UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
Departamento de Ciência da Computação

MAC5701 - Tópicos em Ciência da Computação

Cálculo de Situações, Golog e Revisão de Crenças aplicados a Robôs móveis autônomos.

Eduardo Ribeiro de Castro

Orientadora: Renata Wassermann

20 de junho de 2005

1 Introdução

A construção de um robô móvel autônomo envolve a integração de vários tipos de sistemas, sendo cada um deles uma área inteira para estudos. Módulos de controle de sensores e atuadores, algoritmos de análise de imagens e demais sinais externos, planejamento de ações, e a comunicação/interação com outros agentes, são alguns exemplos.

Dentre todos esses sistemas, o módulo de planejamento do robô pode ser considerado, de certa forma, como o centro da inteligência do robô. Ele é o responsável pela tomada de decisões sobre quais ações o robô deve executar para atingir os seus objetivos. A partir de um conjunto de informações sobre o domínio onde o robô atua, o planejador deve ser capaz de prever as conseqüências de cada seqüência de ações e escolher aquela seqüência que melhor satisfizer a meta desejada.

As informações utilizadas pelo planejador podem ser dados pré-definidos, gravados desde o início na memória do robô, ou dados coletados posteriormente, ao longo da interação do robô com ambiente. No caso de um robô móvel usado para vigilância predial, informações pré-definidas poderiam ser, por exemplo, a lista de características dos sensores e atuadores, a planta do prédio, e a definição do próprio objetivo do robô, o que ele entende por "vigiar". Informações coletadas posteriormente seriam o histórico de todas as ações e observações já efetuadas até o momento. Com base nesses dados, o robô poderia saber que salas ele já visitou, quais faltam visitar e a partir disso traçar uma rota adequada.

Para que um robô seja realmente útil, ele deve ser capaz de se adaptar às mudanças do ambiente exterior. Ele pode fazer isso coletando, esporádica ou continuamente, informações de seus sensores, incorporando-as ao seu banco de dados e checando se as novas informações são compatíveis com o plano já definido. Se for necessário, o robô deve reagir a estas mudanças refazendo os seus planos.

Um problema interessante ocorre quando as informações novas conflitam com as já registradas, tornando o banco de dados inconsistente. O fato de nenhum sensor ser 100% confiável torna este tipo de situação bastante provável. O robô vigilante, por exemplo [4], pode entrar em uma sala e enxergar uma porta onde, de acordo com o mapa, não haveria nada. Nesta situação, ele pode chegar a uma das seguintes conclusões:

a) Não há porta nenhuma, é só uma "ilusão" causada por mau funcionamento do sensor.
b) O mapa está errado e deve ser corrigido. A porta existe e sempre existiu.
c) A porta, de alguma forma, apareceu na parede. O ambiente não é estático como se supunha
d) O robô entrou na sala errada.

Em cada uma dessas conclusões alguma informação nova ou antiga teve que ser alterada no banco de dados para mantê-lo consistente. Não apenas o plano do robô deve ser refeito, mas próprio conhecimento no qual o plano se baseia. Algum critério teve que ser adotado para escolher qual das conclusões possíveis é a mais adequada. O problema de como definir este critério é conhecido como o problema de "Revisão de Crenças".

Nesta monografia foi estudado um dos formalismos clássicos para representação de conhecimento sobre ações: o Cálculo de Situações. Também foi estudada a linguagem Golog, uma linguagem de programação relativamente recente, baseada no cálculo de situações, que foi criada com o objetivo principal de programar robôs. A intenção é usar o

cálculo de situações e a Golog como pontos de partida para atacar o problema da revisão de crenças em robôs.

2 Cálculo de Situações

2.1 Elementos do Cálculo de Situações

O cálculo de situações (McCarthy [12]) foi um dos primeiros formalismos criados para representar os conceitos de causa, efeito e mudança de estado em lógica de primeira ordem. Ele permite definir axiomas de pré-condições e efeitos para as ações do robô, através dos quais é possível gerar automaticamente um plano de ação.

Os elementos básicos do cálculo de situações [8] são os seguintes:

- Ações: termos lógicos que representam as ações do robô/agente, como por exemplo *MoverPara(x)* ou *Parar*.
- Situações: termos lógicos que representam a situação inicial do universo (normalmente chamada de S_0) e todas as situações subseqüentes resultantes da aplicação de ações.
- Fluentes: funções ou predicados que variam de uma situação para outra. Por exemplo, *RoboEm(x,s)*. Como fluentes dependem das situações, por convenção a situação é o último argumento do fluente.
- Termos estáticos ou atemporais: outros termos lógicos, que não variam com as situações. Por exemplo: *Conectadas(Sala1,Sala2)*.

A transição de uma situação para a seguinte é sempre o resultado de uma ação. A função especial *do* serve para representar esta transição: *do(a,s)* representa a situação resultante da aplicação da ação *a* na situação *s*.

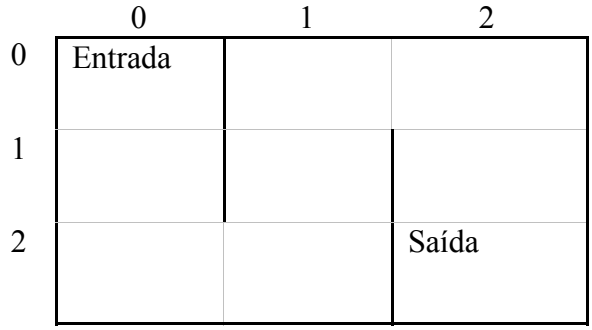
A possibilidade de uma ação ser executada é representada pelo predicado *Poss*. *Poss (a,s)* é verdadeiro se e somente se a ação *a* pode ser executada na situação s.

Com esses elementos básicos é possível definir teorias sobre como o mundo evolui em conseqüência das ações executadas. As teorias conterão os seguintes tipos de axiomas:

- Axiomas de estado inicial: descrevem o estado de alguns ou todos os fluentes no estado inicial S_0.
- Axiomas de Pré-Condições: descrevem os requisitos para execução de ações, pelo predicado *Poss*.
- Axiomas de estados sucessores: descrevem como cada fluente muda de uma situação para a seguinte em conseqüência das ações.
- Axiomas atemporais: descrevem os demais elementos do mundo que não dependem de ações ou situações.

Para exemplificar o uso do cálculo de situações, consideremos um exemplo bem simples, de um robô tentando sair de um labirinto. O labirinto é representado como uma

matriz 3x3 de quadrados. Pode haver paredes separando quadrados adjacentes. O robô começa na entrada do labirinto (posição (0,0)) e quer chegar na saída (posição (2,2)).



Em cálculo de situações, o problema pode ser descrito assim:

Constantes:

S_0: situação inicial
$C_{00}, C_{01}, C_{02}, C_{10}, C_{11}, C_{02}, C_{20}, C_{21}, C_{22}$: células do labirinto.
R: robô.

Fluentes:

Em(o,x,s): objeto o está na célula x na situação s.

Outros Predicados:

Ligação(x,y): as células x e y são adjacentes e não há paredes entre elas.

Ações:

Mover(o,x,y): mover o objeto o da célula x para a célula y.

Os axiomas para o problema são:

a) Axiomas atemporais

$Ligação(C_{00}, C_{01})$
$Ligação(C_{01}, C_{02})$
$Ligação(C_{02}, C_{12})$
$Ligação(C_{12}, C_{11})$
$Ligação(C_{11}, C_{10})$
$Ligação(C_{10}, C_{20})$
$Ligação(C_{20}, C_{21})$
$Ligação(C_{21}, C_{22})$

(Forma do Labirinto – quais células adjacentes estão ligadas)

b) Axiomas de estado inicial

$Em(R,C_{00},S_0)$

(Robô começa na posição (0,0))

c) Axiomas de Pré-Condições

$Poss(Mover(o,x,y),s) \equiv Em(o,x,s) \wedge (Ligação(x,y) \vee Ligação(y,x))$

(Um objeto só pode se mover para células ligadas à célula onde está)

d) Axiomas de Estados sucessores

$$Em(o,x,do(a,s)) \equiv \exists z(a = Mover(o,z,x)) \vee (Em(o,x,s) \wedge \neg \exists z(a = Mover(o,z,x)))$$

(Um objeto só estará em uma célula se ele se moveu para lá ou se já estava lá na anteriormente)

Se alimentarmos um provador de teoremas com estes axiomas, é possível fazer algumas consultas sobre o domínio do problema, tais como:

$Ligação(C_{00},C_{10})?$
$Poss(Mover(R,C_{00},C_{01}), S_0)?$

Se quisermos gerar um plano de ação, precisamos de dois axiomas auxiliares para dois predicados adicionais, *Legal* e *Objetivo*.

$Legal(s) \equiv s=S_0 \vee (s=do(a,s') \wedge Poss(a,s') \wedge Legal(s'))$

$Objetivo(s) \equiv Em(R,C_{22},s)$

O predicado *Legal* indica se uma situação pode ser atingida através de uma seqüência de ações permitidas. O predicado *Objetivo* descreve como deve ser a situação do mundo quando o robô atingir a sua meta. No caso deste problema, o robô deve estar na posição (2,2) do labirinto.

Para buscar um plano para o robô usando os axiomas acima, basta procurarmos por uma situação compatível com os axiomas que seja legal e que na qual o objetivo seja atingido:

$$Axiomas \models \exists s(Objetivo(s) \wedge Legal(s))$$

A resposta obtida, usando Prolog ou outro provador de teoremas, seria:

$s = do(Mover(R,C_{21},C_{22}), do(Mover(R,C_{20},C_{21}), do(Mover(R,C_{10},C_{20}), do(Mover(R,C_{11},C_{10}), do(Mover(R,C_{12},C_{11}), do(Mover(R,C_{02},C_{12}), do(Mover(R,C_{01},C_{02}), do(Mover(R,C_{01},C_{00}),S_0)))))))$

Ou seja, *s* é a situação resultante da seqüência de ações:

$Mover(R,C_{01},C_{00})$, $Mover(R,C_{01},C_{02})$,$Mover(R,C_{02},C_{12})$, $Mover(R,C_{12},C_{11})$, $Mover(R,C_{11},C_{10})$, $Mover(R,C_{10},C_{20})$, $Mover(R,C_{20},C_{21})$, $Mover(R,C_{21},C_{22})$

2.2 Considerações sobre o Cálculo de Situações

A versão do cálculo de situações apresentada acima é a mais simples encontrada na bibliografia. Ela permite descrever a evolução do universo do robô como uma sucessão de linear de situações, sendo que a transição de uma situação para a seguinte ocorre sempre como o resultado de uma única ação. Isto limita bastante os domínios nos quais este formalismo pode ser aplicado. Não é possível, por exemplo, representar duas ações começando exatamente no mesmo instante, já que uma ação deve obrigatoriamente preceder ou suceder a outra. Também não é possível descrever ações ocorrendo paralelamente, e nem mesmo representar a duração das ações.

Outra deficiência a ser notada é a ausência de mecanismos para representar a aquisição de novas informações. Um robô deve ser capaz de decidir em que momento ele precisa de novos dados e de executar voluntariamente uma ação de coleta de dados. Além disso, ele deve ser capaz de reagir a eventos inesperados, como colisões ou falhas de equipamentos.

No cálculo de situações descrito, todas as ações são atômicas. Não há uma forma de tratar com ações complexas, isto é, ações que são composições de outras ações mais simples. Um planejamento pode ser executado de forma mais eficiente e clara se for feito com base em ações de mais alto nível. É mais simples planejar com uma única ação "AbrirPorta" do que com várias ações como "Girar maçaneta para a esquerda", "Puxar porta enquanto segura a maçaneta", "Soltar maçaneta".

Foram propostas várias extensões do cálculo de situações para cobrir essas deficiências. Pinto [5] propôs uma extensão para tratar de concorrência e de duração temporal. Em [6] é apresentada uma extensão para sensoreamento e aquisição de conhecimento. A linguagem Golog, descrita a seguir, adota algumas extensões para facilitar a tarefa de programação e para aumentar a sua expressividade.

3 GOLOG

A linguagem Golog (alGOl in LOGic) foi apresentada por Reiter, Levesque e outros [2 e 1], com a proposta de ser uma linguagem de alto nível para a programação de robôs e agentes inteligentes. Golog, em sua versão original, toma como base o cálculo de situações e incorpora a ele alguns elementos de linguagens estruturadas, visando facilitar o tratamento de ações complexas. As versões posteriores do Golog, ConGolog e IndiGolog, propõem extensões ao cálculo de situações para tratar de ações concorrentes e de aquisição de conhecimento.

3.1 Golog

Golog foi criada para permitir o tratamento de ações complexas dentro do cálculo de situações. Para isto foram incorporados três elementos típicos de linguagens estruturadas: laços (while), desvios condicionais (if) e procedimentos (proc). Também foi incorporado o operador π, para representar a escolha não-determinística de um objeto. Com eles é possível criar sentenças como estas:

- **if** *SinalFechado* **then** *Pare* **else** *Avance* **endIf**
- **while** *(∃bloco) NaMesa(bloco)* **do** *RemoveUmBloco* **endWhile**
- **proc** *RemoveUmBloco (πx)[Pegue(x); JogueFora(x)]* **endProc**

O cálculo de situações é puramente declarativo. A estratégia adotada pelos autores [2] para inserir esses elementos junto aos axiomas do cálculo de situações é através do uso de *macros*:

"Nossa abordagem será definir ações complexas usando alguns símbolos extralógicos (ex: **while**, **if**) que atuem como abreviações para expressões lógicas no cálculo de situações. Estas expressões extralógicas devem ser pensadas como macros que se expandem em fórmulas genuínas do cálculo de situações. Então, definimos abaixo a abreviação *Do(δ, s, s')*, onde *δ* é uma expressão de ação complexa; intuitivamente, *Do(δ, s, s')* será verdadeiro sempre que a situação *s'* é uma atingida após a execução da ação complexa *δ* começando na situação *s*. Note que nossas ações complexas podem ser não determinísticas, isto é, pode haver diferentes execuções terminando em diferentes situações"

A macro *Do* é definida indutivamente, da seguinte forma:

a) Ações primitivas:

$$Do(a,s,s') \stackrel{def}{=} Poss(a[s],s) \wedge s' = d(a[s],s)$$

Aqui, letra *a* representa uma ação primitiva "expurgada" de qualquer referência a situações em fluentes que sejam seus parâmetros; e *a[s]* representa a restauração das situações. Por exemplo, se *a* é *Escolha(ProdutoMaisBarato)*, e *ProdutoMaisBarato* é um fluente, então *a[s]* seria *Escolha(ProdutoMaisBarato(s))*.

b) Ações de Teste

$$Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s = s'$$

ϕ é um pseudo-fluente, (não é uma formula do cálculo de situações), e $\phi[s]$ o fluente com a situação restaurada. Por exemplo:

se ϕ é $(\forall x).NaMesa(x) \wedge \neg Sobre(x, A)$

então $\phi[s]$ é $(\forall x).NaMesa(x, s) \wedge \neg Sobre(x, A, s)$

c) Seqüência de ações

$$Do([\delta_1;\delta_2], s, s') \stackrel{def}{=} (\exists s^*).(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s'))$$

d) Escolha não-determinística de ações

$$Do((\delta_1 \mid \delta_2), s, s') \stackrel{def}{=} (Do(\delta_1, s, s') \vee Do(\delta_2, s, s'))$$

e) Escolha não-determinística de argumentos de ações

$$Do((\pi x)\delta(x), s, s') \stackrel{def}{=} (\exists x) Do(\delta(x), s, s')$$

f) Iteração não-determinística

$$Do(\delta^*, s, s') \stackrel{def}{=} (\forall P).\{(\forall s_1)P(s_1, s_1) \wedge (\forall s_1, s_2, s_3)[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \rightarrow P(s_1, s_3)]\} \rightarrow$$
$$\rightarrow P(s, s')$$

A grosso modo, a expressão acima significa que *s'* pertence ao conjunto de todos os resultados da execução iterativa de δ, zero ou mais vezes, até o número máximo possível de iterações.

A partir das definições acima, temos as definições dos laços (while) e dos desvios condicionais (if):

if ϕ **then** δ_1 **else** δ_2 **endIf** $\stackrel{def}{=} [\phi?;\delta_1] \mid [\neg\phi?;\delta_2]$

while ϕ **do** δ **endWhile** $\stackrel{def}{=} [[\phi?;\delta]^*;\neg\phi?]$

A definição de procedimentos é feita em duas etapas:

a) Primeiro é definida uma macro auxiliar para tratar as chamadas a procedimentos. Para cada procedimento P_k com parâmetros fluentes $t_1,...,t_n$, é definido o predicado P_k na seguinte forma:

$$Do(P_k(t_1,...,t_n), s, s') \stackrel{def}{=} P_k(t_1[s],...,t_n[s], s, s')$$

Na expansão desta macro, os parâmetros t_i são avaliados com base na situação s, a situação na qual foi chamado o procedimento. A execução do procedimento deve resultar na situação *s'*.

b) Depois é feita a definição do corpo de todos os procedimentos, junto com a definição do programa principal. Se houver n procedimentos P_1, ... , P_n, com parâmetros $\vec{v}_1$, ..., $\vec{v}_n$, e o programa principal δ_0, teremos um programa completo em Golog definido desta forma:

proc $P_1(\vec{v}_1)$ δ_1 **endProc**; ...; **proc** $P_n(\vec{v}_n)$ δ_n **endProc**; δ_0

A macro de expansão do programa é dada por:

Do({**proc** $P_1(\vec{v}_1)$ δ_1 **endProc**; ...; **proc** $P_n(\vec{v}_n)$ δ_n **endProc**; δ_0 },s,s')

$$\stackrel{def}{=} (\forall P_1,...,P_n).[\bigwedge_{i=1}^{n}(\forall s_1,s_2,\vec{v}_i).Do(\delta_i,s_1,s_2) \rightarrow Do(P_i(\vec{v}_i),s_1,s_2)] \rightarrow Do(\delta_0,s,s')$$

Quando um programa em Golog é executado, o interpretador do Golog expande a macro do programa descrita acima, convertendo tudo para sentenças em lógica de primeira ordem. Estas sentenças são avaliadas por um provador de teoremas (Prolog, por exemplo), e o resultado obtido é a situação final, ou seja, o histórico de todas as ações executadas pelo programa.

O exemplo de programa em Golog mostrado a seguir foi extraído de [2]. É um programa de controle de um elevador. Primeiro descreveremos o universo do elevador, em cálculo de situações, e em seguida o programa Golog.

Universo do Elevador

Ações primitivas:

turnoff(N)	: desliga o botão do andar N
open	: abre a porta do elevador
close	: fecha a porta do elevador
up(N)	: sobe até o andar N
down(N)	: desce até o andar N

Fluentes:

current_floor(s) = n	: o elevador está no andar n na situação s
on(n,s)	: o botão do andar N está ligado na situação s
next_floor(n,s)	: o próximo andar a ser atendido na situação s é o n

Axiomas de estado inicial:

current_floor(S_0) = 4
on(3,S_0)
on(5,S_0)

Axiomas de precondições (para ações primitivas):

$Poss(up(n),s) \equiv current\_floor(s) < n$
$Poss(down(n),s) \equiv current\_floor(s) > n$
$Poss(open,s) \equiv true$
$Poss(close,s) \equiv true$
$Poss(turnoff,s) \equiv on(n,s)$

Axiomas de estados sucessores:

$$Poss(a,s) \rightarrow [current\_floor(do(a,s)) = m \equiv \{a = up(m) \vee a = down(m) \vee current\_floor(s) = m \wedge \neg(\exists n)a = up(n) \wedge (\exists n)a = down(n)\}]$$

$$Poss(a,s) \rightarrow [on(m, do(a,s)) \equiv on(m,s) \wedge a \neq turnoff(m)]$$

Fluente auxiliar

$$next\_floor(n,s) \equiv on(n,s) \wedge (\forall m).on(m,s) \rightarrow |m - current\_floor(s)| \geq |n - current\_floor(s)|$$

Procedimentos em Golog

proc *serve(n) go_floor(n); turnoff(n); open; close* **endProc**

proc *go_floor(n) (current_floor = n)? | up(n) | down(n)* **endProc**

proc *serve_a_floor (π n)[next_floor(n)?; serve(n)]* **endProc**

proc *park* **if** *current_floor = 0* **then** *open* **else** *down(0);open* **endIf endProc**

proc *control* [**while** *(∃ n)on(n)* **do** *serve_a_floor* **endWhile**]*; park* **endProc**

Na situação inicial, o elevador está parado no andar 4 e há chamadas para os andares 3 e 5. O procedimento *control* é o programa principal, que consiste em servir os andares enquanto houver chamadas. Quando não houver mais chamadas não atendidas, o elevador pára no andar zero e abre. Observe que os fluentes *on* e *current_floor* aparecem nos procedimentos sem o parâmetro *s*. Ele será adicionado quando da expansão da macro.

O programa é executado ao tentarmos provar esta sentença:

$$Axiomas \models (\exists s)Do(control, S_o, s)$$

O resultado obtido seria

s = do(open, do(down(0), do(close, do(open, do(turnoff(5), do(up(5), do(close, do (open), do(turnoff(3), do(down(3), S_0)))))))))

ou seja, o elevador atende o andar 3, depois o 5 e no fim pára no andar zero e abre.

No final das contas, tanto em Golog quanto no cálculo de situações puro tudo se resume a criar um conjunto de axiomas e passá-los para um provador de teoremas. Há algumas diferenças, porém, que vale a pena ressaltar:

- No cálculo de situações, o objetivo deve ser declarado explicitamente, em todos os detalhes. Em Golog, o objetivo está implícito na lógica do programa.
- No cálculo de situações, fica totalmente a cargo do provador de teoremas a tarefa de buscar uma solução que satisfaça o objetivo. Em Golog, o provador de teoremas é guiado pelo programa. Quanto mais determinístico for o programa, menor o espaço de busca do provador. Se o programa for totalmente determinístico, o provador somente segue os passos indicados, sem liberdade para a busca. Neste caso seria melhor usar outras linguagens, como C ou Pascal, que seriam muito mais eficientes. Em outro extremo, também é possível escrever em Golog um programa que deixe tudo a cargo do provador:

  while ¬*Objetivo* **do** *(π a)[Apropriada(a)?;a]* **endWhile**

  O grau de determinismo do programa é uma decisão do programador.
- É possível trabalhar em mais alto nível, com ações complexas, mesmo no cálculo de situações puro. Ficaria a cargo do programador criar novos predicados e funções que consolidassem as informações de baixo nível. Isto, porém, seria muito trabalhoso. Em Golog, este trabalho já está feito.

3.2 ConGolog

ConGolog (Concurrent Golog) [3] é uma versão estendida de Golog que permite representar ações complexas concorrentes. Com ConGolog é possível tratar:

- Processos concorrentes com prioridades distintas
- Interrupções (em alto nível)
- Ações exógenas arbitrárias

ConGolog preserva os operadores **if** e **while** do Golog, e adiciona novas construções para tratamento de concorrência:

if ϕ **then** δ_1 **else** δ_2 **endIf**	: desvio condicional
while ϕ **do** δ **endWhile**	: laço
$(\delta_1 \parallel \delta_2)$	: execução concorrente com prioridades iguais
$(\delta_1 >> \delta_2)$	: execução concorrente com prioridades distintas
$\delta^{\parallel}$	: iteração concorrente
$<\phi \rightarrow \delta>$	: interrupção

Por $(\delta_1 \parallel \delta_2)$ entenda-se a execução de duas ações complexas concorrentemente, sem distinção de prioridades. Em $(\delta_1 >> \delta_2)$, a ação δ_1 tem prioridade de execução sobre δ_2, sendo δ_2 só será executada quando δ_1 estiver bloqueada ou terminada. ($\delta^{\parallel}$) é semelhante à iteração não determinística δ^* definida no item (3.1), com a diferença de que as instâncias de $\delta^{\parallel}$ são executadas concorrentemente, ao invés de em seqüência. E finalmente $<\phi \rightarrow \delta>$ representa uma interrupção, na qual quando a condição ϕ for verdadeira, a ação δ será executada com prioridade máxima.

O cálculo de situações, como foi explicado no item 2.2, não permite a representação de ações concorrentes. Para cada situação há um histórico linear de ações primitivas, em que cada ação só pode ter uma antecessora e uma sucessora imediatas. Em ConGolog esta característica é mantida. O truque da linguagem para permitir ações complexas concorrentes é decompor essas ações em suas ações primitivas e entrelaçá-las durante a execução. É uma estratégia análoga à adotada pelos processadores *multithread*, que simulam a execução simultânea de dois programas executando alternadamente instruções de um e de outro programa.

Para permitir a decomposição de ações complexas, a semântica de Golog foi estendida com dois novos predicados adicionais: *Trans* e *Final*. *Final(δ,s)* indica que o programa *δ* pode terminar legalmente na situação *s*, isto é, não há mais nenhum passo de *δ* a ser executado. Por passo do programa queremos dizer uma ação primitiva ou um teste de condição. *Trans(δ, s, δ', s')* indica que executando um passo do programa *δ* na situação *s* chega-se à situação *s'* com um programa *δ'* sobrando para ser executado.

Com esses novos predicados, pode ser definido um conjunto de axiomas que especificam a forma como os programas devem ser "quebrados" em passos elementares:

a) Programa Vazio

nil – constante especial para representar um programa vazio

$Trans(nil, s, \delta', s') \equiv false$

$Final(nil, s) \equiv true$

O programa vazio é final, e não pode evoluir para nenhuma situação posterior.

b) Ação Primitiva

$$Trans(a,s,\delta',s') \equiv Poss(a[s],s) \wedge \delta' = nil \wedge s' = do(a[s],s)$$

$Final(a, s) \equiv false$

Uma ação primitiva não é final, e ao ser executada resulta em uma nova situação e em um programa remanescente vazio.

c) Teste ou espera

$$Trans(\phi?,s,\delta',s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$

$$Final(\phi?,s) \equiv false$$

Um teste não é final, e ao ser executado resulta na mesma situação em que estava e em um programa remanescente vazio.

d) Seqüência

$$Trans(\delta_1;\delta_2,s,\delta',s') \equiv \exists\gamma.\delta' = (\gamma;\delta_2) \wedge Trans(\delta_1,s,\gamma,s') \vee Final(\delta_1,s) \wedge Trans(\delta_2,s,\delta',s')$$

$$Final(\delta_1;\delta_2,s) \equiv Final(\delta_1,s) \wedge Final(\delta_2,s)$$

Uma seqüência de dois programas é final quando os dois programas são finais. Executar o primeiro passo de uma seqüência de dois programas significa executar o

primeiro passo do primeiro programa ou, se este for vazio, executar o primeiro passo do segundo programa.

e) Escolha não determinística de programas

$$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$
$$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

O programa ($\delta_1|\delta_2$) pode ser considerado terminado se pelo menos um dos programas δ_1 e δ_2 terminou. A execução de um passo do programa ($\delta_1|\delta_2$) equivale à execução de um passo em pelo menos um dos programas δ_1 e δ_2.

f) Escolha não determinística de argumentos

$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x.Trans(\delta_x^v, s, \delta', s')$$
$$Final(\pi v.\delta, s) \equiv \exists x.Final(\delta_x^v, s)$$

O programa anda um passo se existir algum conjunto de parâmetros x para o qual o procedimento δ consiga evoluir um passo.

g) Iteração não determinística

$$Trans(\delta^*, s, \delta', s') \equiv \exists \gamma.(\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s')$$
$$Final(\delta^*, s) \equiv true$$

A iteração do programa δ consegue evoluir um passo se δ o conseguir. δ* é final, já que pode ser executada zero vezes (ver definição de δ* no item 3.1 (f)).

Com base nos 16 axiomas acima, definem-se os axiomas de "quebra" das instruções do ConGolog:

a) desvio condicional

$$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s, \delta', s') \equiv$$
$$\phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s')$$
$$Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s) \equiv$$
$$\phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s)$$

b) laço

$$Trans(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s, \delta', s') \equiv$$
$$\exists \gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s')$$
$$Final(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s) \equiv$$
$$\neg\phi[s] \vee Final(\delta, s)$$

c) execução concorrente com prioridades iguais

$Trans(\delta_1 \,||\, \delta_2, s, \delta', s') \equiv$

$\exists\gamma.\delta' = (\gamma \,||\delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \exists\gamma.\delta' = (\delta_1 \,||\gamma) \wedge Trans(\delta_2, s, \gamma, s')$

$Final(\delta_1 \,||\, \delta_2, s) \equiv$

$Final(\delta_1, s) \wedge Final(\delta_2, s)$

d) execução concorrente com prioridades diferentes

$Trans(\delta_1 >> \delta_2, s, \delta', s') \equiv$

$\exists\gamma.\delta' = (\gamma >> \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee$

$\exists\gamma.\delta' = (\delta_1 >> \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''.Trans(\delta_1, s, \zeta, s'')$

$Final(\delta_1 >> \delta_2, s) \equiv$

$Final(\delta_1, s) \wedge Final(\delta_2, s)$

e) iteração concorrente

$Trans(\delta^{||}, s, \delta', s') \equiv$

$\exists\gamma.\delta' = (\gamma \,||\delta^{||}) \wedge Trans(\delta, s, \gamma, s')$

$Final(\delta^{||}, s) \equiv true$

d) interrupção

$<\phi \rightarrow \delta> \stackrel{def}{=}$ **while** *Interrups_running* **do**

if ϕ **then** δ **else** false **endIf**

endWhile

O fluente *Interrups_running* é um fluente especial usado para indicar quando as interrupções estão habilitadas. Os predicados *Trans* e *Final* para uma interrupção equivalem aos do laço while que a define.

O interpretador ConGolog usa os predicados *Trans* e *Final* no processo de expansão das macros, gerando assim axiomas que permitem o entrelaçamento das ações primitivas que compõem as ações complexas.

ConGolog pode tratar de ações exógenas. Consideram-se como ações exógenas as ações primitivas que podem ocorrer independentemente do programa principal. Para tratá-las, é preciso criar um programa adicional, δ_{EXO}

$$\delta_{EXO} \stackrel{def}{=} (\pi\, a.Exo(a)?;a)*$$

onde o predicado *Exo* deve ser definido pelo programador para indicar quais ações primitivas são exógenas. Para que as ações exógenas sejam reconhecidas, o programa δ_{EXO} deve rodar concorrentemente com o programa principal:

$$\delta \,||\, \delta_{EXO}$$

3.3 IndiGolog

Para rodar programas em Golog e ConGolog é necessário passar para o interpretador, além dos programas em si, todas as informações conhecidas sobre o problema. O interpretador resolve então o programa e retorna o plano com a seqüência completa de ações, antes de qualquer ação real ter sido executada. Mesmo as ações exógenas, suportadas por ConGolog, devem ser conhecidas de antemão para alimentar o interpretador. Obviamente, esta não uma estratégia boa para controlar um robô autônomo, já que as ações exógenas e dados coletados pelos sensores dependem de fatores externos ao robô e não podem ser previstos.

IndiGolog (Incremental Deterministic ConGolog) [9 e 10] estende ConGolog, com elementos para permitir a execução on-line de programas. São adicionados 2 elementos: o predicado *SF* (*sensed fluent*) e o operador $\sum$ (busca).

a) *SF* – Sensed Fluent

O predicado *SF* é usado para indicar quais são as ações primitivas de sensoreamento, na forma $SF(a,s) \equiv \phi_a(s)$. Por exemplo, se uma ação $read_q$ consulta o estado de um predicado *q*, ela é representada como $SF(read_q, s) \equiv q(s)$. Para ações que não envolvem sensoreamento, SF é sempre verdadeiro: $SF(a,s) \equiv true$. Junto com SF, é criado o conceito de *história*, dado pela seqüência de pares *(a,x)*, onde *a* é uma ação e *x* o resultado do sensoreamento. Intuitivamente, a história $(a_1,x_1)\cdot\ldots\cdot(a_n,x_n)$ é aquela na qual as ações $a_1, .. a_n$ ocorrem a partir de alguma situação inicial, e cada ação a_i retorna um valor de sensoreamento x_i.

Na ausência de sensoreamento (ConGolog), uma ação *a* é o próximo passo legal de um programa somente quando

$$Axiomas \models Trans(\delta, s, \delta', do(a,s))$$

Com sensoreamento, a execução de *a* passa a depender também do que foi coletado pelos sensores durante o histórico de ações:

$$Axiomas \cup \{Sensed[\sigma, S_0]\} \models Trans(\delta, s, \delta', do(a,s))$$

onde

- σ é a história das ações e sensoreamento
- $Sensed[\sigma,S_0]$ é a abreviação para a fórmula no cálculo de situações contendo todos os valores coletados pelos sensores na história σ desde a situação inicial S_0.

b) $\sum$ - Busca

Golog e ConGolog rodam off-line, isto é, antes de qualquer ação real ser executada. IndiGolog foi criada para rodar on-line, tratando os eventos externos à medida em que eles ocorrem. Ela não gera o plano de ação todo de uma vez, mas aos pedaços, incrementalmente. Para isto, o programa deve ser separado em partes determinísticas e não

determinísticas. As partes determinísticas são aquelas nas quais a seqüência de passos é totalmente definida pelo programador e não há necessidade de um planejador. As partes não-determinísticas são aquelas para as quais é preciso de alguma avaliação do planejador para encontrar uma seqüência de ações legal. O interpretador executa os segmentos determinísticos passo a passo, e ao encontrar um segmento não-determinístico, chama o planejador para encontrar uma seqüência válida.

O operador de busca $\sum$ serve para indicar quais partes do programa são não-determinísticas. Tomemos como exemplo o programa

$(\delta_1|\delta_2); r?; read_q; p?$

onde $read_q$ é uma ação de sensoreamento. Em ConGolog não seria possível avaliar este programa inteiro, já que o resultado da ação $read_q$ não é conhecido de antemão. A parte não determinística do programa é a escolha entre os subprogramas δ_1 e δ_2. Usa-se então o operador $\sum$ para marcar este segmento.

$\sum \{(\delta_1|\delta_2); r?\}; read_q; p?$

Agora a execução do programa é possível. O interpretador escolhe entre os subprogramas δ_1 e δ_2 aquele que torna o teste $r?$ verdadeiro, gerando a seqüência de ações correspondente. O resto do programa é executado sem necessidade de planejamento.

No IndiGolog não há nenhum mecanismo implícito para separar as partes determinísticas e não-determinísticas, ficando isso totalmente a cargo do programador. Por isso não há uma forma de se saber com antecedência se um programa vai rodar até o fim com sucesso.

4 Revisão de Crenças

Um robô baseado em cálculo de situações possui um banco de dados de axiomas e fatos sobre o ambiente externo e sobre o seu funcionamento interno. O banco de dados evolui continuamente na medida em que os sensores coletam novas informações e o histórico de ações cresce. Na maioria das vezes, a informação nova é simplesmente adicionada ao banco. Mas pode ocorrer uma situação na qual a nova informação contradiga logicamente alguma afirmação já contida no banco. Se a nova informação for incorporada sem nenhum tratamento prévio, o banco de dados ficará inconsistente. Como se sabe, a partir de um conjunto de proposições contraditórias é possível provar qualquer outra proposição. Portanto, um banco de dados inconsistente tem pouca utilidade para qualquer planejamento. Ao adicionarmos a nova informação contraditória à base de conhecimento, é preciso que de alguma forma nós "corrijamos" o banco de dados para que ele se mantenha consistente, num processo conhecido como revisão de crenças.

Seja K base de conhecimento, e φ uma nova sentença não contida em K. De acordo com [7], há três formas possíveis de mudança na base de conhecimento:

- Expansão: ocorre quando adicionamos φ a K e as conseqüências lógicas de φ não contradizem as sentenças contidas em K. Neste caso, φ é simplesmente incorporada a K.
- Revisão: ocorre quando φ é inconsistente com K, e para acomodar φ é necessário remover algumas das sentenças antigas de K.
- Contração: ocorre quando removemos um sentença ψ de K. Neste caso todas as conseqüências lógicas de ψ também devem ser removidas.

Consideremos o exemplo do robô vigilante, citado na introdução deste texto. À medida que ele circula pelo ambiente, o histórico de ações e dados coletados pelos sensores crescem, caracterizando uma expansão da base de conhecimento. Se ele for programado para "esquecer" dados antigos mantendo o banco de dados pequeno, isto caracterizaria uma contração.

O caso da revisão é mais complexo. Voltando ao exemplo citado na introdução: o robô entra em uma sala e encontra uma porta onde de acordo com o mapa não deveria haver nada. O robô pode refazer a base de conhecimento, chegando a uma das seguintes conclusões:

a) Não há porta nenhuma, é só uma "ilusão" causada por mau funcionamento do sensor.
b) O mapa está errado e deve ser corrigido. A porta existe e sempre existiu.
c) A porta, de alguma forma, apareceu na parede. O ambiente não é estático como se supunha
d) O robô entrou na sala errada.

Cada conclusão é resultado da rejeição de alguma informação. Em (a), o dado do sensor é descartado. Em (b), um pedaço do mapa é apagado e refeito. Em (c), um dos axiomas fundamentais sobre o ambiente é reformulado. E em (d), o histórico recente do robô é reconsiderado.

Todas essas conclusões são lógicas e mantêm a base de conhecimentos consistente. Para escolher uma entre elas é necessário um critério extra-lógico. Dois critérios são

comumente adotados. O primeiro é o de atribuir pesos diferentes para sentenças, sendo que as de maior peso são as últimas a serem removidas. Segundo este critério, os axiomas sobre o ambiente e o mapa do prédio teriam maior peso, e a probabilidade de se chegar às conclusões (b) e (c) seriam menores. O outro critério é o de escolher a conclusão que implique na menor alteração na base de conhecimentos. Neste critério, a conclusão (c) também seria a de menor probabilidade, já que ao alterarmos um dos axiomas fundamentais sobre o ambiente, devemos alterar também todas as numerosas sentenças que são conseqüências dele.

Também é possível programar o robô para que ele busque informações adicionais para confirmar ou refutar algumas das conclusões. Ele poderia efetuar testes no sensor, para testar a veracidade da conclusão (a). E poderia explorar mais o ambiente, para testar a conclusão (d).

A revisão de crenças pode ser aplicada ao cálculo de situações. Em [13], por exemplo, é apresentada uma abordagem na qual um novo fluente, *Bel*, serve para indicar o estado de crença em um predicado, a cada situação. Um função *pl(s)* indica a plausibilidade de uma situação. À medida que as situações evoluem, escolhem-se as crenças que se ajustam melhor às situações mais plausíveis.

5 Conclusão

Neste texto tratou-se brevemente do cálculo de situações, um formalismo em lógica de primeira ordem usado para representar ambientes dinâmicos. Tratou-se também da família de linguagens Golog, baseada no cálculo de situações e criada com o objetivo de programar agentes inteligentes e robôs. Finalmente, foi introduzido o problema da revisão de crenças e a forma como ele pode ocorrer em robôs autônomos.

No prosseguimento deste trabalho, será aprofundado o estudo sobre revisão de crenças em robôs. O cálculo de situações, dada a sua simplicidade, será usado como formalismo para a representação de crenças. E Golog, provavelmente, será a linguagem de implementação dos testes.

Referências

[1] Reiter, Raymond. Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. MIT Press, 2001.

[2] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59-84, 1997.

[3] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109-169, 2000.

[4] Correa da Silva, F. S. ; Wasserman, Renata ; Melo, Ana Cristina Vieira de ; Barros, Leliane Nunes de ; Finger, Marcelo . Intelligent Mobile Multi-robotic Systems: some Challenges and Possible Solutions. Proceedings of ICINCO - 2nd International Conference on Informatics in Control, Automation and Robotics, Barcelona, 2005.

[5] Pinto, J. Temporal Reasoning in the Situation Calculus. PhD thesis, Department of Computer Science, University ofToronto, Toronto, Ontario, Canada, Feb. 1994. URL = ftp.cs.toronto.edu/˜cogrobo/jpThesis.ps.Z.

[6] R. Scherl, H. Levesque, and Y. Lespérance. The Situation Calculus with Sensing and Indexical Knowledge, in Moshe Koppel and Eli Shamir, editors, Proceedings of BISFAI'95: The Fourth Bar-Ilan Symposium on Foundations of Artificial Intelligence, pp. 86-95, Ramat Gan and Jerusalem, Israel, June, 1995.

[7] Gärdenfors P. "Belief Revision: An introduction", pp. 1-20 in Belief Revision, Cambridge University Press. 1992

[8] Russel, Stuart e Norvig, Peter. Inteligência Artificial: Tradução da Segunda Edição. Editora Campus, 2004.

[9] An Incremental Interpreter for High-Level Programs with sensing. De Giacomo, Giuseppe; Levesque, Hector. Logical foundation for cognitive agents: contributions in honor of Ray Reiter, pages 86-102. Springer, Berlin, 1999

[10] On the Semantics of Deliberation in Indigolog – from Theory to Implementation Sardina, Sebastian; De Giacomo, Giuseppe; Lespérance, Yves; Levesque, Hector J. Proceedings of Eighth International Conference in Principles of Knowledge Representation and Reasoning (KR-2002), pages 603-614, Toulouse, France, April 2002. Morgan Kaufmann.

[11] McCarthy, John and Hayes, Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. Machine Intelligence, vol 4, Edinburgh University Press, 1969.

[12] McCarthy, John. Situations, Actions and Causal Laws, Technical Report, Stanford University, 1963.

[13] S. Shapiro and M. Pagnucco. Iterated Belief Change and Exogenous Actions in the Situation Calculus. In R. López de Mántaras and L. Saitta (Eds.), Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04), pages 878-882, IOS Press, Amsterdam, The Netherlands, 2004