ROBÓTICA COGNITIVA: PROGRAMAÇÃO BASEADA EM LÓGICA PARA CONTROLE DE ROBÔS

Felipe Werndl Trevizan*
trevisan@ime.usp.br

Leliane Nunes de Barros*
leliane@ime.usp.br

*Departamento de Ciência da Computação
Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP)
Rua do Matão, 1010 – Cidade Universitária – CEP 05508-090 São Paulo – SP – Brasil

ABSTRACT

The goal of the Cognitive Robotics research area is to develop robotic agents capable of high-level functions by using a programming language, based on logics, to describe the robot control program. Besides, such a language can be used to prove properties of the world and to simulate the robot behavior by running its program. This paper shows how a Lego® MindStorms™ robot can be used to implement a software agent capable of performing high level functions specified in IndiGolog – a logical language to write robot control programs, based on Situation Calculus. The application domain example is the classical problem of the Wumpus World for which the construction of a complete intelligent agent requires the integration of several Artificial Intelligence techniques, such as: reactive planning; hierarquical and goal achievement planning; plan execution; reasoning with incomplete information; generation and discrimination of hypotheses about the world state; and belief changes.

KEYWORDS: Planning, on-line planning, Golog, Legolog, Situation Calculus, Lego® MindStorms™.

ARTIGO CONVIDADO:
Versão completa e revisada de artigo apresentado no SBAI-2005
Artigo submetido em 01/06/2006
1a. Revisão em 13/11/2006
2a. Revisão em 30/12/2006
Aceito sob recomendação do Editor Convidado
Prof. Osvaldo Ronald Saavedra Mendez

RESUMO

A área de Robótica Cognitiva tem como principal objetivo desenvolver agentes robóticos capazes de realizar funções de alto-nível, especificando o programa de controle do robô em uma linguagem de programação baseada em lógica. Desta forma, é possível declarar e verificar propriedades do agente como prova de teoremas. Além disso, uma especificação feita em uma linguagem formal pode ser executável, o que permite simular o comportamento do agente através dessa especificação. Este artigo apresenta o desenvolvimento, passo a passo, de um agente para um robô Lego® MindStorms™, usando IndiGolog – uma linguagem para especificação de agentes baseada no Cálculo de Situações. Como exemplo de aplicação, foi escolhido o problema clássico do Mundo do Wumpus para o qual a construção de um agente completo envolve a integração das seguintes técnicas de: planejamento reativo, planejamento para satisfação de metas e realização de tarefas (planejamento hierárquico), execução de ações, raciocínio com informação incompleta, geração e raciocínio hipotético sobre o estado do mundo e mudanças de crença.

PALAVRAS-CHAVE: Planejamento, planejamento *on-line*, Golog, Legolog, Cálculo de Situações, Lego® MindStorms™.

1 INTRODUÇÃO

Um dos problemas de maior interesse na área de Robótica é o desenvolvimento de agentes em ambientes dinâmicos, com observação parcial do mundo e ações não-determinísticas,

probabilísticas ou envolvendo as ainda, duas características (Trevizan et al., 2007). Nesse tipo de problema, o agente deve sensoriar o ambiente para executar ações e monitorar as mudanças do mundo. Tudo isso deve ser realizado durante o planejamento, isto é, enquanto o agente decide qual será a próxima ação a ser executada que o leve mais próximo de seus objetivos. A construção de agentes com tais características possui uma vasta aplicação no mundo real, por exemplo: robôs de busca e resgate (*search and rescue*), navegação de robôs e automação industrial.

Uma das propostas mais conhecidas na área de Robótica Cognitiva usa a linguagem Golog (Levesque et al., 1997): uma linguagem baseada no Cálculo de Situações (Mccarthy, 1963) e implementada como um meta-interpretador Prolog. Apesar do uso de Golog e suas extensões permitir a investigação de novas teorias lógicas de raciocínio (Boutilier et al., 2000; Liu et al., 2004), esses resultados são raramente testados ou demonstrados para agentes robóticos em ambientes complexos e não simulados (Levesque e Pagnucco, 2000), em que a integração de diferentes tipos de comportamento torna o desenvolvimento do agente uma tarefa não trivial.

1.1 Objetivos e contribuições

O objetivo desse trabalho é mostrar como um agente robótico com capacidade de realizar tarefas de alto-nível e raciocínio baseado em lógica, pode ser implementado em um robô Lego® MindStorms™(Trevizan e de Barros, 2004). Essa implementação será feita usando Legolog (Levesque e Pagnucco, 2000), um arcabouço para desenvolvimento de *software* para robôs Lego® MindStorms™, que contém uma extensão da linguagem Golog. Como um estudo de caso, foi escolhido o problema clássico do Mundo do Wumpus (Russel e Norvig, 2003) no qual um agente deve explorar um ambiente hostil, fazendo observações parciais com o objetivo de coletar barras de ouro, evitando localizações que o coloque em perigo. A construção de um agente completo para o Mundo do Wumpus envolve uma combinação de técnicas de Inteligência Artificial, tais como: raciocínio sobre ações no Cálculo de Situações, raciocínio com informação incompleta sobre o estado do mundo, planejamento reativo, planejamento para satisfação de metas e execução de ações.

Apesar do Mundo do Wumpus ter sido um dos primeiros jogos propostos para primeira pessoa e ser discutido em muitos livros de Inteligência Artificial, o desenvolvimento de uma solução completa para ele e implementada em um ambiente não-simulado não foi encontrada na literatura. Além disso, o problema do Mundo do Wumpus ainda é considerado difícil, devido à observabilidade parcial do mundo e envolver tomada de decisão seqüencial, isto é, o agente precisa lembrar os estados anteriores do mundo para tomar decisões. Isso é corroborado pelo interesse que esse problema tem despertado na comunidade de raciocínio não-monotônico, recentemente proposto como tema especial nos eventos internacionais: NRAC05 (*Sixth Workshop on Nonmonotonic Reasoning, Action, and Change*, 2005) e NMR06 (*International Workshop on Non-Monotonic Reasoning*, 2006).

Em suma, a principal contribuição desse trabalho é mostrar como é possível explorar técnicas avançadas de raciocínio lógico da Robótica Cognitiva em robôs Lego® MindStorms™, ou outros robôs para os quais seja possível estender o arcabouço Legolog.[1] Uma vez que robôs Lego® MindStorms™ têm sido usados em competições de Robótica no mundo inteiro e em disciplinas introdutórias de Robótica, o trabalho apresentado nesse artigo cria a possibilidade de novas modalidades de competições e propostas de cursos.

Na Seção 2, é descrita a linguagem Golog e suas extensões (ConGolog e IndiGolog) incluindo uma breve introdução ao Cálculo de Situações. Na Seção 3 é apresentado o robô Lego® MindStorms™ e seu pacote básico RIS (*Robotics Invention System™*), bem como Legolog - um arcabouço para programação lógica de robôs Lego® MindStorms™. Usando o Mundo do Wumpus como estudo de caso, a Seção 4 mostra as fases de construção de um agente usando Legolog: (i) a especificação em Cálculo de Situações das ações do agente; (ii) sua implementação em IndiGolog; (iii) a implementação das ações primitivas do agente no robô; e finalmente (iv) a representação física do ambiente do Mundo do Wumpus.

2 GOLOG: UMA LINGUAGEM PARA ROBÓTICA COGNITIVA

A tarefa de planejamento pode ser trivialmente definida como o seguinte problema: dado a tupla $\langle \mathcal{S}, s_0 \in \mathcal{S}, \mathcal{S}_G \subseteq \mathcal{S}, \mathcal{A} \rangle$, na qual $\mathcal{S}$ é o conjunto de estados, $\mathcal{A}$ é o conjunto de ações (transições), encontrar uma seqüência de ações (plano) que ao ser aplicada em s_0, leve o agente para um estado $s_g \in \mathcal{S}_G$ (estado meta). A complexidade dos algoritmos de planejamento está diretamente ligada à representação escolhida para cada um desses elementos da sua entrada, bem como às restrições associadas a cada ação.

A forma de representação mais usada na área de planejamento é através de variáveis de interesse, chamadas de fluentes. Assim, a definição de um problema é feita utilizando um conjunto de fluentes. Uma valoração para cada item desse conjunto representa um determinado estado. Esse tipo de representação permite expressar sinteticamente problemas com até milhões de estados. Porém, os algoritmos para problemas de planejamento em tais linguagens são ineficientes devido ao seu espaço de estados exponencial (Ghallab

[1] Um exemplo de possível extensão é para os robôs AIBO ™, produzidos pela Sony®, utilizando a interface disponível em `http://www.scs.ryerson.ca/~mes/gti/`

et al., 2004). Para diminuir o espaço de busca, os algoritmos de planejamento empregam: (a) técnicas para tratar conflitos entre ações, e assim transformar a *busca no espaço de estados* numa *busca no espaço de planos* (Kambhampati et al., 1995); (b) heurísticas (Bonet e Geffner, 1998) para orientar a busca de forma eficiente no espaço de estados; ou ainda (c) a modelagem de ações compostas, também chamadas de *ações de alto-nível*, para gerar um plano de ações através de decomposições sucessivas (*planejamento hierárquico*) (Erol et al., 1994).

Golog (Levesque et al., 1997), uma linguagem de programação para agentes inteligentes, permite especificar **restrições** para atingir um conjunto de estados meta desejado. Essas restrições são especificadas através de um *programa de alto-nível*, que pode ser definido como um programa no qual: (i) as instruções primitivas são as ações do agente para o domínio, descritas em Cálculo de Situações (Mccarthy, 1963); (ii) os testes envolvem condições (predicados ou fluentes) dependentes do domínio e que são afetados pelas ações do agente; (iii) os procedimentos, correspondem às ações compostas de planejamento hierárquico (Barros e Iamamoto, 2003); (iv) e ainda, pode conter escolhas não-determinísticas, onde é necessário fazer uma busca por um conjunto de ações futuras (*lookahead*) para garantir que o programa irá terminar com sucesso. Golog busca por uma seqüência de ações que gere uma execução válida do programa de alto-nível do agente. A principal vantagem em procurar uma execução válida de um **programa de alto-nível**, através de escolhas não- determinísticas de decomposições, é a redução do espaço de busca que deveria ser explorado se o agente raciocinasse diretamente sobre suas ações primitivas (forma de planejamento não-hierárquico). Assim as restrições fornecidas pelo programa do agente são usadas para podar o espaço de planos durante a busca por uma execução válida.

Variações da linguagem Golog permitem: (1) especificar programas com ações concorrentes (ConGolog (De Giacomo et al., 2000)); e (2) que ações sejam executadas durante a tarefa de planejamento (IndiGolog (Lespérance e Ng, 2000)), característica chamada aqui de **planejamento *on-line***.

2.1 O Cálculo de Situações

O Cálculo de Situações (Mccarthy, 1963) é um formalismo lógico baseado em lógica de primeira ordem, cuja ontologia inclui: *situações*, representando "fotos" do mundo; *fluentes*, que representam as propriedades do mundo que podem ser alteradas pelo agente; e *ações*, que são responsáveis por alterar o valor verdade dos fluentes, ou seja, que provocam mudanças no mundo. No Cálculo de Situações, a constante s_0 denota a *situação inicial*; a função $do(\alpha, \sigma)$ denota a *situação* resultante da execução da ação α na situação σ; o predicado $poss(\alpha, \sigma)$ representa que é possível executar a ação α na situação σ; e o predicado $holds(\phi, \sigma)$ representa que o fluente ϕ é válido na situação σ.

```
1  :- op(950,xfy,[&]).
2  :- op(500,xfy,[?]).
3  :- op(960,xfy,[|]).
4  :- op(960,xfy,[~]).
5  exec(A1 & A2,S1,S3) :-
6    exec(A1,S1,S2), exec(A2,S2,S3).
7  exec(P?,S,S):- holds(P,S).
8  exec(A1 | A2,S1,S2) :-
9    exec(A1,S1,S2) ; exec(A2,S1,S2).
10 exec(if(P,A1,A2),S1,S2) :-
11   exec(P? & A1 | ~P? & A2,S1,S2).
12 exec(star(E),S1,S2) :-
13   S1=S2;copy(E,E1),exec(E & star(E1),S1,S2).
14 exec(while(P,A),S1,S2) :-
15   copy(P,P1),exec(star(P? & A) & ~P1?,S1,S2).
16 exec(A,S1,S2) :- proc(A,A1), exec(A1,S1,S2).
17 exec(A,S,do(A,S)) :- prim(A), poss(A,S).
18 holds(A=A,_).
19 holds(~P,S) :- not holds(P,S).
20 holds(P,do(A,S)) :-
21   holds(P,S), not affects(A,P).
```

Figura 1: Uma implementação simplificada da linguagem Golog como um meta-interpretador Prolog.

Dada uma especificação de um domínio de planejamento no Cálculo de Situações, a solução para um problema de planejamento nesse domínio pode ser encontrada através de uma prova de teorema. Seja $\mathcal{A}$ o conjunto de axiomas que descrevem as ações do agente, $\mathcal{I}$ o conjunto de axiomas que descreve a situação inicial e $\mathcal{G}$ uma sentença lógica descrevendo a meta do agente. Dessa forma, a prova construtiva de $\mathcal{A} \wedge \mathcal{I} \models \exists S\ (legal(S) \wedge \mathcal{G}(S))$, onde $legal(S) \equiv poss(\alpha_1, s_0) \wedge \ldots \wedge poss(\alpha_n, do(\alpha_{n-1}, do(\ldots, do(\alpha_2, do(\alpha_1, s_0))\ldots)))$, tem como resultado a instanciação da variável S em um termo da forma $do(\alpha_n, do(\alpha_{n-1}, \ldots, do(\alpha_2, do(\alpha_1, s_0))\ldots))$. Essa situação corresponde à seqüência de ações $\langle\alpha_1, \ldots, \alpha_n\rangle$, que quando executada pelo agente na situação inicial s_0, o leva para uma situação que satisfaz a meta. Chamaremos esse tipo de planejamento, de **planejamento *off-line***, isto é, planejamento para metas de alcance (satisfação de estados meta) raciocinando-se diretamente sobre as ações primitivas do agente sem, no entanto, envolver a execução de execução de ações durante a elaboração do plano.

2.2 Golog: um meta-interpretador Prolog

Os programas Golog são executados por um provador de teoremas especializado (Figura 1) para lógica de primeira ordem (Levesque et al., 1997), implementado como um meta-interpretador Prolog. O usuário deve fornecer uma axiomatização $\mathcal{A}$, descrevendo as ações do agente (**conhecimento declarativo**), e um programa de controle c (programa de alto-

nível), especificando o comportamento desejado do agente (**conhecimento procedural**). A execução do programa Golog corresponde a prova construtiva de $\mathcal{A} \models exec(c, s_0, \sigma)$, onde $exec(c, s_0, \sigma) \equiv \exists\sigma(\sigma \wedge legal(\sigma))$ e σ é uma decomposição válida de c. Dessa forma, se uma situação $\sigma = do(\alpha_n, do(\alpha_{n-1}, \ldots, do(\alpha_2, do(\alpha_1, s_0)) \ldots))$ for encontrada pelo meta-interpretador Prolog, a seqüência de ações $\langle\alpha_1, \ldots, \alpha_n\rangle$ corresponde a uma execução válida que pode ser executada pelo agente.

Outra característica de Golog é ser um planejador *off-line*. Isso significa que toda a busca por uma seqüência de ações (execução válida do programa de alto-nível em questão) será realizada antes que qualquer ação seja executada pelo agente.

2.3 Cálculo de Situações e Golog

Como foi mencionado, um termo no Cálculo de Situações da forma $do(\alpha_n, do(\alpha_{n-1}, \ldots, do(\alpha_2, do(\alpha_1, s_0)) \ldots))$ corresponde ao plano $\pi = \langle\alpha_1, \ldots, \alpha_n\rangle$. Em Golog é introduzido o símbolo ";" usado na notação infixa para representar planos como uma composição sequencial de ações. Assim qualquer termo denotando uma ação é um plano e se π_1 é um plano e π_2 é um plano, $\pi_1; \pi_2$ também é um plano.

A execução de planos é definida no meta-interpretador Prolog pelo predicado $Exec$ (Figura 1). $Exec(\pi, s, s')$ é verdadeiro se um plano π leva da situação s para a situação s'. Seja α uma variável representando uma ação e π_1 e π_2 variáveis que representam planos, Golog usa as seguintes abreviações: $Exec(\alpha, s, s')$ para $Poss(\alpha, s) \wedge s' = do(\alpha, s)$ e $Exec(\pi_1; \pi_2, s, s')$ para $\exists s''(\alpha, s)(Exec(\pi_1, s, s'') \wedge Exec(\pi_2, s'', s'))$. Golog também generaliza a definição de plano através de três operadores: $\#$, o operador de escolha não-determinística de execução de ações; $*$, o operador de iteração não-determinística de execução de ações; e ?, o operador que testa se uma condição (fluente) é verdadeira.

Assim, é possível expressar qualquer comando de fluxo de uma linguagem imperativa em Golog, por exemplo, *if* ϕ *then* π_1 *else* π_2 através de $Exec((\phi?; \pi_1)|(\neg\phi?; \pi_2), s, s')$ (Figura 1, linha 10) e *while* ϕ *do* π_1 através de $Exec((\phi?; \pi_1)*; \neg\phi?, s, s')$ (Figura 1, linha 14).

O principal operador da família de linguagens Golog é o operador $Proc(\cdot, \cdot)$. Ele permite definir tarefas compostas através de decomposições sucessivas de um programa de alto-nível. A execução de um procedimento $Proc(A, A1)$ (Figura 1, linha 16) consiste em instanciar as variáveis livres de $A1$ que estão definidas em A e recursivamente executar $A1$, que é decomposto em outros procedimentos ou ações primitivas.

2.4 Extensões de Golog

Uma extensão possível para Golog é permitir a execução concorrente de planos. Esse dialeto é conhecido como Con-Golog (De Giacomo et al., 2000)) e além desses operadores são definidos outros dois: (i) o operador de concorrência com mesma prioridade ($\pi_1 || \pi_1$) entre planos; e o operador de concorrência com maior prioridade para π_1 ($\pi_1 \gg \pi_2$).

A linguagem usada nesse trabalho é chamada IndiGolog (Lespérance e Ng, 2000), uma extensão de Golog para resolver problemas que requerem a execução de ações durante o planejamento, isto é, planejamento *on-line*. Essa característica permite especificar programas capazes de fazer sensoriamento durante a busca por um plano solução e utilizar as percepções obtidas para guiar os novos passos dessa busca. No entanto, IndiGolog realiza planejamento exclusivamente *on-line* e para obter a combinação desses dois tipos de planejamento (*off-line* e *on-line*) é necessário estender o IndiGolog, como será descrito na Seção 4.4.

3 O ROBÔ LEGO® MINDSTORMS™

O robô Lego® MindStorms™ é parte do conjunto básico RIS (*Robotics Invention System™*), no qual o principal componente é o **bloco RCX** (*Robotic Commander Explorer*). Ele contém um microprocessador Hitachi H8/3297 de 16 bits, capaz de controlar até três atuadores e três sensores. Os atuadores são compostos por motores de cinco intensidades para ambas as direções, enquanto os sensores variam entre sensor de luz, temperatura, rotação e botões. Além disso, o RCX também contém uma porta de infravermelho, capaz de se comunicar com a *torre de infravermelho*. Esse último componente, que também integra o conjunto RIS, é ligado na porta serial de um computador para realizar a comunicação entre o bloco RCX e o computador.

Existem compiladores capazes de gerar código para a máquina virtual implementada no *firmware* do bloco RCX, que recebe o código compilado via infravermelho. Um exemplo de compilador desse tipo é o NQC (*not-quite C*), usado nesse trabalho. A linguagem aceita pelo compilador NQC possui comandos para configurar, ativar e desativar os atuadores, receber informações dos sensores, além de possuir concorrência nativa na linguagem. No pacote de *softwares* fornecido pelo RIS, há linguagens de programação visual, chamadas **Robolab** ou **LabVIEW**, nas quais é possível construir programas para o bloco RCX integrando figuras (como peças de quebra-cabeça) e definindo parâmetros para elas. Esse artigo apresenta uma abordagem alternativa a esses ambientes de programação: o uso de lógica formal para o desenvolvimento de programas para o bloco RCX.

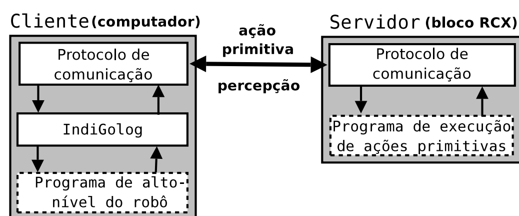

Figura 2: Modelo cliente/servidor do arcabouço Legolog.

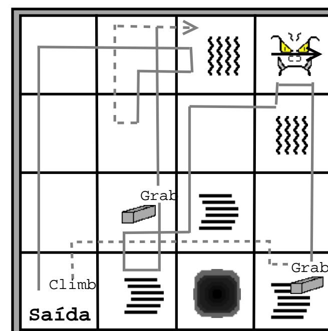

Figura 3: Uma instância do Mundo do Wumpus já explorada pelo agente. Linhas contínuas representam trajetórias de exploração do ambiente e linhas tracejadas representam trajetórias planejadas pelo agente para matar o Wumpus ou sair da caverna. Posições com linhas retas horizontais representam brisa, enquanto linhas onduladas verticais representam o cheiro do Wumpus. A flecha indica que o Wumpus foi morto. Esse mapa caracteriza uma instância do problema de nível de dificuldade alta, envolvendo a execução de 41 ações.

3.1 Legolog

Legolog (Levesque e Pagnucco, 2000) é um arcabouço para desenvolvimento de *software* para o robô Lego® MindStorms™. Esse arcabouço é composto pelo IndiGolog, que ficará instalado no computador, mais a implementação de um protocolo de comunicação entre o bloco RCX e o computador. Esse protocolo permite a troca, via infravermelho, de mensagens durante a execução de um programa no bloco RCX. Assim, Legolog pode ser usado para modelar aplicações do tipo cliente e servidor, em que o bloco RCX é o servidor de atuadores e sensores, e o computador (executando o IndiGolog) é o cliente. A Figura 2 mostra como uma arquitetura do tipo cliente/servidor decompõe um agente implementado usando Legolog, onde:

- **Cliente:** é o pacote executado pelo computador, composto pelo *programa de alto-nível do robô*, uma implementação do protocolo de comunicação (computador → RCX) e do IndiGolog.
- **Servidor:** é o pacote executado pelo bloco RCX, composto pelo p*programa de execução de ações primitivas* e por uma implementação do protocolo de comunicação (RCX → computador).

Dessa forma, um programa de controle de robô usando Legolog para uma determinada aplicação é dividido, basicamente, em duas partes (retângulos tracejados na Figura 2):

Programa de alto-nível do robô. Programa armazenado no computador e que, ao ser executado pelo IndiGolog, faz a geração incremental (*on-line*) de planos de ações primitivas, considerando as percepções do robô.

Programa de execução de ações primitivas. Programa implementado em NQC para ser executado no bloco RCX que especifica como serão executadas as ações primitivas e de sensoriamento (percepções) no robô Lego® MindStorms™.

Através desse modelo, o projetista é capaz de definir *o nível de abstração das ações delegadas ao RCX*, isso é, seu grau de autonomia. Assim, o RCX pode ser programado para simplesmente controlar suas entradas e saídas, tornando ações como *ligue o motor m no sentido s* em primitiva; ou para implementar ações mais complexas, como por exemplo, seguir uma linha, encontrar um objeto, permanecer dentro de uma área limitada, ou ainda, para realizar a tarefa complexa da função **get_boolean_sense**, detalhada na Seção 6. O mais interessante, do ponto de vista da Robótica Cognitiva, é permitir a definição de ações primitivas em um nível maior de abstração, como por exemplo *ir para a posição à frente*.

Como o Legolog já traz a implementação do protocolo de comunicação entre cliente e servidor, após definir o nível de abstração das ações delegadas ao RCX, a construção de um programa de um agente robótico usando Legolog resume-se em: implementar o programa de alto-nível do agente em IndiGolog e o programa de execução de ações primitivas em NQC. Além disso, o projetista deve criar a tabela que traduz o nome das ações em números inteiros e converter as percepções numéricas do robô em fluentes para o IndiGolog.

4 IMPLEMENTAÇÃO DE UM AGENTE USANDO LEGOLOG

Nesta seção, são apresentados os principais passos na construção de um agente usando Legolog, isto é, um agente implementado em IndiGolog para o robô Lego® MindStorms™. Para isso, foi selecionado o problema do Mundo do Wumpus (Russel e Norvig, 2003), um ambiente estático e determinístico com observação parcial que requer tomada de decisão sequencial. Uma solução para esse problema envolve planejamento e execução de ações, bem como a formulação de hipóteses sobre o estado do mundo.

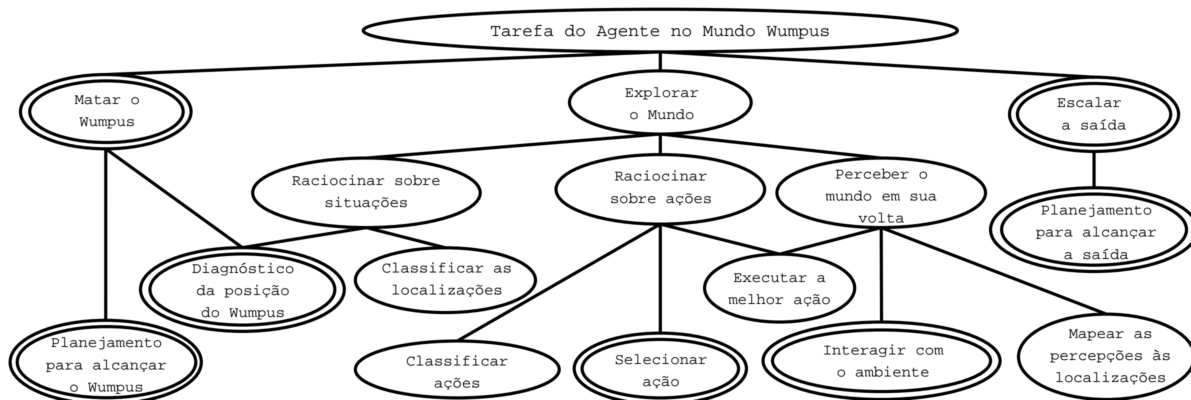

Figura 4: Modelo conceitual de um agente do Mundo do Wumpus. Nesse modelo, sub-problemas de mais alto nível são decompostos em outros mais simples.

4.1 Mundo do Wumpus: busca ao tesouro num ambiente hostil

O problema do Mundo do Wumpus consiste em um agente que deve explorar um reticulado, tendo acesso apenas à informação local. O agente tem como objetivo sair do reticulado, cercado por paredes, com a maior pontuação possível, sendo que para isso, ele deve encontrar o ouro escondido em algumas posições do reticulado, realizando o menor número de movimentos. O agente deve ainda desviar de abismos e do Wumpus, um monstro que poderá devorá-lo. A Figura 3 ilustra uma instância 4×4 resolvida do Mundo do Wumpus.

Um agente para o Mundo do Wumpus possui informação incompleta do mundo, uma vez que sua percepção sobre a presença de abismo (brisa) ou do Wumpus (cheiro), só é possível na adjacência de um abismo ou do Wumpus, respectivamente. Isso caracteriza um agente de observabilidade parcial do mundo. Por essa razão, o agente deve raciocinar através da geração de hipóteses durante a exploração do ambiente, para finalmente classificar as posições como perigosas ou seguras. Além disso, o Wumpus pode ser morto pelo agente com uma flecha atirada em sua direção.

Apesar do Mundo do Wumpus ser um problema estudado em cursos de introdução à Inteligência Artificial, o desenvolvimento do programa completo de um agente para esse domínio não é uma tarefa trivial. Por isso, cursos introdutórios de Inteligência Artificial não o resolvem por completo, por exemplo, em (Russel e Norvig, 2003) são feitas sugestões sobre como modelar somente parte dos comportamentos que um agente inteligente deve realizar.

4.2 Agente do Mundo do Wumpus: modelo conceitual

A Figura 4 mostra um modelo conceitual do agente do Mundo do Wumpus, identificando os diferentes sub-problemas que o agente deve resolver, bem como a relação entre eles. Neste modelo, também se aplica a idéia de decomposição de problemas. Note porém, que apesar de existir uma correspondência entre os sub-problemas e a descrição de ações do agente (ações compostas e ações primitivas), estabelecer essa correspondência não é trivial.

4.3 Agente do Mundo do Wumpus: modelo lógico e implementação

Chamamos de *modelo lógico do agente*, a especificação de suas ações em Cálculo de Situações e de seu programa em IndiGolog (também baseado em lógica). É interessante salientar uma das vantagens da abordagem da Robótica Cognitiva: o modelo lógico do programa do agente é igual à implementação do agente.

O modelo lógico para o agente do Mundo do Wumpus foi inicialmente baseada no livro *Artificial Inteligence: a Modern Approach* (Russel e Norvig, 2003). Porém, nem todos os sub-problemas apresentados do modelo conceitual (Figura 4) foram tratados pelos autores. Nessa figura, as ovais duplas indicam sub-problemas que não são comumente tratados em livros de Inteligência Artificial e que foram especialmente modelados e implementados para o agente deste projeto.

Os fluentes usados para modelar o problema do Mundo do Wumpus foram: **smelly(L)**, **breezy(L)**, **glitter(L)** os quais indicam, respectivamente, que a posição L possui o cheiro do Wumpus, a brisa de um abismo ou o brilho do ouro; **atAgent(L)** representando que o agente está na posição L; **agentDirection(D)** indicando a direção atual D do agente; **visi-**

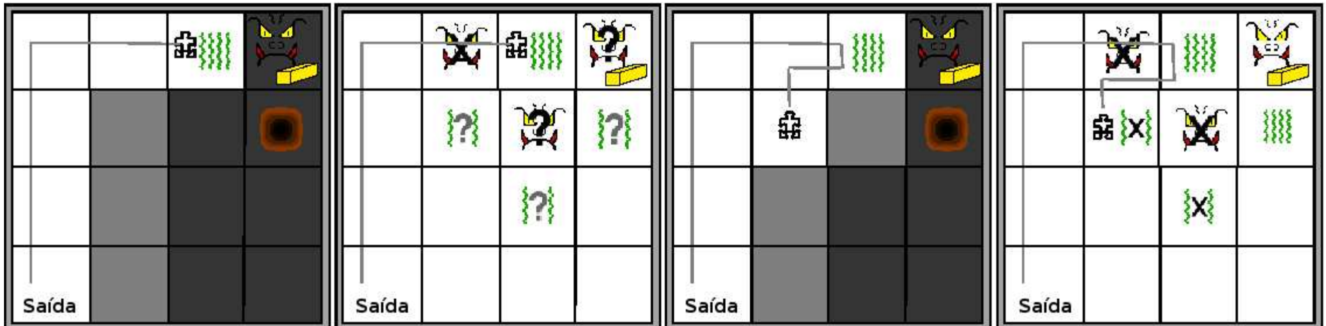

Figura 5: Quatro mapas representando duas situações do agente no Mundo do Wumpus. O primeiro e o terceiro mapa mostram o que o agente já sabe sobre o ambiente: posições brancas indicam casas já visitas; posições pretas indicam casas não visitadas; e posições cinzas indicam casas não visitadas mas inferidas como seguras (i.e, que não possuem percepções de perigo adjacente). O segundo mapa mostra o raciocínio hipotético do agente sobre o primeiro mapa, enquanto o quarto faz o mesmo para o terceiro mapa. Posições marcadas com "?" indicam possíveis localizações do monstro ou de percepções de cheiro; posições marcadas com um "X", indicam que uma hipótese é falsa.

ted(L) e **secure(L)** os quais sinalizam respectivamente que a posição L foi visitada ou que ela é uma localização segura; e **holding(O)**, indica que o agente está segurando o objeto O.

A partir desses fluentes, é possível especificar o conjunto de axiomas do Cálculo de Situações para o agente do Mundo do Wumpus. Esses axiomas são divididos em: (i) *axiomas de estado inicial* $\mathcal{I}$, que descrevem o estado inicial do mundo e (ii) *axiomas de estado sucessor* $\mathcal{A}$, que representam como os fluentes são alterados ou permanecem inalterados com as ações do agente. Por exemplo, o axioma de estado sucessor para o fluente **smelly(L)** é:

```
holds(smelly(L), do(A,S)) :-
  holds(smelly(L), S);
  A==foward, stench(do(A,S)), holds(atAgent(L), do(A,S)).
```

Ele define uma posição **L** como *smelly* na situação **do(A,S)** se **L** já era *smelly* em **S** ou se o agente recebeu a percepção do cheiro do Wumpus ao visitar a posição **L** usando a ação **A**. Da mesma forma, devem ser definidos axiomas (não detalhados aqui por motivo de espaço) para especificar como as ações **turn_clokwise**, **turn_anti_clockwise**, **foward**, **grab**, **shoot**, **percept** e **climb**, afetam os demais fluentes.

4.4 Complementando o modelo lógico: procedimentos em IndiGolog

Para implementar o programa de alto-nível do agente, além dos axiomas do Cálculo de Situações, é necessário resolver os seguintes sub-problemas:

Classificar ações. O conjunto de ações (primitivas ou compostas) que podem ser executadas a partir de uma dada situação são classificadas em **Great**, **Good**, **Medium**, **Risky** e **Deadly**. Essa classificação é implementada por procedimentos em IndiGolog e sua modificação implica em diferentes comportamentos do agente, como por exemplo

```
proc(greatAction,
  if(holding(gold),
    planning([0,0], [climb | []]),
    [sense(glitter), grab] # [tryToKillWumpus])).
```

que define um agente cuja ação de maior prioridade (**greatAction**) é planejar a saída da caverna quando ele já possuir pelo menos uma barra de ouro. Caso contrário, o agente deve pegar o ouro quando receber a percepção de brilho. Finalmente, se nenhuma das ações anteriores for possível, o agente deve tentar matar o Wumpus, caso ele consiga inferir a posição do monstro.

Executar a melhor ação. Para isso foi definido o procedimento IndiGolog **find_action** que seleciona, de modo não-determinístico, qual é a próxima melhor ação a ser executada, de acordo com a classificação de ações. Como IndiGolog realiza planejamento *on-line*, após a melhor ação ser escolhida, ela já é executada.

Diagnóstico da posição do Wumpus e de abismos. Um dos aspectos mais interessantes do problema do Mundo do Wumpus é determinar a posição do Wumpus e de abismos com observação parcial do mundo. Determinar a posição do Wumpus é fundamental para que o agente possa matá-lo, pois ele possui uma única flecha (Figura 5). A determinação de abismos também é importante, pois pode significar ao agente encontrar ou não o ouro (por exemplo, caso a localização do ouro esteja cercada de percepções de abismos), ou ainda, ser útil para descobrir novas posições seguras e minimizar sua navegação.

Esses sub-problemas são chamados de diagnóstico uma vez que, determinar as posições do Wumpus ou dos abismos,

pode *explicar* as observações do agente em situações passadas. O procedimento IndiGolog **tryToKillWumpus**

```
proc(tryToKillWumpus,
  [?(holding(arrow)),
   consultKB(smellyPositions(SmellyPos)),
   startSet(WumpusPos),
   findWumpus(SmellyPos, WumpusPos),
   planKillWumpus(WumpusPos)]).
```

especifica que para tentar matar o Wumpus é necessário que o agente consulte sua base de conhecimento (**consultKB**) e recupere uma lista de posições (pares de coordenadas) em que ele percebeu o cheiro do Wumpus (**SmellyPos**) e, situações passadas. Em seguida, é feita uma chamada ao procedimento **findWumpus** para a geração e discriminação de uma lista de hipóteses sobre as posições possíveis do Wumpus (**WumpusPos**). Se uma lista com apenas uma posição for devolvida, o procedimento **planKillWumpus(WumpusPos)** faz com que o agente planeje para alcançar uma posição adequada para atirar a flecha e, finalmente, matar o Wumpus. Caso, contrário, o agente decide não arriscar perder sua única flecha antes de explorar mais o ambiente.

O procedimento Golog **findWumpus** a seguir, faz a geração e discriminação de hipóteses sobre a posição do Wumpus.

```
proc(findWumpus([[SmeX, SmeY] | SmePos], WumpusPos),
  [abductWumpusAt([SmeX, SmeY+1], south, WumpusPos),
   abductWumpusAt([SmeX, SmeY-1], north, WumpusPos),
   abductWumpusAt([SmeX+1, SmeY], west, WumpusPos),
   abductWumpusAt([SmeX-1, SmeY], east, WumpusPos),
   if(SmePos = [],
       [cut(findWumpus(SmePos, WumpusPos))],
       [endSet(WumpusPos)])]).
```

Nesse procedimento, para cada posição **[SmeX, SmeY]** em que o cheiro do Wumpus foi percebido, suas adjacências são consideradas hipóteses de localização do Wumpus. As hipóteses geradas são discriminadas através do procedimento **abductWumpusAt**:

```
proc(abductWumpusAt([WumX, WumY], IgnDir, WumPos),
  [if(secure([WumX, WumY]), [],
   if(wall([WumX, WumY]), [],
    [possibleWumpusPos([WumX, WumY], IgnoreDir),
    addToSet(WumpusPos, [WumX, WumY])] # [?(inCave)]))]).
```

Inicialmente, é verificado se a posição **[WumX, WumY]** já foi classificada como segura. Caso contrário, o procedimento **possibleWumpusPos** analisa as três posições adjacentes, excluindo a posição na direção *IgnDir* já visitada.

```
proc(possibleWumpusPos([WumX, WumY], TrueStench),
  [?(TrueStench == north) # [
    consultKB(smelly([WumX, WumY + 1])) #
      ?(-visited([WumX, WumY + 1]))],
   ?(TrueStench == south) # [
    consultKB(smelly([WumX, WumY - 1])) #
      ?(-visited([WumX, WumY - 1]))],
   ?(TrueStench == east) # [
    consultKB(smelly([WumX + 1, WumY])) #
      ?(-visited([WumX + 1, WumY]))],
   ?(TrueStench == west) # [
    consultKB(smelly([WumX - 1, WumY])) #
      ?(-visited([WumX - 1, WumY]))]]).
```

Nesse procedimento é verificada a hipótese do Wumpus estar na posição **[WumX, WumY]**, analisando as percepções passadas para detectar conflitos. Um conflito é detectado se uma posição adjacente a **[WumX, WumY]**, que já foi visitada anteriormente, não tenha sido marcada como *smelly*.

Planejamento para alcançar a saída, o Wumpus e posições não-visitadas Para alcançar essas metas, o agente deve atingir uma determinada posição já conhecida, a saber: a saída, uma posição na direção do Wumpus e uma posição segura não-visitada, respectivamente. Para obter a garantia que o agente chegará em tais posições, o caminho escolhido deve ser *seguro*, ou seja, todas as posições contidas nele já devem ter sido marcadas como seguras. A existência de pelo menos um caminho seguro para qualquer posição já visitada é trivialmente provada através da relação entre posição visitada e segura: toda posição visitada é segura e caso essa posição não tenha marcação de cheiro do Wumpus ou brisa, então todas as posições na sua adjacência são seguras também. Outra vantagem em utilizar caminhos seguros é não precisar sensoriar o ambiente para algumas metas do agente, o que resulta em um problema de planejamento *off-line*.

Como foi dito na Seção 2.2, IndiGolog é um planejador exclusivamente *on-line*, o que obriga alterar sua implementação em Prolog para adicionar um novo operador, chamado **planning**. Essa nova primitiva equivale à seguinte relação: $Exec(\text{planning}(pos, \pi_{af}), s, s') \equiv \exists \pi_{sol}, s''(Exec(\pi_{sol}, s, s'') \wedge Exec(\pi_{af}, s'', s') \wedge holds(\text{agentAt}(pos), s''))$, onde pos é a posição que se deseja atingir, π_{af} é o plano que deve ser executado após chegar na posição pos e π_{sol} é o plano obtido de forma *off-line* que leva o agente até a posição pos. Um exemplo de chamada do planejador *off-line* ocorre no procedimento descrito na Seção 4.4, em que o comando **planning([0,0], [climb | []])** é executado pelo IndiGolog.

O algoritmo de busca utilizado para implementar esse planejador é o de busca em profundidade iterativa no espaço de estados (Pereira e Barros, 2004). Sua utilização é feita através da consulta Prolog "`plan(S),exec(S),holds(agentAt[Goal_Pos],S)`", onde **plan(S)** e **exec(S)** são definidos como:

```
exec(s0).
exec(do(A,S)) :- poss(A,S), exec(S).
plan(s0).
plan(do(A,S)) :- plan(S).
```

Esse algoritmo é usado, juntamente com os axiomas de Cálculo de Situações, para inferir que $\exists s \; (plan(s) \wedge exec(s) \wedge G)$, sendo G uma descrição do estado meta. A principal característica desse algoritmo é que ele é ótimo em relação ao tamanho do plano, ou seja, no problema do Mundo do Wumpus ele sempre devolverá o plano que realiza o menor caminho até a meta. Apesar desse algoritmo ter as características desejadas, também é possível usar outros planejadores mais

eficientes, baseados em lógica, que realizam busca no espaço de planos (Pereira e Barros, 2004).

5 UMA MAQUETE PARA O MUNDO DO WUMPUS

Para construir uma maquete do ambiente do Mundo Wumpus foi necessário criar um modo de representar o ambiente considerando os sensores disponíveis do robô Lego® MindStorms™. Como só foi utilizado o sensor de contraste, foram feitas diferentes marcações com diferentes contrastes para que o robô fosse capaz de se movimentar pelo reticulado e captar as percepções de cada posição.

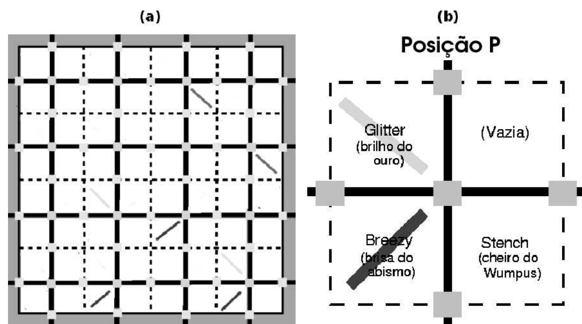

Figura 6: (a) Representação do mundo da Figura 3; e (b) das percepções da posição (0,3) da Figura 3, codificada em quadrantes.

Na Figura 6, as linhas pontilhadas delimitam cada posição do Mundo do Wumpus, já as linha contínuas (linhas guias com marcações de contraste nas intersecções) representam os caminhos possíveis para o robô se movimentar.

As **marcas de percepção** foram criadas para representar as três percepções possíveis para o Mundo do Wumpus (**glitter**, **stench** e **breezy**). Para isso, todas as posições do ambiente foram divididas em quadrantes, aproveitando as linhas guias como delimitadores, onde cada quadrante é responsável por representar uma percepção (Figura 6 (b)).

O terceiro tipo de marca, as **marcas de giro**, são usadas pelo algoritmo de percepção para realizar um giro de 360° dentro de uma mesma posição enquanto o robô procura por marcas de percepção nos quadrantes da posição P. Dessa forma, essas marcas representam que o robô alinhou seu eixo de rotação com o centro da posição P.

Para confeccionar a representação do Mundo do Wumpus, seguindo o modelo acima, foi utilizado lona preta de construção para o fundo e fita crepe para as linhas guias, gerando um alto contraste esses itens. A maquete final da representação do Mundo do Wumpus pode ser vista na Figura 8.

```
void(get_boolean_sense) {
  int light_dif;
  //Reiniciando a variável global
  boolean_sense = FALSE;
  //Girando no fundo até sair da marca de giro
  turnToExitMark(bg_dif_min, bg_dif_max, 1);
  light_dif = light_sensor - base_value;
  //Verificando se ele encontrou a percepção
  if(light_dif >= sense_dif_min &&
                  light_dif <= sense_dif_max) {
    //Percepção encontrada
    boolean_sense = TRUE;
    PlaySound(SOUND_UP);
    //Saindo da marca de percepção e indo para o fundo
    turnToFindMark(bg_dif_min, bg_dif_max, 1);
    //Procurando a linha guia ou a marca de giro
    turnToFindMark(line_dif_min, mark_dif_max, 1);
  } //Se não foi encontrada uma marca,
    //o robô já está na linha
}
```

Figura 7: Programa em NQC que executa um giro de 90° graus procurando por percepções.

6 IMPLEMENTAÇÃO DAS AÇÕES PRIMITIVAS NO ROBÔ

Na decomposição escolhida para a implementação do Mundo do Wumpus usando Legolog, a captação das percepções de uma posição (**glitter**, **stench** e **breezy**), de forma não ordenada, é implementada através da ação primitiva, **percept**. Dessa forma, a percepção do agente é feita de modo autônomo pelo RCX.

Para o robô captar todas as percepções de uma posição, ele deve realizar um giro de 360° coletando as percepções dos quadrantes e depois retornar à posição original. Assim, a ação **percept** é implementada realizando quatro iterações da função **get_boolean_sense** (Figura 7), devolvendo um vetor de quatro *booleans* para o computador, codificado em um número inteiro (Figura 2).

Como o robô pode atingir uma determinada posição por quatro direções diferentes (norte, sul, leste e oeste), o programa de alto-nível do agente usa a direção atual do robô para traduzir o vetor de *booleans* nos fluentes **smelly(L)**, **breezy(L)**, **glitter(L)**, onde L é a sua posição atual. Parte dessa tradução é feita executando até três *shifts* circulares no vetor recebido para recuperar a percepção em uma ordem padrão.

Além da ação **percept**, as seguinte ações foram definidas como primitivas e implementadas no bloco RCX: **turn_clokwise**, **turn_anti_clockwise**, **foward**, **grab**, **shoot** e **climb**. Dessas ações, apenas as três primeiras movem o robô, avançando uma posição, girando 90° no sentido horário e anti-horário, respectivamente. Para as outras ações, **grab**, **shoot** e **climb**, o robô apenas emite sons indicando que elas foram executadas e esperando que o ambiente seja

Figura 8: (a) Robô Lego® MindStorms™ , torre de infravermelho e parte da maquete do Mundo do Wumpus. (b) Representação da maquete na parte (a) usando o padrão estabelecido na Figura 6, onde **A** representa uma posição com um abismo e **R** a posição onde está o robô. (c) Representação da parte (b) usando o padrão estabelecido na Figura 3.

Número da Instância	1	2	3
Posição do Wumpus	[3,3]	[0,2]	[3,1]
Posição do Ouro	[1,1]	[1,2]	[3,0]
Posições dos Abismos	[2,0]	[3,1], [3,3]	
Passos da solução	22	25	18
Tempo médio de decisão para ações *on-line*	0,03s	1,68s	0,05s
Desvio padrão da decisão para ações *on-line*	0,01s	0,42s	0,03s
Tempo médio para elaboração *off-line* de planos	0,03s	1,35s	0,13s
Desvio padrão da elaboração *off-line* de planos	0,01s	1,30s	0,10s
Tempo total gasto na escolha de ações	0,38s	13,25s	9,16s

Tabela 1: Estatísticas para alguns instâncias 4×4 do Mundo do Wumpus com apenas um ouro e probabilidade de 0.2 de uma posição P ser um abismo.

atualizado manualmente.

7 RESULTADOS EXPERIMENTAIS

A Tabela 1 exibe três instâncias do problema do Mundo do Wumpus, com as posições do Wumpus, do ouro e dos abismos especificados nas linhas 2, 3 e 4, respectivamente. Em todos os experimentos feitos, o robô detectou corretamente as posições seguras e perigosas, bem como encontrou o plano ótimo para se aproximar do Wumpus e sair da caverna. Essas capacidades também podem ser verificadas formalmente através da especificação lógica do agente. Os experimentos também mostram que para um plano de tamanho 22, 25 e 18 passos, o tempo médio para a seleção *on-line* de ações foi menor que 1,7 segundos, enquanto o tempo médio para a geração de planos *off-line* foi menor que 1,4 segundos. Para um método de planejamento baseado em um formalismo lógico, sem o uso de heurísticas, esses resultados representam um bom desempenho e demonstram como essa abordagem é viável para testar e investigar teorias formais da Robótica Cognitiva e que é possível a criação de novas modalidades de competições de robôs que envolvam problemas considerados desafios na área de Inteligência Artificial.

8 TRABALHOS CORRELATOS

Tradicionalmente, comparar formalismos para raciocínio sobre ações não é uma tarefa trivial. No entanto, essa tarefa é importante para que seja identificado os pontos fracos e fortes de cada abordagem. Quando essa comparação é feita através de um domínio de teste, é necessário definir métricas precisas de comparação. Por exemplo, a pontuação de um agente para o Mundo do Wumpus pode ser usada como métrica: (i) dois agentes devem ser comparados sob o mesmo conjunto de instâncias do problema, (ii) o tempo gasto para resolver uma mesma instância deve ser medido através dos mesmos procedimentos.

A seguir, é feita um breve discussão sobre outras soluções para o problema do Mundo do Wumpus, propostas recentemente para o tema especial do NMR06 (*International Workshop on Non-Monotonic Reasoning*, 2006) e no NRAC05 (*Sixth Workshop on Nonmonotonic Reasoning, Action, and Change*, 2005).

O trabalho descrito em (Sardina e Vassos, 2005) também propõe uma solução baseada em IndiGolog. Porém, para fazer o diagnóstico de posições (Seção 4.4), essa abordagem propõe algumas modificações no IndiGolog para tratar informação incompleta. Essas modificações são feitas através da extensão da ontologia do Cálculo de Situações: *fluentes* podem ter *valores possíveis* em uma situação; e *known* é um predicado criado para representar que um fluente possui apenas um, e só um, valor em uma dada situação. A abordagem proposta nesse artigo difere de (Sardina e Vassos, 2005) em dois aspectos: (1) não é feita nenhuma extensão no Cálculo

de Situações; (2) para o raciocínio de diagnóstico de posições foram feitas alterações no meta-interpretador Prolog ao invés de alterar a ontologia do Cálculo de Situações.

Uma outra implementação baseada em lógica é dada em (Thielscher, 2005), que usa a linguagem FLUX: uma linguagem da Robótica Cognitiva baseada no Cálculo de Fluentes (Thielscher, 2000). O Cálculo de Fluentes é uma extensão do Cálculo Situações no qual um *estado* S é representado por um conjunto de fluentes que são verdadeiros em S. Essa característica combinada à programação com restrições torna FLUX uma linguagem interessante para Robótica Cognitiva. No entanto, através de alguns experimentos com o agente FLUX para o Mundo do Wumpus foi possível verificar que a instância do problema ilustrada na Figura 4 (que é resolvida pelo agente desse artigo) não foi completamente resolvida, isto é, o agente FLUX não mata o Wumpus para conseguir pegar a barra de ouro.[2] Os autores em (Thielscher, 2005) apontam algumas melhorias futuras que podem ser feitas para resolver tal limitação.

Por último, a implementação proposta em (Shapiro e Kandefer, 2005) usa a linguagem SNePS (Shapiro, 1989) e tem como objetivo propor modelos gerais de inteligência ao invés de desempenho computacional. Assim, o agente SNePS proposto para o Mundo do Wumpus, estende a proposta original do jogo, com a idéia de *temperamento* e tomada de decisões assumindo maior risco. É importante notar que nenhuma das abordagens citadas anteriormente implementam suas soluções em ambientes reais, como é a proposta desse artigo.

9 CONCLUSÕES

Nesse trabalho, foi apresentado como implementar um programa de alto-nível requerido por um agente de planejamento com informação incompleta para o exemplo do Mundo do Wumpus. A solução proposta foi implementada para um ambiente não simulado usando Legolog: um arcabouço para desenvolvimento de software que contém o IndiGolog e a implementação de um protocolo de comunicação entre robôs Lego® MindStorms™ e computadores. O agente implementado foi capaz de resolver instâncias de problemas que outras propostas baseadas em Robótica Cognitiva apresentadas (*Sixth Workshop on Nonmonotonic Reasoning, Action, and Change*, 2005) não foram capazes de resolver. Em termos práticos, esse trabalho mostra como é possível intercalar a execução de ações de alto-nível (procedimentos IndiGolog) com ações primitivas descritas no Cálculo de Situações e implementadas no robô Lego® MindStorms™.

Como trabalhos futuros, pretende-se: (1) explorar o uso de um PDA (*Personal Digital Assistent*) embarcado no robô Lego® MindStorms™, com o objetivo de integrar cliente/servidor e assim possibilitar maior grau de autonomia ao robô; e (2) usar o IndiGolog em robôs mais robustos e como maior capacidade de processamento.

[2] Para realizar esses testes foi usada a implementação disponível em `http://www.fluxagent.org/demos.htm`

AGRADECIMENTOS

Esse projeto foi financiado pela FAPESP (processo 03/08311-3), e pelo CNPq (processo 308530/03-9).

REFERÊNCIAS

Barros, L. N. e Iamamoto, E. (2003). Planejamento de tarefas em golog, *SBAI*.

Bonet, B. e Geffner, H. (1998). HSP: Heuristic Search Planner, *Proc. of AIPS*.

Boutilier, C., Reiter, R., Soutchanski, M. e Thrun, S. (2000). Decision-theoretic, high-level agent programming in the situation calculus, *Workshop on Decision-Theoretic Planning, Proc. KR*.

De Giacomo, G., Lespérance, Y. e Levesque, H. (2000). ConGolog, a concurrent programming language based on the situation calculus, *Artificial Intelligence* **121**(1–2): 109–0.969.

Erol, K., Hendler, J. A. e Nau, D. S. (1994). UMCP: A sound and complete procedure for hierarchical task-network planning, *Proc. AIPS*, pp. 249–254.

Ghallab, M., Nau, D. e Traverso, P. (2004). *Automated Planning: Theory and Practice*, Morgan Kaufman, San Francisco, CA.

International Workshop on Non-Monotonic Reasoning (2006). Lakes District, England.

Kambhampati, S., Knoblock, C. A. e Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning, *Artificial Intelligence* **76**: 167–238.

Lespérance, Y. e Ng, H. (2000). Integrating planning into reactive high-level robot programs, *Proc. of the 2nd International Cognitive Robotics Workshop*.

Levesque, H. e Pagnucco, M. (2000). Legolog: Inexpensive experiments in cognitive robotics, *Proc. of the 2nd International Cognitive Robotics Workshop*.

Levesque, H., Reiter, R., Lespérance, Y., Lin, F. e Scherl, R. (1997). GOLOG: A logic programming language for dynamic domains, *JLP* **31**: 59–84.

Liu, Y., Lakemeyer, G. e Levesque, H. J. (2004). A logic of limited belief for reasoning with disjunctive information, *Proc. KR*.

Mccarthy, J. (1963). *Situations, actions and causal laws*, MIT Press.

Pereira, S. L. e Barros, L. N. (2004). Formalizing planning algorithms: a logical framework for the research on extending the classical planning approach, *Proc. of the ICAPS Workshop: Connecting Planning Theory with Practice*.

Russel, S. e Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*, 2nd. edn, Prentice-Hall, Inc.

Sardina, S. e Vassos, S. (2005). The Wumpus World in IndiGolog: A preliminary report, *Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'05), IJCAI*.

Shapiro, S. C. (1989). The cassie projects: An approach to natural language compentence, *in* J. P. Martins e E. M. Morgado (eds), *EPIA 89: Proc. of the 4th Portuguese Conference on Artificial Intelligence*, Springer, Berlin, Heidelberg, pp. 362–380.

Shapiro, S. C. e Kandefer, M. (2005). A SNePS Approach to The Wumpus World Agent or Cassie Meets the Wumpus, *Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'05), IJCAI*.

Sixth Workshop on Nonmonotonic Reasoning, Action, and Change (2005). Edinburgh, UK.

Thielscher, M. (2000). The fluent calculus: A specification language for robots with sensors in nondeterministic, *Technical Report CL-2000-01*, Artificial Intelligence Institute, Department of Computer Science, Dresden University of Technology.

Thielscher, M. (2005). A FLUX agent for the Wumpus World, *Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'05), IJCAI*.

Trevizan, F. W., Cozman, F. G. e de Barros, L. N. (2007). Planning under risk and knightian uncertainty, *Proc. of the 20th IJCAI*, AAAI (A ser publicado).

Trevizan, F. W. e de Barros, L. N. (2004). Robótica cognitiva, *Relatório técnico de cniciação científica FAPESP 308530/03-9-0.9*, Universidade de São Paulo, Departamento de Ciência da Computação.