3.1: Conjuntos nebulosos para idade

E as funções de pertinência para conjuntos jovem, maduro e idoso, são respectivamente:

$$\mu_{A_1}(x) = \frac{1}{1 + e^{-0.3(x-35)}} \tag{3.4}$$

$$\mu_{A_2}(x) = \frac{1}{1 + \left|\frac{x-45}{11}\right|^6} \tag{3.5}$$

$$\mu_{A_3}(x) = \frac{1}{1 + e^{-0.2(x-60)}} \tag{3.6}$$

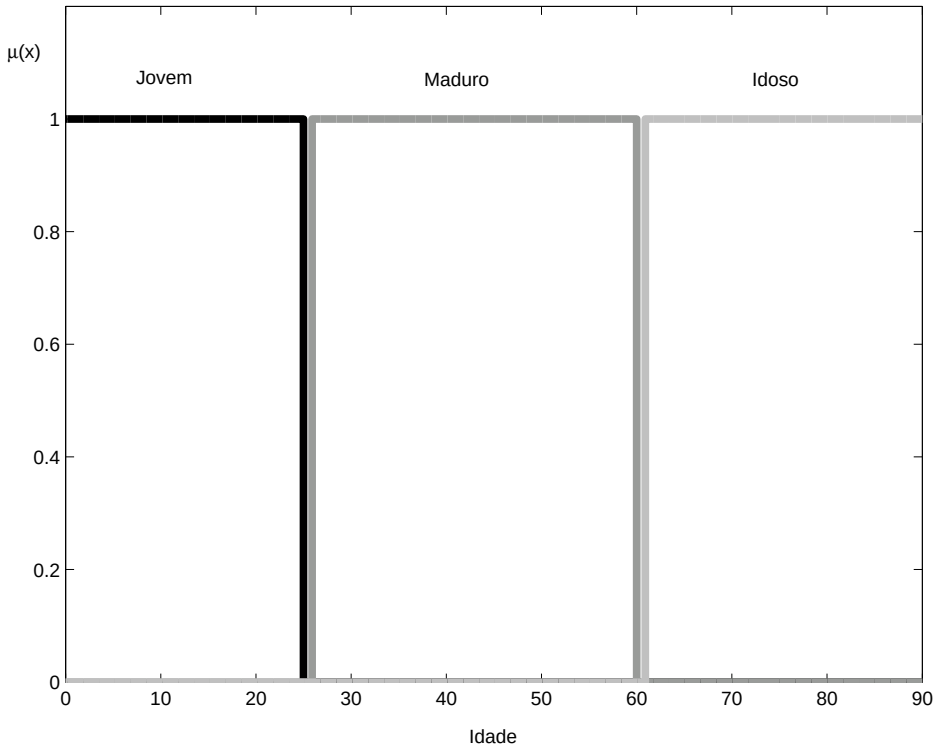

Figura 3.2: Representação tradicional dos conjuntos para idades

Já o gráfico 3.2 é a representação das funções características para conjuntos jovem, maduro e idoso, na tradicional teoria dos conjuntos. Nota-se a variação repentina na delimitação entre eles.

3.2.2 Possibilidade e Probabilidade

É comum a confusão de possibilidade com probabilidade, por possuírem valores similares e significados aparentemente semelhantes.

Enquanto a probabilidade está relacionada com incerteza da ocorrência de um evento no futuro, ou seja, após a ocorrência do evento não há mais incerteza.

Já a possibilidade pode se referir à imprecisão de alguma característica de um evento ocorrido:

A probabilidade se achar um prêmio em uma tampinha de refrigerante é de 1%. Qual a possibilidade dele ser valioso?

Caso se ganhe ou não o prêmio a incerteza está resolvida, porém ainda existe a imprecisão ou dúvida se o prêmio é valioso, barato, muito barato.

3.2.3 Operações com Conjuntos Nebulosos

Todos os operadores dos conjuntos clássicos também são aplicáveis aos conjuntos nebulosos(22). Abaixo mostra-se as operações básicas:

- Igualdade

$$\mu_A(x) = \mu_B(x), \text{ para todo } x \in \mathbb{U} \tag{3.7}$$

- União

$$\mu_{A \cup B}(x) = \max\{\mu_A(x), \mu_B(x)\}, \text{ para todo } x \in \mathbb{U} \tag{3.8}$$

- Intersecção

$$\mu_{A \cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}, \text{ para todo } x \in \mathbb{U} \tag{3.9}$$

- Complemento

$$\mu_{\neg A}(x) = 1 - \mu_A(x), \text{ para todo } x \in \mathbb{U} \tag{3.10}$$

As figuras 3.4, 3.5 e 3.6 ilustram algumas das operações apresentadas acima, tomando como referência os conjuntos A e B que podem ser vistos na figura 3.3.

3.3 Variáveis e Valores Lingüísticos

Variáveis lingüísticas podem ser definidas, informalmente, como variáveis cujos seus valores são palavras ou sentenças na linguagem natural ou artificial ao invés de números(24).

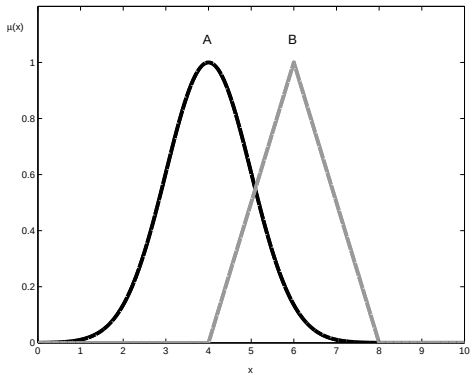

Figura 3.3: Conjuntos nebulosos A e B

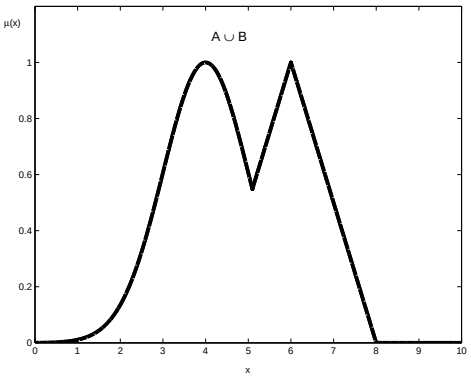

Figura 3.4: Conjunto nebuloso $A \cup B$

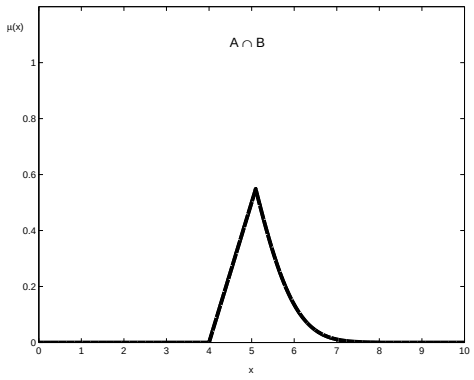

Figura 3.5: Conjunto nebuloso $A \cap B$

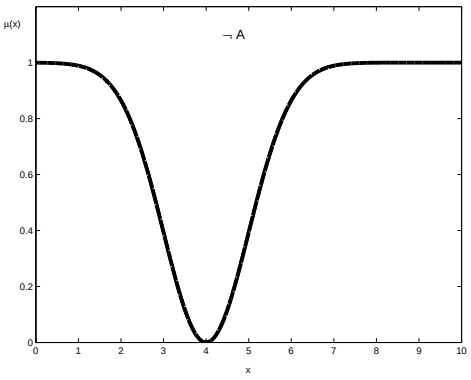

Figura 3.6: Conjunto nebuloso $\neg A$

Um valor lingüístico nomeia um conjunto nebuloso, transmitindo a idéia de qualificação de algum atributo ou característica dessa variável. O conjunto de todos os valores é chamado de *termos lingüísticos*.

Por exemplo a variável lingüística velocidade. Seus valores lingüísticos podem ser: muito baixa, baixa, média, alta e muito alta, como ilustrado na figura 3.7.

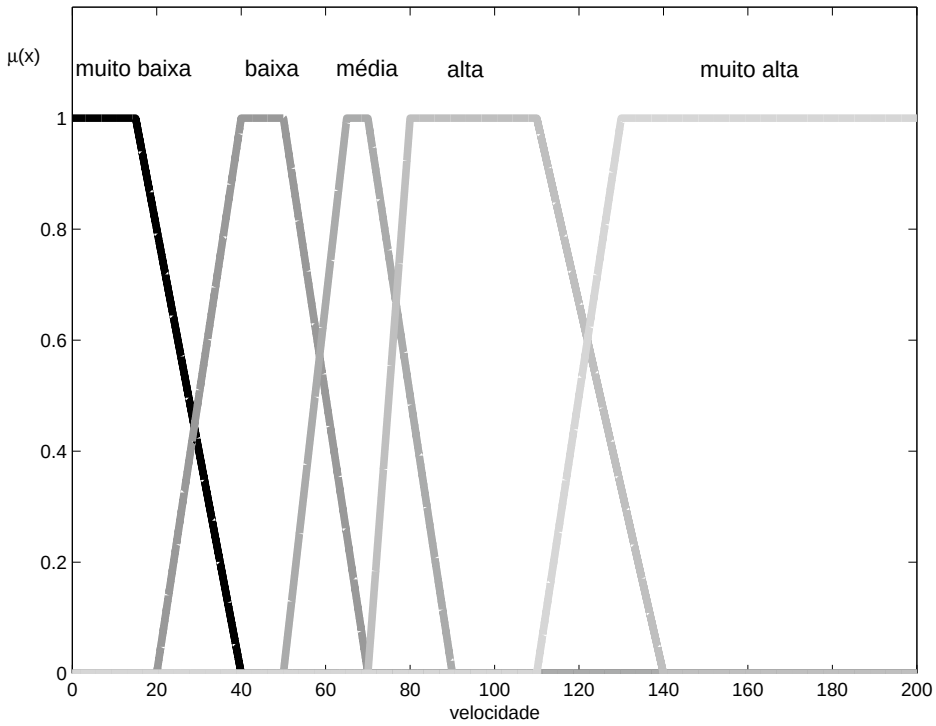

Figura 3.7: Conjuntos de termos para a variável lingüística velocidade

Uma variável lingüística é definida formalmente como uma quíntupla(24):

$$\langle L, T(L), \mathbb{U}, G, M \rangle \,, \quad \text{onde} \tag{3.11}$$

- L é o nome da variável lingüística;
- $T(L)$ é o conjunto de termos de L, isto é, o conjunto de valores lingüísticos;
- $\mathbb{U}$ é o universo de discurso;
- G é a gramática para a geração de termos em $T(L)$;

- M é a regra semântica que associa cada valor lingüístico a um conjunto nebuloso.

3.4 Funções de Pertinência

Geralmente na modelagem dos conjuntos nebulosos usa-se um número restrito de funções. As mais utilizadas são(22):

- Função Triangular: parâmetros (a, b, c), com $a \leq b \leq c$

$$\mu_A(x) = \begin{cases} 0 & \text{se } x \leq a, \\ \frac{x-a}{b-a} & \text{se } a < x \leq b, \\ \frac{c-x}{c-b} & \text{se } b < x \leq c, \\ 0 & \text{se } x > c. \end{cases} \tag{3.12}$$

- Função Trapezoidal: parâmetros (σ, c), com $a \leq b, c \leq d$ e $b < c$

$$\mu_A(x) = \begin{cases} 0 & \text{se } x \leq a, \\ \frac{x-a}{b-a} & \text{se } a < x \leq b, \\ 1 & \text{se } b < x \leq c, \\ \frac{d-x}{d-c} & \text{se } c < x \leq d, \\ 0 & \text{se } x > d. \end{cases} \tag{3.13}$$

- Função Gaussiana: parâmetros (σ, c), com $\sigma > 0$

$$\mu_A(x) = e^{\frac{-(x-c)^2}{2\sigma^2}} \tag{3.14}$$

- Função Sino Generalizada: parâmetros (a, b, c), com $a \neq 0, b > 0$

$$\mu_A(x) = \frac{1}{1 + \left|\frac{x-c}{a}\right|^{2b}} \tag{3.15}$$

- Função Sigmoidal: parâmetros (a, c), com $a \neq 0$

$$\mu_A(x) = \frac{1}{1 + e^{-a(x-c)}} \tag{3.16}$$

- Conjunto *Singleton*: parâmetros (a)

$$\mu_A(x) = \begin{cases} 1 & \text{se } x = a, \\ 0 & \text{se } x \neq a. \end{cases} \tag{3.17}$$

As figuras 3.8, 3.9, 3.10, 3.11, 3.12 e 3.13 são exemplos das funções descritas acima.

A escolha do tipo de função para a representação dos conjuntos devem ser feitas criteriosamente(11), porém nem sempre é fácil, podendo inclusive não estar no alcance do especialista para a aplicação em questão(25). Nestes casos, pode-se recorrer a métodos estatísticos ou técnicas de inteligência artificial como computação evolutiva para auxiliar na determinação das funções e seus parâmetros.

3.5 Sistemas Nebulosos

As primeiras aplicações com sistemas nebulosos surgiram na década de 70, período inicial da Época das Trevas[1] para a inteligência artificial, onde quase todas as pesquisas foram paralisadas por falta de verbas(11).

A primeira aplicação foi desenvolvida por Mamdani e Assilian que implementaram um sistema nebuloso para controle de uma máquina a vapor, publicando em 1974 suas experiências(26), estimulando mais experimentos devido ao sucesso obtido. Um pouco depois, em 1978 foi defendida a primeira dissertação brasileira intitulada *Controle de Processos por Lógica Nebulosa*, por Ricardo Tanscheit e orientada por Jorge M. Barreto. Em 1982 surgiu a primeira aplicação comercial, utilizado em um controlador para um forno de cimento.

Atualmente, sistemas nebulosos são usados em uma infinidade de aplicações em diversas áreas, como citadas na seção 3.1.

Sistemas nebulosos são indicados em problemas onde(25):

[1]Para maiores detalhes sobre a fases históricas da Inteligência Artificial ver (11).

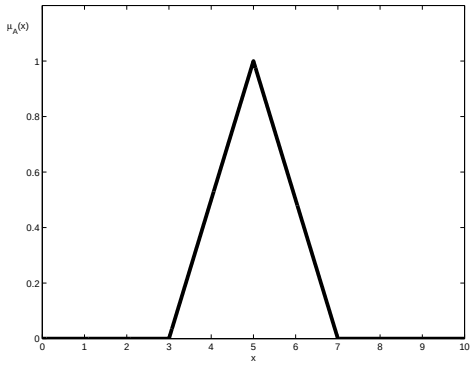

Figura 3.8: Função Triangular

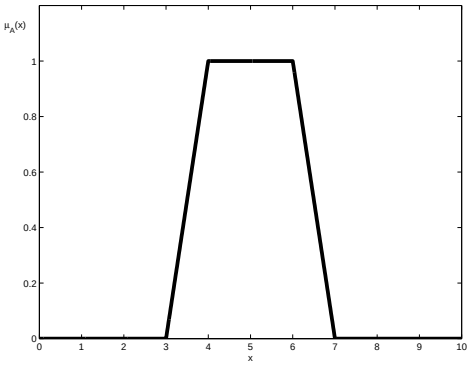

Figura 3.9: Função Trapezoidal

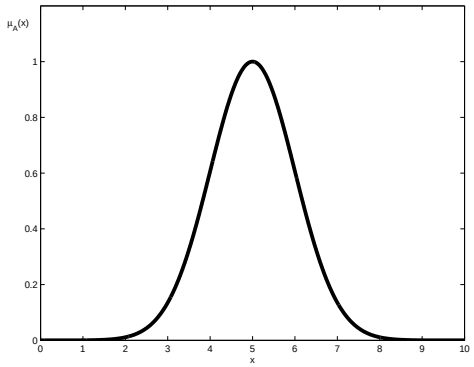

Figura 3.10: Função Gaussiana

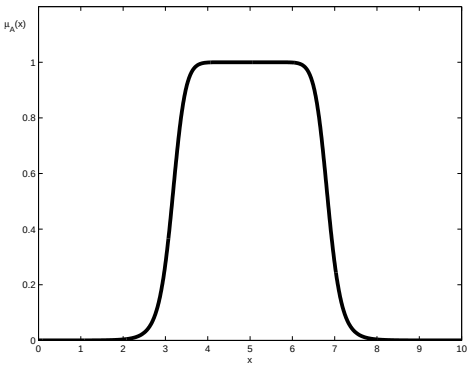

Figura 3.11: Função Sino Generalizada

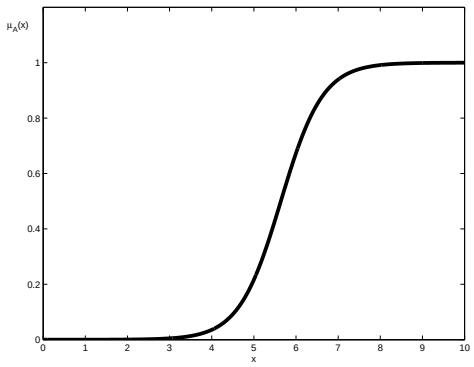

Figura 3.12: Função Sigmoidal

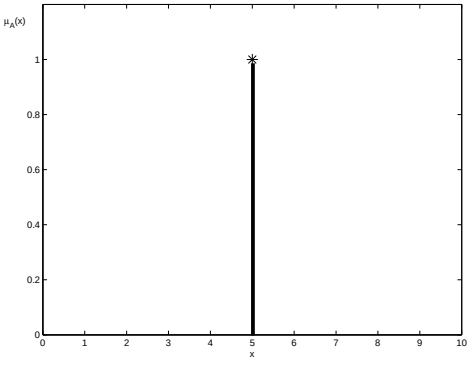

Figura 3.13: Conjunto *Singleton*

- O modelo matemático do sistema não existe ou se existe, é de difícil codificação;
- O modelo matemático é muito complexo para ser avaliado em tempo real, ou necessita de muitos recursos computacionais;
- As variáveis do sistema são contínuas;
- Altos níveis de ruídos afetam o estado dos sensores, ou os sensores são de baixa precisão;
- O processo envolve a interação com um operador humano.

Sistemas nebulosos, também conhecidos como sistemas de inferência nebulosa possuem como estrutura básica 3 componentes conceituais(25):

Base de regras que contém o conjunto de regras nebulosas;

Base de dados que contém os conjuntos nebulosos ou funções de pertinência usadas pelas regras nebulosas;

Motor de inferência que realiza a inferência baseado nas regras nebulosas(raciocínio nebuloso) para obter uma saída ou conclusão.

A figura 3.14 mostra o fluxo das informações em um sistema nebuloso genérico. Inicialmente cada regra é avaliada com base nas entradas, e o resultado de cada regra é combinado para resultar em valor de saída.

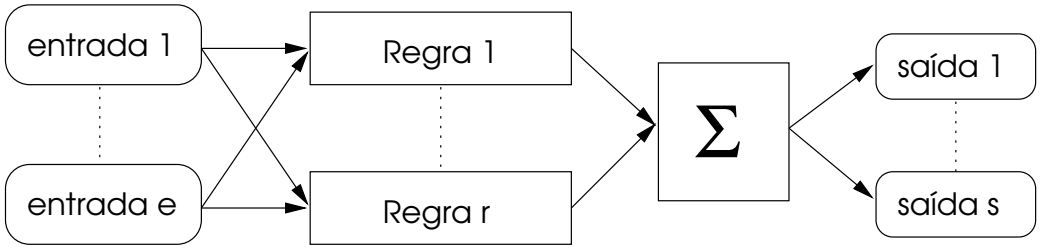

Figura 3.14: Sistema nebuloso genérico

3.5.1 Regras Nebulosas

Regras nebulosas ou declarações condicionais nebulosas são estruturas lógicas que mapeiam um conjunto de entradas do sistema para um conjunto de saídas. Sua elaboração tem origem no conhecimento ou experiência do operador ou especialista humano em relação ao sistema, permitindo elaborar estratégias e diretivas. Elas expressam esse conhecimento por meio de uma forma lingüística.

Elas geralmente são estruturadas da seguinte maneira:

se <*antecedentes*> ***então*** <*conseqüentes*>,

onde os antecedentes exprimem uma condição(premissa) e os conseqüentes a ação ou conclusão a ser executada quando a premissa for verdadeira.

Por exemplo, "**se** a *temperatura é alta* **então** a *velocidade é rápida*", é uma regra nebulosa que relaciona as variáveis lingüísticas temperatura e velocidade com os termos ou conjuntos nebulosos alto e rápido. Assim quando for verdadeira a condição *temperatura alta* o sistema deve tornar a *velocidade rápida*.

3.5.2 Motores de Inferência

Os motores de inferência realizam o processamento das regras nebulosas para obter um conclusão final da ação ou ações que o sistema deve realizar, pois inúmeras regras podem estar ativas no mesmo instante.

Existem vários modelos, que se diferenciam na estrutura dos conseqüentes nas regras nebulosas(25). Os mais conhecidos são(25):

Mamdani que utiliza conjuntos nebulosos também nos conseqüentes das regras nebulosas. Como a saída é nebulosa, resultante da agregação das saídas das regras ativadas, métodos de *defuzificação* são necessários para se obter uma saída final numérica.

Takagi-Sugeno o conseqüente é representado por uma função, geralmente dependente das variáveis de entrada. A saída final é obtida pela média ponderada da saída de cada regra ativada.

3.5.3 Modelo Mamdani

O modelo Mamdani foi utilizado no primeiro sistema nebuloso implementado, como citado na seção 3.5, no controle de uma máquina vapor, onde o conjunto de regras de controle baseadas em variáveis lingüísticas foi obtido de operadores humanos especializados(25).

Nesse modelo, as regras nebulosas assumem o formato:

$$\textbf{\textit{se}}\ x\ \textit{é}\ A\ \textit{conectivo}\ y\ \textit{é}\ B\ \textbf{\textit{então}}\ z\ \textit{é}\ C,$$

onde A, B e C são valores lingüísticos para as variáveis lingüísticas, respectivamente x, y e z. Os conectivos que ligam os antecedentes podem ser **E**(min) ou **OU**(max), operadores definidos na seção 3.2.3. Por exemplo, "**se** a *umidade é alta* **E** a *velocidade é rápida* **então** a *pressão é média*".

O processo de inferência normalmente engloba 4 etapas, nessa ordem:

1. ***Fuzificação*** determinação do grau de pertinência nos conjuntos nebulosos para cada valor de entrada, realizado pelas funções de pertinência que definem cada conjunto nebuloso.

2. **Inferência** para cada regra são aplicados os conectivos que unem cada antecedente, que após a *fuzificação* possuem um grau de pertinência. O resultado é valor numérico, que é aplicado no conseqüente, para indicar seu grau de intensidade, resultando em um conjunto nebuloso. O processo de avaliação de cada regra pode ser executado paralelamente, pois são independentes.

3. **Agregação** como cada regra resulta em um conjunto nebuloso, é necessário combiná-los para resultar em um conjunto único. Geralmente, o operador usado é **OU**(max).

4. ***Defuzificação*** etapa final, onde o conjunto resultante na etapa anterior é processado resultando em um valor de saída numérico.

Todo esse processo pode ser visualizado na ilustração 3.15, que usou como exemplo as seguintes regras:

se *a temperatura é alta* ***E*** *o volume é médio* ***então*** *a velocidade é alta*

se *a temperatura é média* ***E*** *o volume é alto* ***então*** *a velocidade é média*

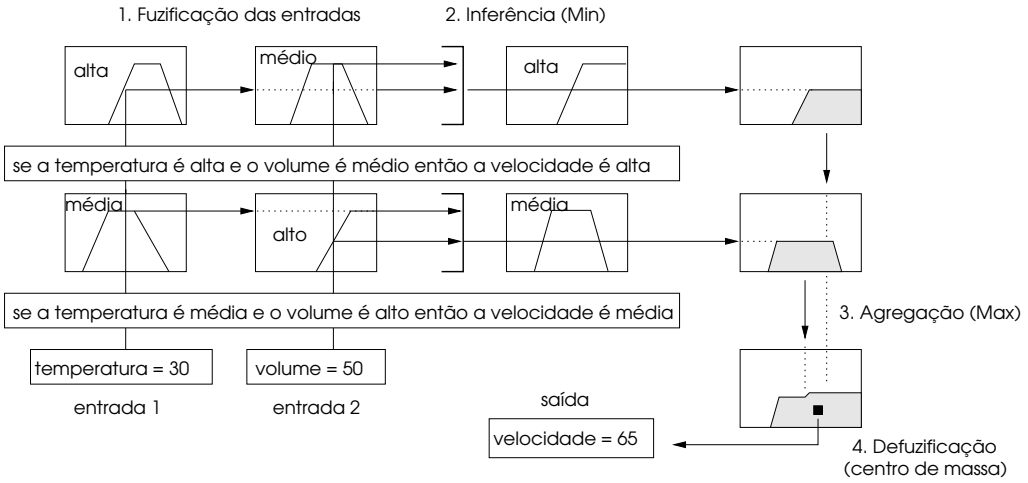

Figura 3.15: Exemplo de um sistema nebuloso Mamdani

Métodos de *Defuzificação*

Existem vários métodos para realizar o processo de *defuzificação*. Os mais utilizados são: centro de massa(gravidade), média dos máximos e centro de área.

As fórmulas 3.18 e 3.19 fornecem o centro de massa(x_C) para funções de pertinência contínuas e discretas, respectivamente. Assim, seja A um conjunto nebuloso definido no universo X, então:

$$x_C = \frac{\int \mu_A(x) x \, dx}{\int \mu_A(x) \, dx} \tag{3.18}$$

$$x_C = \frac{\sum_{i=1}^{n} \mu_A(x_i) x_i}{\sum_{i=1}^{n} \mu_A(x_i)} \tag{3.19}$$

Memórias Associativas Nebulosas

Geralmente um sistema nebuloso é composto por inúmeras regras nebulosas, que podem ser organizadas através de uma matriz nebulosa ou FAM(*fuzzy associative memory)*, onde cada elemento representa uma regra.

Por exemplo, dado as regras nebulosas:

se *o valor é barato* ***E*** *estoque é pouco* ***então*** *o prazo é curto,*
se *o valor é barato* ***E*** *estoque é muito* ***então*** *o prazo é médio,*
se *o valor é caro* ***E*** *estoque é pouco* ***então*** *o prazo é médio,*
se *o valor é caro* ***E*** *estoque é muito* ***então*** *o prazo é longo;*

poderiam ser organizadas na seguinte forma matricial, tabela 3.1, relacionando os antecedentes com o conseqüente:

	pouco	**muito**
barato	curto	médio
caro	médio	longo

Tabela 3.1: Exemplo de uma memória associativa nebulosa

3.6 Sistemas Nebulosos Hierárquicos

Em sistemas nebulosos tradicionais o aumento do número de entradas resultam num aumento exponencial do número de regras. Se um sistema possui n entradas e para cada entrada são definidos m conjuntos nebulosos, então o número total de regras necessário para a implementação de um sistema nebuloso completo será m^n. Por exemplo, suponha-se que o sistema tem 3 entradas e cada entrada 4 conjuntos nebulosos, então o número de regras será

$4^3 = 64$. Agora, se o número de entradas aumentar para 5, resultará em um número de regras: $4^5 = 1024$, um aumento muito considerável que poderia inviabilizar a implementação do sistema(27)(7).

Para tornar esse problema tratável, sistema nebulosos hierárquicos foram propostos, possibilitando um aumento linear do número de regras com incremento do número de entradas(7). A figura 3.16 mostra a estrutura do sistema tradicional e a figura 3.17, estruturas para o sistema hierárquico. Nota-se que o sistema hierárquico é composto por sistemas menores, de dimensões reduzidas.

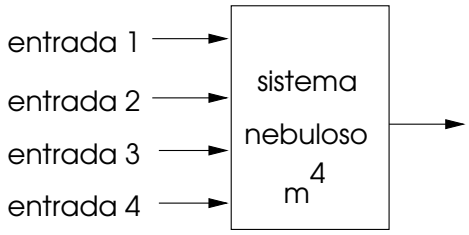

Figura 3.16: Sistema nebuloso tradicional(7)

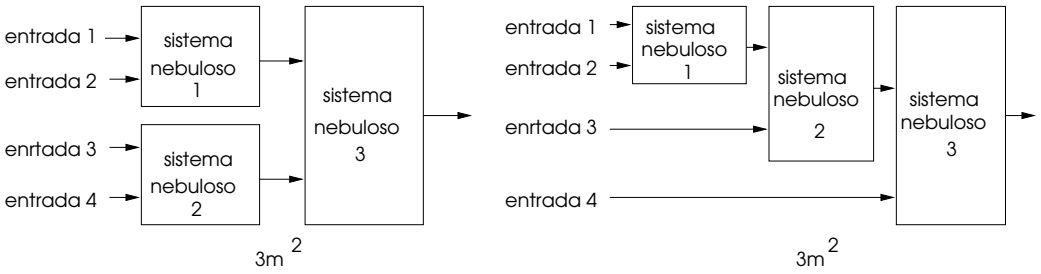

Figura 3.17: Sistema nebuloso hierárquico(7)

Entretanto, nesse sistema as saídas intermediárias não possuem muito significado físico, dificultando a modelagem(7). E quanto maior for o número de camadas, maior será a dificuldade e as perdas do significado(7). Modelos foram propostos para amenizar esses problemas(28)(29). Como no modelo apresentado em (28), as saídas da camada anterior e as entradas da próxima camada são mapeadas em uma variável intermediária, facilitando a implementação e mantendo a estrutura hierárquica. Outro modelo, usado em (7),

foi combinar as variáveis do sistema que se relacionam mais intimamente em um sistema nebuloso distinto, assim sua saída possuía um significado compreensível, para a modelagem da próxima camada.

3.7 Sistemas Nebulosos Hierárquicos e Robôs Móveis Autônomos

Além das estratégias citadas para a implementação de sistemas nebulosos hierárquicos, o modelo chamado *Context-dependent Blending* tem sido usado com sucesso na implementação de sistemas de controle nebulosos para robôs móveis baseados em comportamentos.

Como dito no capítulo 1, seção 1.2, a abordagem reativa deve responder a inúmeras questões para tornar sua implementação viável, robusta e eficiente. O modelo *Context-dependent Blending*(CDB) proposto por Ruspini, Saffiotti e outros(8), procura responder a esses questionamentos utilizando sistemas nebulosos hierárquicos.

Nesse modelo são definidos:

Comportamentos unidade básica nesse modelo. Cada comportamento é modelado como um sistema nebuloso independente, podendo inclusive ser composto por outros comportamentos de menor hierarquia, emergindo um comportamento complexo;

Política de arbitragem de comportamentos que através do uso de meta-regras nebulosas agrupadas em um único módulo, decidem quais comportamentos devem ser ativados em um dado momento;

Fusão de Comportamentos que realiza a junção dos comportamentos ativados e os decodifica em comandos numéricos para serem enviados aos atuadores do robô. E também permite uma mudança ou chaveamento suave entre os comportamentos.

Na figura 3.18 tem-se uma visão geral do modelo CDB.

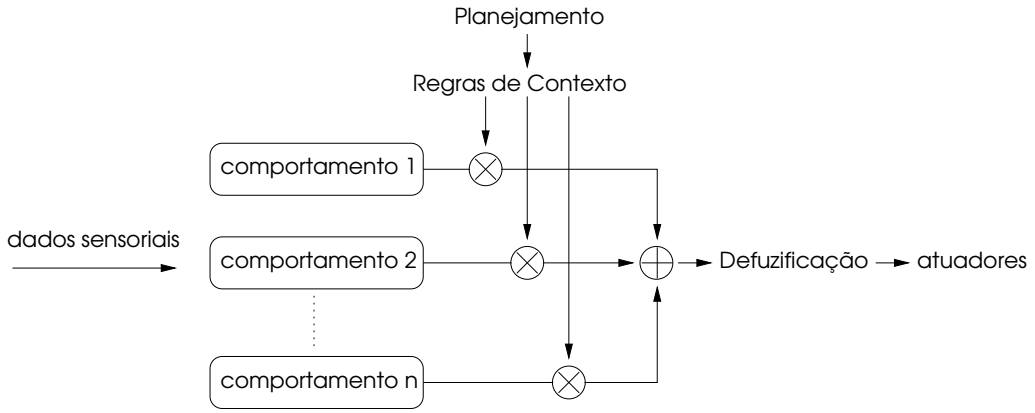

Figura 3.18: Modelo *Context-dependent Blending*(8)

A modelagem dos comportamentos robóticos simples como desviar de obstáculos, seguir parede e outros são modelados na forma tradicional dos sistemas nebulosos. Como por exemplo, essa regra poderia compor o sistema nebuloso para desvio de obstáculo:

se *obstáculo_direita é perto* ***E*** *obstáculo_frente é muito_perto* ***então*** *virar é muito_esquerda*,

onde *obstáculo_direita* e *obstáculo_frente* referem-se às leituras dos sensores e *virar* aos atuadores que mudarão a direção do robô.

A política de arbitragem é composta por meta-regras nebulosas que assumem o seguinte formato:

se <*contexto*> ***então*** <*comportamento*>

Por exemplo, as seguintes meta-regras podem decidir que comportamentos devem ser ativados e dependendo do contexto, fazer o robô seguir a luz e desviar de obstáculos quando encontrar:

se *obstáculo* é perto **então** *Evitar_obstáculo*,

se *obstáculo* é longe **então** *Seguir_a_luz*,

Porém, haverá momentos em que o obstáculo estará parcialmente perto, tornando ambos os comportamento ativos, concorrentemente.

Para a fusão dos comportamentos ativados, que têm *graus de desejabilidade*, resultantes das meta-regras nebulosas, é realizada a agregação dos conjuntos nebulosos originados da inferência de cada comportamento, e posteriormente sua *defuzificação*, como visto na figura 3.18.

O *grau de desejabilidade* indica, como o nome sugere, o quanto é desejado que um comportamento esteja ou seja ativo. A aplicação do grau de desejabilidade permite que em uma dada situação um comportamento tenha mais prioridade ou atuação, sem ignorar os outros. Ele é equivalente a variável intermediária entre camadas descrita em (28) e comentada na seção 3.6. Por exemplo, dado uma meta-regra e uma regra do comportamento ativado por essa meta-regra, respectivamente:

se *luz é forte* ***então*** *Seguir_parede*

se *obstáculo_direita é perto* ***então*** *virar é pouco_esquerda*,

seria equivalente a transformar a regra em:

se *luz é forte* ***E*** *obstáculo_direita é perto* ***então*** *virar é pouco_esquerda.*

Capítulo 4

Implementação

4.1 Introdução

Para a realização dos objetivos que pretende-se atingir, descritos no capítulo 1, implementou-se um sistema completo para controle de robôs móveis autônomos utilizando a abordagem reativa sob o modelo nebuloso hierárquico *Context-dependent Blending*, discutido no capítulo anterior.

A implementação foi baseada nas características, como tipos de sensores e atuadores, do robô Khepera e simulada no *WSU Khepera Robot Simulator* versão 7.2.

4.2 O simulador Khepera

O Khepera é um pequeno robô móvel que possui 2 rodas com 20 níveis para controle de velocidade em cada uma, 10 níveis para avanço e 10 níveis para recuo. Oito sensores de proximidade(sensores infravermelhos) e oito sensores de luminosidade que estão disposto ao seu redor, porém concentrados mais na sua parte frontal(pequenos retângulos), como vistos na figura 4.2.

No simulador é possível construir "mundos", como o ilustrado na figura 4.3, compostos por paredes translúcidas e adicionar no ambiente pequenas luminárias e bolas, tornando-o mais interessante para o robô.

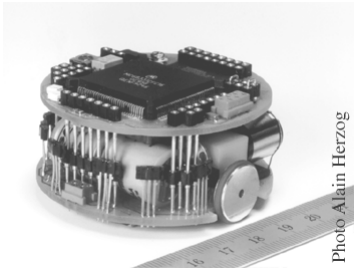

Figura 4.1: Robô real Khepera

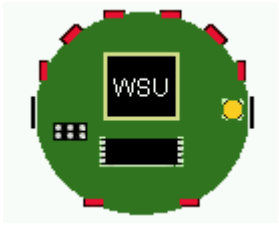

Figura 4.2: Robô do simulador Khepera

A qualidade dos experimentos realizados no simulador tem sido comprovada em testes com o robô real(30).

O *WSU Khepera Robot Simulator* é um simulador gráfico de domínio público e implementado na linguagem Java. E está disponível para *download* juntamente com seu código-fonte e documentação na página: `http://ehrg.-cs.wright.edu/ksim/ksim/ksim.html`

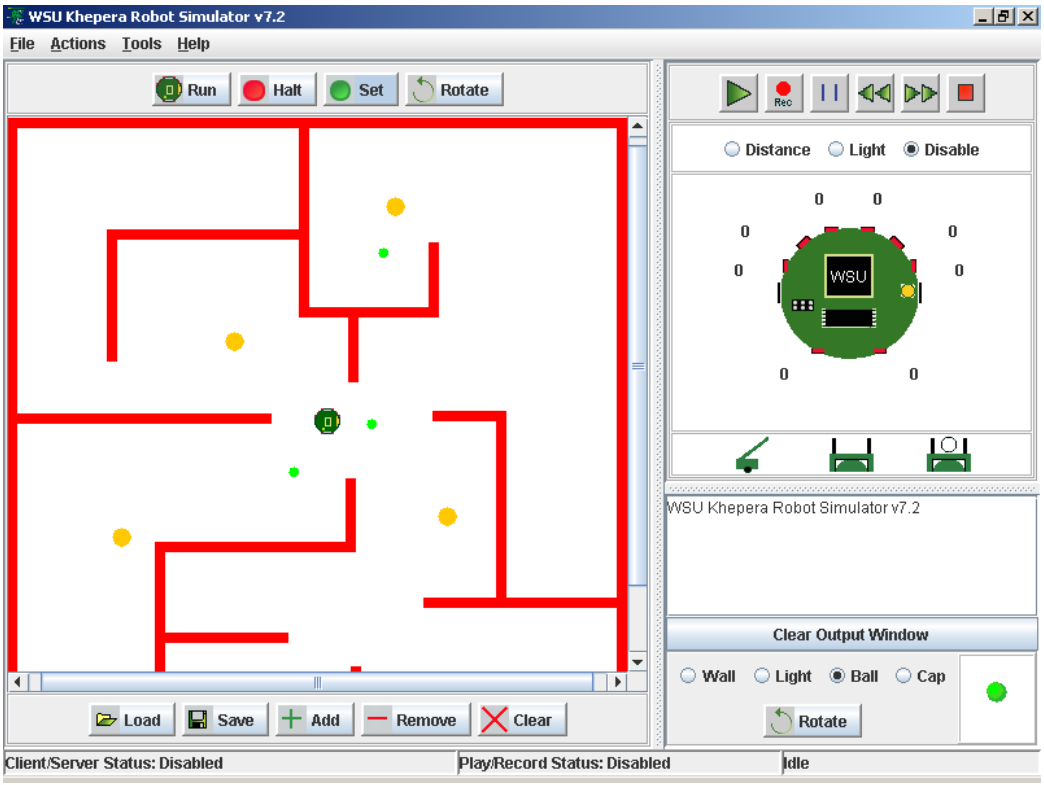

Figura 4.3: Simulador WSU Khepera

4.3 Arquitetura do Sistema

Para avaliar a modularidade e expansibilidade do modelo *Context-dependent Blending*, os comportamentos foram implementados e testados independentemente e posteriormente unidos através das regras de contexto(meta-regras nebulosas) esperando emergir um comportamento complexo, não apenas reativo.

Buscou-se selecionar comportamentos que possibilitassem autonomia e objetivos claros ao robô. Assim, os comportamentos básicos selecionados foram: evitar colisão, seguir parede, procurar por luz e morrer. Esse comportamentos, de menor hierarquia, foram gerenciados pelo comportamento de maior hierarquia, simbolizado pelas meta-regras. A figura 4.4 mostra a arquitetura do sistema implementado.

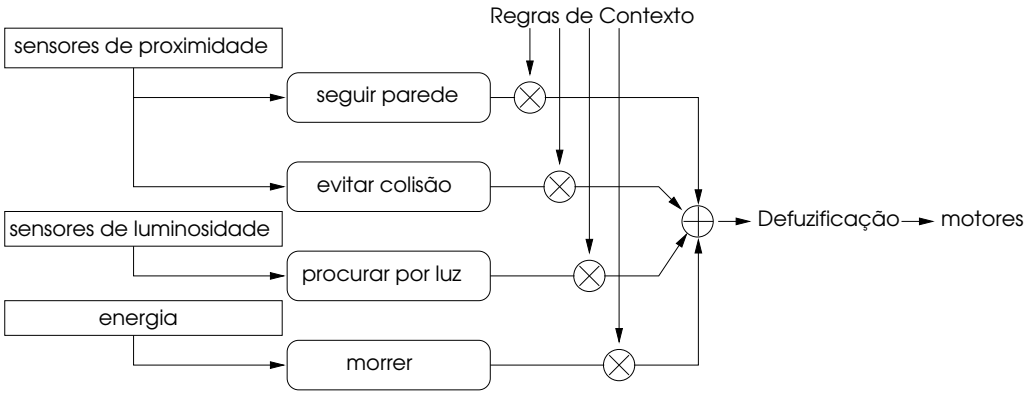

Figura 4.4: Sistema Hierárquico Implementado

Como todo sistema real precisa de energia para funcionar ou sobreviver, no robô também foi implementado um estoque de energia, que é consumida conforme a velocidade que ele se movimenta e reabastecida quando ele pára próximo a um ponto de luz.

A seguir, apresenta-se as meta-regras(regras de contexto), que fornecerão uma compreensão de como o sistema atua e uma descrição dos comportamentos básicos.

4.3.1 Meta-Regras para a Navegação

Como dito na seção 3.7 do capítulo 3, as meta-regras definem um política de arbitragem que permitem decidir o contexto em que os comportamentos deverão ser ativados.

O algoritmo abaixo, procura fornecer um auxílio de como as meta-regras coordenam os comportamentos, lembrando-se que o sistema não está limitado a atuação de um único comportamento, podendo inclusive que todos ou quase todos estejam ativados.

- **se** *energia* é (forte ou normal) **então** comportamento é *seguir parede* e estado é consumindo;
- **se** *energia* é fraca **então** comportamento é *procurar fonte de luz* e estado é consumindo;
- **se** *obstáculo* é próximo **então** comportamento é *evitar obstáculo* e estado é consumindo;
- **se** *luz* é próxima **então** comportamento é *procurar fonte de luz* e estado é carregando;
- **se** *energia* é morto **então** comportamento é *morrer* e estado é consumindo.

Assim, enquanto robô estiver com nível estável de energia ele ficará seguindo a parede, e quando sua energia atingir o nível fraco, consequência do consumo dos motores, ele irá procurar por luz. E ao encontrar, irá se recarregar até ficar no nível máximo, voltando novamente a seguir a parede. Enquanto o robô se recarrega, ele desliga os motores, para não se afastar da fonte de energia. Se durante todo esse processo, algum obstáculo estiver no seu caminho, ele desviará.

No total foram implementadas 32 meta-regras, no formato:

se *luz é ____* ***E*** *proximidade é ____* ***E*** *energia é ____* ***E*** *estado é ____* ***então***
comportamento é ____ e estado é ____

O conjunto de termos T, de cada variável lingüística é:

- T(luz) = {fraca, forte}
- T(proximidade) = {perto, longe}
- T(energia) = {morto, fraco, normal, forte}
- T(estado) = {consumindo, carregando}

Comportamentos não são variáveis lingüística, mas é uma variável que pode assumir os valores que correspondem a cada comportamento, que são {evitar colisão, seguir parede, procurar por luz, morrer}.

4.3.2 Comportamento Evitar Colisão

O comportamento vagar e evitar colisão é indispensável para que o robô explore o ambiente sem colidir com algum obstáculo. Ele se baseia na leituras de 2 sensores de proximidade, que são escolhidos dinamicamente conforme o seu valor. Para cada lado, direito e esquerdo, é escolhido 1 entre os 3 sensores frontais, figura 4.2. Serão selecionados aqueles que indicarem maior valor de proximidade, para um melhor controle.

Esse comportamento é bastante simples, o robô anda em linha reta na velocidade máxima enquanto não encontra nenhum obstáculo, e se encontrar, desvia para o lado oposto do obstáculo.

No total foram implementadas 9 regras nebulosas, no formato:

se *proximidade_esquerda é ____* ***E*** *proximidade_direita é ____* ***então***
velocidade_direita é ____ e velocidade_esquerda é ____ e consumo é ____

O conjunto de termos T, de cada variável lingüística é:

- T(proximidade_esquerda) = {longe, perto, muito perto}
- T(proximidade_direita) = {longe, perto, muito perto}
- T(velocidade_direita) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}
- T(velocidade_esquerda) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}
- T(consumo) = {muito, pouco, nada, reabastecimento pouco, reabastecimento muito, reabastecimento bastante}

4.3.3 Comportamento Procurar por Luz

É outro comportamento também bastante simples. Enquanto o robô não encontrar nenhum foco de luz ele continua seguindo sempre em frente e ao encontrar o menor vestígio, irá em sua direção. Ao chegar próximo o bastante da fonte de luz ele irá desligar os motores para o reabastecimento.

No total foram implementadas 27 regras nebulosas, no formato:

se *luz_esquerda é ____* ***E*** *luz_direita é ____* ***E*** *luz_frente é ____* ***então*** *velocidade_direita é ____ e velocidade_esquerda é ____ e consumo é ____*

O conjunto de termos T, de cada variável lingüística é:

- T(luz_esquerda) = {muita, média, pouca}
- T(luz_direita) = {muita, média, pouca}
- T(luz_frente) = {muita, média, pouca}
- T(velocidade_direita) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}
- T(velocidade_esquerda) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}

- T(consumo) = {muito, pouco, nada, reabastecimento pouco, reabastecimento muito, reabastecimento bastante}

4.3.4 Comportamento Seguir Parede

Nesse comportamento o robô deve manter um pequena distância em relação a parede, procurando segui-la em um dos seus lados, direito ou esquerdo. Nesta implementação, optou-se pelo lado direito, fazendo o robô contornar a parede no sentido anti-horário. Caso o robô não encontre nenhuma parede, ele seguirá continuamente reto.

É interessante observar que para o robô não existe uma definição clara do que é uma parede ou um obstáculo, para ele qualquer coisa que os seus sensores de proximidade acusarem, ele concluirá que é uma parede.

No total foram implementadas 16 regras nebulosas, no formato:

se *proximidade_esquerda é ____* ***E*** *proximidade_direita é ____* ***então*** *velocidade_direita é ____ e velocidade_esquerda é ____ e consumo é ____*

O conjunto de termos T, de cada variável lingüística é:

- T(proximidade_esquerda) = {longe, pouco longe, perto, muito perto}
- T(proximidade_direita) = {longe, pouco longe, perto, muito perto}
- T(velocidade_direita) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}
- T(velocidade_esquerda) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}
- T(consumo) = {muito, pouco, nada, reabastecimento pouco, reabastecimento muito, reabastecimento bastante}

4.3.5 Comportamento Morrer

Esse comportamento é apenas simbólico, apesar de fato ser usado. Ele é útil somente no simulador, para indicar que a energia do robô acabou ele nada pode fazer a não ser ficar parado, denotando sua morte por falta de energia.

No total foram implementadas 4 regras nebulosas, no formato:

se *energia é ____* ***então*** *velocidade_direita é ____ e velocidade_esquerda é ____ e consumo é ____*

O conjunto de termos T, de cada variável lingüística é:

- T(energia) = {morto, fraco, normal, forte}
- T(velocidade_direita) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}
- T(velocidade_esquerda) = {negativa alta, negativa média, negativa baixa, parada, positiva baixa, positiva média, positiva alta}
- T(consumo) = {muito, pouco, nada, reabastecimento pouco, reabastecimento muito, reabastecimento bastante}

4.4 Testes e Resultados

Durante o desenvolvimento de cada comportamento e do sistema final, o robô era testado em diferentes situações, como ambiente com labirintos, com diversas fontes luzes e diversos obstáculos. Esses testes além de certificar o sistema de suas habilidades, também eram úteis para afinamentos nas regras dos comportamentos ou na definição dos conjuntos nebulosos, pois havia situações em que a sensibilidade do robô ou o comportamento não era o "esperado".

Por exemplo, a situação de indecisão que o robô teve que resolver enquanto estava seguindo a parede, mostrado na figura 4.5. Nesse momento o

robô podia tomar duas decisões: virar para esquerda ou para direita. Na primeira modelagem das regras do comportamento seguir parede, o robô sempre desviava para a esquerda, porém havia espaço para ele dobrar para a direita, e continuar seguindo a mesma parede. Apesar de não estar errada a sua decisão, isso simboliza que a definição dos conjuntos nebulosos para os sensores de proximidade precisava de mais níveis de proximidade, para ele saber o quão perto estava.

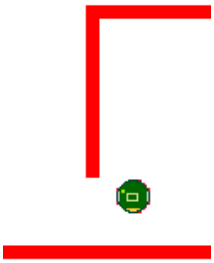

Figura 4.5: Situação de indecisão

Outro fato inusitado foi a *emergência* de um comportamento curioso, algo parecido com o *egoísmo*. O robô foi colocado em um ambiente delimitado por paredes, onde havia apenas uma fonte luz. Enquanto sua energia estava normal ele ficou seguindo a parede, porém ao ficar fraca ele foi em busca da fonte de energia, como era o esperado. Ao encontrar, se recarregou e ficou *eternamente* circulando-a. Isso aconteceu, pois fontes de luz são objetos detectados pelos sensores de proximidade, ele possivelmente achou que fosse uma parede.

Em diversos ambientes, o robô realizou com sucesso seus objetivos, sendo perceptível em alguns casos, quando mais de um comportamento estava atuando, uma certa indecisão na predominância de um comportamento. Por exemplo, durante a transição do comportamento seguir parede para o procurar por luz, ela fazia ambos ao mesmo tempo, o que era esperado.

4.5 Diagrama de Classe

Como o simulador foi implementado na linguagem Java, o código do sistema hierárquico descrito também foi em Java.

Durante o processo de projeto e implementação do sistema procurou-se economizar o máximo os recursos de memória e processamento, para que o sistema fosse portável para robôs com recursos muito limitados, como o *kit* robótico *Lego MindStorms*.

Recentemente foi implementado e disponibilizado para a comunidade uma micro máquina virtual Java para o Lego, chamada *Lejos*. Como os recursos computacionais são bastante escassos no Lego, o Lejos não implementa o sistema de coleta de lixo *(garbage collection)*, ou seja, uma vez criado um objeto, ele sempre permanecerá na memória.

No sistema implementado, todos os objetos necessários para execução do controlador são criados no início do sistema e sempre re-usados, para que não seja necessário criar novos objetos e manter outros sem uso na memória.

A figura 4.6 mostra o diagrama simplificado das classes do sistema e os principais relacionamentos.

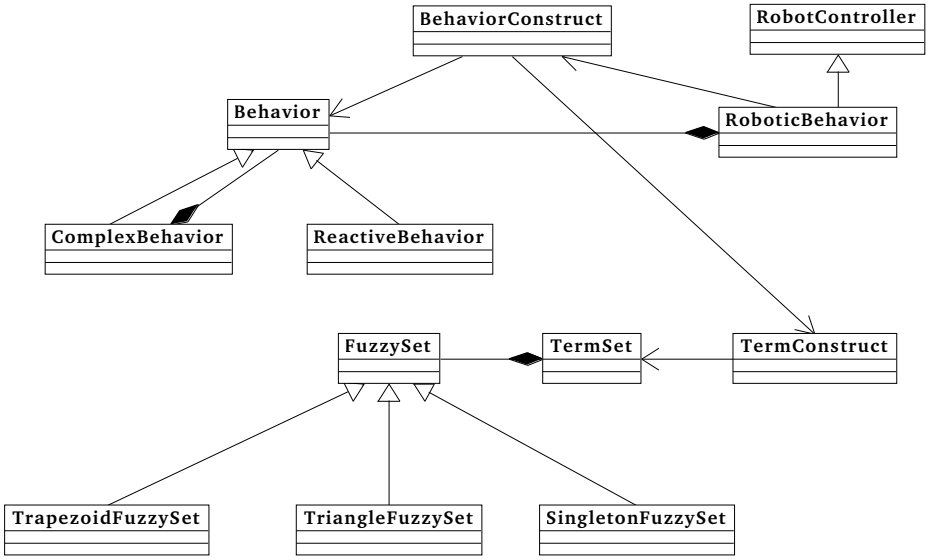

Figura 4.6: Diagrama de Classe do Sistema Hierárquico

Capítulo 5

Conclusão

Modelos hierárquicos nebulosos mostraram-se eficientes para modelagem de sistemas de controle complexos, como o modelo *Context-dependent Blending*, para implementação de comportamentos em robôs móveis autônomos.

O modelo *Context-dependent Blending* conseguiu satisfazer razoavelmente quase todos questionamentos discutidos na seção 1.2. Ele permite que um sistema seja facilmente paralelizável, pois cada comportamento pode ser avaliado de forma independente, antes do processo final de *defuzificação*. Fornece modelos claros para a modelagem de estratégias de controle e planejamento.

O sistema é expansível, permitindo adicionar novos comportamentos sem alterar muito o que já foi desenvolvido, com exceção das meta-regras que precisam abranger os novos comportamentos.

É robusto, caso algum bloco da hierarquia entre em colapso, não significa necessariamente o colapso total de sistema.

O sistema é tolerante a diferentes níveis de ruídos que podem afetar os sensores, sem prejudicar o planejamento; característica inerente de sistemas nebulosos.

Finalmente, a emergência de comportamentos complexos a partir de simples comportamentos, permite aos olhos do observador, concluir que o sistema é *inteligente*.

5.0.1 Sugestões para Trabalhos Futuros

Como o sistema foi modelado e implementado visando robôs com baixo recursos computacionais, ele poderia ser testado nos robôs do *kit Lego MindStorms*. Para tanto, seria necessário modificar os conjuntos nebulosos especificados paro os sensores do Khepera, para corresponderam aos dos sensores do Lego.

Implementar novos comportamentos, utilizando mapas topológicos, como abordados em (8).

Paralelizar os comportamentos e avaliar o seu desempenho.

Referências Bibliográficas

1 BROOKS, Rodney A. *A Robust Layered Control System For a Mobile Robot*. Massachusetts Institute of Technology, 1985.

2 GREAT Moments in Science: every child should know. Disponível em: <http://www.usgennet.org/usa/topic/preservation/science/moments/-chpt5.htm>. Acesso em: 1 jul. 2004.

3 JONES, S. *Automata*. Disponível em: <http://www.culture.com.au/-brain_proj/CONTENT/NETS_02.HTM>. Acesso em: 1 jul. 2004.

4 ROBOT and Adding Machine. Disponível em: <http://www.stoke5399-.freeserve.co.uk/leo/machines-robot.htm>. Acesso em: 1 jul. 2004.

5 HONDA. *ASIMO*. Disponível em: <http://world.honda.com/ASIMO/>. Acesso em: 1 jul. 2004.

6 UNECE. *WORLD ROBOTICS 2004*. Geneva, Switzerland, Out. 2004. Disponível em: <http://www.unece.org/press/pr2004/04stat_p01e.pdf>. Acesso em: 6 nov. 2004.

7 GENTILE, Michela. *Development of a Hierarchical Fuzzy Model for the evaluation of Inherent Safety*. Tese (Doctor of Philosophy), Universidad de las Americas, Puebla, Mexico, ago. 2004.

8 SAFFIOTTI, A. The uses of fuzzy logic for autonomous robot navigation: a catalogue raisonné. *Soft Computing Research journal*, v. 1, n. 4, p.

180–197, 1997. Disponível em: <citeseer.ist.psu.edu/saffiotti97use.html>. Acesso em: 1 nov. 2004.

9 SAFFIOTTI, A. Fuzzy logic in autonomous robotics: behavior coordination. In: *Procs. of the 6th IEEE Int. Conf. on Fuzzy Systems*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1997. p. 573–578. Disponível em: <citeseer.ist.psu.edu/saffiotti97fuzzy.html>. Acesso em: 1 nov. 2004.

10 NEHMZOW, Ulrich. *Mobile Robotics: A Practical Introduction*. London: Springer-Verlag, 2000.

11 BARRETO, Jorge Muniz. *Inteligência Artificial no Limiar do Século XXI*. 3. ed. Florianópolis, SC: Duplic, 2001.

12 ROISENBERG, Mauro. *Emergência da inteligência em agentes autônomos através de modelos inspirados na natureza*. Tese (Doutorado em Engenharia Elétrica) — Curso de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis, 1998.

13 NEARCHOU, Andreas C. Adaptive navigation of autonomous vehicles using evolutionary algorithms. *Artificial Intelligence in Engineering*, v. 13, n. 2, p. 159–173, abr. 1999.

14 BEHRING, C. et al. An algorithm for robot path planning with cellular automata. In: *Proceedings of the Fourth International Conference on Cellular Automata for Research and Industry*. [S.l.]: Springer-Verlag, 2000. p. 11–19. ISBN 1-85233-388-X.

15 ROMERO, Roseli Aparecida Francelin; MARQUES, Eduardo. *Aprendizado em Robôs Móveis via Software e Hardware: ARMOSH*. [S.l.].

16 MARCHI, Jerusa. *Navegação de robôs móveis autônomos: estudo e implementação de abordagens*. Dissertação (Mestrado) — Curso de

Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis, 2001.

17 THE New Encyclopædia Britannica: micropædia. Chicago: Encyclopædia Britannica, 1993.

18 BBC. *Historic Figures: Ctesibius*. Disponível em: <http://www.bbc.co.uk/history/historic_figures/ctesibius.shtml>. Acesso em: 6 nov. 2004.

19 ZEROTH Law of Robotics. Disponível em: <http://en.wikipedia.org/-wiki/Zeroth_Law_of_Robotics>. Acesso em: 6 nov. 2004.

20 PROYAS, Alex. *I, Robot*. 2004. Disponível em: <http://www.irobotmovie.com/>. Acesso em: 6 nov. 2004.

21 CARR, Joseph J.; BROWN, John M. *Sensor Terminology*. Disponível em: <http://zone.ni.com/devzone/conceptd.nsf/webmain/-D4BCE3593A5263B78625681100506DE1>. Acesso em: 6 nov. 2004.

22 JUNIOR, Paulo Roberto Crestani. *Sistemas Inteligentes de Navegação Autônoma: Uma Abordagem Modular e Hierárquica Com Novos Mecanismos de Memória e Aprendizagem*. Dissertação (Mestrado em Engenharia Elétrica) — Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, São Paulo, Dezembro 2001.

23 FREKSA, Christian; KRUSE, Rudolf; MáNTARAS, Ramon López de. *Interview with Prof. Lotfi A. Zadeh*. Disponível em: <http://www.kuenstliche-intelligenz.de/Artikel/-InterviewwithProfLotfiAZadeh.htm>. Acesso em: 8 nov. 2004.

24 ZADEH, L. A. The concept of a linguistic variable and its application to approximate reasoning-i. *Information Sciences*, v. 8, n. 3, p. 199–249, 1975. Abstract.

25 DELGADO, Myriam Regattieri De Biase da Silva. *Projeto Automático de Sistemas Nebulosos: Uma Abordagem Co-Evolutiva*. Tese (Doutorado em Engenharia Elétrica) — Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, SP, fev. 2002.

26 MAMDANI, E. H.; ASSILIAN, S. An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Human-Computer Studies*, v. 51, n. 2, p. 135–147, ago. 1999. Special issue: 1969-1999, the 30th anniversary.

27 MARAM, Satish. *Hierarchical Fuzzy Control of the UPFC and SVC located in AEP's Inez Area*. Dissertação (Master of Sciences in Electrical Engineering) — Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Virginia, USA, mai. 2003.

28 LEE, Ming-Ling; CHUNG, Hung-Yuan; YU, Fang-Ming. Modeling of hierarchical fuzzy systems. *Fuzzy Sets and Systems*, v. 138, n. 2, p. 343–361, set. 2003.

29 MURESAN, Leila. *Interpolation in Hierarchical Fuzzy Rule Bases*. [S.l.], 2001.

30 OLIVEIRA, Luís Otávio de Lacerda. *Mapas auto-organizáveis de Kohonen aplicados ao mapeamento de ambientes de robótica móvel*. Dissertação (Mestrado em Ciências da Computação) — Curso de Pós-Graduação em Ciências da Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, out. 2001.

Anexo A - Artigo
Sistemas Nebulosos Hierárquicos para Implementação de Comportamentos em Robôs Móveis Autônomos

Alexandre Vidal Riso e Mauro Roisenberg

Departamento de Informática e Estatística - INE
Universidade Federal de Santa Catarina - UFSC
Florianópolis, SC, Brasil
vidal@inf.ufsc.br, mauro@inf.ufsc.br

Novembro de 2004

Resumo

A navegação de robôs móveis autônomos em ambientes reais não estruturados é uma tarefa desafiadora, pois o ambiente é caracterizado por um grande número de incertezas, como presença de objetos que se movimentam, lugares bloqueados, superfícies escorregadias. Aliado às restrições sensoriais e computacionais dos robôs, o projeto de um sistema de navegação pode ser inviabilizado diante de tantas complexidades e restrições. Assim, este trabalho avalia e relata a implementação de um sistema nebuloso hierárquico aplicado ao problema da navegação. Modelos de sistemas nebulosos hierárquicos, tendo como base módulos comportamentais, apresentaram grande robustez e simplicidade na busca de soluções para esse problema.

***Palavras-chave**: sistemas nebulosos hierárquicos, robótica, comportamentos robóticos, inteligência artificial.*

1 Introdução

A navegação pode ser considerada como uma das mais importantes capacidades, indispensável a um robô móvel autônomo, pois será através dela que ele obterá seus objetivos; permitindo evitar colisões, atingir um local específico, evitar situações perigosas[1].

A quantidade de incertezas que caracterizam um mundo real e outros problemas podem dificultar muito a navegação: dados com ruídos e imprecisos providos dos sensores, dificuldades para a interpretação dessas informações, limitação do campo de visão, problemas de interação do robô com o ambiente, como escorregamento das rodas, bloqueio dos movimentos ou danos no robô causados por obstáculos não percebidos[2][3].

Diante desse desafio, foram desenvolvidas basicamente 3 abordagens para o tratamento desse problema[4]:

- Abordagem Planejada ou Deliberativa;
- Abordagem Reativa;
- Paradigma Cognitivo ou Híbrido.

A abordagem reativa foi um marco para a navegação robótica, gerando grandes avanços na construção de robôs *inteligentes*.

Rodney Brooks foi pioneiro nesse tipo de abordagem[5]. Ele elaborou uma arquitetura chamada de *Subsumption (Subsunção)*, onde o sistema de controle do robô é decomposto em camadas verticais e concorrentes, sendo cada camada responsável por uma *competência* que especifica um comportamento, como visto na figura 1[6].

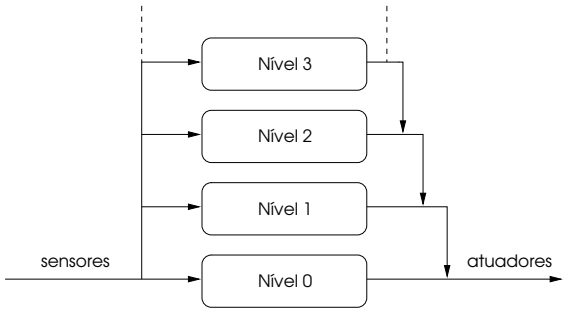

Figura 1: Relação entre as camadas de comportamentos na *Arquitetura de Subsunção*[5]

Uma característica interessante na arquitetura de Subsunção é a *emergência de comportamentos* originadas da interação entre as camadas de ação simples e o ambiente[7].

Essa arquitetura possui inúmeras vantagens que se destacam em relação à abordagem planejada:

- Tempo de resposta rápido aos estímulos, tornando-a adequada ao desenvolvimento de sistemas de controle em tempo-real;
- As camadas são independentes, possibilitando sua reutilização em outros projetos baseados nessa arquitetura;

- Falha em qualquer uma das camadas não causa necessariamente o colapso total do sistema [7];
- Possibilita processamento paralelo, devido à independência entre as camadas;
- Utiliza a estratégia *dividir para conquistar*, reduzindo a complexidade da modelagem[3];
- Comportamentos mais complexos podem emergir da interação de comportamentos mais simples.

Entretanto, no modelo mais puro desse sistema não há nenhum objetivo, plano ou modelo do mundo explícito: *"ele simplesmente reage à situação que tem em mãos"*[7].

Aprimoramentos nesse e nos outros modelos estão advindo de estudos nas áreas de neurociência, psicologia, fisiologia, buscando compreender mais detalhadamente as capacidades instintivas, reativas e cognitivas de animais para serem aplicados e se obter melhores resultados[6][8][7].

1.1 Motivação

Sistemas de navegação baseados na abordagem reativa, principalmente na arquitetura de subsunção, devem responder as seguintes questões durante sua fase de modelagem[3]:

- Como projetar módulos comportamentais robustos e eficientes;
- Como combinar inúmeros comportamentos, podendo ser em algum momento contraditórios;
- Como usar as informações provindas dos sensores, geralmente imprecisas e ruidosas na modelagem e execução dos módulos;
- Como integrar os processos e representações que pertencem a diferentes camadas.

Além desses questionamentos, praticamente qualquer sistema de controle robótico deve possuir características como[5]:

Múltiplos Objetivos Robôs móveis, ao contrário de robôs fixos, que normalmente realizam tarefas repetitivas, podem possuir vários objetivos, como evitar obstáculos, utilizar o mínimo consumo de energia na execução de uma tarefa, atingir um ponto específico;

Múltiplos Sensores Para a realização dos seus objetivos, um robô pode precisar de vários sensores, como sensores infravermelhos, câmera de vídeo, ultra-som;

Robustez Deve suportar às imprecisões de sensores e surpresas que o mundo real pode oferecer;

Expansibilidade Pode necessitar que mais sensores, recursos computacionais, atuadores sejam adicionados para que ele execute outras tarefas, ou que satisfaça novas restrições;

Reusabilidade Necessidade de projetar novos objetivos ou até novos robôs reutilizando o que já foi feito;

Baixo Acoplamento Módulos responsáveis por determinada tarefa podem ser reutilizados, portanto é adequado que ele tenha pouca dependência de outras partes do sistema.

Por isso, o projeto de controladores robóticos não é uma atividade trivial, e necessita de modelos conceituais claros para sua implementação[8], senão esta tarefa pode se tornar inviável diante das complexidades do mundo real e das limitações sensoriais, computacionais e de atuação de um robô.

Neste trabalho foi implementado e avaliado um sistema de controle para robôs móveis com baixos recursos computacionais e sensoriais utilizando sistemas nebulosos hierárquicos, conhecidos como *Context-dependent Blending*.

2 Sistemas Nebulosos

A primeira aplicação com sistemas nebulosos foi desenvolvida por Mamdani e Assilian que implementaram um sistema nebuloso para controle de uma máquina a vapor, publicando em 1974 suas experiências[9], estimulando mais experimentos devido ao sucesso obtido. Um pouco depois, em 1978 foi defendida a primeira dissertação brasileira intitulada *Controle de Processos por Lógica Nebulosa*, por Ricardo Tanscheit e orientada por Jorge M. Barreto. Em 1982 surgiu a primeira aplicação comercial, utilizado em um controlador para um forno de cimento.

Atualmente, sistemas nebulosos são usados em uma infinidade de problemas em diversas áreas, como problemas de controle e automação, sistemas de apoio à decisão, representação do conhecimento, processamento de sinais e sistemas especialistas[10][8].

Sistemas nebulosos são indicados em problemas onde[11]:

- O modelo matemático do sistema não existe ou se existe, é de difícil codificação;
- O modelo matemático é muito complexo para ser avaliado em tempo real, ou necessita de muitos recursos computacionais;
- As variáveis do sistema são contínuas;
- Altos níveis de ruídos afetam o estado dos sensores, ou os sensores são de baixa precisão;
- O processo envolve a interação com um operador humano.

Sistemas nebulosos, também conhecidos como sistemas de inferência nebulosa possuem como estrutura básica 3 componentes conceituais[11]:

Base de regras que contém o conjunto de regras nebulosas;

Base de dados que contém os conjuntos nebulosos ou funções de pertinência usadas pelas regras nebulosas;

Motor de inferência que realiza a inferência baseado nas regras nebulosas(raciocínio nebuloso) para obter uma saída ou conclusão.

A figura 2 mostra o fluxo das informações em um sistema nebuloso genérico. Inicialmente cada regra é avaliada com base nas entradas, e o resultado de cada regra é combinado para resultar em valor de saída.

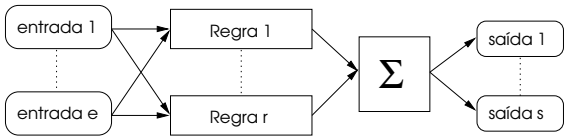

Figura 2: Sistema nebuloso genérico

2.1 Sistemas Nebulosos Hierárquicos

Em sistemas nebulosos tradicionais o aumento do número de entradas resultam num aumento exponencial do número de regras. Se um sistema possui n entradas e para cada entrada são definidos m conjuntos nebulosos, então o número total de regras necessário para a implementação de um sistema nebuloso completo será m^n. Por exemplo, suponha-se que o sistema tem 3 entradas e cada entrada 4 conjuntos nebulosos, então o número de regras será $4^3 = 64$. Agora, se o número de entradas aumentar para 5, resultará em um número de regras: $4^5 = 1024$, um aumento muito considerável que poderia inviabilizar a implementação do sistema[12][13].

Para tornar esse problema tratável, sistema nebulosos hierárquicos foram propostos, possibilitando um aumento linear do número de regras com incremento do número de entradas[13]. A figura 3 mostra a estrutura do sistema tradicional e a figura 4, estruturas para o sistema hierárquico. Nota-se que o sistema hierárquico é composto por sistemas menores, de dimensões reduzidas.

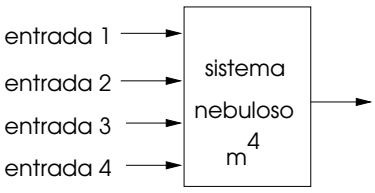

Figura 3: Sistema nebuloso tradicional[13]

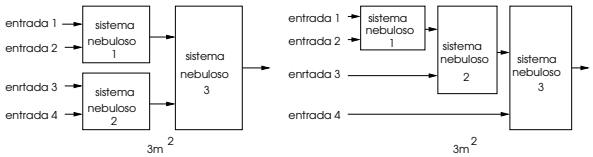

Figura 4: Sistema nebuloso hierárquico[13]

Entretanto, nesse sistema as saídas intermediárias não possuem muito significado físico, dificultando a modelagem[13]. E quanto maior for o número de camadas, maior será a dificuldade e as perdas do significado[13]. Modelos foram propostos para amenizar esses problemas[14][15]. Como no modelo apresentado em [14], as saídas da camada anterior e as entradas da próxima camada são mapeadas em uma variável intermediária, facilitando a implementação e mantendo a estrutura hierárquica. Outro modelo, usado em [13], foi combinar as variáveis do sistema que se relacionam mais intimamente em um sistema nebuloso distinto, assim sua saída possuía um significado compreensível, para a modelagem da próxima camada.

2.1.1 Sistemas Nebulosos Hierárquicos e Robôs Móveis Autônomos

Além das estratégias citadas para a implementação de sistemas nebulosos hierárquicos, o modelo chamado *Context-dependent Blending* tem sido usado com sucesso na implementação de sistemas de controle nebulosos para robôs móveis baseados em comportamentos.

Como dito no capítulo 1, seção 1.1, a abordagem reativa deve responder a inúmeras questões para tornar sua implementação viável, robusta e eficiente. O modelo *Context-dependent Blending*(CDB) proposto por Ruspini, Saffiotti e outros[3], procura responder a esses questionamentos utilizando sistemas nebulosos hierárquicos.

Nesse modelo são definidos:

Comportamentos unidade básica nesse modelo. Cada comportamento é modelado como um sistema nebuloso independente, podendo inclusive ser composto por outros comportamentos de menor hierarquia, emergindo um comportamento complexo;

Política de arbitragem de comportamentos que através do uso de meta-regras nebulosas agrupadas em um único módulo, decidem quais comportamentos devem ser ativados em um dado momento;

Fusão de Comportamentos que realiza a junção dos comportamentos ativados e os decodifica em comandos numéricos para serem enviados aos atuadores do robô. E também permite uma mudança ou chaveamento suave entre os comportamentos.

Na figura 5 tem-se uma visão geral do modelo CDB. A modelagem dos comportamentos robóticos sim-

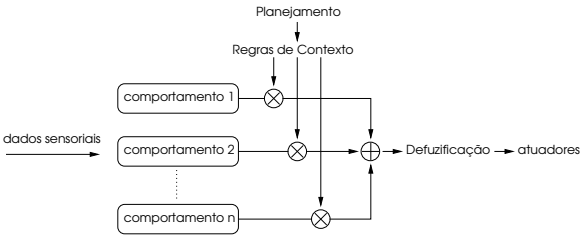

Figura 5: Modelo *Context-dependent Blending*[3]

ples como desviar de obstáculos, seguir parede e outros são modelados na forma tradicional dos sistemas nebulosos. Como por exemplo, essa regra poderia compor o sistema nebuloso para desvio de obstáculo:

se *obstáculo_direita é perto* ***E*** *obstáculo_frente é muito_perto* ***então*** *virar é muito_esquerda*,

onde *obstáculo_direita* e *obstáculo_frente* referem-se às leituras dos sensores e *virar* aos atuadores que mudarão a direção do robô.

A política de arbitragem é composta por meta-regras nebulosas que assumem o seguinte formato:

se <*contexto*> ***então*** <*comportamento*>

Por exemplo, as seguintes meta-regras podem decidir que comportamentos devem ser ativados e dependendo do contexto, fazer o robô seguir a luz e desviar de obstáculos quando encontrar:

se *obstáculo* é perto **então** *Evitar_obstáculo*,
se *obstáculo* é longe **então** *Seguir_a_luz*,

Porém, haverá momentos em que o obstáculo estará parcialmente perto, tornando ambos os comportamento ativos, concorrentemente.

Para a fusão dos comportamentos ativados, que têm *graus de desejabilidade*, resultantes das meta-regras nebulosas, é realizada a agregação dos conjuntos nebulosos originados da inferência de cada comportamento, e posteriormente sua *defuzificação*, como visto na figura 5.

O *grau de desejabilidade* indica, como o nome sugere, o quanto é desejado que um comportamento esteja ou seja ativo. A aplicação do grau de desejabilidade permite que em uma dada situação um comportamento tenha mais prioridade ou atuação, sem ignorar os outros. Ele é equivalente a variável intermediária entre camadas descrita em [14] e comentada na seção 2.1. Por exemplo, dado uma meta-regra e uma regra do comportamento ativado por essa meta-regra, respectivamente:

se *luz é forte* ***então*** *Seguir_parede*

se *obstáculo_direita é perto* ***então*** *virar é pouco_esquerda*,

seria equivalente a transformar a regra em:

se *luz é forte* ***E*** *obstáculo_direita é perto* ***então*** *virar é pouco_esquerda.*

3 Implementação

Para a realização dos objetivos que pretende-se atingir, descritos na Introdução, implementou-se um sistema completo para controle de robôs móveis autônomos utilizando a abordagem reativa sob o modelo nebuloso hierárquico *Context-dependent Blending*.

A implementação foi baseada nas características, como tipos de sensores e atuadores, do robô Khepera e simulada no *WSU Khepera Robot Simulator* versão 7.2.

3.1 O simulador Khepera

O Khepera é um pequeno robô móvel que possui 2 rodas com 20 níveis para controle de velocidade em cada uma, 10 níveis para avanço e 10 níveis para recuo. Oito sensores de proximidade(sensores infravermelhos) e oito sensores de luminosidade que estão disposto ao seu redor, porém concentrados mais na sua parte frontal(pequenos retângulos), como vistos na figura 7.

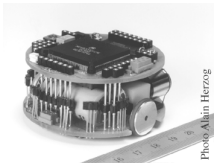

Figura 6: Robô real Khepera

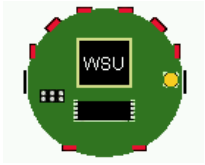

Figura 7: Robô do simulador Khepera

No simulador é possível construir "mundos", como o ilustrado na figura 8, compostos por paredes translúcidas e adicionar no ambiente pequenas luminárias e bolas, tornando-o mais interessante para o robô.

A qualidade dos experimentos realizados no simulador tem sido comprovada em testes com o robô real[16].

O *WSU Khepera Robot Simulator* é um simulador gráfico de domínio público e implementado na linguagem Java. E está disponível para *download* juntamente com seu código-fonte e documentação na página: `http://ehrg.cs.wright.edu/ksim/-ksim/ksim.html`

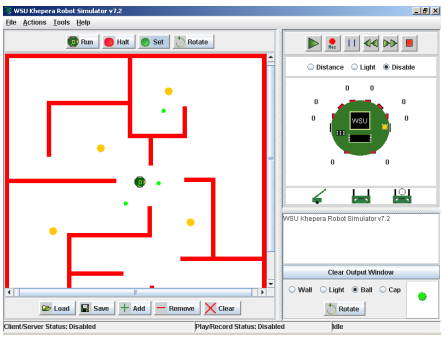

Figura 8: Simulador WSU Khepera

3.2 Arquitetura do Sistema

Para avaliar a modularidade e expansibilidade do modelo *Context-dependent Blending*, os comportamentos foram implementados e testados independentemente e posteriormente unidos através das regras de contexto(meta-regras nebulosas) esperando emergir um comportamento complexo, não apenas reativo.

Buscou-se selecionar comportamentos que possibilitassem autonomia e objetivos claros ao robô. Assim, os comportamentos básicos selecionados foram: evitar colisão, seguir parede, procurar por luz e morrer. Esse comportamentos, de menor hierarquia, foram gerenciados pelo comportamento de maior hierarquia, simbolizado pelas meta-regras. A figura 9 mostra a arquitetura do sistema implementado.

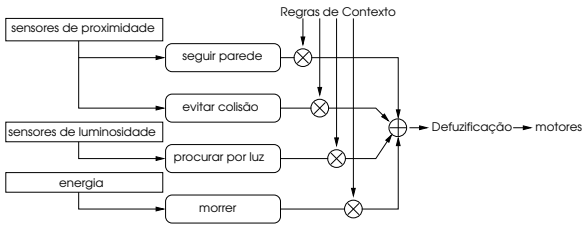

Figura 9: Sistema Hierárquico Implementado

Como todo sistema real precisa de energia para funcionar ou sobreviver, no robô também foi implementado um estoque de energia, que é consumida conforme a velocidade que ele se movimenta e reabastecida quando ele pára próximo a um ponto de luz.

A seguir, apresenta-se as meta-regras(regras de contexto), que fornecerão uma compreensão de como o sistema atua e uma descrição dos comportamentos básicos.

3.2.1 Meta-Regras para a Navegação

Como dito na seção 2.1.1, as meta-regras definem um política de arbitragem que permitem decidir o contexto em que os comportamentos deverão ser ativados.

O algoritmo abaixo, procura fornecer um auxílio de como as meta-regras coordenam os comportamentos, lembrando-se que o sistema não está limitado a atuação de um único comportamento, podendo inclusive que todos ou quase todos estejam ativados.

- **se** *energia* é (forte ou normal) **então** comportamento é *seguir parede* e estado é consumindo;
- **se** *energia* é fraca **então** comportamento é *procurar fonte de luz* e estado é consumindo;
- **se** *obstáculo* é próximo **então** comportamento é *evitar obstáculo* e estado é consumindo;
- **se** *luz* é próxima **então** comportamento é *procurar fonte de luz* e estado é carregando;
- **se** *energia* é morto **então** comportamento é *morrer* e estado é consumindo.

Assim, enquanto robô estiver com nível estável de energia ele ficará seguindo a parede, e quando sua energia atingir o nível fraco, consequência do consumo dos motores, ele irá procurar por luz. E ao encontrar, irá se recarregar até ficar no nível máximo, voltando novamente a seguir a parede. Enquanto o robô se recarrega, ele desliga os motores, para não se afastar da fonte de energia. Se durante todo esse processo, algum obstáculo estiver no seu caminho, ele desviará.

No total foram implementadas 32 meta-regras, no formato:

se *luz é* ____ ***E*** *proximidade é* ____ ***E*** *energia é* ____ ***E*** *estado é* ____ ***então*** *comportamento é* ____ *e estado é* ____

O conjunto de termos T, de cada variável lingüística é:

- T(luz) = {fraca, forte}
- T(proximidade) = {perto, longe}
- T(energia) = {morto, fraco, normal, forte}
- T(estado) = {consumindo, carregando}

Comportamentos não são variáveis lingüística, mas é uma variável que pode assumir os valores que correspondem a cada comportamento, que são {evitar colisão, seguir parede, procurar por luz, morrer}.

3.3 Testes e Resultados

Durante o desenvolvimento de cada comportamento e do sistema final, o robô era testado em diferentes situações, como ambiente com labirintos, com diversas fontes luzes e diversos obstáculos. Esses testes além de certificar o sistema de suas habilidades, também eram úteis para afinamentos nas regras dos comportamentos ou na definição dos conjuntos nebulosos, pois havia situações em que a sensibilidade do robô ou o comportamento não era o "esperado".

Por exemplo, a situação de indecisão que o robô teve que resolver enquanto estava seguindo a parede, mostrado na figura 10. Nesse momento o robô podia tomar duas decisões: virar para esquerda ou para direita. Na primeira modelagem das regras do comportamento seguir parede, o robô sempre desviava para a esquerda, porém havia espaço para ele dobrar para a direita, e continuar seguindo a mesma parede. Apesar de não estar errada a sua decisão, isso simboliza que a definição dos conjuntos nebulosos para os sensores de proximidade precisava de mais níveis de proximidade, para ele saber o quão perto estava.

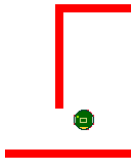

Figura 10: Situação de indecisão

Outro fato inusitado foi a *emergência* de um comportamento curioso, algo parecido com o *egoísmo*. O robô foi colocado em um ambiente delimitado por paredes, onde havia apenas uma fonte luz. Enquanto sua energia estava normal ele ficou seguindo a parede, porém ao ficar fraca ele foi em busca da fonte de energia, como era o esperado. Ao encontrar, se recarregou e ficou *eternamente* circulando-a. Isso aconteceu, pois fontes de luz são objetos detectados pelos sensores de proximidade, ele possivelmente achou que fosse uma parede.

Em diversos ambientes, o robô realizou com sucesso seus objetivos, sendo perceptível em alguns casos, quando mais de um comportamento estava atuando, uma certa indecisão na predominância de um comportamento. Por exemplo, durante a transição do comportamento seguir parede para o procurar por luz, ela fazia ambos ao mesmo tempo, o que era esperado.

4 Conclusão

Modelos hierárquicos nebulosos mostraram-se eficientes para modelagem de sistemas de controle complexos, como o modelo *Context-dependent Blending*, para implementação de comportamentos em robôs móveis autônomos.

O modelo *Context-dependent Blending* conseguiu satisfazer razoavelmente quase todos questionamentos discutidos na seção 1.1. Ele permite que um sistema seja facilmente paralelizável, pois cada comportamento pode ser avaliado de forma independente, antes do processo final de *defuzificação*. Fornece modelos claros para a modelagem de estratégias de controle e planejamento.

O sistema é expansível, permitindo adicionar novos comportamentos sem alterar muito o que já foi desenvolvido, com exceção das meta-regras que precisam abranger os novos comportamentos.

É robusto, caso algum bloco da hierarquia entre em colapso, não significa necessariamente o colapso total de sistema.

O sistema é tolerante a diferentes níveis de ruídos que podem afetar os sensores, sem prejudicar o planejamento; característica inerente de sistemas nebulosos.

Finalmente, a emergência de comportamentos complexos a partir de simples comportamentos, permite aos olhos do observador, concluir que o sistema é *inteligente*.

Referências

[1] NEHMZOW, U. *Mobile Robotics: A Practical Introduction*. London: Springer-Verlag, 2000.

[2] SAFFIOTTI, A. Fuzzy logic in autonomous robotics: behavior coordination. In: *Procs. of the 6th IEEE Int. Conf. on Fuzzy Systems*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1997. p. 573–578. Disponível em: <citeseer.ist.psu.edu/saffiotti97fuzzy.html>. Acesso em: 1 nov. 2004.

[3] SAFFIOTTI, A. The uses of fuzzy logic for autonomous robot navigation: a catalogue raisonné. *Soft Computing Research journal*, v. 1, n. 4, p. 180–197, 1997. Disponível em: <citeseer.ist.psu.edu/saffiotti97use.html>. Acesso em: 1 nov. 2004.

[4] ROMERO, R. A. F.; MARQUES, E. *Aprendizado em Robôs Móveis via Software e Hardware: ARMOSH*. [S.l.].

[5] BROOKS, R. A. *A Robust Layered Control System For a Mobile Robot*. Massachusetts Institute of Technology, 1985.

[6] MARCHI, J. *Navegação de robôs móveis autônomos: estudo e implementação de abordagens*. Dissertação (Mestrado) — Curso de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis, 2001.

[7] ROISENBERG, M. *Emergência da inteligência em agentes autônomos através de modelos inspirados na natureza*. Tese (Doutorado em Engenharia Elétrica) — Curso de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis, 1998.

[8] BARRETO, J. M. *Inteligência Artificial no Limiar do Século XXI*. 3. ed. Florianópolis, SC: Duplic, 2001.

[9] MAMDANI, E. H.; ASSILIAN, S. An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Human-Computer Studies*, v. 51, n. 2, p. 135–147, ago. 1999. Special issue: 1969-1999, the 30th anniversary.

[10] JUNIOR, P. R. C. *Sistemas Inteligentes de Navegação Autônoma: Uma Abordagem Modular e Hierárquica Com Novos Mecanismos de Memória e Aprendizagem*. Dissertação (Mestrado em Engenharia Elétrica) — Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, São Paulo, Dezembro 2001.

[11] DELGADO, M. R. D. B. da S. *Projeto Automático de Sistemas Nebulosos: Uma Abordagem Co-Evolutiva*. Tese (Doutorado em Engenharia Elétrica) — Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, SP, fev. 2002.

[12] MARAM, S. *Hierarchical Fuzzy Control of the UPFC and SVC located in AEP's Inez Area*. Dissertação (Master of Sciences in Electrical Engineering) — Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Virginia, USA, mai. 2003.

[13] GENTILE, M. *Development of a Hierarchical Fuzzy Model for the evaluation of Inherent Safety*. Tese (Doctor of Philosophy), Universidad de las Americas, Puebla, Mexico, ago. 2004.

[14] LEE, M.-L.; CHUNG, H.-Y.; YU, F.-M. Modeling of hierarchical fuzzy systems. *Fuzzy Sets and Systems*, v. 138, n. 2, p. 343–361, set. 2003.

[15] MURESAN, L. *Interpolation in Hierarchical Fuzzy Rule Bases*. [S.l.], 2001.

[16] OLIVEIRA, L. O. de L. *Mapas auto-organizáveis de Kohonen aplicados ao mapeamento de ambientes de robótica móvel*. Dissertação (Mestrado em Ciências da Computação) — Curso de Pós-Graduação em Ciências da Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, out. 2001.

Anexo B - Código-Fonte do Sistema Implementado

```
/**
 * Classe que representa um conjunto nebuloso.
 * Um conjunto nebuloso é caracterizado por uma função, chamada função de
   pertinêcia,
 * que permite definir o quanto é possível, ou a que grau de possibilidade um
   elemento x pertença a um conjunto nebuloso.
 * @author Alexandre
 *
 */
public interface FuzzySet
{
  /**
   * Função que retorna o grau de pertinência(possibilidade) que o elemento x
     pertença a esse conjunto nebuloso.
   * @param x elemento
   * @return grau de possibilidade
   */
  public abstract float fuzzify(float x);

  /**
   * Retorno o valor típico de um conjunto nebuloso. Usado geralmente durante
     o processo de defuzificação, cujo consequentes usam conjuntos Singleton.
   * @return valor típico
   */
  public abstract float getTypicalValue();
}
```

```
/**
 * Conjunto nebuloso representado por uma função triangular.
 * @author Alexandre
 *
 */

public class TriangleFuzzySet implements FuzzySet
{
  private float a, b, c, mbc;

  /**
   * Cria um conjunto nebuloso, passando todos os parâmetros que definem uma
     função triangular.
   * @param a
   * @param b
   * @param c
   */
  public TriangleFuzzySet(float a, float b, float c)
  {
    this.a = a;
    this.b = b;
    this.c = c;
    this.mbc = (b + c)/2;
  }
```

```
  public float fuzzify(float x)
  {
    if(x > a && x <= b)
      return (x - a)/(b - a);
    if(x > b && x < c)
      return (c - x)/(c - b);
    return 0;
  }

  public float getTypicalValue()
  {
    return mbc;
  }
}
```

```
/**
 * Conjunto nebuloso representado por uma função trapezoidal.
 * @author Alexandre
 *
 */

public class TrapezoidFuzzySet implements FuzzySet
{
  private float a, b, c, d, mbc;

  /**
   * Cria um conjunto nebuloso, passando todos os parâmetros que definem uma
     função trapezoidal.
   * @param a
   * @param b
   * @param c
   * @param d
   */
  public TrapezoidFuzzySet(float a, float b, float c, float d)
  {
    this.a = a;
    this.b = b;
    this.c = c;
    this.d = d;
    mbc = (b + c)/2;
  }

  public float fuzzify(float x)
  {
    if(x > a && x < b)
      return (x - a)/(b - a);
    if(x >= b && x <= c)
      return 1;
    if(x > c && x < d)
      return (d - x)/(d - c);
    return 0;
  }

  public float getTypicalValue()
  {
    return mbc;
  }
}
```

```
/**
 * Conjunto nebuloso representado por um valor. Usado geralmente nos
   consequentes das regras nebulosas para facilitar o processo de defuzificação.
 * @author Alexandre
 *
 */
public class SingletonSet implements FuzzySet
{
  private float a;

  /**
   * Cria um conjunto Singleton, passando o valor que o representa.
   * @param a
   */
  public SingletonSet(float a)
  {
    this.a = a;
  }

  public float fuzzify(float x)
  {
    if(x == a)
      return 1;
    return 0;
  }

  public float getTypicalValue()
  {
    return a;
  }
}
```

```
/**
 * Armazena todos os valores lingüísticos (conjuntos nebulosos) de uma variável
   lingüística.
 * @author Alexandre
 *
 */
public class TermSet
{
  protected FuzzySet[] fuzzySet;
  protected float[] fuzzified;

  /**
   * Cria um conjunto de termos.
   * @param fuzzySet valores lingüísticos (conjuntos nebulosos) que definem uma
     variável lingüística.
   */
  public TermSet(FuzzySet[] fuzzySet)
  {
    this.fuzzySet = fuzzySet;
    fuzzified = new float[fuzzySet.length];
  }
  /**
   * Calcula o grau de pertinência de um elemento em todos os conjunto
     nebulosos.
   * @param input
   */
  public void fuzzify(float input)
  {
```

```
    for(int i = 0; i < fuzzySet.length; i++)
      fuzzified[i] = fuzzySet[i].fuzzify(input);
  }
  /**
   * Retorna um conjunto nebuloso pertencente a esse conjunto de termos.
   * @param index índice do conjunto nebuloso
   * @return conjunto nebuloso
   */
  public FuzzySet getFuzzySet(int index)
  {
    return fuzzySet[index];
  }

  /**
   * Retorna a grau de pertinência de um elemento x previamente calculado.
   * @param index índice do conjunto nebuloso
   * @return grau de pertinência
   */
  public float getFuzzifiedValue(int index)
  {
    return fuzzified[index];
  }

  /**
   * Retorna a quantidade de conjuntos nebulosos.
   * @return quantidade
   */
  public int getLength()
  {
    return fuzzySet.length;
  }
}
```

```
/**
 * Classe que fornece conjuntos de termos lingüísticos a serem usados na
   construção de sistemas nebulosos.
 * @author Alexandre
 *
 */
public class TermConstruct
{
  /**
   * Retorna um conjunto de termos lingüísticos para a variável distância.
   * @return conjunto de termos
   */
  public static TermSet getDistanceTerm()
  {
    FuzzySet fuzzySet[] = new FuzzySet[3];
    fuzzySet[0] = new TrapezoidFuzzySet(0, 0, 30, 100); //far
    fuzzySet[1] = new TrapezoidFuzzySet(70, 100, 400, 700); //near
    fuzzySet[2] = new TrapezoidFuzzySet(650, 800, 1023, 1023); //vnear
    return new TermSet(fuzzySet);
  }
  /**
   * Retorna um conjunto de termos lingüísticos para a variável distância,
     porém com maior precisão que o @see TermConstruct#getDistanceTerm().
   * @return conjunto de termos
   */
  public static TermSet getDistanceTermAccurate()
  {
```

```
  FuzzySet fuzzySet[] = new FuzzySet[4];
  fuzzySet[0] = new TrapezoidFuzzySet(0, 0, 50, 100); //far
  fuzzySet[1] = new TrapezoidFuzzySet(70, 100, 200, 250); //near
  fuzzySet[2] = new TrapezoidFuzzySet(200, 300, 500, 700); //vnear
  fuzzySet[3] = new TrapezoidFuzzySet(650, 800, 1023, 1023); //vvnear
  return new TermSet(fuzzySet);
}

/**
 * Retorna um conjunto de termos lingüísticos para a variável luminosidade,
   porém com maior precisão que o @see TermConstruct#getLightTerm().
 * @return conjunto de termos
 */
public static TermSet getLightTermAccurate()
{
  FuzzySet fuzzySet[] = new FuzzySet[3];
  fuzzySet[0] = new TrapezoidFuzzySet(0, 0, 150, 200); //vlight
  fuzzySet[1] = new TrapezoidFuzzySet(150, 200, 300, 500); //mlight
  fuzzySet[2] = new TrapezoidFuzzySet(450, 500, 1023, 1023); //flight
  return new TermSet(fuzzySet);
}
/**
 * Retorna um conjunto de termos lingüísticos para a variável luminosidade.
 * @return conjunto de termos
 */
public static TermSet getLightTerm()
{
  FuzzySet fuzzySet[] = new FuzzySet[2];
  fuzzySet[0] = new TrapezoidFuzzySet(0, 0, 450, 500); //vlight
  fuzzySet[1] = new TrapezoidFuzzySet(450, 500, 1023, 1023); //flight
  return new TermSet(fuzzySet);
}
/**
 * Retorna um conjunto de termos lingüísticos para a variável velocidade.
 * @return conjunto de termos
 */
public static TermSet getVelocityTerm()
{
  FuzzySet fuzzySet[] = new FuzzySet[7];
  fuzzySet[0] = new SingletonSet(-8); //nfast
  fuzzySet[1] = new SingletonSet(-5); //nmedium
  fuzzySet[2] = new SingletonSet(-2); //nlow
  fuzzySet[3] = new SingletonSet(0); //stop
  fuzzySet[4] = new SingletonSet(2); //plow
  fuzzySet[5] = new SingletonSet(5); //pmedium
  fuzzySet[6] = new SingletonSet(8); //pfast
  return new TermSet(fuzzySet);
}
/**
 * Retorna um conjunto de termos lingüísticos para a variável energia.
 * @return conjunto de termos
 */
public static TermSet getEnergyTerm()
{
  FuzzySet fuzzySet[] = new FuzzySet[4];
  fuzzySet[0] = new TrapezoidFuzzySet(0, 0, 0, 10); //morto
  fuzzySet[1] = new TrapezoidFuzzySet(9, 10, 1980, 2000); //weak
  fuzzySet[2] = new TrapezoidFuzzySet(1980, 2000, 4980, 5000); //normal
  fuzzySet[3] = new TrapezoidFuzzySet(4980, 5000, 10000, 10000); //strong
  return new TermSet(fuzzySet);
```

```
  }
  /**
   * Retorna um conjunto de termos lingüísticos para a variável velocidade,
     porém com menor precisão que @see TermConstruct#getDistanceTerm().
   * @return conjunto de termos
   */
  public static TermSet getDistanceTermFewAccurate()
  {
    FuzzySet fuzzySet[] = new FuzzySet[2];
    fuzzySet[0] = new TrapezoidFuzzySet(0, 0, 140, 250); //far
    fuzzySet[1] = new TrapezoidFuzzySet(200, 250, 1023, 1023); //near
    return new TermSet(fuzzySet);
  }
  /**
   * Retorna um conjunto de termos lingüísticos para a variável consumo de
     energia.
   * @return conjunto de termos
   */
  public static TermSet getEnergyUseTerm()
  {
    FuzzySet fuzzySet[] = new FuzzySet[6];
    fuzzySet[0] = new SingletonSet(-0.8f); //very
    fuzzySet[1] = new SingletonSet(-0.25f); //few
    fuzzySet[2] = new SingletonSet(0); //nothing
    fuzzySet[3] = new SingletonSet(0.25f); //few
    fuzzySet[4] = new SingletonSet(0.4f); //very
    fuzzySet[5] = new SingletonSet(6); //vvvvery
    return new TermSet(fuzzySet);
  }

  /**
   * Retorna um conjunto de termos lingüísticos para a variável estado.
   * @param length quantidade de estados a serem criados
   * @return conjunto de termos
   */
  public static TermSet getStateTerm(int length)
  {
    FuzzySet fuzzySet[] = new FuzzySet[length];
    for(int i = 0; i < length; i++)
      fuzzySet[i] = new SingletonSet(i);
    return new TermSet(fuzzySet);
  }
}
```

```
import java.util.Arrays;

/**
 * Super-classe que representa os comportamentos robóticos.
 * @author Alexandre
 *
 */

public abstract class Behavior
{
  protected int fam [][][];
  protected TermSet antecedent[];
  protected float denominator;
  protected String name;
  protected float fuzzificationResult [];
  protected float[] defuzzificationResult ;
```

```
/**
 * Cria um comportamento robótico.
 * @param fam (fuzzy associative memory) Memória Associativa Nebulosa
 * @param antecedent conjunto de termos usados nos antecedentes das regras
   nebulosas
 * @param name nome do comportamento
 */
public Behavior(int fam [][][], TermSet antecedent[], String name)
{
  this.fam = fam;
  this.antecedent = antecedent;
  denominator = 0;
  this.name = name;
   fuzzificationResult  = new float[fam.length];
}
/**
 * Realiza todo o processo inferência das regras nebulosas e defuzificação.
 * @param sensors entradas do sistema
 * @return resultado final do processo
 */
public float[] actuate(float sensors [])
{
   fuzzify(sensors);
  return defuzzify();
}

/**
 * Realiza apenas o processo de fuzificação.
 * @param sensors entradas do sistema
 */
public void fuzzify(float sensors [])
{
  for(int i = 0; i < antecedent.length; i++)
    antecedent[i].fuzzify(sensors[i]);
  Arrays.fill(fuzzificationResult, 0);
  int jump = 0;
  int i = 0;
  do
  {
    step:
    {
      float fuzzifiedMin = Float.MAX_VALUE;
      float fuzzifiedTemp = 0;
      jump = 1;
      for(int j = 0; j < fam[i][0].length; j++)
      {
        fuzzifiedTemp = antecedent[j].getFuzzifiedValue(fam[i][0][j]);
        if(fuzzifiedTemp == 0)
        {
          int k = 1;
          for(k += j; k < antecedent.length; k++)
            jump *= antecedent[k].getLength();
          break step;
        }
        if(fuzzifiedTemp < fuzzifiedMin)
          fuzzifiedMin = fuzzifiedTemp;
      }
      fuzzificationResult[i] = fuzzifiedMin;
    }
```

```
        i += jump;
      }while(i < fam.length);
    }
    /**
     * Retorna o nome do comportamento.
     */
    public String toString()
    {
      return name;
    }
    /**
     * Retorna o número de consequentes de uma regra nebulosa.
     * @return número de consequentes
     */
    public int getConsequentLength()
    {
      return fam[0][1].length;
    }
    /**
     * Realiza o processo de defuzificação.
     * @return resultado do processo
     */
    public abstract float[] defuzzify();
}
```

```
import java.util.Arrays;
/**
 * Classe que representa um comportamento reativo ou comportamentos simples.
 * @author Alexandre
 *
 */
public class ReactiveBehavior extends Behavior
{
    protected TermSet consequent[];

    /**
     * Cria um comportamento reativo.
     * @param fam (fuzzy associative memory) Memória Associativa Nebulosa
     * @param antecedent conjunto de termos usados nos antecedentes das regras
       nebulosas
     * @param consequent conjunto de termos usados nos consequentes das regras
       nebulosas
     * @param name nome do comportamento
     */
    public ReactiveBehavior(int fam [][][], TermSet antecedent[], TermSet
      consequent[], String name)
    {
      super(fam, antecedent, name);
      this.consequent = consequent;
      defuzzificationResult = new float[fam[0][1].length];
    }

    public float[] defuzzify()
    {
      defuzzifyPartlyNumerator();
      for(int i = 0; i < defuzzificationResult.length; i++)
        defuzzificationResult[i] /= denominator;
      return defuzzificationResult;
    }
    /**
```

```
    * Etapa inicial do processo de Defuzificação. Processa apenas o numerador,
      utilizando apenas conjuntos Singleton.
    * @return resultado do processo
    */
  public float[] defuzzifyPartlyNumerator()
  {
    denominator = 0;
    Arrays.fill(defuzzificationResult, 0);
    for(int i = 0; i < fuzzificationResult.length; i++)
    {
      if(fuzzificationResult[i] == 0)
        continue;
      for(int j = 0; j < defuzzificationResult.length; j++)
        defuzzificationResult[j] += fuzzificationResult[i] *
          consequent[j].getFuzzySet(fam[i][1][j]).getTypicalValue();
      denominator += fuzzificationResult[i];
    }
    return defuzzificationResult;
  }
  /**
    * Etapa posterior a @see ReactiveBehavior#defuzzifyPartlyNumerator.
      Processa apenas o denominador, utilizando apenas conjuntos Singleton.
    * @return denominador
    */
  public float defuzzifyPartlyDenominator()
  {
    return denominator;
  }
}
```

```
import java.util.Arrays;
/**
 * Classe que representa um comportamento complexo, composta por
   comportamentos mais simples.
 * @author Alexandre
 *
 */
public class ComplexBehavior extends Behavior
{
  protected ReactiveBehavior[] behaviors;
  protected boolean activatedBehaviors[];
  protected int behaviorSensors[][];
  protected float sensors [][];
  protected int nextState;

  protected float defuzzifyPartlyNumerator[][];
  protected float defuzzifyPartlyDenominator[];
  /**
    * Cria um comportamento complexo.
    * @param fam (fuzzy associative memory) Memória Associativa Nebulosa
    * @param antecedent conjunto de termos usados nos antecedentes das regras
      nebulosas
    * @param behaviors comportamentos de menor hierarquia a serem utilizados
    * @param behaviorSensors matriz, que contém os índices de matrizes, que
      relaciona os dados da matriz de entrada do sistema complexo com a matriz
      de entrada dos sistemas de menor hierarquia. Utilizado para distribuir
      corretamente os dados de entrada do sistema complexo para os sistemas de
      menor hierarquia
    * @param name nome do comportamento
    */
```

```
public ComplexBehavior(int fam[][][], TermSet antecedent[],
  ReactiveBehavior[] behaviors, int behaviorSensors [][], String name)
{
  super(fam, antecedent, name);
  this.behaviors = behaviors;
  this.behaviorSensors = behaviorSensors;
  sensors = new float[behaviors.length][];
  for(int i = 0; i < behaviorSensors.length; i++)
    sensors[i] = new float[behaviorSensors[i].length];
  activatedBehaviors = new boolean[behaviors.length];

   defuzzificationResult = new float[behaviors[0].getConsequentLength()];
  //bh regra = fuzzy
  //regra acao = num
  defuzzifyPartlyNumerator = new float[behaviors.length][];
  //regra = den
  defuzzifyPartlyDenominator = new float[behaviors.length];
  nextState = 0;
}
/**
 * Distribui a entrada do sistema(saída dos sensores) para os comportamentos
    de menor hierarquia.
 * @param sensorsOutput entrada do sistema(saída dos sensores)
 */
public void setSensors(float sensorsOutput[])
{
  for(int i = 0; i < behaviors.length; i++)
    for(int j = 0; j < behaviorSensors[i].length; j++)
      this.sensors[i][j] = sensorsOutput[behaviorSensors[i][j]];
}

public float[] defuzzify()
{
  for(int i = 0; i < behaviors.length; i++)
  {
    behaviors[i].fuzzify(sensors[i]);
    defuzzifyPartlyNumerator[i] = behaviors[i].defuzzifyPartlyNumerator();
    defuzzifyPartlyDenominator[i] =
      behaviors[i].defuzzifyPartlyDenominator();
  }

  denominator = 0;
  Arrays.fill(activatedBehaviors, false);
  Arrays.fill(defuzzificationResult, 0);

  double winner = 0;
  for(int i = 0; i < fam.length; i++)
  {
    System.out.println(fuzzificationResult[i]);
    if(fuzzificationResult[i] == 0)
      continue;

    activatedBehaviors[fam[i][1][0]] = true;

    if(fam[i][1].length > 1)
    {
      if(fuzzificationResult[i] > winner)
      {
        winner = fuzzificationResult[i];
        nextState = fam[i][1][1];
```

```
            }
          }
          /*
            TODO
            Implementar a aplicação grau de desejabilidade, através do min
            de cada comportamento. Isso pode ser feito através da
            passagem desse valor para o método de defuzificação de cada
            comportamento de menor hierarquia. O grau de desejabilidade
            fica armazenado em fuzzificationResult[].
          */
          for(int j = 0; j < defuzzificationResult.length; j++)
            defuzzificationResult[j] += defuzzifyPartlyNumerator[fam[i][1][0]][j];
          denominator += defuzzifyPartlyDenominator[fam[i][1][0]];
        }
        for(int i = 0; i < defuzzificationResult.length; i++)
          defuzzificationResult[i] /= denominator;
        return defuzzificationResult;
    }
    /**
     * Retorna o estado do sistema.
     * @return estado
     */
    public int getNextState()
    {
        return nextState;
    }
    /**
     * Retorna o comportamentos ativados.
     * @return comportamentos ativados
     */
    public boolean[] getActivatedBehaviors()
    {
        return activatedBehaviors;
    }
    /**
     * Retorna os comportamentos que compõem esse comportamento complexo.
     * @return comportamentos
     */
    public Behavior[] getBehaviors()
    {
        return behaviors;
    }
}
```

```
/**
 * Classe que fornece os comportamentos robóticos já definidos.
 * @author Alexandre
 *
 */
public class BehaviorConstruct
{
    /**
     * Retorna o comportamento evitar colisão.
     * @return comportamento
     */
    public static ReactiveBehavior getCollisionAvoid()
    {
        TermSet antecedent[] = new TermSet[2];
        TermSet consequent[] = new TermSet[3];
        antecedent[0] = TermConstruct.getDistanceTerm();
```

```
    antecedent[1] = TermConstruct.getDistanceTerm();
    consequent[0] = TermConstruct.getVelocityTerm();
    consequent[1] = TermConstruct.getVelocityTerm();
    consequent[2] = TermConstruct.getEnergyUseTerm();

    int fam [][][] = {{{0, 0}, {6, 6, 0}}, {{0, 1}, {4, 6, 0}},
    {{0, 2}, {2, 6, 1}}, {{1, 0}, {6, 4, 0}}, {{1, 1}, {2, 6, 1}}, {{1, 2},
    {2, 6, 1}}, {{2, 0}, {6, 2, 1}}, {{2, 1}, {6, 2, 1}}, {{2, 2}, {3, 6,
      1}}};

    return new ReactiveBehavior(fam, antecedent, consequent,
      "CollisionAvoid");
  }
  /**
   * Retorna o comportamento seguir parede.
   * @return comportamento
   */
  public static ReactiveBehavior getWallFollower()
  {
    TermSet antecedent[] = new TermSet[2];
    TermSet consequent[] = new TermSet[3];
    antecedent[0] = TermConstruct.getDistanceTermAccurate();
    antecedent[1] = TermConstruct.getDistanceTermAccurate();
    consequent[0] = TermConstruct.getVelocityTerm();
    consequent[1] = TermConstruct.getVelocityTerm();
    consequent[2] = TermConstruct.getEnergyUseTerm();

    int fam [][][] = {{{0, 0}, {6, 6, 0}}, {{0, 1}, {6, 1, 1}}, {{0, 2}, {6, 6,
      0}},
    {{0, 3}, {4, 6, 1}}, {{1, 0}, {3, 6, 1}}, {{1, 1}, {6, 2, 1}}, {{1, 2}, {3,
      6, 1}},
    {{1, 3}, {3, 6, 1}}, {{2, 0}, {1, 6, 0}}, {{2, 1}, {3, 6, 1}}, {{2, 2}, {0,
      6, 0}},
    {{2, 3}, {3, 6, 1}}, {{3, 0}, {0, 6, 0}}, {{3, 1}, {0, 6, 0}}, {{3, 2}, {0,
      6, 0}},
    {{3, 3}, {0, 6, 0}}};

    return new ReactiveBehavior(fam, antecedent, consequent, "WallFollower");
  }
  /**
   * Retorna o comportamento morrer.
   * @return comportamento
   */
  public static ReactiveBehavior getDying()
  {
    TermSet antecedent[] = new TermSet[1];
    TermSet consequent[] = new TermSet[3];
    antecedent[0] = TermConstruct.getEnergyTerm();
    consequent[0] = TermConstruct.getVelocityTerm();
    consequent[1] = TermConstruct.getVelocityTerm();
    consequent[2] = TermConstruct.getEnergyUseTerm();

    int fam [][][] = {{{0}, {3, 3, 2}}, {{1}, {3, 3, 1}}, {{2}, {3, 3, 1}},
    {{3}, {3, 3, 1}}};

    return new ReactiveBehavior(fam, antecedent, consequent, "Dying");
  }
  /**
   * Retorna o comportamento procurar por fonte de luz.
   * @return comportamento
```

```
 */
public static ReactiveBehavior getSearchLight()
{
  TermSet antecedent[] = new TermSet[3];
  TermSet consequent[] = new TermSet[3];
  antecedent[0] = TermConstruct.getLightTermAccurate();
  antecedent[1] = TermConstruct.getLightTermAccurate();
  antecedent[2] = TermConstruct.getLightTermAccurate();
  consequent[0] = TermConstruct.getVelocityTerm();
  consequent[1] = TermConstruct.getVelocityTerm();
  consequent[2] = TermConstruct.getEnergyUseTerm();

  int fam [][][] = {{{0, 0, 0}, {3, 3, 5}}, {{0, 0, 1}, {3, 3, 5}},
  {{0, 0, 2}, {3, 3, 5}}, {{0, 1, 0}, {3, 3, 5}}, {{0, 1, 1}, {3, 3, 5}},
  {{0, 1, 2}, {3, 3, 5}}, {{0, 2, 0}, {3, 3, 5}}, {{0, 2, 1}, {3, 3, 5}},
  {{0, 2, 2}, {3, 3, 5}}, {{1, 0, 0}, {3, 3, 5}}, {{1, 0, 1}, {3, 3, 5}},
  {{1, 0, 2}, {3, 3, 5}}, {{1, 1, 0}, {3, 3, 5}}, {{1, 1, 1}, {6, 6, 1}},
  {{1, 1, 2}, {3, 6, 2}}, {{1, 2, 0}, {3, 3, 5}}, {{1, 2, 1}, {3, 6, 2}},
  {{1, 2, 2}, {2, 6, 2}}, {{2, 0, 0}, {3, 3, 5}}, {{2, 0, 1}, {3, 3, 5}},
  {{2, 0, 2}, {3, 3, 5}}, {{2, 1, 0}, {3, 3, 5}}, {{2, 1, 1}, {6, 3, 2}},
  {{2, 1, 2}, {6, 6, 1}}, {{2, 2, 0}, {3, 3, 5}}, {{2, 2, 1}, {6, 3, 1}},
  {{2, 2, 2}, {6, 6, 0}}};

  return new ReactiveBehavior(fam, antecedent, consequent, "SearchLight");
}
/**
 * Retorna o comportamento complexo consumir de luz e seguir de parede. Ver
   o relatório para mais detalhes.
 * @return comportamento
 */
public static ComplexBehavior getLightConsumer()
{
  TermSet antecedent[] = new TermSet[4];
  ReactiveBehavior behaviors[] = new ReactiveBehavior[4];
  antecedent[0] = TermConstruct.getStateTerm(2);
  antecedent[1] = TermConstruct.getEnergyTerm();
  antecedent[2] = TermConstruct.getDistanceTermFewAccurate();
  antecedent[3] = TermConstruct.getLightTerm();
  behaviors[0] = getCollisionAvoid();
  behaviors[1] = getWallFollower();
  behaviors[2] = getSearchLight();
  behaviors[3] = getDying();

  int fam [][][] = {{{0, 0, 0, 0}, {3, 0}}, {{0, 0, 0, 1}, {3, 0}},
  {{0, 0, 1, 0}, {3, 0}}, {{0, 0, 1, 1}, {3, 0}}, {{0, 1, 0, 0}, {2, 1}},
  {{0, 1, 0, 1}, {0, 0}}, {{0, 1, 1, 0}, {0, 0}}, {{0, 1, 1, 1}, {0, 0}},
  {{0, 2, 0, 0}, {1, 0}}, {{0, 2, 0, 1}, {1, 0}}, {{0, 2, 1, 0}, {1, 0}},
  {{0, 2, 1, 1}, {1, 0}}, {{0, 3, 0, 0}, {1, 0}}, {{0, 3, 0, 1}, {1, 0}},
  {{0, 3, 1, 0}, {1, 0}}, {{0, 3, 1, 1}, {1, 0}},

  {{1, 0, 0, 0}, {3, 0}}, {{1, 0, 0, 1}, {3, 0}}, {{1, 0, 1, 0}, {3, 0}},
  {{1, 0, 1, 1}, {3, 0}}, {{1, 1, 0, 0}, {2, 1}}, {{1, 1, 0, 1}, {1, 0}},
  {{1, 1, 1, 0}, {0, 0}}, {{1, 1, 1, 1}, {0, 0}}, {{1, 2, 0, 0}, {2, 1}},
  {{1, 2, 0, 1}, {1, 0}}, {{1, 2, 1, 0}, {1, 0}}, {{1, 2, 1, 1}, {1, 0}},
  {{1, 3, 0, 0}, {1, 0}}, {{1, 3, 0, 1}, {1, 0}}, {{1, 3, 1, 0}, {1, 0}},
  {{1, 3, 1, 1}, {1, 0}}};

  int distribution [][] = {{0, 1}, {2, 3}, {4, 5, 6}, {7}};

  return new ComplexBehavior(fam, antecedent, behaviors, distribution,
```

```
        "LightConsumer");
  }
}
```

```
import edu.wsu.KepheraSimulator.RobotController;
/**
 * Classe acessada pelo  WSU Khepera Robot Simulator para controlar o robô.
 * @author Alexandre
 *
 */
public class RoboticBehavior extends RobotController
{
  private ComplexBehavior behavior;
  private float energy;
  private BehaviorUI ui;
  private float[] distanceSensor;

  /**
   * Cria o comportamento robótico que controlará o robô.
   *
   */
  public RoboticBehavior()
  {
    behavior = BehaviorConstruct.getLightConsumer();
    energy = 2500;
    //energy = 9000;
    ui = new BehaviorUI(behavior.getBehaviors());
    distanceSensor = new float[6];
    setWaitTime(10);
  }
  /**
   * When an object implementing the interface Controller is used to create a
     thread, the starting thread causes the objects doWork method to be called.
      This method is called repeatedly and may perform any task. The time
     needed for doWork to complete is not significant with respect to the next
     invocation of this method.
   * @throws Exception
   */
  public void doWork() throws Exception
  {
    for(int i = 0; i < distanceSensor.length; i++)
      distanceSensor[i] = getDistanceValue(i);

    float distanceMinorLeft = (distanceSensor[0] > distanceSensor[1])?
      distanceSensor[0]: distanceSensor[1];
    distanceMinorLeft = (distanceMinorLeft > distanceSensor[2])?
      distanceMinorLeft : distanceSensor[2];

    float distanceMinorRight = (distanceSensor[3] > distanceSensor[4])?
      distanceSensor[3]: distanceSensor[4];
    distanceMinorRight = (distanceMinorRight > distanceSensor[5])?
      distanceMinorRight : distanceSensor[5];

    float distance = (distanceMinorLeft > distanceMinorRight)?
      distanceMinorLeft : distanceMinorRight;

    float rightW = (distanceSensor[4] > distanceSensor[5])? distanceSensor[4] :
      distanceSensor[5];
    float center = (distanceSensor[2] > distanceSensor[3])? distanceSensor[2] :
      distanceSensor[3];
```

```
    float leftL = (getLightValue(0) > getLightValue(1))? getLightValue(1) :
      getLightValue(0);
    float centerL = (getLightValue(2) > getLightValue(3))? getLightValue(3) :
      getLightValue(2);
    float rightL = (getLightValue(4) > getLightValue(5))? getLightValue(5) :
      getLightValue(4);

    float light = (leftL < centerL)? leftL : centerL;
    if(light > rightL)
      light = rightL;

    float stateB[] = {distanceMinorLeft, distanceMinorRight, center, rightW,
      leftL, centerL, rightL, energy};
    float state[] = {behavior.getNextState(), energy, distance, light};

    behavior.setSensors(stateB);
    float action[] = behavior.actuate(state);
    ui.setEnergy(energy);
    ui.setBehaviors(behavior.getActivatedBehaviors());
    setLeftMotorSpeed((int)action[0]);
    setRightMotorSpeed((int)action[1]);
    energy += action[2];
  }
  /**
   * Indicates that the application has finished using the controller, and that
     any resources being used may be released. The starting thread invokes the
     object's close method only when doWork has returned and is not scheuled
     to be called again.
   * @throws Exception
   */
  public void close() throws Exception
  {
    ui.dispose();
  }
}
```

```
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JLabel;
import java.awt.Container;
import javax.swing.JScrollPane;
import javax.swing.Box;
import javax.swing.SwingConstants;
import java.util.Arrays;
import java.text.NumberFormat;
/**
 * Classe que mostra os comportamentos ativados em um dado instante.
 * @author Alexandre
 *
 */
public class BehaviorUI extends JFrame
{
  private JTextArea behaviorsText;
  private JLabel energy;
  private JLabel behaviorsLabel;
  private Behavior[] behaviors;
  private boolean shownBehavior[];
  private NumberFormat numberFormat;
```

```
  static final long serialVersionUID = 01L;
  /**
   * Cria a interface.
   * @param behaviors todos os comportamentos do sistema
   */
  public BehaviorUI(Behavior[] behaviors)
  {
    super("Comportamentos Ativados");

    this.behaviors = behaviors;
    shownBehavior = new boolean[behaviors.length];

    numberFormat = NumberFormat.getInstance();
    numberFormat.setMaximumFractionDigits(3);

    Container container = getContentPane();
    Box box = Box.createVerticalBox();
    container.add(box);

    energy = new JLabel("Energia: ???", SwingConstants.CENTER);
    box.add(energy);

    behaviorsLabel = new JLabel("Comportamentos:", SwingConstants.LEFT);
    box.add(behaviorsLabel);

    behaviorsText = new JTextArea(6, 10);
    behaviorsText.setEditable(false);
    box.add(new JScrollPane(behaviorsText));

    setSize(160, 220);
    setVisible(true);
  }
  /**
   * Mostra o valor da energia.
   * @param energy energia
   */
  public void setEnergy(float energy)
  {
    this.energy.setText("Energia: " + numberFormat.format(energy));
  }
  /**
   * Mostra os comportamentos ativos.
   * @param behavior comportamentos
   */
  public void setBehaviors(boolean behavior[])
  {
    if (!(Arrays.equals(shownBehavior, behavior)))
    {
      for(int i = 0; i < behavior.length; i++)
        shownBehavior[i] = behavior[i];
      behaviorsText.append("--------------------
      --------------------\n");
      for(int i = 0; i < behavior.length; i++)
        if(behavior[i])
          behaviorsText.append(behaviors[i].toString() + "\n");
    }
  }
}
```