Planejamento no

Cálculo de Situações usando a Linguagem Golog

Edward Mitsuo Iwanaga Iamamoto

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIAS

Área de concentração: Ciência da Computação

Orientadora : Profa. Dra. Leliane Nunes de Barros

— São Paulo, 30 de Novembro de 2005 —

PLANEJAMENTO NO CÁLCULO DE SITUAÇÕES USANDO A LINGUAGEM GOLOG

Esse exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Edward Mitsuo Iwanaga Iamamoto e aprovada pela comissão julgadora.

São Paulo, 30 de Novembro de 2005.

Banca Examinadora:

Profa. Dra. Leliane Nunes de Barros (orientadora) - IME/USP
Prof. Dr. Flávio Soares Correa da Silva - IME/USP
Prof. Dr. Paulo Eduardo Santos - FEI

Resumo

Golog é uma linguagem de alto nível criada para especificar programas para agentes robóticos. No entanto, como acontece em situações do mundo real, Golog deve permitir a programação de agentes capazes de tratar eventos exógenos e imprevistos. Para isso, uma primeira solução seria construir programas prevendo todas as contingências do mundo. No entanto, isso pode levar ao desenvolvimento de agentes robóticos de alto custo e baixo desempenho. Uma outra possibilidade é o planejamento com execução e monitoramento, que, em caso de falha, permite criar um plano para corriji-la. Isso pode ser feito por um planejador para satisfação de metas. Nesse trabalho, mostramos que o meta-interpretador Golog realiza planejamento de forma semelhante aos planejadores hierárquicos e que, baseado nessa técnica, Golog pode resolver problemas de planejamento para satisfação de metas através da definição de **procedimentos meta**. Chamamos esse tipo de planejamento de **GGPP** (**G**olog **G**oal **P**rocedures based **P**lanner). Para comparar o desempenho do GGPP com outros planejadores, foram implementados e analisados três outros planejadores para satisfação de meta baseados no Cálculo de Situações. Os resultados obtidos mostraram que a estratégia proposta pode ser mais eficiente que os esses outros planejadores.

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Planejamento clássico 1

1.3 Planejamento hierárquico 2

1.4 A linguagem Golog e o planejamento hierárquico 3

1.5 Objetivos 4

1.6 Organização 5

1.7 Notação 6

2 Planejamento clássico **9**

2.1 O Cálculo de Situações 10

2.1.1 Suposição de mundo fechado e o problema da persistência 13

2.2 Planejando no Cálculo de Situações 16

2.3 A representação de ações STRIPS 20

2.3.1 Planejamento com STRIPS 22

2.4 Resumo 31

3 Planejamento hierárquico **33**

3.1 Noção informal de planejamento hierárquico 34

3.2 Definições 36

3.3 Um algoritmo de planejamento hierárquico 40

3.4 Indecidibilidade do planejamento hierárquico 41

3.5 Expressividade do planejamento hierárquico 42

3.6 Planejamento com tarefas meta 45

3.7 Críticas 47

3.7.1 Crítica de pré-condições *observáveis* e *planejáveis* 49

3.8 O planejador SHOP2 51

3.9 Resumo 54

4 GOLOG **55**

4.1 A linguagem Golog 55

4.2 Sintaxe e semântica da linguagem Golog 57

4.2.1 Caracterização do predicado *Trans* 59

4.2.2 Caracterização do predicado Final 64

4.3 Os predicados *Do* e *Trans** 66

4.4 O interpretador Golog 67

4.5 Interrupções 72

4.6 Indigolog 73

4.7 Um exemplo de programa Golog 74

5 Golog e planejamento hierárquico 77

5.1 A proposta de Baral & Son: o operador htn 78

5.2 A proposta de Gabaldon: representações de redes de tarefas 80

5.2.1 Redes de tarefas totalmente ordenadas em Golog 81

5.2.2 Redes de tarefas parcialmente ordenadas em Golog: os fluentes *nexec* e *pred* . 82

5.2.3 Redes de tarefas parcialmente ordenadas em Golog: menos expressividade . . . 83

5.3 Solução de Baral & Son versus solução de Gabaldon 85

5.4 O meta-interpretador Golog como um planejador hierárquico 85

5.4.1 Definição de problema e soluções 86

5.4.2 Restrições de fluentes do planejamento hierárquico na linguagem Golog 91

6 Satisfação de metas em Golog 93

6.1 Necessidade da satisfação de metas em Golog 94

6.2 Planejamento no Cálculo de Situações: o planejador IDDP 95

6.2.1 O predicado *plan* e a estratégia de busca em profundidade iterativa 96

6.2.2 O predicado *exec* e a instanciação de planos genéricos 98

6.2.3 O predicado *planning* e o planejador IDDP 99

6.3 Planejamento no Cálculo de Situações: o planejador WSPBF 100

6.3.1 O procedimento *actionSequence* e a instanciação de planos 102

6.3.2 O procedimento *plans* e a estratégia de busca em profundidade iterativa 103

6.4 Comparação entre IDDP e WSPBF 103

6.5 Tarefas-meta em Golog 105

6.5.1 Satisfação como possível conseqüência de programa Golog 105

6.5.2 Procedimentos meta 107

7 Análise de desempenho **111**

7.1 Teste 1 112

7.2 Teste 2 115

7.3 Teste 3 118

7.4 Teste 4 121

8 Conclusões **125**

A O predicado *badSituations* **127**

B Implementação do meta-interpretador Golog. **131**

Lista de Figuras

2.1 Exemplo de estados no mundo dos blocos. 10

2.2 Axiomatização Prolog do Mundo dos Blocos no Cálculo de Situações. 14

2.3 O domínio do Mundo dos Blocos em Prolog. 18

2.4 Trace da consulta `?-sobre(b,c,S)`. 18

2.5 Busca Prolog para a consulta `?-sobreMesa(c,S)`. 19

2.6 Operadores STRIPS para *desempilha*(a, b) e *empilha*(a, b). 22

2.7 Um plano na representação STRIPS. 23

2.8 Um espaço de estados no Mundo dos Blocos. 25

2.9 Planos obtidos na busca em largura pelo espaço de estados 26

2.10 Uma busca em largura pelo espaço de estados com STRIPS 27

2.11 Um espaço de planos. 28

2.12 Planejamento POP. 29

2.13 A ação OP-3 ameaça o vínculo causal $Start \xrightarrow{a} OP\text{-}2$. 30

2.14 Promoção para eliminar uma ameaça em um plano. 31

2.15 O planejador POP. 32

3.1 Representação gráfica de uma rede de tarefas. [Erol, 1995] 38

3.2 Duas decomposições possíveis da tarefa não-primitiva $ir(sp, rj)$. 39

3.3 Um algoritmo de planejamento hierárquico [Erol, 1995]. 40

3.4 Recursão de decomposições hierárquicas. 41

3.5 Algoritmo de transformação de problema clássico em problema de planejamento HTN [Erol, 1995]. 44

3.6 Transformação de um operador STRIPS em uma rede de tarefas que pode ser usada nos métodos $(achieve[livre(b)], d)$ e $(achieve[sobreMesa(a), d])$. 45

3.7 Os operadores $empilha(X, Y)$ e $desempilha(X, Y)$ convertidos em tarefas meta. 46

3.8 Seqüência de decomposições da tarefa meta $achieve[livre(b)]$. 47

3.9 Solução para executar a tarefa meta $achieve[livre(b)]$ evitando laços de decomposições recursivas. 48

3.10 $achieve[livre(b)]$ e $achieve[sobreMesa(a)]$. 50

3.11 Novos métodos sem tarefas meta oriundas de pré-condições observáveis. 51

3.12 Algoritmo SHOP2 em linguagem de programação genérica. 52

3.13 Um passo de decomposição do SHOP2 sempre escolhe a tarefa *mais à esquerda* da rede de tarefas. 53

4.1 As variações da linguagem Golog. 74

4.2 Programa GOLOG para criar uma torre de tamanho n no Mundo dos Blocos 75

5.1 Métodos da tarefa não-primitiva p representados como um procedimento Golog p. . . . 81

5.2 Rede de tarefas totalmente ordenada e o programa Golog correspondente. 81

5.3 Representação gráfica de uma rede de tarefas parcialmente ordenada. 84

5.4 Exemplo de rede que não pode ser codificada sem $pred$ e $nexec$. 84

5.5 Representação gráfica da decomposição da tarefa q. 89

5.6 Representação gráfica de duas decomposições da tarefa t. 90

6.1 Domínio do mundo dos blocos em Cálculo de Situações. 96

6.2 Problema no mundo dos blocos em Cálculo de Situações. 96

6.3 O planejador IDDP. 97

6.4 Mecanismo de busca em profundidade iterativa. 98

6.5 Mecanismo de instanciação de planos genéricos: falha. 99

6.6 Mecanismo de instanciação de planos genéricos: sucesso. 100

6.7 Domínio do mundo dos blocos em Cálculo de Situações para o planejador WSPBF. . . 101

6.8 O algoritmo simplificado do planejador WSPBF. 102

6.9 O planejador WSPBF descrito em Prolog, chamado de SCP (Situation Calculus Planner.104

6.10 Não há execução de $criaTorre(3)$ que satisfaça sobre(a,b) e sobre(b,c). 107

6.11 Rede de tarefas para $achieve[livre(b)]$ e $achieve[sobreMesa(a)]$. 108

6.12 Procedimentos meta para satisfação dos fluentes *livre*, *sobreMesa*, *sobre* do Mundo dos Blocos. 109

7.1 Problemas de planejamento para o **Teste 1**. $A1$, $A2$, $A3$, $A4$ e $A5$ são estados iniciais de problemas para o mesmo estado meta G. 112

7.2 Problemas de planejamento para o **Teste 2**. $B1$, $B2$, $B3$, $B4$, $B5$, $B6$, $B7$ e $B8$ descrevem cada uma das diferentes metas dos 8 problemas para o mesmo estado inicial I. 116

7.3 Problemas de planejamento do **Teste 3**. As ilustrações representam os estados iniciais e $D1$, $D2$, $D3$, $D4$, $D5$ e $D6$, os estados meta correspondentes. 118

A.1 O algoritmo completo do planejador WSPBF. 128

A.2 Uma situação inicial e um estado meta. 128

A.3 Uma definição de *badSituation*. 129

Lista de Tabelas

4.1 Sintaxe e semântica dos operadores da linguagem Golog. 58

7.1 Desempenho com mesma meta e número variável de objetos. 113

7.2 Comparação no tamanho do plano solução. 117

7.3 Desempenho em problemas com interação entre submetas. 119

7.4 Problemas com interação entre submetas usando o operador de concorrência melhorado. 123

Capítulo 1

Introdução

1.1 Motivação

O processo de escolha e ordenação de ações para atingir uma meta pré-definida é chamado de planejamento. A área de **Planejamento em Inteligência Artificial** estuda os aspectos computacionais para realizar esse processo. Uma motivação prática no seu estudo é a necessidade de ferramentas para lidar com problemas complexos que envolvam muitas variáveis, agentes, recursos, tarefas e objetivos. Em muitos casos, uma ferramenta de planejamento pode indicar uma solução mais eficiente e não identificada por planejadores humanos [Gallab, M et al., 2004]. Outra motivação é equipar agentes robóticos com planejadores automatizados para realizarem tarefas complexas sem a necessidade de intervenção humana.

1.2 Planejamento clássico

Formalmente, caracterizamos um problema de planejamento como um modelo de estados, a saber:

- um conjunto não-vazio finito de estados $\mathcal{S}$;

- um estado inicial $S_0 \in \mathcal{S}$;
- um conjunto S_G de estados $S_G \subseteq \mathcal{S}$, chmado de estado meta (ou *Goal*);
- um conjunto finito de ações aplicáveis $A(s)$ para cada estado $s \in \mathcal{S}$ (o conjunto de ações descreve um **domínio de planejamento** $\mathcal{D}$);
- uma função $f : \mathcal{S} \times A \rightarrow \mathcal{S}$ que mapeia a transição de um estado $s \in \mathcal{S}$ para outro estado $s' \in \mathcal{S}$ após a execução de uma ação a em s.

O espaço de estados pode ser modelado como um grafo dirigido onde cada vértice corresponde a um estado do mundo e cada arco corresponde a uma ação cuja execução transforma um estado do mundo em outro estado do mundo [Korf, 1987]. Nesse contexto, o **problema de planejamento** consiste em encontrar um caminho pelo grafo, i.e., uma seqüência de ações, que chamamos de plano, que leve do estado inicial S_0 até um estado meta $s \in S_G$.

Essa caracterização de **planejamento clássico** (não-hierárquico) [Weld, 1994] baseia-se nas seguintes suposições:

- **onisciência**, i.e., em qualquer estado temos conhecimento total sobre o mundo;
- **determinismo de ações**, i.e, para cada par estado-ação (s,a) temos $| f(s,a) | \leq 1$, ou seja, a execução de uma ação a em um estado s sempre produzirá uma transição para um único estado s';
- **causa de mudança única**, i.e, todas as mudanças que ocorrem no mundo são devido à execução de ações do agente para o qual o plano foi criado;
- **tempo atômico**, i.e., as ações não têm duração, cada transição de estado é instantânea.

1.3 Planejamento hierárquico

O planejamento hierárquico [Erol, 1995] baseia-se nas mesmas suposições de planejamento clássico. A diferença principal entre o planejamento hierárquico e o planejamento clássico não-hierárquico está na definição de metas de planejamento e na descrição de domínio de planejamento $\mathcal{D}$. Enquanto o planejamento não-hierárquico planeja para *atingir* um estado meta, o planejamento hierárquico

planeja para *executar* uma **rede de tarefas**. Uma rede de tarefas é uma coleção de tarefas que precisa ser executada obedecendo a um conjunto de restrições, ou seja, é um plano parcialmente especificado. Uma **rede de tarefas primitiva** contêm apenas ações (tarefas) **primitivas** $a \in A(s)$.

As redes de tarefas podem conter **tarefas não-primitivas** e **métodos**. Tarefas não-primitivas não podem ser executadas diretamente, pois elas representam ações complexas que podem envolver outras tarefas [Erol, 1995]. Uma tarefa não-primitiva é uma ação composta cuja execução deve ser planejada, i.e, é necessário encontrar um plano executável de ações primitivas que execute essa tarefa não-primitiva. Um método descreve como uma tarefa não-primitiva pode ser executada através da execução de uma outra rede de tarefas.

Um **problema de planejamento hierárquico** [Erol, 1995] é formalmente definido pela tupla $\langle d, I, \mathcal{D} \rangle$, onde d é uma rede de tarefas, I é um estado inicial e $\mathcal{D}$ é um *domínio de planejamento hierárquico* composto por tarefas primitivas, tarefas não-primitivas e métodos. A solução para esse problema é uma rede de tarefas primitiva, executável a partir da situação I, obtida pela substituição de todas as tarefas não-primitivas de d, utilizando-se os métodos de $\mathcal{D}$.

1.4 A linguagem Golog e o planejamento hierárquico

A área de **robótica cognitiva** se preocupa em desenvolver agentes robóticos ou de software com funções cognitivas de alto nível, envolvendo o raciocínio sobre metas, ações, planos, estados mentais de agentes, execução colaborativa de tarefas, percepção, comunicação, etc. Na linha da robótica cognitiva que este trabalho segue, o raciocínio de alto nível é baseado em lógica, ou seja, a representação do conhecimento de um agente, seus objetivos e a situação do mundo são expressos através de uma linguagem lógica.

O Grupo de Robótica Cognitiva da Universidade de Toronto desenvolveu a linguagem **Golog** [Levesque, H. J. et al., 1997] baseada em **Cálculo de Situações** da Lógica de Predicados de Primeira Ordem. Nessa linguagem, o comportamento de um agente é descrito por um conjunto de procedimentos. Um procedimento Golog é composto por ações primitivas e chamadas a outros procedimentos. Golog têm sido utilizada em pesquisa e aplicações que envolvem problemas complexos

como, por exemplo, gerenciamento de robôs para jogar futebol [Arroz, M et al., 2003] e composição de serviços em *Semantic Web* [McIlraith, S. and Son, T. C., 2001].

Redes de tarefas do planejamento hierárquico assemelham-se a procedimentos Golog. Ambos possuem ações primitivas, e uma tarefa não-primitiva assemelha-se a uma chamada de procedimento Golog. Construir programas de controle na linguagem Golog não é uma tarefa trivial, assim como não é trivial modelar ações compostas no planejamento hierárquico. A semelhança entre o planejamento hierárquico e Golog sugere que compreender melhor esse dois paradigmas de raciocínio sobre ações, pode facilitar a adaptação de técnicas desenvolvidas em um paradigma para serem aplicadas ao outro.

Alguns trabalhos já fizeram associações entre o planejamento hierárquico e Golog. Gabaldon propõe [Gabaldon, 2002] uma série de predicados que permitem descrever redes de tarefas em programas Golog, enquanto Baral & Son propõem [Baral, C. and Son, T. C., 1999] um operador especial *htn* que implementa um algoritmo de planejamento hierárquico. Entretanto, esses trabalhos mostram apenas *como* realizar planejamento hierárquico em Golog, sem se aprofundar nas implicações da relação entre planejamento hierárquico e Golog .

1.5 Objetivos

Nesse trabalho, mostraremos que algumas das técnicas desenvolvidas no planejamento hierárquico podem melhorar o desempenho de programas Golog. Mais especificamente, os objetivos desse trabalho são:

1. Apresentar alguns algoritmos de planejamento clássico baseados no formalismo do Cálculo de Situações e na representação de ações STRIPS [Reiter, 2001a] [Pereira, 2002];
2. Apresentar a definição de planejamento hierárquico, e mostrar como o planejamento hierárquico também pode planejar para atingir um estado meta [Erol, 1995], aspecto esse não explorado na literatura de planejamento ou pelos trabalhos de Gabaldon e Baral & Son;
3. Estabelecer a correspondência entre a execução de um programa Golog e o planejamento hierárquico;

4. Mostrar que as tarefas de planejamento hierárquico do tipo **tarefa meta** [Erol, 1995] podem ser utilizadas como procedimentos Golog para atingir um estado meta de forma mais eficiente que outras abordagens não-hierárquicas baseadas no Cálculo de Situações [Reiter, 2001a] [Pereira, 2002].

1.6 Organização

Essa dissertação está organizada da seguinte maneira:

Capítulo 2. Introduzimos o formalismo lógico da Linguagem do Cálculo de Situações que é utilizada para modelagem de mundos dinâmicos. Também introduzimos a representação de ações STRIPS como outra maneira de representar mundos dinâmicos. Apresentamos algoritmos de planejamento não-hierárquico que utilizam essas representações.

Capítulo 3. Definimos formalmente o planejamento clássico hierárquico e apresentamos um algoritmo de planejamento hierárquico genérico. Também analisamos a sua expressividade e indecidibilidade. No final, apresentamos o planejador hierárquico SHOP2 que realiza planejamento hierárquico trabalhando apenas com *redes de tarefas totalmente ordenadas.* O objetivo desse capítulo é apresentar os principais conceitos de planejamento hierárquico, em especial as **tarefas meta** cuja utilização em Golog é o assunto principal desse trabalho.

Capítulo 4. Apresentamos a linguagem Golog, que é utilizada para especificação de alto nível do comportamento de agentes robóticos. Definiremos os principais operadores dessa linguagem e como eles são implementados em Prolog. O objetivo desse capítulo é mostrar como um programa Golog é processado por um meta-interpretador em Prolog.

Capítulo 5. Comparamos as propostas de [Gabaldon, 2002] e [Baral, C. and Son, T. C., 1999] para planejamento hierárquico em Golog. Argumentamos que não é necessário utilizar os recursos apresentados nesses dois trabalhos para realizar planejamento hierárquico em Golog, uma vez que a execução de programas Golog é semelhante ao planejador SHOP2. O objetivo desse capítulo é deixar claro como Golog pode naturalmente realizar o planejamento hierárquico.

Capítulo 6. Apresentamos dois algoritmos de planejamento não-hierárquico orientado à satisfação de metas e baseados no Cálculo de Situações: o **IDDP** [Pereira, 2002] e o **WSPBF** [Reiter, 2001a]. Propomos que o conceito de tarefas meta aplicado a Golog pode realizar planejamento orientado a metas de forma semelhante ao planejamento hierárquico. O objetivo desse capítulo é mostrar a principal contribuição de nosso trabalho, os **procedimentos meta**, i.e., que podemos especificar programas Golog para satisfação explícita de metas através de tarefas meta do planejamento hierárquico especificadas como procedimentos.

Capítulo 7. Apresentamos os resultados de testes comparativos entre os planejadores IDDP, WSPBF, SCP (*SituationCalculusPlanner*, um WSPBF sem Golog) e um planejador que usa os procedimentos meta em Golog. O objetivo desse capítulo é mostrar que os procedimentos meta em Golog podem ser uma alternativa mais eficiente de planejamento para atingir estados meta baseado no Cálculo de Situações.

Capítulo 8. Apresentamos a conclusão desse trabalho, suas principais contribuições e indicamos como ele pode ser estendido para utilização em projetos de robótica cognitiva.

1.7 Notação

Tentamos reproduzir o mais fielmente possível as notações utilizadas por [Reiter, 2001a] e [Erol, 1995]. Entretanto, como nesse trabalho serão abordados o Cálculo de Situações, planejamento hierárquico, e programação Golog implementada em Prolog, pode haver alguma confusão pois as notações de cada um desses temas são diferentes.

No Cálculo de Situações, utiliza-se a notação de lógica de primeira ordem. As variáveis, constantes e ações possuem a primeira letra minúscula (x, δ, desempilha(x,y), enquanto os predicados começam com letras maiúsculas ($Poss(a, s)$, $SobreMesa(x)$).

Programas Prolog convencionam que variáveis começam com letras maiúsculas (X), enquanto predicados, ações e constantes sempre começam com letras minúsculas ($livre(X)$, $desempilha(X, Y)$, s_0).

A linguagem Golog é codificada em Prolog, mas todas as definições de seus operadores foram feitas

utilizando-se a notação da lógica de primeira ordem como fazem os criadores de Golog [Reiter, 2001a] e [Levesque, H. J. et al., 1997]. Nas ocasiões em que apresentamos programas Golog, utilizamos a convenção Prolog.

Sendo assim, dependendo do contexto e assunto abordado, uma seção pode adotar uma notação diferente da adotada por outra seção.

No Capítulo 2, prevalece a notação da lógica de primeira ordem exceto quando se faz referência a programas Prolog. No Capítulo 3, que descreve o planejamento hierárquico mantemos a notação de [Erol, 1995] em que fluentes também aparecem com letras minúsculas. No Capítulo 4, as seções 4.1,4.2, 4.3 e 4.5 utilizam a notação da lógica de primeira ordem, enquanto as seções 4.4 e 4.7 utilizam a notação Prolog. Nos outros capítulos, em geral, podemos supor que sempre se está usando a notação da lógica de primeira ordem, exceto quando se faz referência um programa Golog ou Prolog, caso em que usamos a notação Prolog.

Capítulo 2

Planejamento clássico

O surgimento dos primeiros provadores automáticos de teoremas como o **GPS** (*General Problem Solver*) [Newell and Simon, 1961] e o **LT** (*Logic Theorist*) [Newell, A. et al., 1957], juntamente com a linguagem do **Cálculo de Situações** [McCarthy, 1963] permitiu a realização de algumas das primeiras tentativas de planejamento através de prova de teoremas. Um provador de teoremas é, basicamente, um construtor de sentenças lógicas que, partindo de um conjunto inicial de axiomas, gera sentenças utilizando um conjunto de regras especificadas também por axiomas. Nessa abordagem, que chamaremos aqui de **planejamento lógico**, axiomas do Cálculo de Situações, definindo como as ações modificam o mundo, são utilizados por um provador de teoremas para gerar sentenças que correspondem a planos. Como esses planos são gerados obedecendo as especificações do Cálculo de Situações podemos garantir que são planos corretos, i.e., planos executáveis. Para resolver um problema de planejamento, basta restringir a construção de planos apenas à aqueles cuja execução, a partir do estado inicial S_0, atingem o estado meta S_G do problema.

Em abordagens não-lógicas, a representação de ações é feita por operadores do tipo **STRIPS** [Fikes, R. and Nilsson, N. J., 1971] que são estruturas de dados que representam as pré-condições e efeitos de ações através de listas de literais. Estados do mundo e planos também são representados por estruturas de dados. Algoritmos de planejamento manipulam essas estruturas para resolver um problema de planejamento.

A seguir, veremos as definições do Cálculo de Situações, dos operadores STRIPS, e exemplos de como podemos utilizá-los para planejamento. O Cálculo de Situações é um pré-requisito para compreender a linguagem Golog, e o exemplo de planejador baseado em STRIPS ajudará a entender como o **planejamento hierárquico** resolve um problema de planejamento clássico através de planejamento orientado a metas no Capítulo 3.

Nesse Capítulo, e também nos demais, quase todos os exemplos serão baseados no **Mundo dos Blocos** [Nilsson, 1980] que consiste em blocos de construção cúbicos sobre uma mesa que podem ser empilhados uns sobre os outros. Somente um bloco pode ser empilhado diretamente sobre outro bloco. As ações que podem ser executadas são *empilhar* e *desempilhar* um bloco sobre outro. Um bloco pode ter, nem sempre simultaneamente, as seguintes propriedades: *estar sobre a mesa*, *estar sobre outro bloco*, e *estar livre de blocos sobre si.* Na Figura 2.1, temos dois exemplos de estados no Mundo dos Blocos com cinco blocos distintos $\{a, b, c, d, e\}$

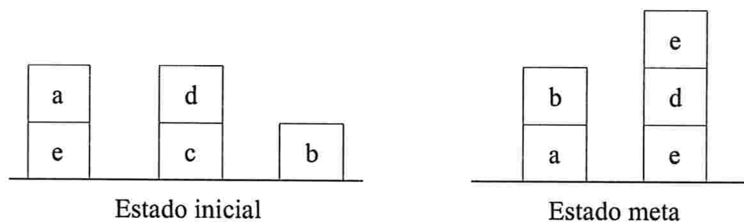

Figura 2.1: Exemplo de estados no mundo dos blocos.

2.1 O Cálculo de Situações

O Cálculo de Situações [McCarthy, 1963] é uma linguagem de Lógica de primeira ordem que permite especificar ações e como elas alteram o estado do mundo. Nessa linguagem, há três tipos distintos de termos: **fluentes**, **situações** e **ações**.

- **Ação.** Modificações no estado do mundo são realizadas apenas através da execução de ações. Um termo do tipo ação é da forma $A(\vec{x})$, onde A é o nome da ação e $\vec{x}$ é um conjunto de

parâmetros. Exemplos:

$$empilha(b,d) \qquad \text{e} \qquad desempilha(d,c)$$

significando, respectivamente, as ações de empilhar o bloco b sobre o bloco d, e a ação de desempilhar o bloco d de sobre o bloco c[1].

- **Situação.** Corresponde a um estado do mundo que resulta da execução de uma seqüência de ações a partir de um estado inicial. Dessa forma, uma situação representa um histórico de ações, ou seja, um plano. O termo situação que representa o histórico vazio, sem a execução de nem uma ação, é representado pela constante s_0, e é chamado de **situação inicial**. Usa-se o símbolo funcional *do* para construir situações a partir de outras situações, onde $do(\alpha, s)$ denota a situação resultante após a execução da ação α a partir da situação s. Por exemplo,

$$do(empilha(b,\ d),\ s_0)$$

denota a situação resultante após empilhar o bloco b sobre o bloco d a partir da situação inicial s_0. Outro exemplo,

$$do(a_3,\ do(a_2,\ do(a_1,s_0)))$$

denota a situação após a execução, das ações a_1, a_2 e a_3 a partir da situação inicial (note que as ações são executadas na ordem inversa em que aparecem, da direita para a esquerda). Para simplificação de notação, às vezes usamos $do(\vec{a})$, onde $\vec{a} = (a_1,\ a_2,\ ...,\ a_n)$ para representar a situação $do(a_n,\ ...,\ do(a_2,\ do(a_1,s_0))...\ )$.

- **Fluente.** Um fluente é uma propriedade do mundo cujo valor pode ser modificado por alguma ação. Um conjunto de fluentes com valores verdadeiros ou falsos descrevem um estado do mundo ou um conjunto de estados. Um termo do tipo fluente é da forma $F(\vec{x}, s)$, onde F é o nome do fluente, e $\vec{x}$ é um conjunto de parâmetros e s é um termo situação. Por exemplo, se tiver valor verdadeiro, o fluente

[1]Durante todo o trabalho, nos axiomas do Cálculo de Situações ou em programas Prolog, termos que começam com letras minúsculas designam constantes ou predicados, e os termos que começam com letras maiúsculas designam variáveis.

$$SobreMesa(d, do(desempilha(d,c), s_0))$$

denota que a propriedade do bloco d *estar sobre a mesa* é verdadeira na situação $do(desemplíha(d, c), s_0)$.[2]

Uma axiomatização no Cálculo de Situações é feita através de quatro tipos de axiomas:

- **Axiomas de Estado inicial** descrevem os valores dos fluentes na situação inicial s_0. São da forma $F(\vec{v}, s_0)$, significando que o fluente de nome F com parâmetros $\vec{v}$ é verdadeiro na situação inicial. Exemplo:

$$SobreMesa(b, s_0)$$

denotando que o bloco b está sobre a mesa na situação s_0.

- **Axiomas de Pré-condições** descrevem as condições em que uma ação pode ser executada. Deve haver um axioma de pré-condição para cada ação. Axiomas de pré-condição são da forma

$$Poss(\alpha(\vec{v}), s) \leftarrow \phi(s),$$

onde ϕ é uma conjunção de fluentes e o predicado *poss* denota que é possível executar a ação $\alpha(\vec{v})$ na situação s. Esse axioma representa o fato de que é possível executar a ação $\alpha(\vec{v})$, se todos os fluentes de ϕ são verdadeiros na situação s. Por exemplo:

$$empilha(x,y,s) \leftarrow Livre(x,s) \wedge Livre(y,s) \wedge SobreMesa(x,s),$$

denota que é possível empilhar um bloco x sobre um bloco y, na situação s, se os blocos x e y estão livres e o bloco x está sobre a mesa.

- **Axiomas de Estado Sucessor** são axiomas que descrevem como as ações modificam os valores dos fluentes. É necessário um axioma de estado sucessor para cada fluente. Um axioma de estado sucessor é da forma

[2]Em algumas especificações de Cálculo de Situações, os fluentes não possuem um termo situação como parâmetro. Em vez disso, reifica-se F através do predicado $holds(F(\vec{v}), s)$, onde *holds* é verdadeiro se o fluente $F(\vec{v})$ é verdadeiro na situação s.

$$\Phi(\vec{v}, do(e,s)) \leftarrow (\Phi(\vec{v}, s) \wedge e \neq \alpha) \vee e = \beta$$

denotando que o fluente Φ é verdadeiro em uma situação $do(e, s)$ se Φ já era verdadeiro na situação anterior s e e é diferente da ação α, ou se e é igual à ação β. Em outras palavras, isso significa que a ação α torna Φ falso e a ação β torna Φ verdadeiro. Exemplo:

$$\boldsymbol{Sobre(x,y,do(e,s)) \leftarrow (Sobre(x,y,s) \wedge e \neq desempilha(x,y)) \vee e = empilha(x,y)}$$

significando que o bloco x está sobre o bloco y na situação $do(e, s)$ se x já estava sobre y na situação anterior s e a ação e não foi desempilhar x de sobre y, ou se a ação e executada foi a ação de empilhar x sobre y.

Na Figura 2.2, mostramos a axiomatização em Prolog do *Mundo dos Blocos* descrita através de axiomas do Cálculo de Situações. Temos as duas ações: $empilha(x, y)$ e $desempilha(x, y)$, representando, respectivamente, a ação de empilhar um bloco x, que está sobre a mesa, sobre um bloco y, e a ação de desempilhar um bloco x de sobre um bloco y colocando-o sobre a mesa. Há três fluentes: $Livre(x, s)$, $SobreMesa(x, s)$ e $Sobre(x, y, s)$, significando, respectivamente, que não há blocos sobre o bloco x, que o bloco x está sobre a mesa, e que o bloco x está sobre o bloco y na situação s. Os axiomas de estado inicial de (2.1) a (2.5) descrevem que existem três blocos a, b e c, sendo que a e b estão sobre a mesa, c está sobre a, e que b e c estão livres na situação inicial. Os axiomas de pré-condição(2.6) e (2.7) descrevem, respectivamente, se é possível executar as ações $empilha(x, y)$ e $desempilha(x, y)$ em uma situação s. Os axiomas de estado sucessor (2.8), (2.9) e (2.10) descrevem, respectivamente, se os fluentes $SobreMesa(x, s)$, $Sobre(x, y, s)$ e $Livre(x, s)$ possuem valor verdadeiro em uma situação s.

Os axiomas de (2.1) a (2.10) definem um **domínio de planejamento** $\mathcal{D}$ no Cálculo de Situações.

2.1.1 Suposição de mundo fechado e o problema da persistência

Formalmente, a descrição do estado inicial no domínio da Figura 2.2 está incompleta, pois descreve apenas os fluentes que têm valor verdadeiro, não especificando os fluentes que têm valor falso. Por

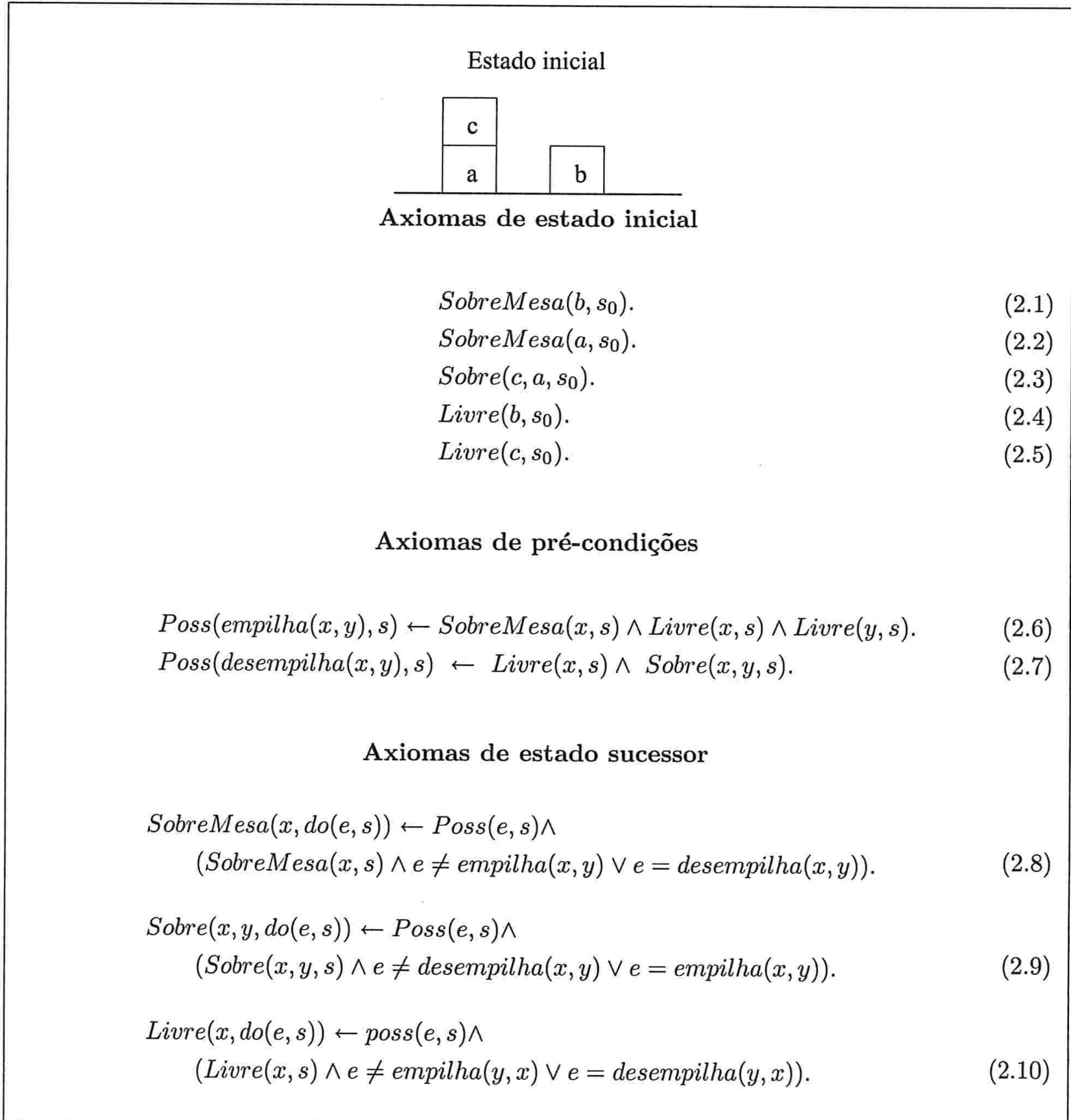

Figura 2.2: Axiomatização Prolog do Mundo dos Blocos no Cálculo de Situações.

exemplo, não declaramos que $Livre(a, s_0)$, $Sobre(a, b, s_0)$, $Sobre(b, c, s_0)$ ou $SobreMesa(c, s_0)$ são falsos. Nesse trabalho será adotada a *suposição de mundo fechado* [Reiter, 1978], i.e., a suposição de que o valor de um fluente é falso caso ele não tenha sido declarado verdadeiro ou caso não tenha sido possível inferir, através dos axiomas, que seu valor é verdadeiro. Essa suposição permite descrever de forma sintética o estado do mundo, pois, em geral, o número de fluentes com valor falso é muito maior que o de fluentes com valor verdadeiro, tanto na situação inicial como em qualquer outra.

A linguagem **Prolog** [Roussel, 1975] [Colmerauer, A. et al., 1973], na qual iremos implementar todos os algoritmos de planejamento deste trabalho, já faz essa suposição de mundo fechado através de seu mecanismo de **negação por falha**, i.e., o valor de um predicado é automaticamente considerado falso caso o interpretador Prolog não tenha conseguido inferir que seu valor é verdadeiro.

Diferentemente de nossa suposição de mundo fechado e uso de axiomas de estado sucessor, a descrição original do Cálculo de Situações define **axiomas de efeito** e **axiomas de frame**. Um axioma de efeito descreve como e quais fluentes são afetados por uma ação. Axiomas de frame descrevem os fluentes que não são afetados por uma ação. Precisamos desses dois tipos de axiomas para deduzir o valor de um fluente após a execução uma seqüência de ações. Após a execução de uma ação a partir da situação inicial podemos deduzir, através dos axiomas de efeitos, os valores dos fluentes que essa ação afetou, mas sem os axiomas de frame não conseguimos deduzir o que aconteceu com os valores dos outros fluentes que não são afetados por essa ação. Sem os axiomas de frame, a suposição de mundo fechado fará com que todos os fluentes, cujos valores eram verdadeiros antes da execução da ação, tenham valor falso, não reproduzindo o comportamento dos mundos dinâmicos .

Devido ao fato da quantidade de fluentes que não são afetados por uma ação ser muito maior que a quantidade de fluentes que o são, o número de axiomas de frame aumenta exponencialmente com o número de ações e o número fluentes de um domínio. Para um interpretador Prolog, o uso de axiomas de estado sucessor é preferível aos axiomas de frame, pois o número de axiomas de estado sucessor a serem avaliados é muito menor do que seria o número de axiomas de frame.

Esse problema de encontrar uma forma sintética de especificar como as ações afetam, ou não, os fluentes de um domínio é conhecido como **problema da persistência** ou **problema do quadro** (*frame problem*) [McCarthy and Hayes, 1969]. Foram sugeridas soluções para o problema da per-

sistência como a **circunscrição** [Shanahan, 1997] e os **axiomas de estado sucessor** [Elkan, 1992] [Reiter, 1991]. Nesse trabalho, adotamos os axiomas de estado sucessor para solucionar o problema da persistência uma vez que essa é a solução também adotada pela linguagem Golog.

2.2 Planejando no Cálculo de Situações

Na definição de [Green, 1969], dada uma teoria de domínio $\mathcal{D}$ e uma fórmula $\phi(s)$, a tarefa de planejamento (lógico) consiste em encontrar uma seqüência de ações $\vec{a}$ tal que:

$$\mathcal{D} \models (\exists \vec{a}) Legal(\vec{a}, s_0) \wedge \phi(do(\vec{a}, s_0))$$

onde $do([a_1, ..., a_n], s)$ é uma abreviação para

$$do(a_n, do(a_{n-1}, ..., do(a_1, s)...))$$

e onde $Legal([a_1, ..., a_n], s)$ significa que

$$Poss(a_1, s) \wedge ... \wedge Poss(a_n, do([a_1, ..., a_{n-1}], s))$$

Em outras palavras, planejamento é a tarefa de encontrar uma seqüência de ações que seja executável a partir de uma situação s_0 que leve ao estado meta $do(\vec{a}, s_0)$, especificado pela fórmula ϕ, ao qual se quer chegar.

Tomando a axiomatização da Figura 2.2 como o domínio $\mathcal{D}$, e $\phi(s) \equiv Sobre(a, b, s)$, uma solução $\vec{a}$ pode ser obtida através da seqüência de inferências (Prolog) a seguir:

$Poss(desempilha(c, a), s_0)$	por (2.3), (2.5) e (2.7)	(2.11)
$Livre(a, do(desempilha(c, a), s_0))$	por (2.11) e (2.10)	(2.12)
$Livre(b, do(desempilha(c, a), s_0))$	por (2.11), (2.4) e (2.10)	(2.13)
$SobreMesa(a, do(desempilha(c, a), s_0))$	por (2.11),(2.2) e (2.8)	(2.14)
$Poss(empilha(a, b), do(desempilha(c, a), s_0))$	por (2.12), (2.13),(2.14) e (2.6)	(2.15)
$Sobre(a, b, do(empilha(a, b), do(desempilha(c, a), s_0)))$	por (2.15) e (2.9)	(2.16)

A prova de (2.16) demonstra que existe uma situação $s = do(empilha(a,b), do(desempilha(c,a), s_0))$ em que $\phi(s)$ é verdadeira. Finalmente, as provas (2.11) e (2.15) demonstram que *Legal([desempilha(c,a), empilha(a,b)],*s_0*)* é verdadeiro.

Em teoria, se especificarmos esse mesmo domínio $\mathcal{D}$ como um programa Prolog, o interpretador Prolog deveria ser capaz de encontrar o plano solução para um problema de planejamento através de uma consulta de $\phi(s)$. Na Figura 2.3, apresentamos a axiomatização do Mundo dos Blocos da Figura 2.2 adaptada para a linguagem Prolog.

De fato, uma consulta como `?-sobre(b,c,S)` obtém uma solução `S = do(empilha(b,c),s0)`. Na Figura 2.4, temos o trace dessa consulta. O interpretador Prolog realiza a unificação de S utilizando, nessa seqüência, as cláusulas das linhas 9, 6, 1, 4 e 5 da Figura 2.3.

No entanto, dependendo do estado meta ou da ordem em que são declaradas as cláusulas representando os axiomas do domínio $\mathcal{D}$, o interpretador Prolog não consegue obter a resposta para o problema. Por exemplo, se a consulta fôr `?-sobreMesa(c,S)`, o interpretador não consegue obter uma resposta pois ocorre um estouro de pilha.

A causa dessa falha está relacionada com o mecanismo de resolução do interpretador Prolog que realiza uma busca em profundidade pelas cláusulas, i.e., ele sempre irá unificar o i-ésimo resolvente da cauda de uma cláusula antes de tentar unificar o (i+1)-ésimo resolvente dessa mesma cláusula.

Na Figura 2.5, mostramos como o interpretador Prolog realiza essa tentativa de unificação na consulta `?-sobreMesa(c,S)`. Primeiro, ele tenta unificar $sobreMesa(c,S)$ com a cláusula (I). Isso o leva a tentar unificar o resolvente $poss(E,S')$ dessa cláusula com a cláusula (II), o axioma de pré-condição da ação *empilha*. A seguir, tenta unificar o resolvente $sobreMesa(X,S')$ de (II) com a cláusula (III), o que resultará em falha. Tenta unificar $sobreMesa(X,S')$ com a cláusula (IV), o que também resultará em falha. Então, tenta unificar $sobreMesa(X,S')$ com a cláusula (V), que o leva a uma tentativa de resolver $poss(E',S'')$ com (VI). Surge aqui uma nova instância da cláusula do axioma de pré-condição da ação *empilha*. O processo continua gerando infinitos ciclos de resolventes *poss* e *sobreMesa* que em algum momento irá estourar a pilha de execução do interpretador Prolog.

```
1  sobreMesa(b,s0).
2  sobreMesa(a,s0).
3  sobre(c,a,s0).
4  livre(b,s0).
5  livre(c,s0).

6  poss(empilha(X,Y),S) :- sobreMesa(X,S), livre(X,S), livre(Y,S), X \= Y.
7  poss(desempilha(X,Y),S) :- livre(X,S), sobre(X,Y,S), X \= Y.

8  sobreMesa(X,do(E,S)) :- poss(E,S), (sobreMesa(X,S), E \= empilha(X,Y); E=desempilha(X,Y)).

9  sobre(X,Y,do(E,S)) :- poss(E,S), (sobre(X,Y,S), E \= desempilha(X,Y); E=empilha(X,Y)).

10 livre(X,do(E,S)) :- poss(E,S),(livre(X,S), E \= empilha(Y,X); E=desempilha(Y,X)).
```

Figura 2.3: O domínio do Mundo dos Blocos em Prolog.

```
[trace]  ?- sobre(b,c,S).
   Call: (8) sobre(b, c, _G288)
   Call: (9) poss(_G338, _G339)
   Call: (10) sobreMesa(_G341, _G339)
   Exit: (10) sobreMesa(b, s0)
   Call: (10) livre(b, s0)
   Exit: (10) livre(b, s0)
   Call: (10) livre(_G342, s0)
   Exit: (10) livre(b, s0)
   Call: (10) b\=b
   Fail: (10) b\=b
   Redo: (10) livre(_G342, s0)
   Exit: (10) livre(c, s0)
   Call: (10) b\=c
   Exit: (10) b\=c
   Exit: (9) poss(empilha(b, c), s0)
   Call: (9) sobre(b, c, s0)
   Fail: (9) sobre(b, c, s0)
   Call: (9) empilha(b, c)=empilha(b, c)
   Exit: (9) empilha(b, c)=empilha(b, c)
   Exit: (8) sobre(b, c, do(empilha(b, c), s0))

S = do(empilha(b, c), s0)

Yes
```

Figura 2.4: Trace da consulta `?-sobre(b,c,S)`.

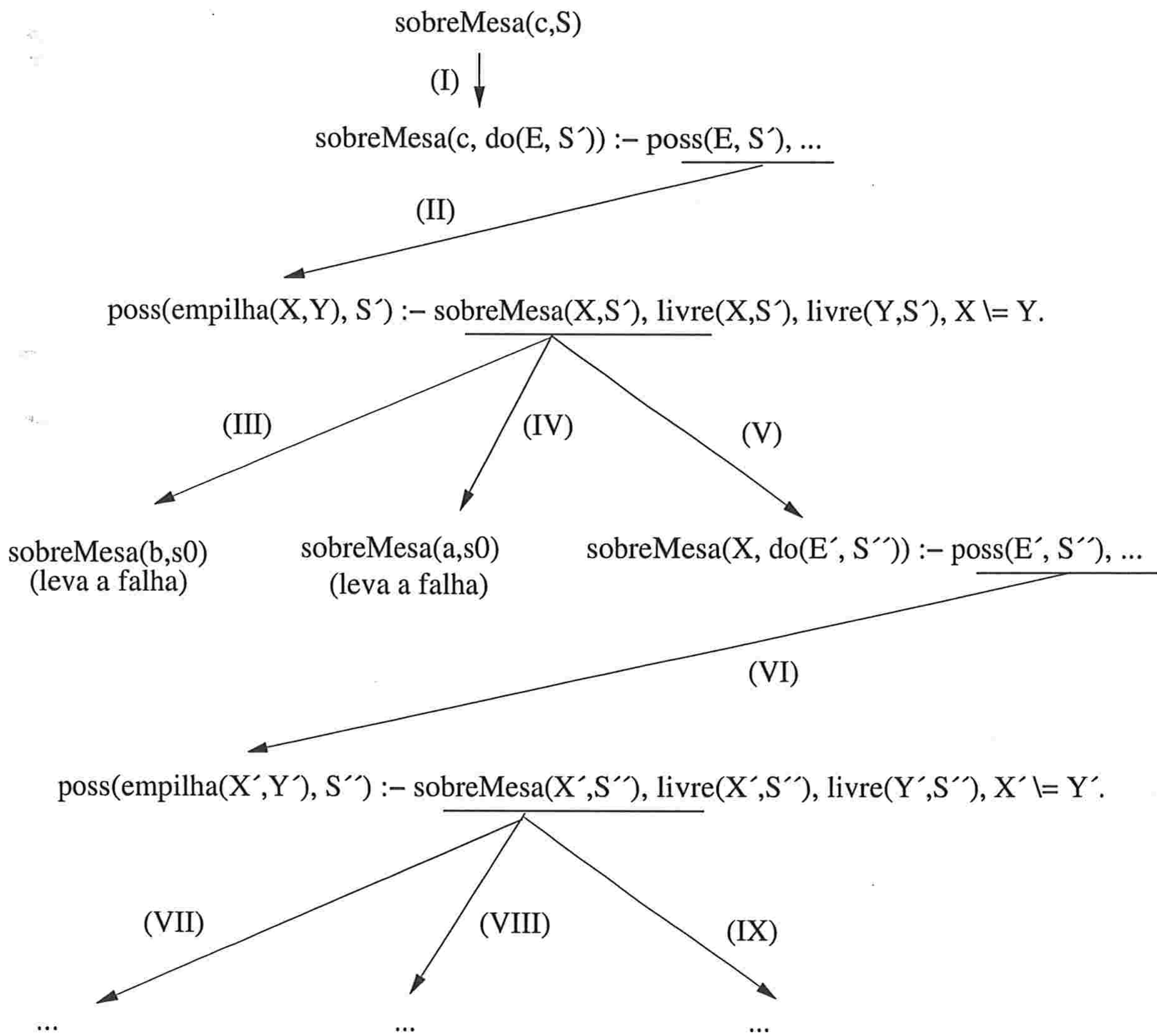

Figura 2.5: Busca Prolog para a consulta `?-sobreMesa(c,S)`.

Isso ocorre porque, a partir de um certo momento, em (II), os fluentes dos quais se quer saber o valor ficam com todos os seus parâmetros totalmente desvinculados de uma situação totalmente definida, i.e, como nenhuma ação ainda foi definida, não há como inferir os valores que os fluentes terão nos passos seguintes. A consulta da Figura 2.4 só deu certo porque em algum momento foi possível instanciar os fluentes com os blocos e a situação inicial s_0.

Uma forma de resolver esse problema é forçar o interpretador Prolog a fazer todas as buscas possíveis com planos de tamanho n antes de realizar buscas com planos de tamanho $n+1$, começando com o plano de tamanho zero que é representado pela situação s_0. Isso pode ser feito se, além dos axiomas do Cálculo de Situações, definirmos o predicado *Plan* como

$$plan(s_0).$$

$$plan(do(A, S)) : -plan(S).$$

uma consulta como

```
?- plan(S), sobreMesa(c,S).
S = do(desempilha(c, a), s0)
```

produzirá uma solução para o problema de planejamento. O problema é resolvido porque, partindo da situação inicial, podemos saber os valores dos fluentes após cada nova ação executada. Além disso, é feita uma busca em profundidade iterativa que garante que, se existir um plano solução de menor tamanho, ele será o primeiro a ser encontrado. Esse mecanismo de busca é o ponto principal dos planejadores **IDDP**, **WSPBF** e **SCP** que discutiremos no Capítulo 6.

2.3 A representação de ações STRIPS

Os operadores **Strips** [Fikes, R. and Nilsson, N. J., 1971] [Fikes, R. et al., 1972] são outra forma de representar acões em um mundo dinâmico. STRIPS deriva de uma simplificação dos axiomas do Cálculo de Situações. Sua estrutura não é tão expressiva quanto a dos axiomas do Cálculo de

Situações e nem serve para realizar inferências lógicas, mas é mais adequada para algoritmos de planejamento.

A estrutura de um operador STRIPS a é constituída por quatro componentes:

- **Action**, basicamente o nome da ação e seus parâmetros;
- **Pre(a)**, uma conjunção de átomos (fluentes) que deve ser verdadeira para que a ação que o operador representa possa ser aplicada. É equivalente às pré-condições do Cálculo de Situações;
- **Add(a)**, uma lista de *efeitos positivos* que especifica os fluentes que passam a ser verdadeiros após a execução da ação a; e
- **Del(a)**, uma lista de *efeitos negativos*, isto é, os fluentes que deixam de ser verdadeiros imediatamente após a execução da ação a.

Algumas variações da linguagem STRIPS apresentam os efeitos positivos e negativos em uma única lista, utilizando um símbolo de negação ($\neg$) para diferenciar os efeitos negativos dos positivos. Na Figura 2.6, mostramos os operadores STRIPS e as representações gráficas que correspondem às ações $desempilha(a, b)$ e $empilha(a, b)$ do Mundo dos Blocos. No lado esquerdo do retângulo que representa um operador ficam dispostas as pré-condições da ação e no lado direito ficam os efeitos da ação.

Na linguagem STRIPS, o estado do mundo (ou situação) é descrito por uma lista de átomos representando os fluentes que são verdadeiros. Também é feita a suposição de mundo fechado, i.e., considera-se que os fluentes que não estiverem na lista têm valor falso.

Quando um operador *op* é aplicado em uma situação, ele a altera da seguinte forma:

(i) os fluentes contidos na lista *Add* de *op* são adicionados à lista da situação; e

(ii) os fluentes contidos na lista *Del* de *op* são removidos da lista que representa a situação.

Tomemos, por exemplo, um estado do mundo s_0 = {*Livre(a), Sobre(a,b), SobreMesa(b)* }. Se aplicarmos o operador STRIPS que representa a ação $desempilha(a, b)$ da Figura 2.6, temos que

```
Op(Action : desempilha(a,b),
   Pre    : {Livre(a), Sobre(a,b)},
   Add    : {Livre(b), SobreMesa(a)},
   Del    : {Sobre(a,b)}
  )

Op(Action : empilha(a,b),
   Pre    : {Livre(a), Livre(b), SobreMesa(a)},
   Add    : {Sobre(a,b)},
   Del    : {Livre(b), SobreMesa(a)}
  )
```

Sobre(a,b)		¬ Sobre(a,b)
	desempilha(a,b)	Livre(b)
Livre(a)		SobreMesa(a)

SobreMesa(a)		¬ Livre(b)
Livre(a)	**empilha(a,b)**	Sobre(a,b)
Livre(b)		¬ SobreMesa(a)

Figura 2.6: Operadores STRIPS para $desempilha(a, b)$ e $empilha(a, b)$.

remover de S_0 o fluente $sobre(a, b)$ que está em Del e adicionar os fluentes $sobreMesa(a)$ e $livre(b)$ que estão em Add, transformando o estado do mundo original em S_1 = {*Livre(a), SobreMesa(b), SobreMesa(a), Livre(b)* }. Note que só podemos aplicar o operador $desempilha(a, b)$ porque o estado do mundo s_0 possui os fluentes $Livre(a)$ e $Sobre(a, b)$ da lista Pre.

2.3.1 Planejamento com STRIPS

Um plano STRIPS é uma estrutura composta de quatro elementos:

- **Steps:** uma lista de elementos S_i:op_i, onde S_i é um passo do plano e op_i é um operador STRIPS associado a esse passo;
- **Orderings:** uma lista de restrições de ordem da forma $S_m \prec S_n$ significando que o passo S_m deve ser executado antes do passo S_n ($\prec$ é o operador de ordem parcial);
- **Bindings:** uma lista de restrições de variáveis da forma $v = x$ ou $v \neq x$ onde v é uma variável de algum passo e x pode ser uma constante ou uma variável;

- **Links:** uma lista de **vínculos causais** (*causal links*) da forma $S_{add} \xrightarrow{c} S_{need}$ especificando que o passo S_{add} foi adicionado ao plano para satisfazer a pré-condição c do passo S_{need}. Uma especificação de plano pode prescindir da lista de vínculos causais, mas alguns algoritmos de planejamento analisam a informação contida nela para solucionar um problema de planejamento (como o **POP** que mostraremos mais à frente).

---

plan(
    Steps: $\{S_1 : empilha(X,Y),\ S_2 : desempilha(Z,W),\ S_3 : desempilha(M,N)\}$,
    Orderings: $\{S_2 \prec S_1\}$,
    Bindings: $\{X = a,\ Y = b,\ Z = c,\ W = X,\ M = u,\ N = v\}$,
    Links: $\{S_2 \xrightarrow{livre(a)} S_1\}$
)

Figura 2.7: Um plano na representação STRIPS.

Na Figura 2.7, mostramos um exemplo de plano STRIPS. Esse plano possui três passos $S_1 = empilha(a,b)$, $S_2 = desempilha(c,a)$ e $S_3 = desempilha(u,v)$, com a restrição de S_2 ter que ser executado antes de S_1. O vínculo causal $S_2 \xrightarrow{livre(a)} S_1$ indica que S_2 foi adicionado ao plano para satisfazer a pré-condição $livre(a)$ do passo S_1. É importante observar que essa é, na verdade, a descrição de um conjunto de planos. Não há restrições de ordem para o passo S_3, por isso ele poderia ser executado a qualquer momento antes de S_2, depois de $S1$ ou entre esses dois passos. Isso significa que o plano da Figura 2.7 representa o conjunto dos seguintes planos:

$$S_3 \prec S_1 \prec S_2$$

$$S_1 \prec S_3 \prec S_2$$

$$S_1 \prec S_2 \prec S_3$$

Um plano em que a ordem de execução de todos os seus passos está bem determinada, representando uma única seqüência de passos, é chamado de **plano totalmente ordenado**. Um plano representando várias seqüências de passos, é chamado de **plano parcialmente ordenado**.

Um problema de planejamento em STRIPS é caracterizado pela tripla $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, onde $\mathcal{O}$ é o conjunto de todos os operadores que descrevem as ações do domínio, $\mathcal{I}$ é um estado inicial, e $\mathcal{G}$ representa o conjunto de estados meta. A seguir, apresentaremos duas maneiras de resolver um problema $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$.

Planejador Strips: busca pelo espaço de estados

No planejamento pelo **espaço de estados**, ações e estados do mundo são vistos como um grafo orientado, onde os estados são os vértices do grafo, cada ação é um arco que liga um estado a outro estado. Esse tipo de planejamento consiste em encontrar um caminho por esse grafo que leve do estado inicial $\mathcal{I}$ a um estado pertencente ao conjunto do estado final $\mathcal{G}$. Na Figura 2.8, mostramos o espaço de estados do Mundo dos Blocos com apenas três blocos a, b e c. A transição de um estado para outro só pode ser feita através de ações de empilhar e desempilhar blocos.

Para resolver um problema de planejamento, basta fazer, por exemplo, uma busca em largura por esse espaço de estados. Vamos tomar como exemplo o estado inicial $\mathcal{I} = \{Sobre(a, b),\ Sobre(b, c)\}$ e o estado meta $\mathcal{G} = \{Sobre(c, b),\ Sobre(b, a)\}$. Começamos do estado inicial com um plano vazio em que nenhuma ação foi executada e vamos gerando os caminhos de tamanho 1, depois todos os de tamanho 2, e assim por diante até encontrar um que leve ao estado meta. Na Figura 2.9, mostramos os caminhos que são gerados durante a busca. Durante o processo de busca, podemos utilizar os planos de tamanho n já gerados para gerar os planos de tamanho $n+1$. Note que omitimos os planos que levam a estados já visitados por outros planos, como, por exemplo, o plano *desempilha(a,b), desempilha(b,c), empilha(b,c)*. Como o espaço de estado é finito, essa medida garante que a busca irá terminar devolvendo um plano solução ou descobrindo que o problema não tem solução.

Na Figura 2.10, mostramos um algoritmo que usa os operadores STRIPS para realizar uma busca em largura pelo espaço de estados. O algoritmo começa verificando se o estado inicial $\mathcal{I}$ pertence a $\mathcal{G}$, caso em que o plano vazio já é solução do problema. Caso contrário, chama-se a função $Plan(P)$, onde P é um conjunto vazio de planos. A função $Plan(P)$ recebe um conjunto P de todos os planos de tamanho n e faz com que a função *GeraPlanos* devolva o conjunto $Paux$ de todos os planos executáveis de tamanho $n+1$. A seguir, $Plan$ verifica se $Paux$ possui algum plano p em $Paux$ que

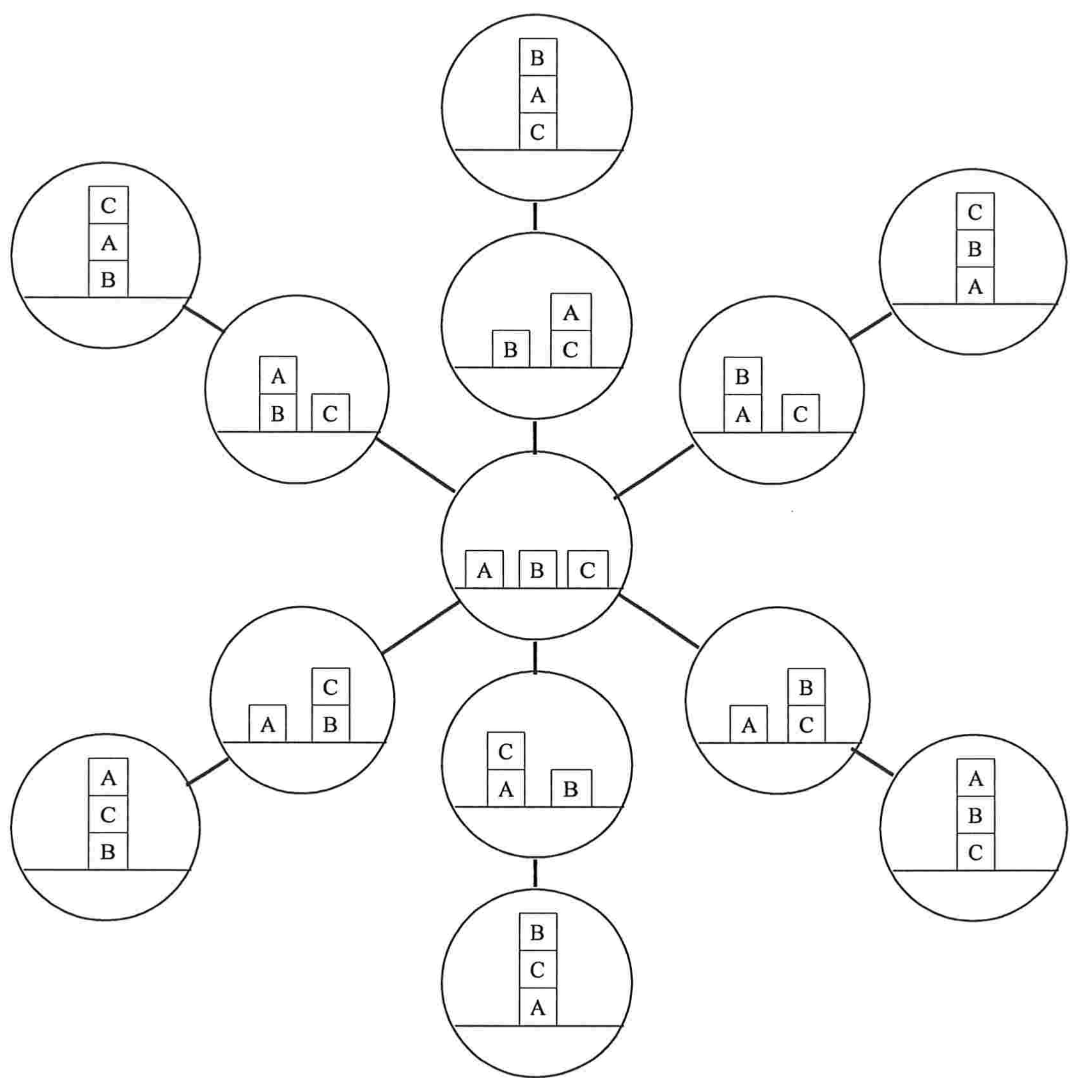

Figura 2.8: Um espaço de estados no Mundo dos Blocos.

```
Tamanho 1
   desempilha(a,b)

Tamanho 2
   desempilha(a,b) ≺ desempilha(b,c)

Tamanho 3
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(c,a)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(c,b)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(a,b)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(a,c)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(b,a)

Tamanho 4
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(c,a) ≺ empilha(b,c)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(c,b) ≺ empilha(a,c)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(a,b) ≺ empilha(c,a)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(a,c) ≺ empilha(b,a)
   desempilha(a,b) ≺ desempilha(b,c) ≺ empilha(b,a) ≺ empilha(c,b) - plano solução
```

Figura 2.9: Planos obtidos na busca em largura pelo espaço de estados

atinja o estado meta. Se houver, ele devolve p, se não, ele chama $Plan(Paux)$ para procurar por uma solução entre todos os planos de tamanho maior que $n+1$. Se houver uma solução para o problema, alguma instância de *Plan* irá devolver um plano solução de tamanho mínimo.

Planejador Strips: busca pelo espaço de planos

O **espaço de planos** pode ser visto como um grafo orientado onde cada vértice corresponde a um plano parcialmente ordenado e cada arco corresponde a uma adição de passo, uma adição de restrição (de ordem ou de variável), ou adição de um vínculo causal. Na Figura 2.11 temos um exemplo de espaço de planos. Enquanto o espaço de estados possui um número finito de vértices, o espaço de planos pode ter um número infinito de vértices pois o número de planos pode ser infinito.

O planejamento pelo espaço de planos parte de um plano inicial vazio e procura por um caminho no grafo do espaço de planos que leve a um plano que executado a partir do estado inicial atinja o estado meta. Planejadores como TWEAK [Chapman, 1987] e SNLP [Soderland, S. and Weld, D., 1991]

```
Entrada: ⟨ 𝒪, ℐ, 𝒢 ⟩

Se ℐ está em 𝒢 então
   devolve o plano vazio
Senão
   P ← {}
   Plan(P)

Função Plan(P)
   Paux ← GeraPlanos(P)
   Se Paux = {} então
      devolve falha
   Senão
      Se existe um plano p em Paux que atinge 𝒢 então
         devolve p
      Senão
         Plan(Paux)

Função GeraPlanos(P)
   Paux ← {}
   Para cada plano p em P faça
      Para cada operador op em 𝒪 faça
         Se op é executável após p então
            adiciona o plano p ≺ op a Paux
   Devolve Paux
```

Figura 2.10: Uma busca em largura pelo espaço de estados com STRIPS

realizam planejamento pelo espaço de planos utilizando operadores Strips. Para exemplificar o planejamento pelo espaço de planos, vamos apresentar o planejador **POP** (*Partial Order Planner*) [Russell, S. and Norwig, P., 1995].

O planejador POP 2.15 recebe $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ e devolve em $plan$ o plano solução, se existir. $plan$ começa com um plano inicial composto por apenas duas pseudo-ações, $Start$ e $Finish$, onde $Start \prec Finish$. $Start$ não possui pré-condições e possui como efeitos todos os fluentes de $\mathcal{I}$, e $Finish$ não possui efeitos e possui como pré-condições os fluentes de $\mathcal{G}$. Nenhum passo a ser adicionado posteriormente pode anteceder $Start$, e nenhum pode suceder $Finish$. Esse plano inicial serve para orientar o planejador na escolha das ações que deverão ser adicionadas para completar o plano solução.

Enquanto no planejador da Figura 2.10 os passos do plano eram adicionados na ordem em que seriam executados, e o critério de escolha de passo a ser adicionado levava em conta apenas se a

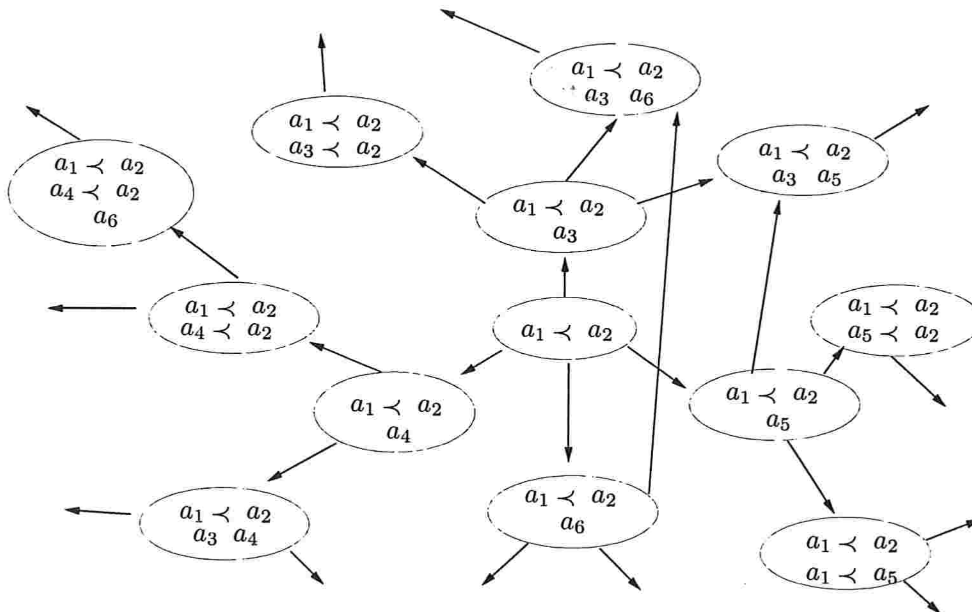

Figura 2.11: Um espaço de planos.

ação era executável no estado atual, o planejador POP adiciona os passos levando em conta se eles podem satisfazer as pré-condições das ações do plano. POP não necessariamente adiciona passos na ordem em que serão executados, e não necessariamente impõe restrições de ordem entre todos os passos do plano, i.e., POP trabalha com planos parcialmente ordenados em vez de planos totalmente ordenados.

A Figura 2.12 nos dá uma idéia de como o planejador POP constrói um plano. São dados: $\mathcal{I} = \{a, b\}$, $\mathcal{G} = \{c, d, e\}$ e $\mathcal{O}$ com três operadores *OP-1*, *OP-2* e *OP-3*. O plano inicial possui os passos *Start* e *Finish*. Convencionaremos que as pré-condições circuladas estão satisfeitas por algum passo do plano e que as não circuladas ainda não estão satisfeitas. Uma seta que parte de um passo S_{add} em direção a uma pré-condição x de um passo S_{need} indica que existe um vínculo causal $S_{add} \xrightarrow{a} S_{need}$, ou seja, o passo S_{add} foi adicionado para satisfazer a pré-condição x do passo S_{need}.

Na Figura 2.12, o plano inicial possui três pré-condições não satisfeitas: as pré-condições c, d e e do passo *Finish*. Para satisfazer c e e, o planejador POP adiciona *OP-1* para ser executado antes de

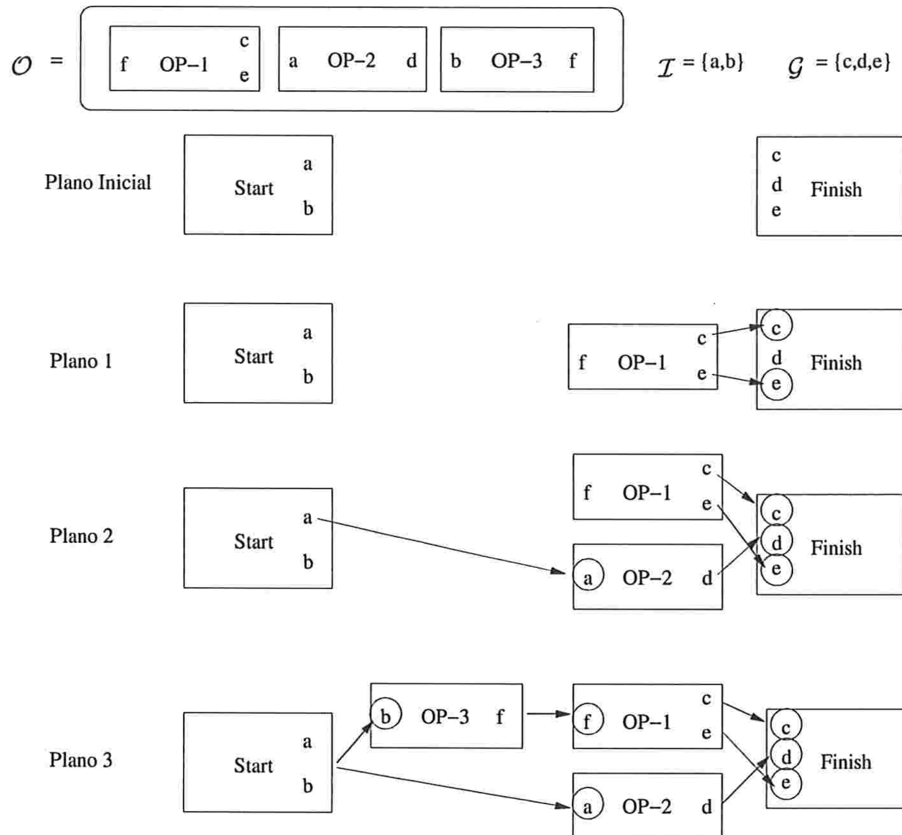

Figura 2.12: Planejamento POP.

Finish, criando o plano 1. Agora existem duas pré-condições para serem satisfeitas: d de *Finish* e f de *OP-1*. A seguir, POP adiciona *OP-2* para satisfazer a pré-condição d e cria-se o plano 2. Sobra a pré-condição f para ser satisfeita. Note que a pré-condição a de *OP-2* já é satisfeita pelos efeitos do passo *Start*. Por fim, a adição de *OP-3* satisfaz f e o plano parcialmente ordenado 3 que está com todas as pré-condições satisfeitas é solução do problema.

Note que os passos só são adicionados se puderem contribuir para satisfazer alguma pré-condição não satisfeita e, por isso, esse tipo de planejamento é chamado de **planejamento orientado a metas**. Uma vantagem desse planejamento em relação ao planejador que vimos na Figura 2.10, é que apenas os planos com ações relevantes são verificados. Não se perde tempo verificando todos os

planos possíveis.

Um problema que pode ocorrer durante a construção do plano é a possibilidade de ser adicionado um passo que desfaz o trabalho já feito por outro passo, i.e., adiciona-se um passo que torna uma pré-condição que estava satisfeita em não-satisfeita. Sabemos que uma pré-condição *pre* está satisfeita se o plano possui um vínculo causal $S_{add} \xrightarrow{pre} S_{need}$. Chamamos de **ameaça** (*threat*), a possibilidade de um passo do plano ser executado de forma a desfazer o trabalho feito por outro passo. Por exemplo, vamos supor que o operador *OP-3* também tenha como efeito $\neg a$. O plano resultante aparece na Figura 2.13.

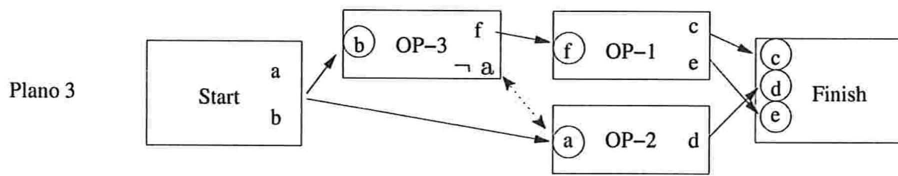

Figura 2.13: A ação OP-3 ameaça o vínculo causal $Start \xrightarrow{a} OP\text{-}2$.

Da forma como está especificado, esse plano está inconsistente. Existe a possibilidade do passo com *OP-3* ser executado antes de *OP-2*, pois não há restrição de ordem entre esses dois passos. Se isso ocorrer, o passo *OP-2* não poderá ser executado porque o passo *OP-3* desfez uma pré-condição de *OP-2*, já satisfeita pelo planejador através da ação *Start*. Há duas maneiras para se eliminar uma ameaça que um passo S_{threat}. A primeira é adicionar a restrição de ordem $S_{threat} \prec S_{add}$ (**democão**). A segunda maneira é adicionar a restrição de ordem $S_{need} \prec S_{threat}$ (**promoção**). No nosso exemplo, a ameaça é eliminada pela promoção de *OP-3* (a ameaça não pode ser eliminada por uma democão porque nenhum passo pode vir antes de *Start*). A Figura 2.14 mostra como fica o plano após a resolver a ameaça.

No algoritmo POP, após a adição de cada passo, é feita uma verificação para detecção e eliminação de ameaças. A cada passo adicionado também adicionam-se vínculos causais para indicar quais os fluentes que esse passo satisfez. Nos casos em que não é possível resolver uma ameaça por promoção, ou democão, ocorre uma falha.

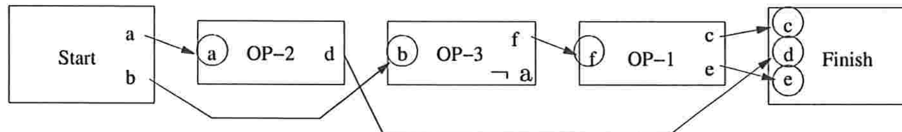

Figura 2.14: Promoção para eliminar uma ameaça em um plano.

2.4 Resumo

Neste capítulo, introduzimos a linguagem do Cálculo de Situações, os operadores STRIPS, e como utilizá-los para modelar mundos dinâmicos. Mostramos como um provador automático de teoremas pode obter uma solução para problemas de planejamento no Cálculo de Situações. Vimos que podemos utilizar os operadores STRIPS para realizar planejamento pelo espaço de estados e pelo espaço de planos, de forma progressiva e orientada a metas. A forma como a busca em profundidade iterativa do planejador pelo espaço de estados é feita será muito utilizada nos planejadores **IDDP**, **WSPBF** e **SCP** que discutiremos no Capítulo 6. O entendimento do planejador POP que descrevemos neste Capítulo será importante no Capítulo 3 para entender como o planejamento hierárquico pode realizar planejamento orientado à satisfação de metas.

```
1  Function POP(𝓘,𝓖,𝓞) retorna plan

2      plan ← CriaMínimo(𝓘, 𝓖)
3      Enquanto Verdadeiro faça
4          Se Solução(plan) então retorna plan
5          S_need, c ← SelecionaSubMeta(plan)
6          EscolheOperador(plan, 𝓞, S_need, c)
7          ResolveAmeaças(plan)
8      end

9  function SelecionaSubMeta(plan) retorna S_need, c

10     Pegue um passo de plano S_need ∈ STEPS(plan)
11       com uma pré-condição c que ainda não foi atingida
12     Devolve S_need, c

13 procedure EscolheOperador(plan, 𝓞, S_need, c)

14     Escolha um passo  S_add ∈ 𝓞 ∪ Steps(plan) onde c ∈ Add(S_add)
15     Se não houver tal passo então Falha
16     Adicione o link causal S_add --c--> S_n a Links(plan)
17     Adicione a restrição de ordem S_add ≺ S_need a Orderings(plan)
18     Se S_add é um passo vindo de 𝓞
19         Adicione S_add a Steps(plan)
20         Adicione Start ≺ S_add ≺ Finish a Orderings(plan)

21 procedure ResolveAmeaças(plan)

22     Para cada S_threat que ameaça um link S_i --c--> S_j ∈ Links(plan)
23         Escolha um dos dois:
24              Promoção: adicione S_threat ≺ S_i a Orderings(plan)
25              Demoção: adicione S_j ≺ S_threat a Orderings(plan)
26         Se ¬ Consistente(plan) então Falha
27     end
```

Figura 2.15: O planejador POP.

Capítulo 3

Planejamento hierárquico

O **planejamento hierárquico** (*Hierarchical Task-Network planning*) é uma alternativa à abordagem clássica de planejamento que têm sido utilizada em aplicações práticas como, por exemplo, no planejamento de operações militares [Wilkins, 1988] e no planejamento semi-automático para montagem dos foguetes Ariane-V da ESA, a Agência Espacial Européia [Parrod, Y. and Valera, S., 1993].

Nesse capítulo, iremos apresentar os conceitos fundamentais do planejamento hierárquico. Primeiro, na Seção 3.1, daremos uma noção intuitiva de planejamento hierárquico. A seguir, na Seção 3.2 definiremos uma linguagem para o planejamento hierárquico e, na Seção 3.3 um algoritmo que resolve problemas de planejamento nessa linguagem. Logo após, na Seção 3.4, explicaremos porque o planejamento hierárquico é indecidível. Na Seção 3.5), argumentaremos porque ele é mais expressivo que o planejamento clássico e também veremos que o planejador hierárquico pode fazer planejamento clássico orientado a metas de forma semelhante ao planejador POP. Apresentaremos o conceito de **crítica**, na Seção 3.7, para melhorar o desempenho de um planejador hierárquico e, por fim, explicaremos como o planejador hierárquico **SHOP2** funciona (Seção 3.8). Esses tópicos ajudarão a entender como o planejamento hierárquico está relacionado com a execução de programas Golog (Capítulo 5).

3.1 Noção informal de planejamento hierárquico

O conceito básico do planejamento hierárquico é a idéia de que um problema de planejamento pode ser composto por outros problemas de planejamento. Por sua vez, esses problemas também são compostos por outros problemas de planejamento, criando uma hierarquia de problemas de planejamento.

O planejamento hierárquico diferencia-se do planejamento clássico não hierárquico em três aspectos principais:

- O conceito de **redes de tarefas** (*task-networks*). Inexistente no planejamento clássico não-hierárquico, uma rede de tarefas é muito semelhante a um plano parcialmente ordenado, i.e., possui um conjunto de passos (tarefas) e restrições sobre elas. Uma rede de tarefas difere de um plano parcialmente ordenado por possuir, além de passos que são ações primitivas, passos que são ações abstratas (*tarefas compostas*) representando outras redes de tarefas.
- Diferentes tipos de ações. Enquanto o planejamento clássico não-hierárquico define apenas um tipo de ação, o planejamento hierárquico define **tarefas primitivas** e **tarefas compostas** (usa-se o termo *tarefa*, no lugar do termo *ação*). As tarefas primitivas são análogas às ações do planejamento clássico. Uma tarefa composta é uma ação abstrata que pode ser realizada por uma rede de tarefas compostas ou primitivas.
- Diferentes tipos de metas de planejamento. Enquanto a meta do planejamento clássico não-hierárquico é criar um plano que atinja um estado meta, o planejamento hierárquico tem como meta decompor uma rede de tarefas até gerar um plano composto apenas por tarefas primitivas.

Isso é feito através da substituição das tarefas compostas pelas redes de tarefas que elas representam. Essa substituição é chamada de **decomposição hierárquica**. Cada vez que ocorre uma decomposição hierárquica, podem surgir conflitos entre as tarefas já existentes na rede e as novas tarefas adicionadas. Esses conflitos devem ser resolvidos através da adição de novas restrições.

O planejamento hierárquico assemelha-se à execução de um programa composto por vários pro-

cedimentos. O programa é análogo a uma rede de tarefas inicial e cada chamada de procedimento é análoga a uma decomposição hierárquica. Têm-se a impressão de que a construção do plano que executa a rede de tarefas inicial é feita agrupando-se pequenos planos pré-fabricados (as redes de tarefas adicionadas em cada decomposição). De fato, cada tarefa composta é como um plano pré-fabricado, pois a redes de tarefas que elas representam já têm parte dos seus conflitos, ou até todos eles, resolvidos pelo projetista do domínio.

Intuitivamente, o planejador hierárquico pode ser mais eficiente porque, a cada decomposição hierárquica, ele adiciona, de uma só vez, vários passos cujos conflitos já estão previamente resolvidos, sobrando poucos conflitos para resolver. No planejador não-hierárquico, adiciona-se um passo de cada vez e não há conflitos previamente resolvidos. Outro fator importante é o número de opções nos pontos de escolha de cada tipo de planejador. Em geral, os pontos de escolha dos planejadores não-hierárquicos, as escolhas de ações, permitem escolher uma entre várias ações, dezenas ou mais delas. No planejador hierárquico, os pontos de escolha, as escolhas de decomposições, permitem escolher, uma de, em geral, duas ou três decomposições.

Um especialista de domínio especifica uma rede de tarefas de uma tarefa composta baseado em seu conhecimento de quais tarefas primitivas e/ou compostas são necessárias para realizar a tarefa de alto nível. Ele também procura especificar em que ordem os passos de uma rede de tarefas devem ser executados e quais são as restrições adicionais que devem ser obedecidas por esses passos. Como em um plano parcialmente ordenado, a rede de tarefas representa um conjunto de planos, e quanto maior fôr a quantidade de restrições adicionada a ela, menor será o número de planos nesse conjunto. É interessante que as tarefas compostas sejam especificadas por redes de tarefas o mais restritas possível. Isso diminui o número de planos que o planejador deve analisar, aumentando sua eficiência. Entretanto, uma tarefa composta, cuja rede de tarefas represente apenas um plano, é pouco flexível, têm seu uso restrito a poucas situações. Cabe ao especialista de domínio achar o equilíbrio adequado entre eficiência e flexibilidade na hora de especificar uma rede de tarefas.

3.2 Definições

Trabalhos sobre planejamento hierárquico como SIPE [Wilkins, 1988], NOAH [Sacerdoti, 1975], NONLIN [Tate, 1977] e DEVISER [Vere, 1983] enfatizam a parte algorítmica dos sistemas de planejamento hierárquico, sem formalizá-lo. Erol [Erol, 1995] foi o primeiro a propor uma formalização para o planejamento hierárquico definindo os seguintes conceitos:

- **Estados do mundo**: equivalentes aos estados do planejamento clássico, são representados, como na linguagem STRIPS por uma lista de fluentes.

- **Tarefa primitiva**: têm a forma $do[p(\vec{v})]$ onde p é o nome da tarefa primitiva e $\vec{v}$ são os parâmetros da tarefa[1]. As tarefas primitivas são ações simples que podem ser executadas diretamente por um agente. Elas são representadas por operadores STRIPS.

- **Tarefa composta**: têm a forma $perform[t(\vec{v})]$ onde t é o nome da tarefa composta e $\vec{v}$ são parâmetros. Uma tarefa composta é uma tarefa de alto nível, que representa um conjunto de tarefas (primitivas ou compostas). Para uma tarefa composta ser executável, todas as tarefas que estão nesse conjunto devem ser primitivas.

  Em muitas ocasiões, não indicaremos explicitamente as tarefas primitivas e tarefas compostas através de suas formas $do[p(\vec{v})]$ e $perform[t(\vec{v})]$. Será mais comum usar formas simplificadas $p(\vec{v})$ e $t(\vec{v})$, e o contexto dos exemplos deixará implícita a natureza dessas tarefas.

- **Tarefa meta**: têm a forma $achieve[f]$, onde f é um fluente. Uma tarefa meta é uma tarefa composta especial cuja execução torna o valor do fluente f verdadeiro.

  As tarefas meta e tarefas compostas são chamadas genericamente de **tarefas não-primitivas**.

- **Plano**: é uma seqüência de tarefas primitivas.

- **Rede de tarefas**: têm a forma $[(t_1 : \alpha_1)....(t_m : \alpha_m), \phi]$, onde cada α_i é uma tarefa, cada t_i é um rótulo para α_i, e ϕ é uma conjunção de restrições. ϕ pode conter os conectivos usuais de

[1] Não confundir o ***do*** usado por Erol com o símbolo funcional ***do*** do Cálculo de Situações

conjunção, negação e disjunção. Para fins de simplificação, em vez da forma $(t_i : \alpha_i)$, muitas vezes iremos utilizar apenas os rótulos t_i para nos referirmos a tarefas.

As restrições são das seguintes formas:

- $(v = c)$, significando que a variável v tem o valor da constante c;
- $(v_1 = v_2)$, significando que a variável v_1 é igual à variável v_2;
- $(t_1 \prec t_2)$, que são restrições de ordem significando que a tarefa rotulada por t_1 deve ser executada antes da tarefa com rótulo t_2;
- (f, t), significando que um fluente f deve ser verdadeiro antes da execução da tarefa de rótulo t;
- (t, f), significando que um fluente f é verdadeiro após a execução da tarefa de rótulo t; e
- (t, f, t'), significando que o fluente f é verdadeiro depois da execução tarefa de rótulo t e antes da execução da tarefa de rótulo t'.

Na Figura 3.1, temos uma representação gráfica da rede de tarefas a seguir:

$$
\begin{aligned}
d = [\ & (t_1 : achieve[livre(v_1)])(t_2 : achieve[livre(v_2)])(t_3 : do[move(v_1, v_3, v_2)]), \\
& (t_1 \prec t_3) \wedge (t_2 \prec t_3) \wedge \\
& (t_1, livre(v_1), t_3) \wedge (t_2, livre(v_2), t_3) \wedge (sobre(v_1, v_3), t_3) \wedge \\
& \neg(v_1 = v_2) \wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]
\end{aligned}
$$

Temos uma rede de tarefas com um tarefa primitiva $t_3 : move(v_1, v_2, v_3)$, mover o bloco v_1 de sobre o bloco v_2 para cima do bloco v_3, que é precedida pelas tarefas meta $t_1 : achieve[livre(v_1)]$ e $t_2 : achieve[livre(v_2)]$, a precedência é indicada pelas setas. Os elementos $livre(v_1)$ e $livre(v_1)$ representam, respectivamente, as restrições $(t_1, livre(v_1), t_3)$ e $(t_2, livre(v_2), t_3)$. O elemento $: sobre(v_1, v_3)$ representa a restrição $(sobre(v_1, v_3), t_3)$.

- **Rede de tarefas primitivas**: é uma rede de tarefas contendo apenas tarefas primitivas. É uma estrutura muito semelhante a um plano parcialmente ordenado, mas a rede de tarefas primitivas é mais expressiva pois possui restrições como as do tipo (t, f, t') que não existem em um plano parcialmente ordenado do planejamento clássico não-hierárquico.

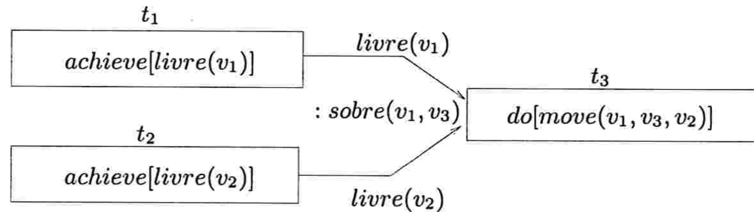

Figura 3.1: Representação gráfica de uma rede de tarefas. [Erol, 1995]

- **Método**: é uma construção da forma (α, d), onde α é uma tarefa não-primitiva e d é uma rede de tarefas. Um método descreve que uma maneira de decompor a tarefa não-primitiva α é através da rede de tarefas d, i.e., executar todas as tarefas contidas na rede d obedecendo as suas restrições é uma maneira alternativa de realizar a tarefa α. Pode haver mais de um método para uma mesma tarefa não-primitiva.

  As redes de tarefa dos métodos para tarefas meta $achieve[f]$ possuem uma estrutura especial. Possuem apenas uma tarefa primitiva $do[f(\vec{v})]$ que tem o efeito de tornar f verdadeiro. Essa rede de tarefas também possui tarefas meta $achieve[pre_i]$ para cada pré-condição da tarefa $do[f(\vec{v})]$. A Figura 3.1 é um exemplo de rede de tarefas para os métodos $(achieve[sobre(v_1, v_2)], d)$ e $(achieve[livre(v_3)], d)$. Note que a mesma rede de tarefas d serve para as tarefas meta $achieve[sobre(v_1, v_2)]$ e $achieve[livre(v_3)]$ porque a tarefa primitiva $do[move(v_1, v_3, v_2)]$ tem $sobre(v_1, v_2)$ e $livre(v_3)$ como efeitos.

  Define-se também um método para $achieve[f]$ quando o fluente f já está satisfeito. Esse método especifica uma rede de tarefas com uma tarefa primitiva *dummy* sem efeito e sem pré-condições com a única restrição $(f, dummy)$. Esse método é chamado de **método nulo**.

- **Decomposição hierárquica**. É o processo de substituição de uma tarefa não-primitiva t_i, pertencente a uma rede de tarefas d, por uma rede de tarefas d_i especificada por um método (t_i, d_i). Na Figura 3.2, temos dois exemplos de decomposição hierárquica para a tarefa não-primitiva $ir(sp, rj)$. A primeira decomposição $D1$ foi feita usando um método que especifica que para ir de São Paulo para o Rio de Janeiro deve-se ir à estação de trem e, em seguida, pegar um trem para o Rio de Janeiro. A segunda decomposição usa um método que realiza a

tarefa $ir(sp, rj)$ através de uma viagem de avião.

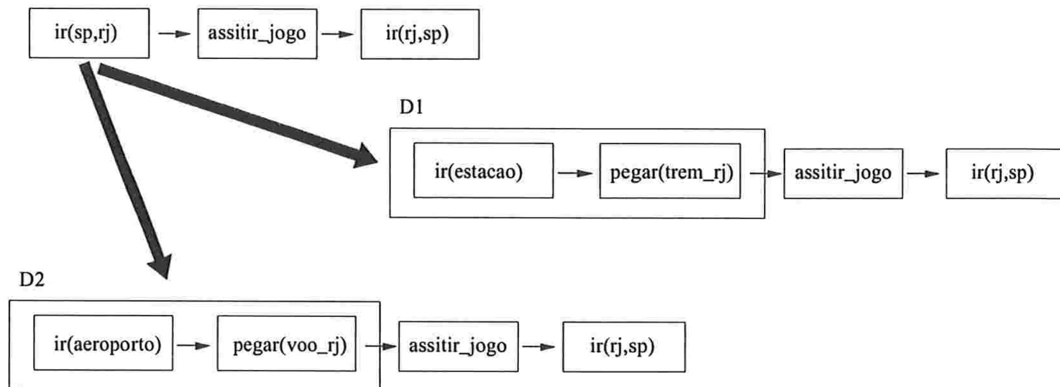

Figura 3.2: Duas decomposições possíveis da tarefa não-primitiva $ir(sp, rj)$.

- Um **problema de planejamento hierárquico** é representado como uma tupla $\langle d, I, \mathcal{D} \rangle$, onde d é uma rede de tarefas, I é uma situação inicial, e $\mathcal{D}$ é um conjunto de tarefas e métodos do domínio de planejamento.

- Uma **solução** para um problema de planejamento hierárquico $\langle d, I, \mathcal{D} \rangle$ é uma rede de tarefas primitivas sem conflitos obtida por sucessivas decomposições hierárquicas da rede de tarefas d e que pode ser executada a partir da situação I. Essa rede de tarefas primitiva especifica um plano parcialmente ordenado, mas note que, diferentemente do planejamento não-hierárquico, essa definição de solução não menciona um estado meta a ser atingido pelo plano solução. O objetivo de planejamento hierárquico é encontrar um meio de realizar as tarefas de uma rede de tarefas inicial. Isso não significa que o planejamento hierárquico não consiga realizar planejamento para satisfação de metas. O objetivo de atingir um estado meta pode ser especificado implicitamente nas redes de tarefas, ou explicitamente através de tarefas meta. Na seção 3.5, veremos como o problema clássico de planejar para atingir um fluente pode ser expresso como um problema de planejamento hierárquico.

3.3 Um algoritmo de planejamento hierárquico

O mecanismo básico de um planejador hierárquico para resolver um problema $\langle d, I, \mathcal{D}\rangle$ é um gerador de redes de tarefas, que usando métodos de $\mathcal{D}$, aplica sucessivas decomposições hierárquicas na rede de tarefas inicial d, até chegar a uma rede de tarefas primitiva que possa ser executada a partir da situação inicial I. Na Figura 3.3, temos um algoritmo de planejamento hierárquico.

1.**Entrada**: um problema de planejamento $\langle d_0, I, \mathcal{D}\rangle$.
2. Se d contêm apenas tarefas primitivas então

    Resolva os conflitos em d e devolva o resultado. *
    Se os conflitos não puderem ser resolvidos devolve *Falha*

3. Escolha não-deterministicamente uma tarefa não-primitiva t em d. *
4. Escolha não-deterministicamente um método $m = (t, d_t)$ para d. *
5. Substitua t por d_t.
6. Use críticas para encontrar interações entre as tarefas de d e tratá-las. (passo opcional)
7. Vá para a o passo 2.

* ponto de *backtracking*

Figura 3.3: Um algoritmo de planejamento hierárquico [Erol, 1995].

Esse algoritmo de planejamento hierárquico realiza uma busca pelo espaço de todas as decomposições hierárquicas possíveis da rede de tarefas inicial d. Quando ele chega no passo 2, tenta resolver os conflitos da rede de tarefas d e, quando não consegue resolvê-los, realiza *backtracking* nas escolhas de decomposições de tarefas não-primitivas. Eventualmente, o algoritmo acabará por tentar todas as possíveis seqüências de decomposições de d e caso nenhuma gere uma solução, o algoritmo devolve uma falha. Note que o passo 6 do algoritmo é opcional e inclui o uso de **críticas**. As críticas são critérios utilizados para se detectar ramos do espaço de decomposições que podem ser podados. Discutiremos mais sobre críticas na seção 3.7.

3.4 Indecidibilidade do planejamento hierárquico

Como demonstrou [Erol, 1995], o planejamento hierárquico é indecidível, ou seja, não há garantias de que a busca do planejamento pare em algum momento se o problema não tiver solução. Intuitivamente, isso se deve à possibilidade de redes de tarefas especificarem recursões, i.e., uma rede de tarefas d pode possuir uma tarefa não-primitiva que se decompõe em outra rede igual a d.

De certa forma, o planejador hierárquico da seção 3.3 é um planejador que faz busca pelo espaço de planos, pois uma rede de tarefas é parecida com um plano parcialmente ordenado. Ele começa com um plano d pouco detalhado e prossegue inserindo grupos de passos (as redes de tarefas das decomposições) até chegar a um plano totalmente detalhado (a rede de tarefas primitiva final). Também podemos dizer que o planejador hierárquico faz uma busca pelo **espaço de decomposições**, ou seja, ele faz uma busca por todas as seqüências de decomposições aplicáveis a uma rede de tarefas até encontrar uma que leve a uma rede de tarefas primitiva que seja executável a partir da situação inicial. Como o espaço de planos, o espaço de decomposições pode ser infinito apesar da rede de tarefas inicial ter um número finito de tarefas primitivas e não-primitivas, pois pode ocorrer uma recursão infinita de decomposições.

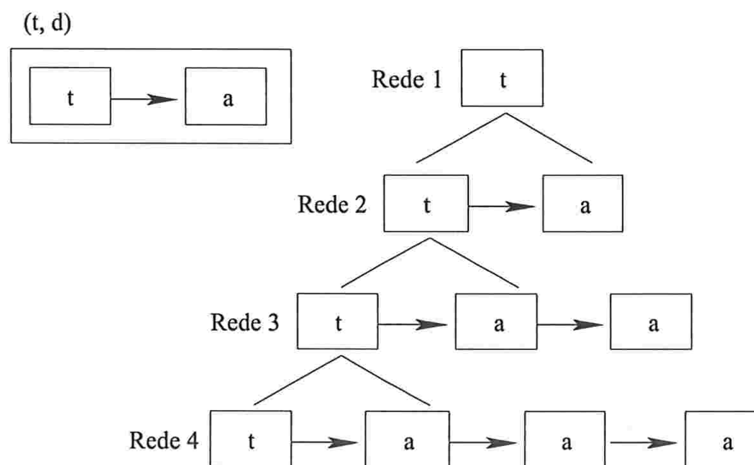

Figura 3.4: Recursão de decomposições hierárquicas.

A Figura 3.4 mostra um exemplo de recursão de tarefa não-primitiva. Nela, temos uma tarefa

não-primitiva t que é decomposta pelo método $m = (t, d)$. O método (t, d) decompõe a tarefa d em apenas duas subtarefas a e t. Nesse caso, se não houver outro método para decompor t, um método nulo, por exemplo, o planejador hierárquico nunca terminará de fazer decomposições, pois sempre haverá uma tarefa t para ser decomposta. Mesmo havendo métodos que levem ao fim dessa recursão de decomposições, a política de escolha de métodos pode levar a uma recursão infinita de decomposições. Ainda que a política de escolha de métodos procure priorizar os métodos não recursivos, as recursões serão inevitáveis se o problema de planejamento não tiver solução, pois, depois de falhar com os métodos não-recursivos, o planejador terá que testar os métodos recursivos.

Técnicas para detecção de ciclos de ações como em [Kambhampati, 1995], para podar ramos do espaço de busca que possuam ciclos, poderiam eliminar o problema da indecidibilidade. No entanto, tais técnicas podem não ser úteis porque métodos recursivos são intencionalmente especificados para criar ciclos, e algumas soluções de problemas de planejamento precisam desses ciclos. Além disso, ao eliminar os ramos do espaço de busca que possuem ciclos, corremos o risco de deixar de verificar uma solução de um problema. Dessa forma, trocaríamos a indecidibilidade pela incompletude.

3.5 Expressividade do planejamento hierárquico

Durante algum tempo, acreditou-se que a linguagem do planejamento clássico fosse equivalente à linguagem do planejamento hierárquico, i.e., que, com algum esforço, os problemas de planejamento hierárquico poderiam ser expressos na linguagem do planejamento clássico e vice-versa. O trabalho de Erol [Erol, 1995] demonstra que qualquer problema de planejamento clássico pode ser expresso como um problema de planejamento hierárquico, mas que a recíproca não é verdadeira. Portanto, a linguagem do planejamento hierárquico é mais expressiva. Intuitivamente, isso se deve ao fato de não haver, em STRIPS, algo que equivalha às redes de tarefas, às tarefas compostas, e à decomposição hierárquica. Os planos parcialmente ordenados da linguagem STRIPS são semelhantes às redes de tarefas, mas a rede de tarefas é mais flexível por poder representar restrições do tipo (f, t), (t, f) e (t, f, t'). Por outro lado, todos os elementos que aparecem em um problema de planejamento clássico STRIPS (estados, metas e ações) também aparecem no planejamento hierárquico.

Um problema de planejamento clássico pode ser expresso como um problema de planejamento hierárquico com o uso de **tarefas meta**. Por exemplo, um problema clássico $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, onde $\mathcal{G} = \{f_1, f_2\}$ pode ser expresso como o problema de planejamento hierárquico $\langle d, \mathcal{I}, \mathcal{D} \rangle$, onde d é uma rede de tarefas composta pelas tarefas *achieve[f_1]* e *achieve[f_2]*. Por outro lado, o problema de planejamento hierárquico de executar uma rede de tarefas como *desempilha(a,b)* $\prec$ *empilha(a,b)*, com o estado inicial $sobre(a, b, s_0)$, não pode ser expresso como um problema de planejamento clássico baseado em STRIPS pois não há, na linguagem STRIPS, como expressar a execução de tarefas como uma meta. Além disso, como o estado inicial e o estado final após a execução dessas duas tarefas são iguais, um planejador não-hierárquico devolveria um plano vazio.

O algoritmo de Erol para transformar um problema de planejamento clássico em planejamento hierárquico está descrito na Figura 3.5. Basicamente, ele utiliza as informações sobre efeitos e pré-condições dos operadores STRIPS para criar métodos para tarefas meta. Para especificar uma tarefa meta $achieve[f]$, o algoritmo procura pelas ações que têm f como efeito. Cada uma dessas ações será transformada na tarefa primitiva de um método para $achieve[f]$. Se duas ações possuem o efeito f, então haverá dois métodos para $achieve[f]$. O passo 4 da transformação cria os **métodos nulos** para cada tarefa meta.

A informação das pré-condições dos operadores STRIPS é utilizada para completar a descrição dos métodos para as tarefas meta. Todas as pré-condições pre_i de um operador tornam-se tarefas meta $achieve[pre_i]$ dentro da rede de tarefas de um método. A Figura 3.6 nos dá uma visão mais clara do que ocorre durante a transformação dos operadores STRIPS. Nessa figura, o operador do mundo dos blocos para a ação $desempilha(a, b)$ foi transformado em uma rede de tarefas que serve para os métodos $(achieve[livre(b)], d)$ e $(achieve[sobreMesa(a)], d)$, onde d é a rede de tarefas da Figura 3.6. Podemos interpretar que esses métodos representam o fato de que um meio de tornar $livre(b)$ verdadeiro é através da execução de $desempilha(a, b)$ e um meio de tornar $sobreMesa(a)$ verdadeiro também é executando a ação $desempilha(a, b)$. O que é apenas uma outra maneira de dizer que $livre(b)$ e $sobreMesa(a)$ são os dois efeitos da ação $desempilha(a, b)$. Note que a transformação introduz as restrições do tipo (t, f, t'), indicadas na figura por $sobre(a, b)$ e $livre(a)$. As restrições *(achieve[livre(b)], livre(b), desempilha(a,b))* e *(achieve[sobre(a,b)], sobre(a,b), desempilha(a,b))* lembram os vínculos causais dos planos no planejamento não-hierárquico. Na transformação de um problema

1. **Entrada**: um problema STRIPS $\langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, onde $\mathcal{I}$ é o estado inicial, $\mathcal{G}$ é o estado meta e $\mathcal{O}$ é o conjunto dos operadores STRIPS do domínio.

2. Para cada operador $o \in \mathcal{O}$, crie uma tarefa primitiva $f_o \in \mathcal{D}$ com as mesmas pré-condições e efeitos de o.

3. Para cada efeito l de cada operador $o =$ [Pre: pre_1, pre_2, ..., pre_k][Pos: l_1, l_2, ..., l_r] declare um método $(achieve[l], r) \in \mathcal{D}$ como ilustrado na Figura 3.6.

   $(achieve[l], r)$, onde r é a rede
   $[(n_1 : achieve[pre_1]])...(n_k : achieve[pre_k])(n : f_o),$
   $(n_1 \prec n) \wedge ...(n_k \prec n) \wedge$
   $(n_1, pre_1, n) \wedge ...(n_k, pre_k, n)]$

4. (*) Para cada pré-condição p de cada operador $o =$ [Pre: pre_1, pre_2, ..., pre_k][Pos: l_1, l_2, ..., l_r], declare um método $(achieve[p], r) \in \mathcal{D}$, onde r é a rede $[(n : dunny, (p, n)]$ caso esse método ainda não exista em $\mathcal{D}$

5. Declare $\mathcal{I}$ como o estado inicial do problema de planejamento hierárquico

6. Seja $\mathcal{G} = g_1, ..., g_k$. Declare uma rede de tarefas d

   $[(n_1 : achieve[g_1])...(n_k : achieve[g_k])(n : dummy)$
   $(n_1 \prec n) \wedge ...(n_k \prec n) \wedge$
   $(n_1, g_1, n) \wedge ...(n_k, g_k, n)],$
   onde *dummy* é uma ação sem efeitos e sem pré-condições.

Figura 3.5: Algoritmo de transformação de problema clássico em problema de planejamento HTN [Erol, 1995].

de planejamento clássico em problema de planejamento hierárquico, também é necessário definir ações primitivas que descrevem as mesmas pré-condições e efeitos descritos pelos operadores STRIPS. A Figura 3.6 não mostra mas além dos métodos $(achieve[livre(b)], d)$ e $(achieve[sobreMesa(a), d])$ também deve ser definida uma tarefa primitiva $desempilha(a, b)$ com pré-condições $sobre(a, b)$ e $livre(a)$ e efeitos $livre(b)$ e $sobreMesa(a)$.

Depois de transformar o conjunto $\mathcal{O}$ dos operadores STRIPS em um conjunto de tarefas meta de $\mathcal{D}$, o algoritmo de transformação define a rede de tarefas d a ser executada no problema de planejamento hierárquico. A rede d possui apenas tarefas meta $achieve[g_i]$, onde g_i são todos os fluentes que devem ser verdadeiros no estado meta $\mathcal{G}$ do problema original de planejamento clássico. O estado inicial do problema de planejamento clássico e do problema hierárquico são iguais. Na Figura 3.7, temos os

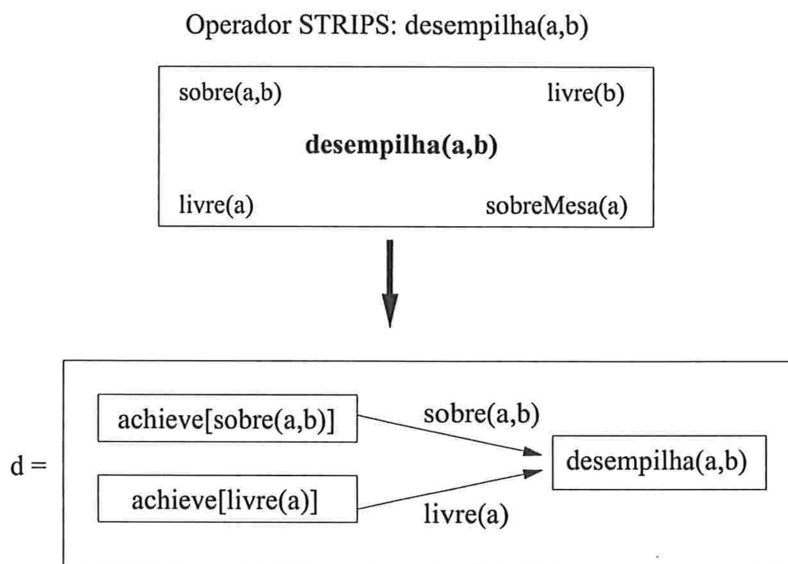

Figura 3.6: Transformação de um operador STRIPS em uma rede de tarefas que pode ser usada nos métodos $(achieve[livre(b)], d)$ e $(achieve[sobreMesa(a), d])$.

métodos e as redes de tarefas que seriam gerados a partir dos operadores STRIPS $empilha(X,Y)$ e $desempilha(X,Y)$.

3.6 Planejamento com tarefas meta

Na Figura 3.8, temos um exemplo de problema hierárquico equivalente ao problema clássico para encontrar um plano para atingir um estado meta $\{livre(b)\}$. Inicialmente, a tarefa meta *achieve[livre(b)]* de $d1$ é decomposta com o método $(achieve[livre(b)], d')$ gerando a rede $d2$. Como $d2$ não é um rede de tarefas primitiva, decomposições devem ser feitas nas tarefas não-primitivas de $d2$, $achieve[livre(a)]$ e $achieve[sobre(a,b)]$. O planejador escolhe realizar a decomposição de *achieve[sobre(a,b)]* com o método $(achieve[sobre(a,b))], d'')$, gerando a rede $d3$ e assim por diante.

É fácil notar que a seqüência de decomposições da Figura 3.8 assemelha-se ao processo de planejamento do planejador POP da Figura 2.15. Além disso, cada adição de tarefa primitiva sempre

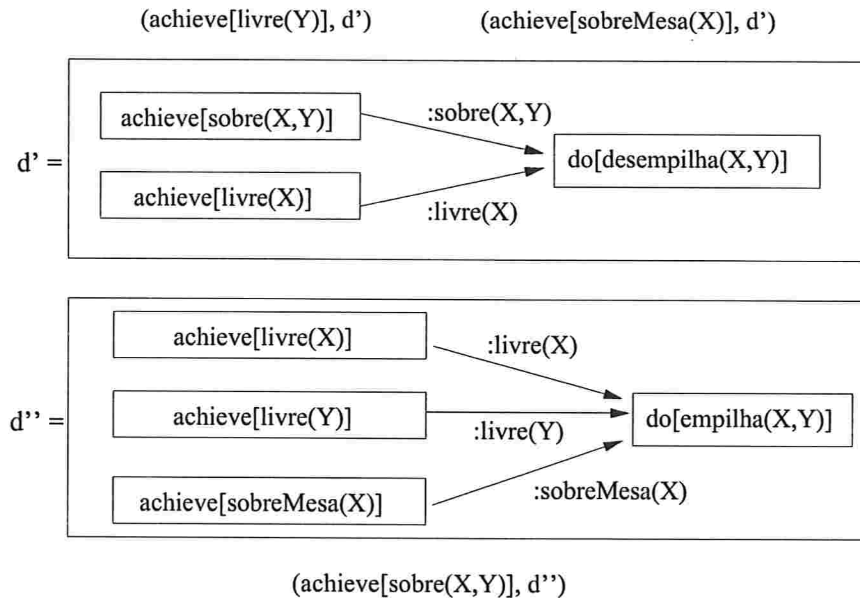

Figura 3.7: Os operadores $empilha(X, Y)$ e $desempilha(X, Y)$ convertidos em tarefas meta.

satisfaz alguma pré-condição de alguma tarefa primitiva já presente na rede de tarefas. Cada tarefa meta $achieve[f]$ funciona como se fosse uma pré-condição ou submeta ainda não satisfeita. Podemos ver que, dessa forma, o planejamento hierárquico, que é orientado à execução de tarefas, acaba realizando um planejamento orientado à satisfação de metas como no POP. O planejamento hierárquico com tarefas meta só difere do planejador POP por representar vínculos causais como restrições do tipo (t, f, t').

É importante notar que a seqüência de decomposições da Figura 3.8 gerou um ciclo de decomposições, pois a tarefa meta $achieve[sobre(a, b)]$ aparece na rede $d2$, é decomposta, e reaparece na rede $d4$. Criando um ciclo *empilha-desempilha-empilha*. Se as escolhas dos métodos utilizados nas decomposições fossem outras, ou se as escolhas das tarefas a serem decompostas fossem diferentes, esse ciclo poderia ser evitado. De fato, a Figura 3.9 apresenta uma seqüência de decomposições que resolve o problema de planejamento hierárquico. Nessa figura, foram escolhidas as decomposições nulas, uma vez que $sobre(a, b)$ e $livre(a)$ já são satisfeitas no estado inicial $\mathcal{I}$.

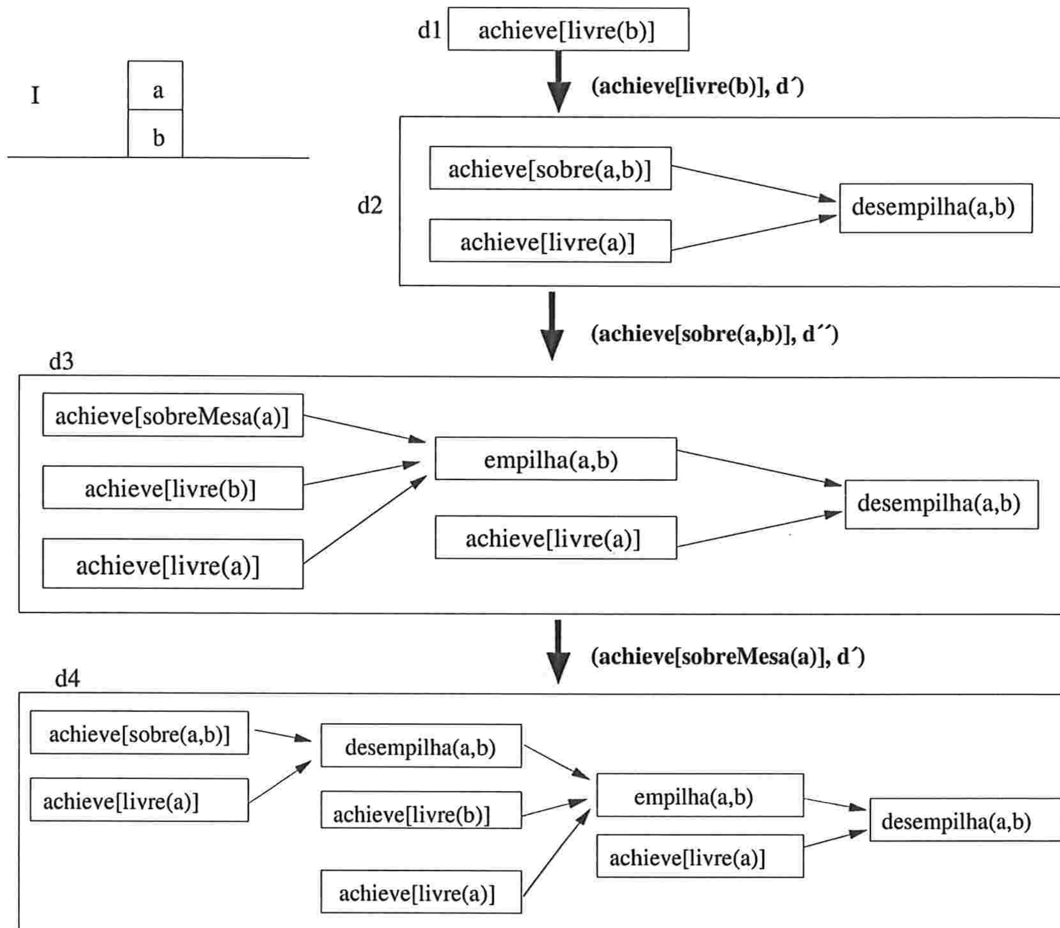

Figura 3.8: Seqüência de decomposições da tarefa meta *achieve*[*livre*(*b*)].

3.7 Críticas

O planejamento hierárquico realiza busca pelo espaço de todas as seqüências de decomposições possíveis a partir de uma rede de tarefas inicial. Em muitos casos, algumas decomposições levam a redes de tarefas cujos conflitos são insolúveis independentemente das decomposições que vierem a ser feitas depois. Quando isso acontece, podemos eliminar da busca todas as seqüências que derivem dessa seqüência com conflito insolúvel. Chamamos de **crítica** [Erol, K. et al., 1995] a análise de uma rede de tarefas que permite dizer se essa rede pode ou não gerar uma solução para um problema de

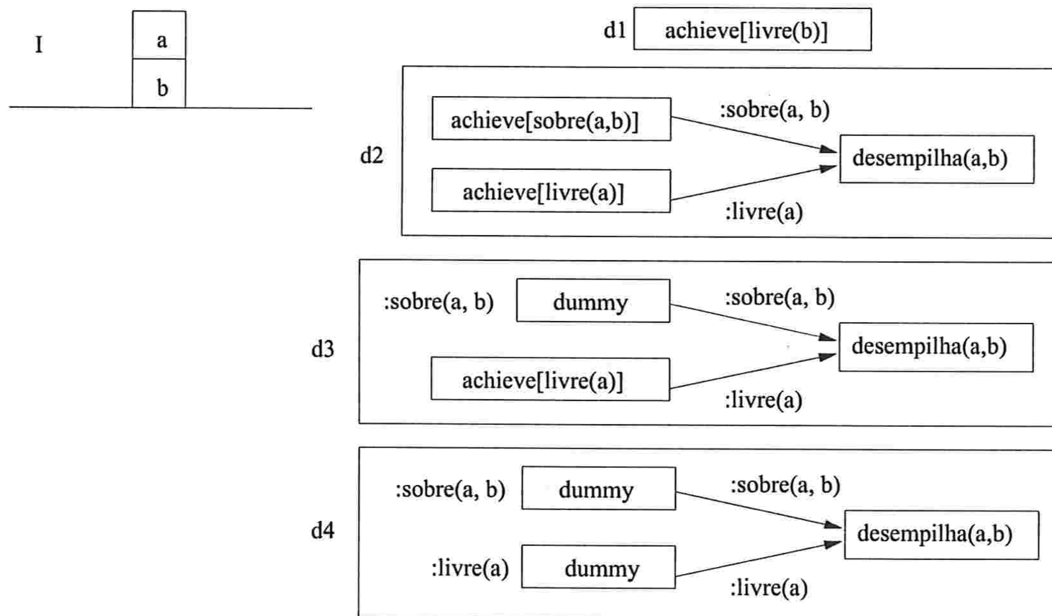

Figura 3.9: Solução para executar a tarefa meta *achieve*[*livre*(b)] evitando laços de decomposições recursivas.

planejamento hierárquico, permitindo realizar um corte no espaço de busca.

Existem críticas independentes de domínio e críticas dependentes de domínio. Erol [Erol, 1995] apresenta uma crítica independente de domínio que analisa a conjunção de restrições de ϕ na rede de tarefas e verifica se ϕ é falsa. Se ϕ fôr falsa, ocorre um corte no ramo da árvore de busca que deriva dessa rede de tarefas. Se o valor de ϕ fôr verdadeiro ou ainda não puder ser determinado, o planejador pode continuar desenvolvendo sua busca a partir dessa rede.

Quanto antes um conflito insolúvel fôr detectado, maior será o ganho obtido por uma crítica. As críticas dependentes de domínio podem ser mais precoces, portanto, mais eficientes, do que as críticas independentes de domínio na detecção desses conflitos, pois podem conter conhecimento sobre conflitos que não estão disponíveis em ϕ. Por exemplo, podemos ter uma crítica que determine que certas combinações de decomposições sempre produzem conflitos. Assim, o corte pode ser feito antes mesmo dessas decomposições ocorrerem.

Podemos ter críticas com relação aos seguintes conflitos:

- **conflitos entre efeitos e pré-condições.** Esse tipo de crítica trata os conflitos tipicamente estudados em planejamento de ordem parcial, ou seja, quando o efeito de uma ação ameaça a satisfação de uma pré-condição de outra ação do plano;
- **conflitos no uso de recursos.** Este tipo de crítica se aplica a domínios que envolvam a utilização de recursos dos seguintes tipos: (i) consumíveis, relacionados à satisfação de ações; (ii) disputados, que determinam onde as ações deverão ser executadas, como por exemplo, processadores ou máquinas;
- **eliminação de pré-condições redundantes.** Essa crítica identifica e aproveita oportunidades através do reconhecimento de metas que foram atingidas por mais do que uma ação, eliminando a necessidade de realizar decomposições redundantes.

Vejamos um exemplo de crítica dependente de domínio. Tomemos uma rede de tarefas para realizar a compra de cinco itens diferentes com limite de gasto total de cem unidades monetárias. A compra de cada item i é representada por uma tarefa composta C_i. Existem vários métodos para cada tarefa C_i, onde cada método para C_i compra o item i por um preço diferente. Suponha que sabemos de antemão que nenhum produto custa menos que dez unidades monetárias. Se na decomposição de C_1 para a inclusão do primeiro item, verificamos que esse item custará setenta unidades, é fácil ver que não será possível realizar a compra de todos os outros itens e que qualquer decomposição a partir daí será inútil e que devemos tentar outra decomposição para C_1. A verificação desse estouro de orçamento não é parte do algoritmo de planejamento hierárquico, mas pode ser facilmente implementada como uma verificação a ser feita após cada decomposição.

3.7.1 Crítica de pré-condições *observáveis* e *planejáveis*

Nesta seção, iremos sugerir uma crítica para o mundo dos blocos que evita o surgimento de alguns ciclos de decomposições como o que aparece na Figura 3.8. A análise desta crítica não é feita durante

o processo de planejamento hierárquico, mas durante a criação das tarefas meta na transformação de um problema de planejamento clássico em um problema de planejamento hierárquico.

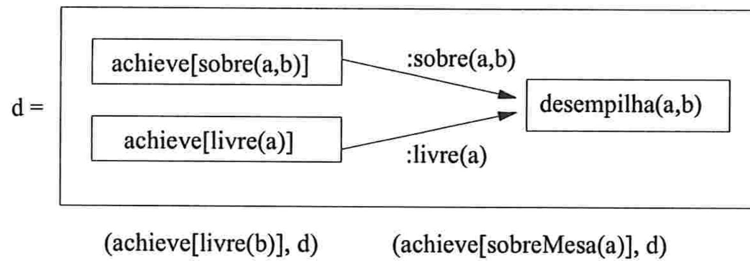

Figura 3.10: $achieve[livre(b)]$ e $achieve[sobreMesa(a)]$.

A transformação sugerida por Erol gera tarefas meta como a da Figura 3.10. Nessa figura, temos que a tarefa meta $achieve[livre(b)]$, cuja finalidade é tornar $livre(b)$ verdadeiro, pode ser executada através da execução da rede de tarefas d. Entretanto, uma das tarefas meta de d, $achieve[sobre(a,b)]$ têm como objetivo tornar a pré-condiçào $sobre(a,b)$ verdadeira, o que é incompatível com $livre(b)$. Em outras palavras, o que d especifica é que para tornar $livre(b)$ verdadeiro, antes devemos tornar $livre(b)$ falso colocando o bloco a sobre b. É esse tipo de especificação das redes de tarefas das tarefas meta que cria ciclos do tipo empilha-desempilha-empilha.

Observemos que na rede d da Figura 3.10 a execução da tarefa $achieve[sobre(a,b)]$ entra em conflito com o objetivo de tornar $livre(b)$ verdadeiro, enquanto que o mesmo não ocorre com a tarefa meta $achieve[livre(a)]$, cuja execução não compromete o objetivo de tornar $livre(b)$ verdadeiro. Assim, definiremos dois tipos de pré-condições: **observáveis** e **planejáveis**.

Definição 3.1: pré-condição observável é a pré-condição p que, na transformação de Erol, irá gerar uma tarefa meta $achieve[p]$ dentro de uma rede de tarefas d de um método $(achieve[f], d)$ tal que a execução de $achieve[p]$ torna f falso. Ela deve ser transformada em uma restrição de fluente do tipo (p,t), onde t é a tarefa primitiva da rede d.

Definição 3.2: pré-condição planejável é aquela pré-condição p que não é pré-condição observável. Ela é transformada normalmente pelo processo de Erol.

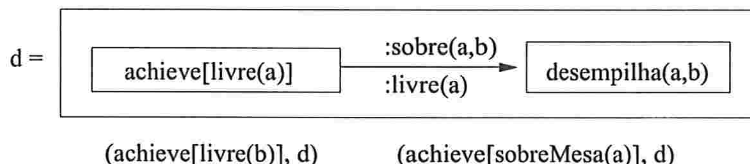

Figura 3.11: Novos métodos sem tarefas meta oriundas de pré-condições observáveis.

Por exemplo, transformar $sobre(a, b)$ em uma restrição : $sobre(a, b)$ na Figura 3.11, evita os ciclos empilha-desempilha-empilha. Por esse critério, as pré-condições observáveis dão origem a restrições que devem apenas ser observadas, enquanto as outras pré-condições se tornam tarefas meta cuja execução deve ser planejada. Dessa forma, as tarefas meta da Figura 3.10 passariam a ser como na Figura 3.11. Essa nova transformação poda todas as seqüências de decomposições em que uma tarefa meta $achieve[f]$ contêm dentro de si a tarefa meta que torna f falso. Embora, não ocorra durante o processo de planejamento hierárquico, vamos chamar essa transformação de crítica porque ela permite a poda de várias seqüências de decomposições que levariam a infinitas decomposições. Esse critério pode ser utilizado em transformações de outros problemas de planejamento clássico em problemas de planejamento hierárquico.

3.8 O planejador SHOP2

SHOP2 [Nau, D. S. et al., 2003] [Nau, D. S. et al., 2001], a segunda versão do planejador SHOP (Simple Hierarchical Ordered Planner), é considerado um planejador hierárquico eficiente e prático. Implementado na linguagem LISP, além de ter ganho um dos quatro maiores prêmios da Competição Internacional de Planejamento *International Planning Competition* em 2002, ele têm sido utilizado em pesquisas acadêmicas, em laboratórios governamentais (dos EUA) e aplicações industriais [Nau et al., 2004].

Uma característica importante do planejador SHOP2 é a de gerar redes de tarefas primitivas totalmente ordenadas. Além disso, SHOP2 adota uma política de escolha de tarefas para decomposição mais restritiva que a de planejadores como NONLIN [Tate, 1977], SIPE-2 [Wilkins, 1988], O-PLAN

[Currie, K. and Tate, A.,] e UMCP [Erol, 1995]. SHOP2 sempre escolhe para decompor uma tarefa que não possua outras tarefas não-primitivas que a preceda. Além disso, cada decomposição hierárquica de uma tarefa t por um método $m = (t, d)$ deve ser seguida por outra decomposição de alguma tarefa t' de d até que se chegue a uma tarefa primitiva (busca em profundidade). A Figura 3.12 descreve em linguagem de programação genérica o algoritmo do planejador SHOP2 de [Nau, D. S. et al., 2001].

```
1  Função SHOP2(S,M,L)
2     Se M é vazio então devolve Falha
3     Escolha uma tarefa t ∈ M sem predecessores
4     ⟨r,R1⟩ ← Redução(S,t)
5     Se r = Falha então devolve Falha
6     Escolha um operador o aplicável a r em S
7     S1 ← o estado produzido da aplicação de o em S.
8     L1 ← a lista de proteção produzida de L pela aplicação de o a r
9     M1 ← a rede produzida a partir de M pela substituição de t por R1
10    P ← SHOP2(S1,M1,L1)
11    Devolve cons(o,P)

12 Função Redução(S,t)
13    Se t é uma tarefa primitiva então devolve <t,NIL>
14    Senão
15       Se nenhum método é aplicável a t em S então devolve <Falha,NIL>
16    Escolha um método m aplicável a t em S
17    R ← a rede obtida pela de composição de t com o método m
18    r ← qualquer tarefa em R sem predecessores
19    ⟨r1,R1⟩ ← redução(S,r)
20    Se r1 = Falha então devolve ⟨Falha,NIL⟩
21    R2 ← a rede parcialmente ordenada obtida trocando r por R1 em R
22    Devolve ⟨r1,R2⟩
```

Figura 3.12: Algoritmo SHOP2 em linguagem de programação genérica.

O algoritmo possui apenas duas funções recursivas *SHOP2* e *Redução*. A entrada é dada por um estado inicial S, uma rede de tarefas parcialmente ordenada M, e uma lista de condições protegidas L. A lista L é utilizada para que o planejador possa mapear quais pré-condições de tarefas devem

ser protegidas de alterações. A princípio, o planejador não precisa da lista L, mas ela pode ajudar a evitar conflitos durante o processo de planejamento. O algoritmo descrito aqui não entra em detalhes de como isso é feito, pois queremos apenas destacar a política de decomposição do planejador SHOP2.

A função *Redução(S,t)* realiza a decomposição hierárquica recursiva da tarefa t até chegar a uma tarefa primitiva que seja executável na situação S. Note que em cada instância de *Redução(S,t)*, a tarefa t sempre é uma tarefa sem predecessores. A função *SHOP2* também sempre escolhe para decompor uma tarefa sem predecessores. Uma conseqüência dessa política de escolha e da decomposição recursiva é que, a cada instância de $SHOP2$, a situação S sempre é totalmente conhecida, pois cada chamada de *Redução* devolve uma tarefa primitiva r que é executável a partir de S, gerando uma situação S_1 que é utilizada como entrada da próxima instância de $SHOP2$. O conhecimento do estado do mundo no momento das decomposições é um dos motivos de sua eficiência, pois dá ao SHOP2 maior informação para decidir quais métodos podem ou não ser utilizados.

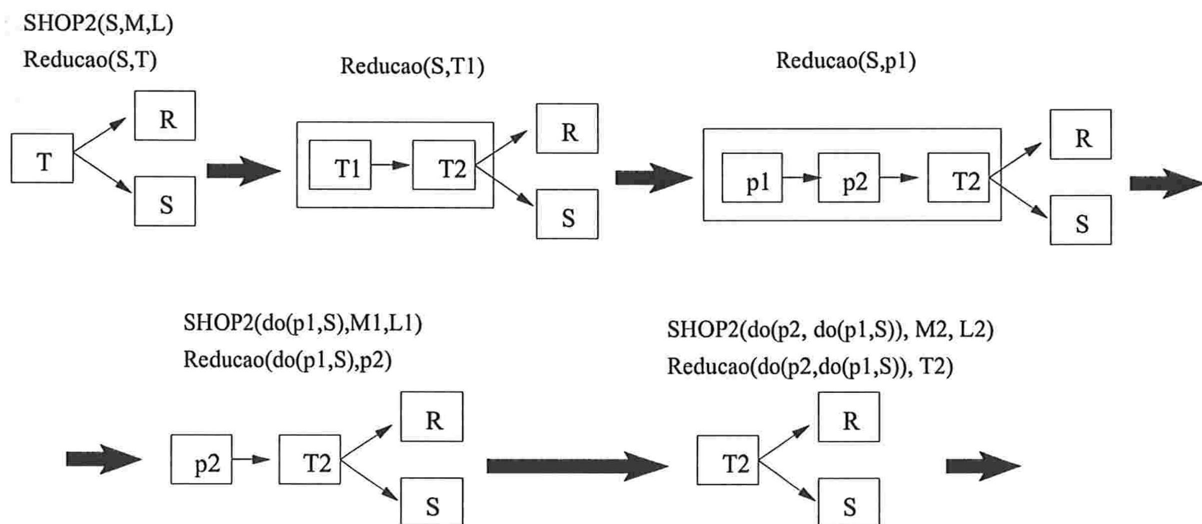

Figura 3.13: Um passo de decomposição do SHOP2 sempre escolhe a tarefa *mais à esquerda* da rede de tarefas.

Na Figura 3.13, temos um exemplo de como ocorre a decomposição no SHOP2. Nessa figura, temos uma rede de tarefas composta pelas tarefas não-primitivas T, R, e S que deve ser executada a partir de uma situação inicial I. A primeira instância de SHOP2 deve escolher para decomposição

uma tarefa sem predecessores. Só pode escolher a tarefa T e chama *Redução(S,T1)*. A tarefa T é decomposta nas subtarefas não-primitivas $T1$ e $T2$, chama-se *Redução(S,T1)*, pois $T1$ é a única tarefa entre $T1$ e $T2$ sem predecessores. $T1$ é decomposta nas tarefas primitivas $p1$ e $p2$. Como $p1$ é primitiva as decomposições recursivas acabam. Redução devolve $\langle p1, p1 \prec p2 \prec T2 \rangle$. A próxima instância de $SHOP2$ irá trabalhar com a situação após $p1$ e com a rede que sobra após a execução de $p1$. Como $p2$ é a única tarefa sem predecessores, chama-se *Redução(do(p1,S),p2)*. Já que $p2$ é uma tarefa primitiva, a recursão de *Redução* não continua. A próxima instância de $SHOP2$ irá trabalhar com a situação resultante da execução de $p1$ e $p2$ a partir de S e com a rede que sobrou. A próxima tarefa escolhida será $T2$. O processo continua até que não haja mais tarefas na rede e a primeira instância de $SHOP2$ irá devolver um plano na forma $p1 \prec p2 \prec ... \prec pn$

3.9 Resumo

Nesse Capítulo, apresentamos uma noção intuitiva de planejamento hierárquico bem como a definição formal de planejamento hierárquico. Apresentamos um algoritmo de planejamento hierárquico genérico não decidível, i.e, que pode não parar se não existir uma solução para o problema, pois o espaço de decomposições pode ser infinito. Vimos também que a linguagem do planejamento hierárquico é mais expressiva que a linguagem de planejamento clássico não-hierárquico e que qualquer problema de planejamento clássico pode ser expresso como um problema de planejamento hierárquico, mas que o contrário não é verdade. Vimos que o problema clássico de satisfação de estados meta é resolvido pelo planejamento hierárquico através de tarefas meta cuja decomposição hierárquica lembra o funcionamento do planejador clássico POP.

Além disso, vimos que críticas permitem podar ramos da árvore de busca do planejador hierárquico e definimos a noção de uma pré-crítica para o Mundo dos Blocos para evitar ciclos do tipo empilha-desempilha-empilha. Por fim, apresentamos o planejador SHOP2. O entendimento do planejamento hierárquico e do SHOP2 será importante para compreender as semelhanças entre a execução de um programa Golog e o planejamento hierárquico no Capítulo 5.

Capítulo 4

GOLOG

GOLOG [Levesque, H. J. et al., 1997] é uma linguagem criada pelo *Grupo de Robótica Cognitiva* da Universidade de Toronto que oferece vantagens para aplicações em domínios dinâmicos como programação de alto nível de agentes robóticos, controle de processos, simulação de eventos discretos, etc. Muitos dos trabalhos têm usado GOLOG para lidar com execução e sensoriamento [Reiter, 2001b] [De Giacomo, G. and Levesque, H. J., 1999] [De Giacomo, G. et al., 1998], raciocínio temporal [Reiter, 1998], sistemas multi-agentes [Shapiro, S. et al., 1997], gerenciamento de robôs para jogar futebol [Arroz, M et al., 2003] e *Semantic Web* [McIlraith, S. and Son, T. C., 2001].

Neste capítulo, apresentaremos a linguagem Golog. Explicaremos sua sintaxe, sua semântica, e o funcionamento do meta-interpretador implementado em Prolog.

4.1 A linguagem Golog

Um dos principais objetivos do *Grupo de Robótica Cognitiva* da Universidade de Toronto ao propor a linguagem Golog [Levesque, H. J. et al., 1997] foi o de facilitar a especificação de sistemas robóticos em ambientes complexos. Em geral, as linguagens de programação de robôs permitem especificar comportamentos complexos através de programas de controle sem manter explicitamente um modelo

do ambiente em que o robô deverá atuar. Ainda que algum modelo mental do ambiente tenha sido considerado pelo projetista, esse modelo está implicitamente descrito no funcionamento do programa, implicando em grandes dificuldades de manutenção, modificação do sistema, e verificação formal de propriedades.

Uma das propostas de Golog é permitir que o programador mantenha uma representação explícita do ambiente. Isso é conseguido utilizando-se sentenças do Cálculo de Situações que descrevem mundos dinâmicos. Além disso, Golog define operadores para descrever ações complexas usando símbolos extra-lógicos como, **while**, **if** e chamadas de procedimentos, por exemplo. Tais operadores são como *macros* que, quando executadas, desdobram-se em uma ou mais sentenças do Cálculo de Situações. A execução bem sucedida de um programa Golog, que é uma expressão formada por esses operadores, gera uma seqüência de tarefas primitivas. As sentenças do Cálculo de Situações geradas durante a execução do programa Golog formam uma prova de que essa seqüência de tarefas primitivas é executável.

Na definição de [Green, 1969], dadas uma teoria de domínio $\mathcal{D}$ e uma fórmula $\phi(s)$, a tarefa de planejamento consiste em encontrar uma seqüência de ações $\vec{a}$ tal que:

$$\mathcal{D} \models (\exists \vec{a}) Legal(\vec{a}, s_0) \wedge \phi(do(\vec{a}, s_0))$$

em que $do(\vec{a}, s)$ é uma abreviação para $do([a_1, ..., a_n], s_0)$, que por sua vez é a abreviação de:

$$do(a_n, do(a_{n-1}, ..., do(a_1, s_0)...)),$$

$\phi(do(\vec{a}, s_0))$ significa as fórmulas do estado meta satisfeitas no estado $do(\vec{a}, s_0)$,

e $Legal([a_1, ..., a_n], s)$ é uma abreviação para

$$Poss(a_1, s) \wedge ... \wedge Poss(a_n, do([a_1, ..., a_{n-1}], s))$$

Em outras palavras, planejamento é a tarefa de encontrar uma seqüência de ações que seja executável a partir de uma situação s_0 que atinja o estado meta especificado pela fórmula ϕ.

Planejamento em Golog é a **execução de um programa de alto nível**, ou seja, a transformação de um programa de alto nível em um programa executável. O interpretador Golog transforma o programa original em um programa executável sem de fato executar ações no ambiente real, i.e., Golog gera um plano de ações primitivas. Formalmente [De Giacomo, G. et al., 2000a], dados uma teoria de domínio $\mathcal{D}$ e um programa δ, o problema da execução deum programa de alto nível é encontrar uma seqüência de ações $\vec{a}$ tal que:

$$\mathcal{D}, \delta \models (\exists \vec{a}) Do(\delta, s_0, do(\vec{a}, S_0))$$

onde $Do(\delta, s, s')$ significa que o programa δ, executado a partir da situação s_0, termina de forma legal na situação $do(\vec{a}, S_0)$.

Enquanto o planejamento clássico faz uma busca pelo espaço de todos os planos possíveis até encontrar algum que atinja o estado meta, um interpretador de programa Golog limita a sua busca apenas aos planos que obedecem às especificações do programa. Apesar disso, Golog permite um certo grau de não-determinismo através do uso de operadores de escolha não-determinística (veja operadores Golog na Tabela 4.1) que indicam pontos de escolha na execução do programa. Operadores não-determinísticos permitem especificar comportamentos mais flexíveis, o que é particulamente útil em ambientes dinâmicos. Assim, a linguagem Golog permite que o programador possa definir o grau de não-determinismo em um programa, indo de um *script* totalmente determinístico até um programa altamente não-determinístico.

4.2 Sintaxe e semântica da linguagem Golog

Como as linguagens procedurais, Golog possui operadores como *if*, *while* e chamadas de procedimentos. Além disso, Golog possui outros operadores como, por exemplo, a **escolha não determinística de ações** e a **escolha não-determinística de argumentos**. Na Tabela 4.1 listamos as palavras-chave que representam os operadores Golog, juntamente com as suas sintaxes e uma breve descrição de sua semântica.

A semântica de Golog é chamada de **semântica computacional** pois baseia-se na computação

Op	Nome	Uso	Semântica
:	Seqüência de ações	$\delta_1 : \delta_2$	δ_1 e δ_2 são programas. Todas as ações de δ_2 devem ser executadas depois das ações de δ_1.
?	Teste de condição espera	$?(\phi)$	ϕ é uma expressão. O programa só continua se a avaliação de ϕ é bem sucedida.
#	Escolha não-determinística de ação	$\delta_1 \# \delta_2$	δ_1 e δ_2 podem ser ações primitivas ou programas. Um dos dois programas é escolhido para execução
pi	Escolha de argumentos	$pi(x, \delta)$	x é uma variável e δ é um programa. Escolha um valor para x, e todas as ocorrências de x no programa δ terão esse valor.
while	While	$while(\phi, \delta)$	ϕ é uma expressão e δ é um programa Enquanto a expressão ϕ fôr verdadeira executar o programa δ
if	If	$if(\phi, \delta_1, \delta_2)$	ϕ é uma expressão, δ_1 e δ_2 são programas Se a expressão ϕ é verdadeira, execute o programa δ_1, caso contrário, execute o programa δ_2
iter	Iteração	$iter(\delta)$	δ é um programa Execute o programa δ zero ou mais vezes
conc	Concorrência	$conc(\delta_1, \delta_2)$	δ_1 e δ_2 são programas Os programas δ_1 e δ_2 podem ser processados concorrentemente. Em algumas versões de Golog, o operador é representado por $\|\|$
prconc	Concorrência com prioridade	$prconc(\delta_1, \delta_2)$	δ_1 e δ_2 são programas Os programas δ_1 e δ_2 são processados concorrentemente, mas, sempre que possível, o processamento de δ_1 tem prioridade
iterconc	Concorrência de iterações	$iterconc(\delta)$	δ é um programa Uma ou mais instâncias do programa δ podem ser processadas concorrentemente
proc	Definição de procedimentos	$proc(P(\vec{v}), \delta)$	δ é um programa, $\vec{v}$ é um conjunto de argumentos. Definição do procedimento P com argumentos de entrada $\vec{v}$ e o programa δ sendo o corpo do procedimento.

Tabela 4.1: Sintaxe e semântica dos operadores da linguagem Golog.

de pequenos passos de um programa, ou pequenas **transições** de programa. Adota-se essa semântica porque é mais adequada para lidar com concorrência [De Giacomo, G. et al., 2000a]. Aqui, os pequenos passos são ações primitivas e testes para verificar se uma condição é verdadeira na *situação atual*. Essas transições de programa são definidas através do predicado $Trans$, onde $Trans(\delta, s, \delta', s')$ significa que a configuração de programa (δ, s) pode sofrer uma transição para a configuração de programa (δ', s'). Intuitivamente, podemos entender que quando um programa δ sofre uma transição a partir se s, ocorre a execução de um pequeno passo de δ, e δ' é o que ainda resta para ser executado do programa e s' é a situação resultante da execução desse pequeno passo.

Também precisamos introduzir o predicado $Final$, onde $Final(\delta, s)$ é verdadeiro se a configuração de programa (δ, s) é uma configuação final, i.e., se o programa δ pode ser considerado legalmente terminado na situação s. É conveniente também definir o símbolo nil para representar o programa vazio, que denota que não há mais nada para ser executado.

4.2.1 Caracterização do predicado $Trans$

A seguir, apresentamos os axiomas de equivalência que caracterizam o predicado $Trans$ de acordo com [De Giacomo, G. et al., 2000b].

Ações primitivas

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

Esse axioma define que um programa que consiste apenas de uma ação primitiva a sofre uma transição para o programa vazio nil se é possível executar a ação $a[s]$ na situação s. O termo $a[s]$ corresponde à ação a com o valor de seus argumentos na situação s. Por exemplo, seja a ação $mover(b, room(b), room(c))$, que significa mover um objeto b da sala onde b está ($room(b)$) para a sala onte está o objeto c ($room(c)$). Os locais onde b e c estão diferem de uma situação para outra. Portanto, $Trans$ deve avaliar os valores de $room(b)$ e $room(c)$ na situação s para saber se $mover(b, room(b), room(c))$ pode ser executada.

Ação de Teste/Espera

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

Este axioma define que um programa composto por um teste (operador ?) da condição ϕ na situação s sofre transição se a condição ϕ é verdadeira na situação s, resultando em um programa vazio, sem alterar o estado do mundo. Pelo fato de não alterar o estado do mundo, os testes são chamados de *pseudo-ações*. A razão para os testes serem considerados *ações de espera* está relacionada com o fato do interpretador Golog realizar a execução do programa como uma busca. A cada falha de teste, o interpretador realiza *backtracking* até encontrar alguma execução, se existir, que consiga passar pelo teste.

Seqüência

$$\begin{aligned} Trans(\delta_1 : \delta_2, s, \delta', s') \equiv \quad & \exists \gamma . \delta' = (\gamma : \delta_2) \ \wedge \ Trans(\delta_1, s, \gamma, s') \ \vee \\ & Final(\delta_1, s) \ \wedge \ Trans(\delta_2, s, \delta', s') \end{aligned}$$

Este axioma determina que se dois programas δ_1 e δ_2 estão em seqüência (operador :), sempre deve-se realizar uma transição de δ_1, se possível, i.e., se δ_1 não estiver na configuração final. Caso contrário realiza-se a transição de δ_2.

Escolha não-determinística de ações

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

Este axioma determina que a transição de um programa na configuração $\delta_1 \# \delta_2$ é a transição do programa δ_1 ou a transição do programa δ_2. Um dos programas é escolhido para ser executado e o outro é descartado.

Escolha não-determinística de argumento

$$Trans(pi(\nu, \delta), s, \delta', s') \equiv \exists x.Trans(\delta^{\nu}_{x}, s, \delta', s')$$

Este axioma especifica que, na transição do programa δ, deve-se escolher um argumento x que deve ser substituído por todas as ocorrências de ν em δ, resultando no programa δ^{ν}_{x} em s no qual é feita a transição de δ^{ν}_{x} para δ'.

Iteração

$$Trans(iter(\delta), s, \delta', s') \equiv \exists \gamma.(\delta' = \gamma : iter(\delta)) \wedge Trans(\delta, s, \gamma, s')$$

O operador *iter* representa a execução de um programa zero ou mais vezes. Este axioma especifica que o programa $iter(\delta)$ sofre uma transição para um programa constituído de δ após uma transição em s seguido de um novo $iter(\delta)$, o que permite que δ seja executado várias vezes. Dado que $iter(\delta)$, por definição, pode ser considerada uma configuração final de programa (ver a definição de Final na Seção 4.2.2), uma das possíveis transições de $iter(\delta)$ é não sofrer transição.

Exemplo de transição com o operador *iter*:

$$\delta \equiv iter(a_1 : a_2 : a_3) \qquad\qquad \delta' \equiv a_2 : a_3 : iter(a_1 : a_2 : a_3)$$

If-Then-Else

$$Trans(if(\phi, \delta_1, \delta_2), s, \delta', s') \equiv (\phi[s] \wedge Trans(\delta_1, s, \delta', s')) \vee (\neg\phi[s] \wedge Trans(\delta_2, s, \delta', s'))$$

Este axioma define que a transição do programa $if(\phi, \delta_1, \delta_2)$ será a transição do programa δ_1 se a expressão ϕ fôr verdadeira, caso contrário, será a transição do programa δ_2. Este axioma define um *if* sincronizado, i.e, o teste de ϕ é seguido **imediatamente** pela transição de δ_1 ou de δ_2. Devido ao operador de concorrência *conc*, existe a possibilidade de algum programa γ, executando concorrentemente com o programa $if(\phi, \delta_1, \delta_2)$, ser intercalado entre o teste $?(\phi)$ e a transição de δ_1 (ou δ_2). O programa γ pode alterar o valor de ϕ de forma que a escolha de δ_1 (ou δ_2) feita por if não seja mais válida. A transição de δ_1 (ou δ_2) logo em seguida ao teste de ϕ garante que a escolha

de if não seja afetada por ações concorrentes.

Note que o axioma $Trans$ para o operador if indica que o operador if pode ser implementado a partir dos operadores de teste (?), escolha não-determinística de ações (#) e seqüência (:) da seguinte maneira:

$$(?(\phi)\ :\ \delta_1)\ \#\ (?(\neg\phi)\ :\ \delta_2)$$

While

$$Trans(while(\phi,\delta),s,\delta',s') \equiv \exists\gamma.(\delta' = \gamma : while(\phi,\delta))\ \wedge\ \phi[s]\ \wedge\ Trans(\delta,s,\gamma,s')$$

Este axioma define que o programa δ deve ser executado enquanto a condição ϕ fôr verdadeira, ou seja, ele define um laço *while*. Esse laço é implementado pela transição do programa $while(\delta)$ para o programa constituído pela transição de δ em s seguida por $while(\delta)$. Note que, assim como o operador if, o teste de ϕ é sincronizado de forma que ocorre uma transição de δ imediatamente após o teste. Exemplo de transição com operador *while*:

$$\delta \equiv while(\phi,\ a_1 : a_2 : a_3) \qquad\qquad \delta' \equiv a_2 : a_3 : while(\phi,\ a_1 : a_2 : a_3)$$

Note que o operador *while* pode ser implementado a partir dos operadores de iteração (*iter*), teste (?), seqüência da seguinte maneira:

$$iter(\delta)\ :\ ?(\neg\phi)$$

Concorrência

$$\begin{aligned} Trans(conc(\delta_1,\delta_2),s,\delta',s') \equiv\ & \exists\gamma.\delta' = (conc(\gamma,\delta_2))\ \wedge\ Trans(\delta_1,s,\gamma,s')\ \vee \\ & \exists\gamma.\delta' = (conc(\delta_1,\gamma))\ \wedge\ Trans(\delta_2,s,\gamma,s') \end{aligned}$$

Este axioma define que um programa composto por dois programas δ_1 e δ_2 em concorrência sofre transição para um programa $conc(\delta',\delta_2)$ com δ' sendo o resultado de uma transição de δ_1 ou transição para um programa $conc(\delta_1,\delta')$ com δ' sendo o resultado de uma transição de δ_2. Esse axioma permite intercalar a execução das ações primitivas do programa δ_1 com a execução das ações primitivas do

programa δ_2.

Concorrência com prioridade

$$
\begin{aligned}
&Trans(prconc(\delta_1, \delta_2), s, \delta', s') \equiv \\
&\qquad \exists\gamma.\delta' = (prconc(\gamma, \delta_2)) \ \wedge \ Trans(\delta_1, s, \gamma, s') \ \vee \\
&\qquad \exists\gamma.\delta' = (prconc(\delta_1, \gamma)) \ \wedge \ Trans(\delta_2, s, \gamma, s') \ \wedge \ \neg\exists\zeta, s''.Trans(\delta_1, s, \zeta, s'')
\end{aligned}
$$

Este axioma define uma concorrência semelhante à do operador de concorrência *conc*. A única diferença está no fato de transições de δ_2 nunca ocorrerem enquanto existir a possibilidade de realizar transiçoes em δ_1.

Iteração com concorrência

$$Trans(iterconc(\delta), s, \delta', s') \ \equiv \ \exists\gamma.\delta' = (conc(\gamma, iterconc(\delta)) \ \wedge \ Trans(\delta, s, \gamma, s')$$

Este axioma define que um programa δ poder ser executado zero ou mais vezes, sendo que cada nova instância de δ pode ser executada concorrentemente com as outras instâncias de δ.

Chamada de procedimento

$$Trans(p, s, \delta', s') \ \equiv \ \exists\rho, \ \rho = proc(p, \delta) \ \wedge Trans(\delta, s, \delta', s')$$

Esse axioma define que um programa composto por uma chamada a um procedimento p sofre transição para o programa δ', com δ' sendo o resultado da transição de δ que é o corpo do procedimento p. Uma chamada de procedimento é feita substituindo-se a chamada pelo corpo δ do procedimento seguida imediatamente por uma transição de δ em s.

4.2.2 Caracterização do predicado Final

A seguir, apresentamos os axiomas de equivalência que caracterizam o predicado *Final* de acordo com [De Giacomo, G. et al., 2000b].

A1. Programa vazio

$$Final(nil, s) \equiv Verdadeiro$$

A2. Ação primitiva

$$Final(a, s) \equiv Falso$$

A3. Ação de Teste/Espera

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

A4. Seqüência

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

A5. Escolha não-determinística de ações

$$Final(\delta_1 \# \delta_2, s) \equiv Final(\delta_1, s) \ \vee \ Final(\delta_2, s)$$

A6. Escolha não-determinística de argumentos

$$Final(pi(\nu, \delta), s) \equiv \exists x.Final(\delta_x^\nu, s)$$

A7. Iteração

$$Final(iter(\delta), s) \equiv Verdadeiro$$

A8. If-Then-Else

$$Final(if(\phi, \delta_1, \delta_2), s) \equiv \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s)$$

A9. While

$$Final(while(\phi, \delta), s) \equiv \neg\phi[s] \vee Final(\delta, s)$$

A10. Concorrência

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

A11. Concorrência com prioridade

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

A12. Iteração com concorrência

$$Final(iterconc(\delta), s) \equiv Verdadeiro$$

Esses axiomas podem ser intuitivamente entendidos da seguinte maneira:

A1. (nil, s) é uma configuração final.

A2. (a, s) não pode ser considerado uma configuração final, já que ainda resta uma tarefa primitiva a para ser executada.

A3. $(?(\phi), s)$ não pode ser considerado uma configuração final, já que ainda resta um teste da condição ϕ para ser executada.

A4. $(\delta_1 : \delta_2, s)$ pode ser considerada uma configuração final se δ_1 e δ_2 são configurações finais.

A5. $(\delta_1 \# \delta_2, s)$ pode ser considerada uma configuração final se δ_1 ou δ_2 são configurações finais.

A6. $(pi(\nu, \delta), s)$ pode ser considerada uma configuração final se existe um valor x que substituído em todas as ocorrências de ν em δ resultam em um programa δ^{ν}_{x} que está em configuração final em s.

A7. $(iter(\delta), s)$ é uma configuração final já que $iter(\delta)$ pode ser executado zero vezes.

A8. $(if(\phi, \delta_1, \delta_2), s)$ pode ser considerada uma configuração final se $\phi[s]$ é verdadeiro e (δ_1, s) está em uma configuração final ou se $\phi[s]$ é falso e (δ_2, s) está em uma configuração final.

A9. $(while(\phi, \delta), s)$ pode ser considerada uma configuração final se $\phi[s]$ é falso ou se (δ, s) está em uma configuração final.

A10. $(conc(\delta_1 : \delta_2), s)$ pode ser considerada uma configuração final se δ_1 e δ_2 são configurações finais.

A11. $(prconc(\delta_1 : \delta_2), s)$ pode ser considerada uma configuração final se δ_1 e δ_2 são configurações finais.

A12. $(iterconc(\delta), s)$ é uma configuração final já que $iterconc(\delta)$ pode ser executado zero vezes.

Com a definição de *Final* podemos definir quando o programa termina legalmente, enquanto que com a definição de *Trans* podemos definir como ocorre cada passo da execução de um programa Golog. A seguir, veremos como é definida a transição completa de um programa através dos predicados *Trans∗* e *Do*.

4.3 Os predicados *Do* e *Trans**

O predicado $Trans*(\delta, s, \delta', s')$ é verdadeiro se a configuração de programa δ' pode ser atingida na situação s' a partir de sucessivas transições da configuração de programa δ na situação s. Intuitivamente, *Trans∗* define quais são todas as possíveis configurações de programa. A seguinr, apresentamos os axiomas que caracterizam *Trans∗* [De Giacomo, G. et al., 2000b]:

$$Verdadeiro \supset Trans*(\delta, s, \delta, s)$$
$$Trans(\delta, s, \delta'', s'') \wedge Trans*(\delta'', s'', \delta', s') \supset Trans*(\delta, s, \delta', s')$$

Esses axiomas definem o predicado *Trans∗* recursivamente. O primeiro axioma define que um programa δ é uma configuração atingível para um programa δ, ou seja, o programa não sofre transição. O segundo axioma define que uma configuração de programa δ' é atingível para um programa δ se δ' é o resultado de uma transição de uma configuração δ'' que é atingível a partir de δ. Em outras palavras, *Trans∗* define o processo de execução de um programa Golog, constituído por sucessivas transições do programa inicial.

É importante notar que juntamente com cada transição de programa também ocorre uma transição de situação. A necessidade de atualizar a situação vem do fato de ser necessário carregar um histórico

das ações que foram *executadas* até o momento. Esse histórico permite que seja possível utilizar o Cálculo de Situações para saber se uma ação primitiva é executável ou se um teste de fluente é bem-sucedido.

Com a definição de $Trans*$ e $Final$, podemos enunciar o axioma que caracteriza o predicado Do:

$$Do(\delta, s, s') \equiv \exists \delta'.Trans*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

que define a execução de um programa Golog δ em s, gerando uma situação s' como o problema de encontrar s' através de sucessivas transições de δ sendo que δ' é uma configuração final. Com esse axioma definimos a tarefa de encontrar uma execução legal de um programa na linguagem Golog. Na seção 4.4, veremos como o interpretador Golog implementa essas definições.

4.4 O interpretador Golog

A seguir, apresentaremos a implementação em Prolog de algumas das definições vistas nas seções anteriores. A implementação completa do interpretador pode ser vista no Apêndice B.

$Trans$ - Ação primitiva

$$trans(A, S, nil, do(A, S)) \ :- \ primitive(A),\ sub(now, S, A, AS),\ poss(AS, S). \tag{4.1}$$

Essa é a implementação do axioma $Trans$ para uma ação primitiva. Se A é uma ação primitiva, então o resolvente $sub(now, S, A, AS)$ gera a ação AS que é a ação A com os seus parâmetros substituídos pelos valores que teriam na situação S. AS é então avaliada quanto à sua executabilidade através do resolvente $poss(AS, S)$ utilizando os axiomas do Cálculo de Situações. Se AS fôr executável, irá devolver a situação resultante $do(A, S)$ e um programa vazio nil. Note que este é um dos únicos pontos onde a linguagem Golog precisa utilizar o Cálculo de Situações.

$Trans$ - **Teste de condição/Espera**

$$trans(?(C), S, nil, S) \ :- \ holds(C, S). \tag{4.2}$$

Essa é a implementação do axioma $Trans$ para o teste de uma condição C. O predicado *holds* verifica se a condição C é verdadeira (veja a implementação no Apêndice B). Em caso afirmativo, a situação resultante devolvida é S. Isso significa que nenhuma transição de situação ocorreu de fato, apesar de ter ocorrido uma transição de programa (o programa, que é um teste, sofre transição para o programa vazio nil). Por esse motivo, os testes são também chamados de *pseudo-ações.*

Note que C pode ser uma condição complexa envolvendo mais de um fluente. Esses fluentes estão desprovidos de argumentos situação mas *holds* usa as sentenças $restoreSitArg$, declaradas no domínio do programa, para restaurar os argumentos situação a esses fluentes com a situação atual. Assim, os valores dos fluentes podem ser avaliados pelos axiomas do Cálculo de Situações.

Note que somente as regras $Trans$ para ações primitivas e testes fazem a ligação dos programas Golog com o domínio no Cálculo de Situações. Note também que somente na transição com ação primitiva ocorre de fato uma transição de situação. No caso dos outros operadores Golog como if, por exemplo, não ocorrem transições de situação pois eles são apenas mecanismos de controle que não realizam ações no mundo. Se observarmos todos os axiomas que caracterizam o predicado $Trans$ veremos que, à exceção dos testes e das ações primitivas, todos recorrem a uma definição recursiva de $Trans$. Assim, quando o interpretador avalia uma expressão, esta acaba se decompondo em uma série de avaliações de ações primitivas e testes de condições.

$Trans$ - **Seqüência**

$$trans(P1 : P2, S, P2r, Sr) \quad :- final(P1, S), trans(P2, S, P2r, Sr). \tag{4.3}$$

$$trans(P1 : P2, S, P1r : P2, Sr) : -trans(P1, S, P1r, Sr). \tag{4.4}$$

Essa é a implementação Prolog do axioma $Trans$ para o operador de seqüências de ações. A regra em

4.3 diz que se, na seqüência $P1 : P2$, o programa $P1$ estiver em uma configuração final, isto é, não é mais necessário continuar processando o programa $P1$, a transição ocorrerá como se o programa fosse constituído apenas pelo programa $P2$. A regra em 4.4 diz que se, na seqüência $P1 : P2$, ainda houver algo para ser processado em $P1$, o programa $P1 : P2$ sofrerá uma transição para $P1r : P2$ onde $P1r$ é o programa resultante da transição de $P1$, a partir de S, para Sr. Note que a transição de situação é dependente da situação Sr devolvida por $trans(P2, S, P2r, Sr)$ ou $trans(P1, S, P1r, Sr)$.

Trans - Escolha não-determinística de ações

$$trans(P1\#P2, S, Pr, Sr) \ :- \ trans(P1, S, Pr, Sr); \ trans(P2, S, Pr, Sr). \tag{4.5}$$

Essa é a implementação do axioma $Trans$ para o operador de escolha de ação. O interpretador deve escolher entre transição de $P1$ e a transição de $P2$ para ser a transição Pr do programa $P1\#P2$. Em Prolog, a trnasição de $P1$ sempre será escolhida primeiro. Isso não significa que a transição de $P2$ nunca venha a ser escolhida. Um *backtracking* ocorrerá se $P1$ não levar a uma configuração final. Nesse caso, o meta-interpretador irá realizar a transição de $P2$ para ser a transição Pr.

Trans - Concorrência

$$trans(conc(P1, P2), S, conc(P1r, P2), Sr) \ :- trans(P1, S, P1r, Sr). \tag{4.6}$$

$$trans(conc(P1, P2), S, conc(P1, P2r), Sr) \ :- trans(P2, S, P2r, Sr). \tag{4.7}$$

O operador *conc* permite implementar a concorrência entre dois programas $P1$ e $P2$. Concorrência não deve ser confundida com *paralelismo* como se houvesse dois processadores, cada um interpretando um dos programas. Concorrência deve ser entendida como dois programas que competem pelo direito de serem interpretados, com o interpretador alternando entre a execução de um programa e outro. Em outras palavras o interpretador intercala as instruções dos dois programas.

A regra 4.6 define que a transição de um programa $conc(P1, P2)$ será $conc(P1r, P2)$, onde $P1$

e $P2$ são programas e $P1r$ é uma transição de $P1$. Na regra 4.7, a transição de $conc(P1, P2)$ é $conc(P1, P2r)$, com $P2r$ sendo a transição de $P2$. É mais fácil como ocorre a intercalação aplicando várias transições seguidas a um programa como, por exemplo, $conc(P1, P2)$, onde $P1$ é o programa $a1 : a2$, $P2$ é o programa $a3 : a4$, e $a1$, $a2$, $a3$ e $a4$ são ações primitivas. Se aplicarmos quatro transições seguidas teremos o seguinte:

```
?- trans(conc(a1:a2, a3:a4), s0, Pr, Sr)                          (4.6)
  Pr = conc(nil:a2, a3:a4)
  Sr = do(a1, s0)

?- trans(conc(nil:a2, a3:a4), do(a1, s0), Pr, Sr)                 (4.6)
  Pr = conc(nil, a3:a4)
  Sr = do(a2, do(a1, s0))

?- trans(conc(nil, a3:a4), do(a2, do(a1, s0)), Pr, Sr)            (4.6)
  Pr = conc(nil, nil:a4)
  Sr = do(a3, do(a2, do(a1, s0)))

?- trans(conc(nil, nil:a4), do(a3, do(a2, do(a1, s0))), Pr, Sr)   (4.6)
  Pr = conc(nil, nil)
  Sr = do(a4, do(a3, do(a2, do(a1, s0))))
```

Tudo ocorre como se tivéssemos processado o programa $a1 : a2$ e o programa $a3 : a4$ em seqüência. Isso porque somente a regra 4.6 foi usada. Se utilizássemos a regra 4.7 na segunda transição teríamos a seguinte seqüência:

```
?- trans(conc(a1:a2, a3:a4), s0, Pr, Sr)                          (4.6)
  Pr = conc(nil:a2, a3:a4)
  Sr = do(a1, s0)

?- trans(conc(nil:a2, a3:a4), do(a1, s0), Pr, Sr).                (4.7)
  Pr = conc(nil:a2, nil:a4)
  Sr = do(a3, do(a1, s0)) ;

?- trans(conc(nil:a2, nil:a4), do(a3, do(a1, s0)), Pr, Sr).       (4.6)
  Pr = conc(nil, nil:a4)
  Sr = do(a2, do(a3, do(a1, s0)))

?- trans(conc(nil, nil:a4), do(a2, do(a3, do(a1, s0))), Pr, Sr).  (4.6)
  Pr = conc(nil, nil)
  Sr = do(a4, do(a2, do(a3, do(a1, s0))))
```

Podemos imaginar que os programas $P1$ e $P2$ são como duas linhas de montagem e que o meta-interpretador é o único operário que deve operar essas linhas alternando seu tempo entre uma linha

e outra. O mecanismo de *backtracking* do meta-interpretador permite produzir todas as possíveis intercalações executáveis de $P1$ e $P2$.

$Trans$ - Procedimentos

As chamadas de procedimentos em Golog são feitas através da seguinte cláusula.

$$trans(P, S, Pr, Sr) \ :- \ sub(now, S, P, PArgsS), \ proc(PArgsS, E), \ trans(E, S, Pr, Sr). \quad (4.8)$$

P corresponde a uma chamada de procedimento $p(\vec{v})$. A função de $sub(now, S, P, PArgsS)$ é substituir os valores de $\vec{v}$ pelos valores que terá na situação em que p fôr executada no mundo real. A unificação de $proc(PArgsS, E)$ funciona como uma busca pelo código E de execução do procedimento $p(\vec{v})$. Esse código será usado em $trans(E, S, Pr, Sr)$ para gerar a transição de P para Pr. Na prática, a chamada de procedimento p nada mais é do que a substituição de p pelo código E especificado pelo procedimento p.

*Trans** e *Do*

$$do(P, S, Sr) : -trans{*}(P, S, Pr, Sr), final(Pr, Sr). \quad (4.9)$$

$$trans{*}(P, S, P, S). \quad (4.10)$$

$$trans{*}(P, S, Pr, Sr) \ :- \ trans(P, S, PP, SS), \ trans{*}(PP, SS, Pr, Sr). \quad (4.11)$$

Essas cláusulas implementam os predicados $Trans*$ e Do. A execução de um programa Golog é sempre feita através de uma consulta $do(\delta, s0, S)$[1], onde, geralmente, δ é o nome de um procedimento ou um programa Golog, $s0$ é a situação inicial (ou uma situação qualquer), e S é a variável que receberá a situação resultante. A situação devolvida em S é o plano de ações primitivas que o

[1]Como, em Prolog, não podemos utilizar *Do*, uma vez que termos com iniciais maiúsculas são interpretados como variáveis, escrevemos *do*, que não pode ser confundido com o *do* do Cálculo de Situações.

interpretador devolverá se conseguir executar legalmente o programa δ a partir da situação inicial.

A implementação em Prolog faz com que a consulta Do funcione como uma chamada a um mecanismo de busca de execuções legais do programa δ. Do chama $Trans*$ para encontrar uma configuração de programa Pr possível de ser obtida a partir do programa P na situação S e depois, através de $Final$, verifica se essa configuração Pr pode ser considerada uma execução legal de P. Em caso negativo, ocorre um *backtracking* e $Trans*$ tentará encontrar outra configuração Pr. A cada *backtraking*, o interpretador tentará uma nova forma de executar o programa P através de uma outra escolha de ação, outra intercalação de ações ou outra escolha de argumento.

A possibilidade de realizar *backtrackings* em programa Golog permite que seja possível especificar comportamentos robóticos mais flexíveis daqueles que seriam possíveis em outras linguagens procedurais. Em outras linguagens, se durante a execução das instruções de um programa algo der errado, é necessário chamar uma rotina de tratamento de falha. Com Golog, devido à representação do domínio no Cálculo de Situações, é possível saber quando ocorrem essas falhas e devido ao mecanismo de busca implementado através de Do, é possível encontrar mais de uma forma de executar um programa.

4.5 Interrupções

Giacomo *et.al* [De Giacomo, G. et al., 2000b] descrevem como implementar um operador de **interrupção** em Golog. Uma interrupção $\phi \rightarrow \delta$ dispara a execução do programa δ quando a condição ϕ torna-se verdadeira. A definição de interrupção é feita utilizando outros operdadores Golog da seguinte maneira:

$$< \phi \rightarrow \delta > \stackrel{def}{=} while(interrupt_running, if(\phi, \delta, nil))$$

Vejamos como a interrupção funciona [De Giacomo, G. et al., 2000b]. Assume-se que o fluente *interrupt_running* é verdadeiro. Quando a interrupção assume o controle, ela executa o programa δ repetidamente até a condição ϕ tornar-se falsa, liberando o controle da execução para outro programa. Para terminar com o laço *while*, utiliza-se uma ação primitiva especial *stop_interrupt*, cujo único efeito é tornar *interrupt_running* falso. Também existe uma ação primitiva *start_interrupt* que torna

interrupt_running verdadeiro, ou seja, uma ação que ativa a interrupção.

4.6 Indigolog

A linguagem Golog que apresentamos nas seções anteriores corresponde a um dos dialetos Golog. Há três dialetos: **Golog** [Levesque, H. J. et al., 1997], **ConGolog** [De Giacomo, G. et al., 2000b] e **IndiGolog** [De Giacomo, G. et al., 2002]. O dialeto que apresentamos é o ConGolog. A linguagem ConGolog é uma extensão da linguagem Golog original e IndiGolog é uma extensão de Congolog. A Figura 4.1 representa graficamente como cada linguagem estende a outra.

Originalmente, Golog não possuía operadores de concorrência. Estes foram adicionados na linguagem ConGolog. Como Golog e ConGolog não diferem significativamente entre si, é comum usar o nome Golog para se referir a ambos os dialetos. Esses dois dialetos possuem em comum o fato de sua execução realizar um planejamento deliberativo, i.e., nenhuma ação é executada realmente no mundo real (*execução on-line*). Apenas é criado um plano de ações que teoricamente seria executado com sucesso no mundo real. A base para tal confiança no sucesso do plano está no fato do domínio do Cálculo de Situações permitir inferir o estado do mundo em qualquer situação.

A confiança dos planos obtidos de programas Golog depende da fidelidade com que o domínio representa o mundo real onde será executado o plano. Em alguns domínios, no entanto, não é possível especificar totalmente a situação inicial, a interação com ações externas ou resultados imprevistos de ações. Por isso, foi desenvolvida a linguagem IndiGolog que lida com essas questões através de uma execução *on-line* do programa Golog, i.e., as ações primitivas são executadas à medida que vão sendo avaliadas pelo interpretador. Com essa abordagem é possível realizar tarefas de sensoriamento e monitoramento para medir os valores de fluentes desconhecidos na situação inicial e para detectar falhas nos resultados de ações ou ações de agentes externos.

A reatividade dos agentes robóticos a eventos externos é implementada através de interrupções que disparam quando algum sensor detecta uma mudança no ambiente produzida por um evento externo.

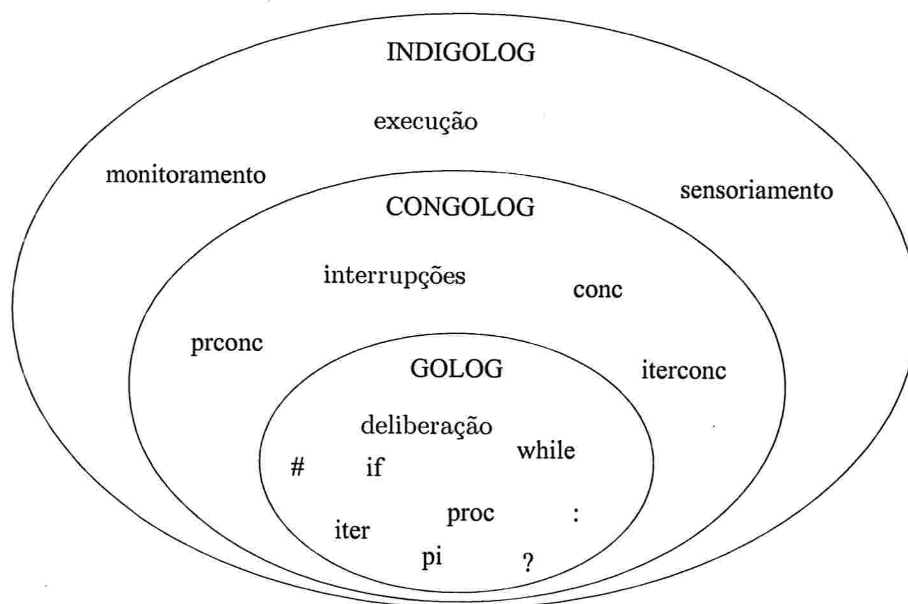

Figura 4.1: As variações da linguagem Golog.

Uma das desvantagens da execução *on-line* é a impossibilidade de se realizar *backtracking*, pois uma vez executada uma ação, ela não pode ser desfeita, embora seus efeitos o possam. Por isso, Indi-Golog dá ao programador a possibilidade do interpretador entrar em modo de execução deliberativa quando fôr conveniente.

4.7 Um exemplo de programa Golog

Um programa Golog possui duas partes: *uma descrição de domínio no Cálculo de Situações* e *um conjunto de declarações de procedimentos na linguagem Golog.* Na figura 4.2, temos um exemplo de programa Golog consistindo de três declarações de procedimentos e cujo domínio é o Mundo dos Blocos.

No domínio do Mundo dos Blocos, os fluentes $sobre$, $sobreMesa$ e $livre$ têm os significados usuais. O fluente $torre$, onde $torre(x, m)$ significa que há uma torre de m blocos cujo topo é o bloco

```
1  proc(criaTorre(n),      % Constrói uma torre de n blocos
2    pi([x,m,r], ?(torre(x,m)) :
3              if(m =< n, ?(r is n-m) : completaN(x,r),
4                   ?(r is m-n) : removeN(x,r)))
5  ).

6  proc(completaN(x,n),    % Coloca n blocos sobre a torre cujo topo é o bloco x
7    ?(n = 0) #
8    pi([y,r], ?(sobreMesa(y)) : empilha(y,x) : ?(r is n-1) : completaN(y,r) #
9    pi([y,z,r], ?(sobre(y,z)) : move(y,z,x) : ?(r is n-1) : completaN(y,r)
10 ).

11 proc(removeN(x,n),      % Remove n blocos da torre cujo topo é o bloco x.
12   ?(n = 0) #
13   pi([y,r], ?(sobre(x,y)) : desempilha(x,y): ?(r is n-1) : removeN(y,r)
14 ).

15 % Domínio no Cálculo de Situações
16 primitive(move(A,B,C)).
17 primitive(empilha(A,B)).
18 primitive(desempilha(A,B)).

19 poss(move(A,B,C),S)     :- livre(A,S), livre(C,S), sobre(A,B,S), A \= C.
20 poss(empilha(A,B),S)    :- livre(A,S), livre(B,S), sobreMesa(A,S), A \= B.
21 poss(desempilha(A,B),S) :- livre(A,S), sobre(A,B,S).

22 sobreMesa(A,do(E,S)) :- sobreMesa(A,S), not(E = empilha(A,B));
23                         E = desempilha(A,B).
24 sobre(A,B,do(E,S))   :- sobre(A,B,S), E \= desempilha(A,B), E \= move(A,B,C);
25                         E = empilha(A,B);
26                         E = move(A,C,B).
27 livre(A,do(E,S))     :- livre(A,S), E \= move(B,C,A), E \= empilha(B,A);
28                         E = move(B,A,C);
29                         E = desempilha(B,A).

30 % Estado inicial
31 livre(a,s0).
32 sobreMesa(a,s0).
33 livre(b,s0).
34 sobreMesa(b,s0).
35 sobreMesa(c,s0)
36 sobre(d,c,s0).
37 livre(d,s0).
38 torre(a,1,s0).
39 torre(d,2,s0).
40 torre(b,1,s0).

41 restoreSitArg(sobre(A,B), S, sobre(A,B,S)).
42 restoreSitArg(sobreMesa(A), S, sobreMesa(A,S)).
43 restoreSitArg(livre(A), S, livre(A,S)).
44 restoreSitArg(torre(A,M), S, torre(A,M,S)).
```

Figura 4.2: Programa GOLOG para criar uma torre de tamanho n no Mundo dos Blocos

x. Esse programa define três procedimentos: $criaTorre$, $completaN$ e $removeN$. O procedimento $criaTorre(n)$ funciona escolhendo uma torre de m blocos com um bloco x no topo. Se essa torre possui menos de n blocos, então é feita uma chamada para o procedimento $completaN$ para completar a torre com $n - m$ blocos. Se a torre escolhida possui mais de n blocos, é chamado o procedimento $removeN$ para retirar os $m - n$ blocos excedentes da torre.

No programa da Figura 4.2, observamos que os procedimentos ($proc$) fazem referências às ações e fluentes do Cálculo de Situações. Porém, existe uma diferença importante a se notar: nos procedimentos, os fluentes $sobre$, $sobreMesa$ e $torre$ aparecem sem argumentos situação. Os argumentos situação são restaurados aos fluentes na regra $Trans$ de teste (4.2) pelo resolvente $holds$ que utiliza as declarações $restoreSitArg$. No Apêndice B, temos a implementação de $holds$.

Para construir uma torre de três blocos, por exemplo, basta realizar a consulta Prolog `?- do(criaTorre(3),s0,S).`

que irá devolver a solução

```
S = do(move(d,c,b), do(empilha(b,a),s0)),
```

ou seja, o plano $\{empilha(b,a) \prec move(d,c,a)\}$.

Capítulo 5

Golog e planejamento hierárquico

Redes de tarefas hierárquicas são usadas extensivamente em sistemas de planejamento atualmente em uso. Assim, é uma característica desejada que linguagens de programação e execução de ações como Golog sejam compatíveis com redes de tarefas. Os trabalhos de Gabaldon [Gabaldon, 2002] e Baral & Son [Baral, C. and Son, T. C., 1999] mostram como representar redes de tarefas do planejamento hierárquico na linguagem Golog. Como veremos a seguir, esses dois trabalhos usam abordagens diferentes. A primeira propõe alterações no meta-interpretador Prolog da linguagem Golog adicionando um novo operador, chamado *htn*, que permite que Golog trabalhe com redes de tarefas. A segunda proposta, não altera o meta-interpretador mas adiciona meta-fluentes ao domínio de planejamento para expressar fatos a respeito do estado da execução de uma ação.

Nesse trabalho, mostraremos que é possível fazer planejamento hierárquico em Golog sem a necessidade de especificar mudanças no meta-interpretador ou da adição de meta-fluentes. Nesse capítulo, faremos um breve resumo dessas propostas.

5.1 A proposta de Baral & Son: o operador htn

Baral & Son [Baral, C. and Son, T. C., 1999] partem do princípio de que a linguagem Golog e a linguagem do planejamento hierárquico possuem estratégias similares, mas que apresentam vantagens uma em relação à outra. As vantagens do planejamento hierárquico com relação ao Golog são:

- a ordem parcial em uma rede de tarefas é diretamente representada por um conjunto de restrições de ordem e
- a rede de tarefas permite expressar as restrições de fluentes do tipo (f, t), (t, f), chamadas de pré-condição e pós-condição de tarefas respectivamente, e ainda restrições do tipo (t_1, f, t_2), chamadas de condições temporais.

Por sua vez, Golog possui as seguintes vantagens com relação ao planejamento hierárquico:

- Golog permite construções do tipo ALGOL, tais como seqüências e laços. Estas construções podem ser simuladas em uma rede de tarefas mas as construções do estilo ALGOL são mais naturais para um programador e
- Golog usa uma lógica de ações sofisticada baseada no Cálculo de Situações que permite especificar efeitos de ações e axiomas sobre as propriedades do mundo. Isso não é possível com a linguagem do planejamento hierárquico. Além disso, devido à maior expressividade da lógica, é possível estender, com mais facilidade, algumas das restrições feitas em planejamento clássico.

Baral & Son sugerem unir as vantagens das duas linguagens através da extensão de Golog com um **operador htn** que permite representar uma rede de tarefas em um programa Golog. O operador htn é da forma $htn(d, \phi)$, onde d é uma lista de tarefas e ϕ é uma lista de restrições. Essa linguagem estendida é chamada de **ConGolog+HTN**.

Para implementar o operador *htn*, foram definidos [1] os seguintes predicados: *method*, *primitive*, *comp* e *reduces*, que não fazem parte do domínio de Cálculo de Situações, a saber:

- ***Method(α, htn(d, ϕ))*** - define que a tarefa não-primitiva α pode ser decomposta pela rede de tarefas dada por $htn(d, \phi)$;
- ***Primitive(d)*** [2] - é verdadeiro se d é uma lista de tarefas primitivas;
- ***Comp(htn(d,ϕ), α, s)*** - é verdadeiro quando $primitive(d)$ é verdadeiro e quando α é uma seqüência de tarefas totalmente ordenada que contêm todas as tarefas de d que, executada a partir da situação s, satisfaz as restrições de ϕ.
- ***Reduces(htn(d,ϕ),htn(d',ϕ'))*** - é verdadeiro se $htn(d, \phi)$ é uma rede de tarefas e $htn(d', \phi')$ é a rede de tarefas resultante de uma decomposição hierárquica feita em alguma tarefa não-primitiva de $htn(d, \phi)$.

Além disso, o meta-intrepretador Golog deve ser modificado para incluir as seguintes cláusulas para $Trans$ e $Final$:

$Final(htn([], \phi))$.

$Trans(htn(d, \phi), s, \delta, s_1) \leftarrow Primitive(d) \wedge Comp(htn(d, \phi), \alpha, s) \wedge Trans(\alpha, s, \delta, s_1)$.

$Trans(htn(d, \phi), s, \delta, s_1) \leftarrow \neg Primitive(d) \wedge Reduces(htn(d, \phi), htn(d_1, \phi_1)) \wedge Trans(htn(d_1, \phi_1), s, \delta, s_1)$.

Assim, o meta-interpretador ConGolog+HTN transforma uma rede de tarefas $htn(d, \phi)$, em um programa δ que é uma seqüência de tarefas primitivas obtida pela ordenação total de todas as tarefas da rede de tarefas d que satisfazem todas as restrições em ϕ. O programa δ é uma solução do problema de planejamento hierárquico $\langle d, s, \mathcal{D} \rangle$, onde s é a situação atual no momento de executar o operador $htn(d, \phi)$ e $\mathcal{D}$ é o domínio contendo: métodos, tarefas, redes de tarefas e os axiomas

[1] Também são definidos muitos outros predicados para especificar as restrições de ordem entre tarefas e as restrições de fluentes [Baral, C. and Son, T. C., 1999].

[2] Pode ser visto como uma extensão do *primitive* que define as ações primitivas de Golog.

originais do Cálculo de Situações. A cada *backtracking* no operador htn, uma nova solução para o problema é devolvida.

O operador htn funciona como uma chamada a um planejador hierárquico externo ao meta-interpretador Golog que devolve todas as possíveis soluções totalmente ordenadas do problema de planejamento hierárquico especificado pela rede de tarefas $htn(d, \phi)$. Entretanto, o operador htn não devolve todas as soluções de uma só vez. Primeiramente, ele devolve uma possível solução. Caso a execução do programa Golog falhe em algum ponto subsequente, ocorre um *backtracking* até o operador htn, que devolverá outra possível solução. Os *backtrackings* de htn proseeguem até que seja encontrada alguma solução que consiga levar o programa Golog até o fim de sua execução, ou até que não existam mais soluções de htn para serem testadas.

É importante notar que a execução do operador htn é independente do meta-interpretador Golog, i.e., das cláusulas $Trans$ definidas anteriormente (Seção 4.2).

5.2 A proposta de Gabaldon: representações de redes de tarefas

Gabaldon [Gabaldon, 2002] considera que procedimentos Golog são equivalentes aos métodos do planejamento hierárquico. Na Figura 5.1, temos um exemplo de como três métodos (p, d_1), (p, d_2) e (p, d_3) de um domínio de planejamento hierárquico podem ser representados como um único procedimento Golog p. No procedimento Golog p, o operador de escolha não-determinística ($\#$) permite que o meta-interpretador escolha uma dentre três maneiras de executar o procedimento p.

A principal contribuição do trabalho de Gabaldon é propôr três maneiras diferentes de representar uma rede de tarefas em Golog, sem a necessidade de estender o meta-interpretador, a saber:

- a primeira, representa redes de tarefas totalmente ordenadas em Golog;
- a segunda, representa redes de tarefas parcialmente ordenadas do planejamento hierárquico definindo alguns fluentes e ações extras na linguagem do Cálculo de Situações, o que permite representar qualquer rede de tarefas de planejamento hierárquico em Golog;
- a terceira, representa redes de tarefas parcialmente ordenadas sem utilizar os fluentes extras,

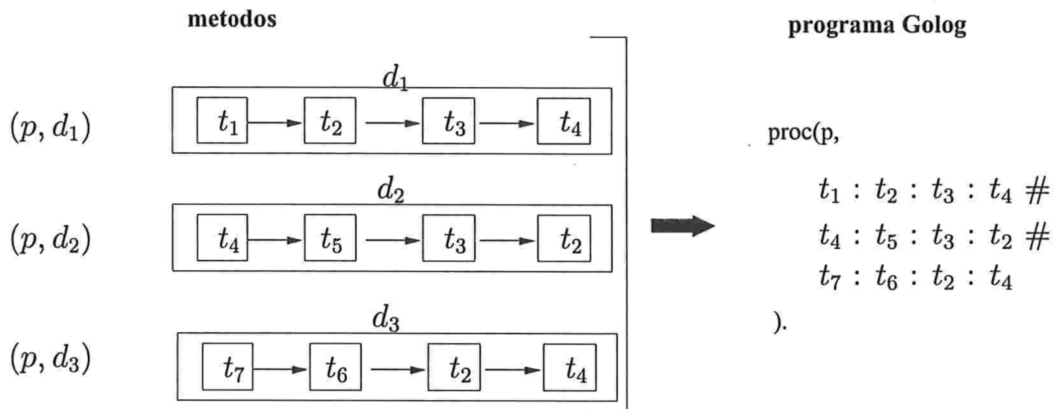

Figura 5.1: Métodos da tarefa não-primitiva p representados como um procedimento Golog p.

isto é, somente com os operadores da linguagem Golog definidos no Capítulo 4, o que faz com que nem toda rede de tarefas de planejamento hierárquico possa ser representada em Golog.

5.2.1 Redes de tarefas totalmente ordenadas em Golog

Uma rede de tarefas totalmente ordenada pode ser trivialmente representada como um programa seqüêncial em Golog através do operador ":". Na Figura 5.2, temos um exemplo de como uma rede de tarefas totalmente ordenada do planejamento hierárquico pode ser transformada num programa Golog.

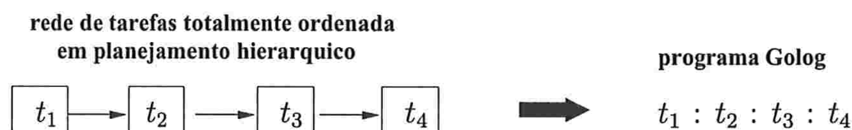

Figura 5.2: Rede de tarefas totalmente ordenada e o programa Golog correspondente.

5.2.2 Redes de tarefas parcialmente ordenadas em Golog: os fluentes *nexec* e *pred*

A segunda proposta de Gabaldon permite que uma rede de tarefas parcialmente ordenada do planejamento hierárquico seja codificada como um programa Golog através da definição de dois fluentes e duas ações extras no domínio de Cálculo de Situações, a saber:

- $Executing(p(x), s)$, significa que a ação primitiva ou procedimento p está sendo executada na situação s;
- $Terminated(p(x), s)$, significa que a ação primitiva ou procedimento p terminou de ser executada na situação s;
- $start(p(x))$, é a ação que inicia a execução de $p(x)$, i.e., torna $Executing(p(x))$ verdadeiro;
- $end(p(x))$, é a ação que termina a execução de $p(x)$, i.e., torna $Executing(p(x))$ falsa e $Terminated(p(x))$ verdadeira.

Note que *Executing* e *Terminated* são meta-fluentes que descrevem propriedades de ações e procedimentos. Os axiomas de estado sucessor que definem como os valores verdade dos fluentes *Executing* e *Terminated* podem ser alterados são:

$$Executing(p(\vec{v}), do(a, s)) \leftarrow a = start(p(\vec{v})) \vee (Executing(p(\vec{v}), s) \wedge a \neq end(p(\vec{v})))$$
$$Terminated(p(\vec{v}), do(a, s)) \leftarrow a = p(\vec{v}) \vee a = end(p(\vec{v})) \vee Terminated(p(\vec{v}), s)$$

Seja d uma rede de tarefas e t uma tarefa de d. Gabaldon define ainda os seguintes fluentes em um domínio Golog:

- $Nexec(t, s) \equiv \neg Executing(t, s) \wedge \neg Terminated(x, s)$, significa que a tarefa t não começou a ser executada na situação s e
- $Pred(t, d, s)$ significa que $Terminated(t', s)$ é verdadeiro na situação s para todas as tarefas $t' \in d$ que precedem a tarefa t.

O predicado *Pred* é utilizado para representar restrições de ordem entre tarefas em Golog sem ser preciso utilizar o operador de seqüência (:). Assim, podemos descrever uma rede de tarefas d

parcialmente ordenada com tarefas $t_1, t_2, ..., t_n$, com o seguinte programa Golog:

$(\text{pred}(t_1) \wedge \text{nexec}(t_1)) \rightarrow t_1 \; ||$
$(\text{pred}(t_2) \wedge \text{nexec}(t_2)) \rightarrow t_2 \; ||$
...
$(\text{pred}(t_n) \wedge \text{nexec}(t_n)) \rightarrow t_n$

No início da execução desse programa, os fluentes $Terminated$ e $Executing$ devem ter valor falso para todas as ações. A interrupção (operador $\rightarrow$) de uma tarefa t faz com que ela só possa ser executada quando todas as tarefas que a precedem estiverem terminadas. No início, o programa executa as tarefas sem predecessores. Conforme as ações são executadas, os fluentes $Terminated$ se tornam verdadeiros até que todas as tarefas que precedem a tarefa t terminem, fazendo a interrupção iniciar a sua execução t.

5.2.3 Redes de tarefas parcialmente ordenadas em Golog: menos expressividade

Na terceira proposta [Gabaldon, 2002], Gabaldon sugere que, em alguns casos, podemos codificar uma rede de tarefas parcialmente ordenada como um programa Golog de uma forma mais simples, sem a necessidade de definir os meta-fluentes $Executing$ e $Terminated$. Para isso, uma rede de tarefas parcialmente ordenada é representada por um programa Golog utilizando apenas os operadores de comcorrência (||) e de seqüência (:). Como exemplo, temos um procedimento *move*(v_1,v_2,v_3) que retira todos os blocos de sobre v_1 e retira todos os blocos de sobre v_3 antes de passar o bloco v_1 que está sobre v_2 para cima do bloco v_3. Na Figura 5.3, temos uma representação gráfica dessa rede de tarefas.

Repare que *liberar*(v_1) e *liberar*(v_3) não têm uma ordem de execução definida devido ao operador de execução concorrente de ações (||), uma vez que esse operador poderá escolher qualquer uma das duas tarefas para começar a ser executada primeiro. Por sua vez, o operador de seqüência (||) deixa bem definida a ordem em que as tarefas *desempilha*(v_1, v_2) e *empilha*(v_1, v_3) serão executadas.

O problema com essa representação para redes de tarefa é que nem sempre é possível representar

proc move(v_1,v_2,v_3)
(liberar(v_1) || liberar(v_3)) :
desempilha(v_1, v_2) : empilha(v_1, v_3)
).

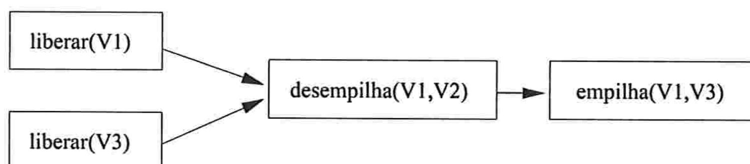

Figura 5.3: Representação gráfica de uma rede de tarefas parcialmente ordenada.

uma rede de tarefas de planejamento hierárquico com certas restrições de ordem, com apenas operadores de seqüência e execução concorrente. Na Figura 5.4 apresentamos um exemplo de rede de tarefas que não pode ser representada diretamente com os operadores da linguagem Golog, isto é, sem o uso dos fluentes *nexec* e *pred*.

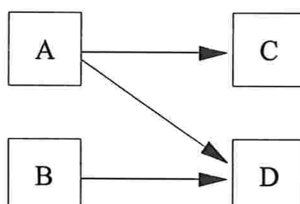

Figura 5.4: Exemplo de rede que não pode ser codificada sem *pred* e *nexec*.

A rede de tarefas da Figura 5.4 poderia ser representada em Golog das seguintes maneiras:

1. com o programa $A : (C||D)$. No entanto, não é possível introduzir as relações de ordem entre B e D;

2. com o programa $A : (C||(B : D))$, não seria possível representar a ordem total que satizfaz a ordem parcial da figura: $B \prec A \prec C \prec D$. O problema dessa modificação é que ela adiciona uma restrição de ordem entre A e B que não existe na rede de tarefas da figura;

3. com o programa para $(A||B) : (C||D)$, isto é, para representar o fato de que A e B não

possuem restrição de ordem entre si devemos usar o operador || entre A e B modificando o primeiro program. Entretanto, isso acaba por adicionar uma restrição de ordem entre B e C que não existe na rede de tarefas da figura. Programas diferentes irão sempre adicionar restrições a mais ou deixar de representar alguma restrição.

5.3 Solução de Baral & Son versus solução de Gabaldon

Apesar de diferentes, a solução de Baral & Son e a solução genérica de Gabaldon têm em comum o fato de fazerem grandes modificações na descrição do domínio e do modo de funcionamento de programas Golog. Na proposta de Baral e Son, são definidos fluentes para representar redes de tarefas e um planejador hierárquico, operador *htn* que é chamado pelo meta-interpretador Golog.

Em Gabaldon, são adicionadas meta-ações e meta-fluentes, *nexec* e *pred*, que agem sobre as ações e fluentes do domínio do Cálculo de Situações. A vantagem dessa extensão do Cáculo de Situações é a de permitir a representação de qualquer rede de tarefas, isto é, qualquer ordem parcial de tarefas.

A terceira proposta de Gabaldon de representar redes de tarefas somente com os operadores || e : é muito mais simples para um programador, apesar de menos expressiva. Seguindo essa mesma idéia, na próxima seção, mostramos que planejamento hierárquico pode ser realizado naturalmente através das chamadas de procedimentos Golog, correspondência essa que não foi apontada explicitamente por nenhum dos trabalhos anteriores. Mostraremos como a especificação de problemas e plano solução de planejamento hierárquico podem ser interpretadas nos programas Golog. Finalmente, fazemos uma comparação da execução de um programa Golog com o planejador SHOP2 (Seção 3.13).

5.4 O meta-interpretador Golog como um planejador hierárquico

Os trabalhos apresentados nas seções anteriores criam extensões para representar redes de tarefas. Enquanto que a linguagem *ConGolog+HTN* chama um planejador hierárquico, as três propostas em [Gabaldon, 2002] não deixam claro como a execução de programas Golog (com ou sem extensões da

linguagem) realizam planejamento hierárquico.

Nessa seção mostraremos que o próprio meta-interpretador da linguagem Golog é um planejador hierárquico, independente da adição de novos fluentes e ações aos domínios ou ainda, novas cláusulas $Trans$ e $Final$ no meta-interpretador Golog.

5.4.1 Definição de problema e soluções

Como vimos no Capítulo 3, um **problema de planejamento hierárquico** é representado como uma tupla $\langle d, I, \mathcal{D}\rangle$, onde d é uma rede de tarefas, I é uma situação inicial, e $\mathcal{D}$ é um conjunto de tarefas e métodos do domínio de planejamento. Uma **solução** para um problema de planejamento hierárquico $\langle d, I, \mathcal{D}\rangle$ é uma rede de tarefas primitivas, que satisfaz todas as restrições de tarefas e de ordens, obtida por sucessivas decomposições hierárquicas da rede de tarefas d e que pode ser executada a partir da situação I. Essa rede de tarefas primitiva especifica um plano parcialmente ordenado. O principal objetivo de planejamento hierárquico é encontrar um meio de executar as tarefas de uma rede de tarefas inicial.

Por outro lado [De Giacomo, G. et al., 2000a], dados um programa δ, uma teoria de domínio D, composta pelos axiomas do Cálculo de Situações e um conjunto de procedimentos; o **problema da execução de Golog** é encontrar uma seqüência de ações $\vec{a}$ tal que:

$$D, \delta \models (\exists \vec{a}) Do(\delta, s_0, do(\vec{a}, s_0))$$

onde $Do(\delta, s, s')$ significa que o programa δ, é executável a partir da situação s_0, levando à situação $do(\vec{a}, S_0)$. Em outras palavras, o problema da execução é a busca por uma seqüência $\vec{a}$ de ações primitivas, executável a partir da situação s_0, que obedeçam às especificações do programa Golog δ

O principal objetivo do meta-interpretador Golog é encontrar um meio de executar um programa que, como mostrou Gabaldon, se assemelha a uma rede de tarefas. Assim, planejamento em Golog é a **execução de um programa de alto nível**, ou seja, a transformação de uma rede de tarefas de alto nível em uma rede de tarefas executável.

O meta-interpretador Golog transforma o programa original em um programa executável através de sucessivas chamadas a procedimentos, i.e., sucessivas decomposições de tarefas de alto nível, gerando um plano de ações primitivas.

Como no planejamento hierárquico, Golog limita a sua busca apenas aos planos que obedecem às especificações do programa. Também permite um certo grau de não-determinismo através do uso de operadores de escolha não-determinística que indicam pontos de *backtracking* na execução do programa. Isso é o que ocorre no algoritmo não determinístico de planejamento hierárquico apresentado no Capítulo 3 (Seção 3.3).

Finalmente, fazemos um paralelo entre as definições de domínios $\mathcal{D}$ e D. Em Golog, os axiomas do Cálculo de Situações equivalem às especificações das tarefas primitivas de planejamento hierárquico enquanto que os procedimentos Golog equivalem aos métodos de planejamento hierárquico, pois, como visto em [Gabaldon, 2002], um procedimento t pode ser visto como o conjunto de todos os métodos para uma tarefa não-primitiva t[3].

É importante notar que os trabalhos anteriores deixaram de fazer um paralelo importante entre o planejamento hierárquico e Golog: falta uma definição de tarefas meta para a linguagem Golog sendo essa uma das principais contribuições desse trabalho, como será visto no Capítulo 6.

Redes de tarefas versus programas

Já vimos que nem toda rede de tarefas pode ser transformada em um programa Golog sem definirmos meta-fluentes e ações como fez Gabaldon. O que não foi discutido ainda é a transformação de um programa Golog p em uma rede de tarefas d.

Quando um programa Golog possui os operadores de seqüência (:), execução concorrente (||), escolha não-determinística (#) e teste (?), é trivial construir a rede de tarefas equivalente ao programa. Já para programas contendo os operadores if, *while*, *iter* e *iterconc* é preciso transformá-los

[3]No restante do texto, consideraremos os termos *tarefa primitiva* e *ação primitiva* equivalentes tanto no contexto do planejamento hierárquico como no contexto da linguagem Golog. Também consideraremos que chamadas de procedimento em Golog são tarefas não-primitivas.

em procedimentos especiais para que seja possível identificar a rede de tarefas correspondente, como mostramos a seguir:

```
proc(if(f,a1,a2),
    ?(f) : a1 #
    ?(-f) : a2
).

proc(while(f,a),
    ?(-f) #
    ?(f) : a : while(f,a)
).

proc(iter(a),
    nil #
    a : iter(a)
).

proc(iterconc(a),
    nil #
    a || iterconc(a)
).
```

Golog e o planejador SHOP2

Como dissemos anteriormente, o algoritmo de planejamento hierárquico aplica sucessivas decomposições hierárquicas em uma rede de tarefas inicial até gerar uma rede de tarefas primitiva. Uma decomposição hierárquica é a substituição de uma tarefa não-primitiva por uma rede de tarefas especificada pelo método que informa como executá-la. Por sua vez, quando o meta-interpretador Golog executa um programa que faz uma chamada a um procedimento (*proc*), ele substitui a chamada de procedimento pelo programa especificado no procedimento. Os nomes dos procedimentos correspondem às tarefas não-primitivas, e as declarações de procedimentos correspondem aos métodos

no planejamento hierárquico. No exemplo a seguir, ilustramos como ocorre o processamento do programa $q : r : s$, onde q é uma chamada ao procedimento q.

```
    proc(q,
       q1 : q2 : q3 :
    ).

% Configuração do programa inicial
                                  q : r : s

% Configuração do programa após aplicação da regra Trans para o procedimento q.
                             q1 : q2 : q3 : r : s
```

Na Figura 5.5, mostramos a representação gráfica da decomposição hierárquica de uma tarefa q de uma rede de tarefas equivalente ao programa $q : r : s$.

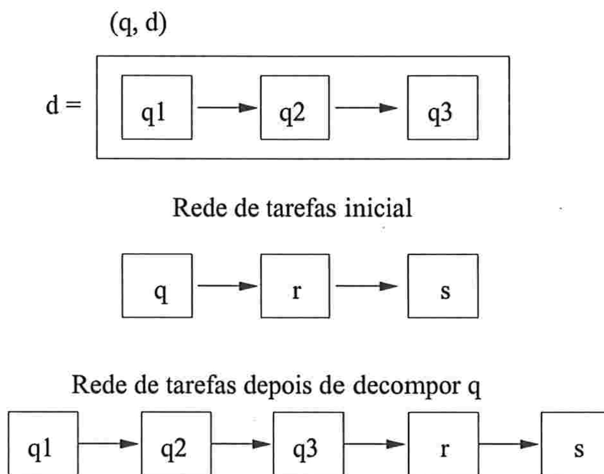

Figura 5.5: Representação gráfica da decomposição da tarefa q.

Observe agora o seguinte procedimento:

proc(t,

$t_1 : t_2$ #

$t_3 : t_4$

).

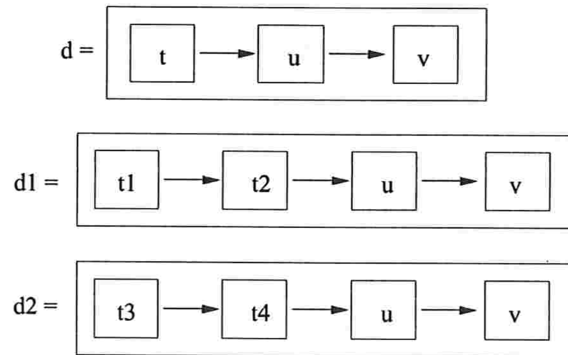

Figura 5.6: Representação gráfica de duas decomposições da tarefa t.

Esse procedimento especifica dois métodos para decompor a tarefa t pertencente à rede de tarefas d. Na Figura 5.6, mostramos duas possíveis redes de tarefas $d1$ e $d2$ resultantes da decomposição de d usando o procedimento t, isto é, os dois métodos especificados para a tarefa composta t. Nesse exemplo, quando o meta-interpretador Golog processa a chamada do procedimento t em um programa $t : u : v$, a configuração de programa resultante não é apenas um dos dois programas $t1 : t2 : u : v$ ou $t3 : t4 : u : v$, mas sim a especificação completa da escolha não determinística de ações, ou seja, $(t1 : t2 \# t3 : t4) : u : v$. Isso ocorre porque a regra $Trans$ para chamada de procedimento (regra 4.8) substitui a chamada de t por **todo** o corpo de programa especificado no procedimento t, isto é:

Assim, a escolha do método a ser usado na decomposição é feita pelo meta-interpretador Golog na execução do programa $(t1 : t2 \# t3 : t4) : u : v$. Golog tentará executar primeiro $t1 : t2$ e caso falha, ele fará um backtracking para a escolha $t3 : t4$. Isso é definido pela a regra $Trans$ para escolha de ação (regra 4.5):

$$trans(P1\#P2, S, Pr, Sr) \;:-\; trans(P1, S, Pr, Sr);\; trans(P2, S, Pr, Sr).$$

Uma outra diferença está na política de escolha da próxima tarefa não-primitiva que deve ser decomposta. Enquanto o planejador hierárquico da Seção 3.3 faz uma escolha não-determinística da próxima tarefa a ser decomposta, o meta-interpretador sempre irá processar/decompor as chamadas

de procedimento **na ordem em que devem ser executadas no programa**, isto é, uma chamada de procedimento só é processada se não existir outras chamadas de procedimento que a preceda. Essa é a política de escolha da próxima ação não-primitiva a ser decomposta do planejador hierárquico SHOP2 (seção 3.8).

Como uma conseqüência direta da política de escolha de ações não-primitivas, Golog, assim como SHOP2, constrói planos de ações da esquerda para direita, ou seja, de maneira progressiva. Isso permite que as ações primitivas que não possuam ações não-primitivas à sua esquerda, possam ser de fato executadas no ambiente real. Apesar de impedir que o meta-interpretador faça backtracking na escolha de ações caso ocorra alguma falha, a execução de ações antes de um plano completo ter sido gerado pode ser uma característica desejada em muitas aplicações no mundo real. Por exemplo, em ambientes com informação incompleta do mundo.

5.4.2 Restrições de fluentes do planejamento hierárquico na linguagem Golog

As semelhanças apontadas nas duas seções anteriores nos mostram que o meta-interpretador da linguagem Golog, mesmo sem o uso de quaisquer extensões como propostas por [Gabaldon, 2002] e [Baral, C. and Son, T. C., 1999], faz planejador hierárquico de modo semelhante ao planejador SHOP2. No entanto, com base apenas nas semelhanças apontadas, o meta-interpretador não pode ser considerado um planejador hierárquico de acordo com as definições em [Erol, 1995], mas sim uma instância particular daquele algoritmo mais geral. Como vimos na Seção 5.2, algumas redes de tarefas como as da Figura 5.4 não podem ser representadas na linguagem Golog. Além disso, Golog não é capaz de representar as restrições de tarefas do tipo (t, f), (f, t) e (t, f, t').

Considere a seguinte restrição de fluente (t_1, f, t_2). Em Golog, não há como garantir que o fluente f mantenha o valor verdadeiro entre a execução das tarefas t_1 e t_2. Isso é devido ao fato do meta-interpetador Golog, ao contrário do planejador da Figura 3.3, não ser capaz de usar uma estratégia de proteção de intervalos onde fluentes devem ser verdadeiros. Essa estratégia foi herdada pelos planejadores hierárquicos dos planejadores clássicos que fazem busca pelo espaço de planos.

Essa limitação de Golog pode ser compensada com o planejamento para satisfação de metas,

como veremos no Capítulo 6.

Capítulo 6

Satisfação de metas em Golog

No capítulo anterior, foi mostrado que o meta-interpretador Golog realiza planejamento hierárquico de forma semelhante (sem restrições de fluentes) aos planejadores do tipo HTN. Entretanto, não mencionamos como as **tarefas meta** do Capítulo 3 são representadas em Golog. Assim como o planejamento hierárquico pode resolver problemas de planejamento não-hierárquico através de tarefas meta, como mostrado no Capítulo 3, é desejável que Golog também se comporte dessa maneira. Nas abordagens de Gabaldon [Gabaldon, 2002] e Baral & Son [Baral, C. and Son, T. C., 1999], as tarefas meta não são mencionadas. Nesse capítulo, propomos que tarefas meta especificadas como **procedimentos meta** permitem que Golog realize planejamento orientado à satisfação de metas na linguagem Golog, sem a necessidade de se fazer uma chamada a um planejador dedicado para satisfação de meta no Cálculo de Situações, i.e., um algoritmo de planejamento não-hierárquico, independente de domínio, que raciocine diretamente sobre os axiomas de Cálculo de Situações, como Reiter propôs originalmente [Reiter, 2001a].

A principal contribuição desse trabalho, é mostrar que planejamento orientado à satisfação de metas em Golog pode ser visto como a decomposição de **procedimentos meta**. Além disso, será mostrado que o uso de **procedimentos meta** em Golog é mais eficiente que planejamento para satisfação de metas no Cálculo de Situações.

6.1 Necessidade da satisfação de metas em Golog

Golog é uma linguagem de alto nível criada para especificar programas para agentes robóticos, e como acontece em situações do mundo real, Golog deve permitir a programação de agentes capazes de tratar eventos exógenos imprevistos. Para isso, existem pelo duas estratégias que um programador pode adotar:

1. especificar programas Golog que prevejam todas as contingências possíveis, ou
2. permitir que o agente robótico seja informado ou faça o sensoriamento das mudanças do estado do mundo para que ele possa ainda assim, planejar para atingir seus objetivos.

Nem sempre é possível prever todas as contingências que irão surgir durante a operação do agente robótico, principalmente se o projetista não possue informação completa sobre o comportamento do ambiente onde o agente irá operar (por exemplo, sondas de exploração planetária). Além disso, a especificação de tais programas é difícil e pode ocupar muito espaço de armazenamento, o que pode ser um problema se queremos criar agentes robóticos de baixo custo e alto desempenho.

A segunda opção envolve planejamento com execução e monitoramento, i.e., planejar enquanto se executa as ações do plano, tanto de atuação como de sensoriamento do ambiente. Em programas IndiGolog, feitos para planejamento e execução, a resposta padrão para essas situações é devolver que ocorreu uma falha. Entretanto, poderíamos continuar a execução do programa apenas criando um plano que *conserte* a pré-condição alterada. Isso pode ser feito por um planejador para satisfação de meta.

Por exemplo, durante a execução de um programa IndiGolog pode ocorrer uma interrupção devido a alguma alteração imprevista do estado do mundo que torna uma ação primitiva impossível de ser executada, ou seja, uma de suas pré-condições passa a não estar satisfeita (por exemplo, por não ser capaz de fazer a proteção de fluentes quando executa ações durante o planejamento).

Veremos aqui, quatro planejadores diferentes para satisfação de meta baseados no Cálculo de Situações:

1. o **IDDP** [Pereira, 2002], um planejador não-hierárquico descrito em Prolog, proposto por um dos alunos do grupo de Lógica, Inteligência Artificial e Métodos Formais do IME-USP;
2. o **WSPBF** [Reiter, 2001a], um algoritmo de planejamento não-hierárquico proposto por Reiter para a linguagem Golog;
3. o **SCP**, uma versão do planejador WSPBF sem a camada Golog, apenas com o cálculo de Situações; e
4. o planejamento com **procedimentos meta**, que correspondem às **tarefas meta** implementadas como procedimentos Golog para a satisfação de metas de um determinado domínio de aplicação.

6.2 Planejamento no Cálculo de Situações: o planejador IDDP

O **IDDP** (*Iterative Deepening Deductive Planner*) [Pereira, 2002] é um planejador que sintetiza planos, como efeito colateral da prova de teoremas, empregando uma estratégia de busca em profundidade iterativa no espaço de estados do domínio de planejamento.

No IDDP, tanto o domínio (Figura 6.1) quanto o problema (Figura 6.2) de planejamento são descritos através de axiomas do Cálculo de Situações. Conforme originalmente proposto, o IDDP usa o axioma de frame universal para tratar o problema do frame. Nesse trabalho, entretanto, adotaremos uma versão modificada do IDDP, onde o axioma de frame universal é substituído por axiomas de estados sucessores. Essa nova versão do planejador, também implementada em Prolog, é apresentada na Figura 6.3.

Assim, dados um conjunto de axiomas de pré-condições e estados sucessores *Dom* (Figura 6.1), descrevendo um domínio de planejamento, um conjunto de axiomas de estado inicial e estados meta *Prob* (Figura 6.2), descrevendo um problema de planejamento, e o conjunto de axiomas *Iddp*, que descrevem o planejador (Figura 6.3), a tarefa de planejamento consiste em obter-se uma prova construtiva para o seguinte teorema:

$Dom \cup Prob \cup Iddp \models \exists s\ planning(s)$

```
1 % Axiomas de pré-condições:

2 poss(empilha(A,B),S) :- livre(A,S), livre(B,S), sobreMesa(A,S).
3 poss(desempilha(A,B),S) :-  livre(A,S), sobre(A,B,S).

4 % Axiomas de estado sucessor:
5 sobreMesa(A,do(E,S)) :- sobreMesa(A,S), E \= empilha(A,B); E = desempilha(A,B).

6 sobre(A,B,do(E,S)) :- sobre(A,B,S), E \= desempilha(A,B); E = empilha(A,B).

7 livre(A,do(E,S)) :- livre(A,S), E\= empilha(B,A); E = desempilha(B,A).
```

Figura 6.1: Domínio do mundo dos blocos em Cálculo de Situações.

```
1 % Axiomas de estado inicial:
2 sobreMesa(b,s0).
3 sobreMesa(a,s0).
4 sobre(c,a,s0).
5 livre(b,s0).
6 livre(c,s0).

7 % Estados meta
8 goal(S) :-  sobre(a,b,S).
```

Figura 6.2: Problema no mundo dos blocos em Cálculo de Situações.

ou, equivalentemente:

$$Dom \cup Prob \cup Iddp \models \exists s\ Plan(s) \wedge exec(s) \wedge Goal(S)$$

6.2.1 O predicado *plan* e a estratégia de busca em profundidade iterativa

Como podemos ver na Figura 6.3, o único mecanismo de controle de busca existente no IDDP é aquele implementado pelo predicado *plan*. Usando esse predicado, garantimos que, para todo $i \geq 0$, o IDDP examina todos os planos de tamanho i, antes daqueles de tamanho $i+1$. Para entender como isso acontece, observe a Figura 6.4: ao ser feita a consulta `?- plan(S)`, o sistema Prolog

```
1 planning(S) :- plan(S), exec(S), goal(S).

2 plan(s0).
3 plan(do(A,S)) :- plan(S).

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

Figura 6.3: O planejador IDDP.

pode resolvê-la usando a cláusula 2 ou a cláusula 3 do programa IDDP. Usando a cláusula 2, o sistema obtém `S=s0`. Nesse ponto, caso uma solução alternativa seja solicitada, o sistema retrocede na última escolha realizada e utiliza a cláusula 3, instanciando `S` com `do(A,S1)` e transformando a consulta original em `?- plan(S1).` Analogamente, essa nova consulta também pode ser resolvida pelo uso das cláusulas 2 e 3. Utilizando a cláusula 2, o sistema obtém `S1=s0` e, conseqüentemente, `S=do(A,s0)`; e utilizando a cláusula 3, o sistema instancia `S1` com `do(B,S2)` e transforma a consulta em `?- plan(S2)`... Claramente, a cada retrocesso realizado pelo sistema, o plano é estendido em uma ação. Sendo assim, a primeira solução encontrada para $plan(S)$ será o plano de tamanho $i = 0$ (s_0); a partir daí, no i-ésimo retrocesso, a solução encontrada será um plano de tamanho i.

Um fato importante a ser observado é que os planos encontrados como solução para $plan(S)$ são "genéricos", i.e. são compostos por ações representadas por variáveis ainda não instanciadas. Quando um plano desse tipo é passado para o predicado *exec*, como veremos a seguir, o sistema tenta instanciar suas variáveis de todas as formas possíveis, até que o predicado *goal* seja satifeito ou até que todas as possibilidades sejam esgotadas. Nesse último caso, ocorre uma falha na prova de $exec(S)$ e o sistema retrocede automaticamente ao predicado $plan(S)$, obtendo um outro plano genérico, contendo apenas uma ação a mais. Obviamente, esse comportamento é justamente o que garante que o IDDP realize uma busca em profundidade iterativa.

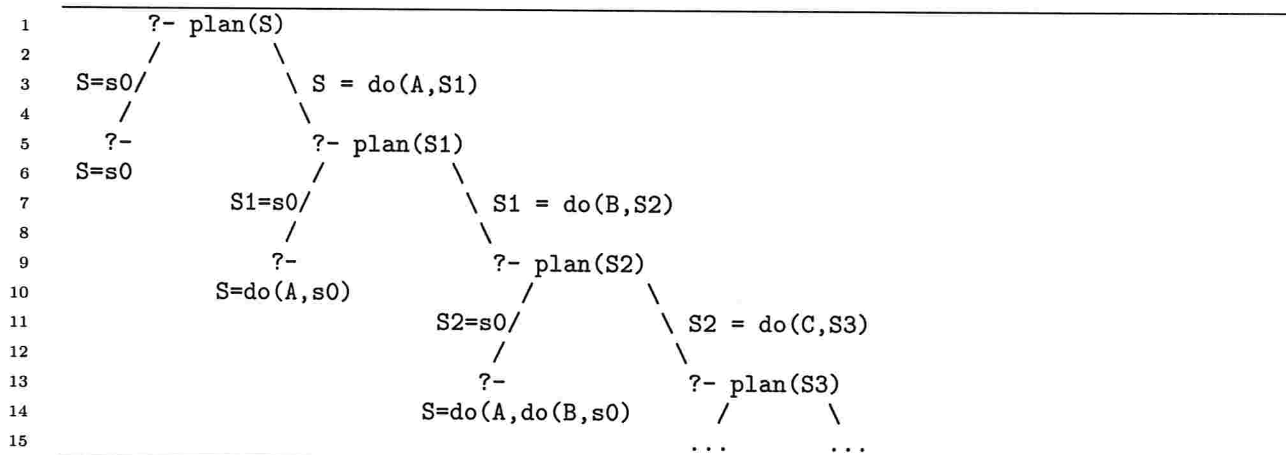

Figura 6.4: Mecanismo de busca em profundidade iterativa.

6.2.2 O predicado *exec* e a instanciação de planos genéricos

Para entender como o predicado *exec* é utilizado para instanciar um plano genérico, obtido como solução de $plan(S)$, considere o problema especificado na Figura 6.2. Nesse problema, temos inicialmente os blocos a e b sobre a mesa e o bloco c sobre o bloco a; e a meta é encontrar uma situação em que o bloco a esteja sobre o bloco b. Como não há solução para esse problema com menos de duas ações, vamos considerar que o plano genérico a ser instanciado por *exec* seja $do(A, do(B, s0))$.

Considere, então, a consulta `?- exec(do(A,do(B,s0))), goal(do(A,do(B,s0))).` Inicialmente, o predicado *exec*, que é recursivo à esquerda, é desdobrado em chamadas ao predicado *poss*, até que a situação incial s_0 seja atingida. Nesse ponto, o predicado *exec* é satisfeito trivialmente (base da definição recursiva). Resta agora ao sistema, satisfazer as submetas *poss*, originadas pela execução recursiva de *exec*. Observe que a satisfação dessas submetas produz pontos de retrocesso aos quais o sistema deverá retroceder (escolhendo outra ação possível), sempre que a prova da submeta *goal* for mal sucedida (veja a Figura 6.5).

Sendo assim, após um número finito de retrocessos realizados pelo sistema, a escolha certa de ações será feita e, finalmente, a submeta *goal* poderá ser satisfeita (veja a Figura 6.6).

```
1      ?- exec(do(A,do(B,s0))), goal(do(A,do(B,s0)))
2      |
3      ?- exec(do(B,s0)), poss(A,do(B,s0)), goal(do(A,do(B,s0)))
4      |
5      ?- exec(s0), poss(B,s0), poss(A,do(B,s0)), goal(do(A,do(B,s0)))
6      |
7      ?- poss(B,s0), poss(A,do(B,s0)), goal(do(A,do(B,s0))) <== ponto de retrocesso
8      |
9      | B = empilha(b,c)
10     |
11     ?- poss(A,do(empilha(b,c),s0), goal(do(A,do(empilha(b,c),s0)))
12     |
13     | A = desempilha(b,c)
14     |
15     ?- goal(do(desempilha(b,c),do(empilha(b,c),s0)))
16     |
17     falha
```

Figura 6.5: Mecanismo de instanciação de planos genéricos: falha.

Caso não existisse uma solução com apenas duas ações para esse problema, então a consulta `?- exec(do(A,do(B,s0))), goal( do(A,do(B,s0)))` terminaria com falha. Entretanto, como o planejador IDDP é equivalente à consulta `?- plan(S), exec(S), goal(S)`, a falha em provar `?- exec(S), goal(S)` obriga o sistema a retroceder até `plan(S)`. Nesse caso, um novo plano genérico S com três passos seria obtido e uma nova tentativa de provar `?- exec(S), goal(S)` seria iniciada.

6.2.3 O predicado *planning* e o planejador IDDP

O predicado *planning* apenas faz chamadas aos predicados *plan*, *exec* e *goal*, constituindo um algoritmo muito simples, do tipo gerar-e-testar. Conforme vimos, a geração dos estados é controlada por um mecanismo de profundidade iterativa. Entretanto, um ponto interessante a ressaltar é que os testes, realizados pelo predicado *plan*, são aplicados uma única vez em cada possível plano do domínio. Diferente de um algoritmo convencional de busca em profundidade iterativa em que, cada vez que a profundidade é aumentada, todos os planos testados na iteração anterior são testados novamente.

O planejador IDDP é uma proposta simples e elegante de planejamento no Cálculo de Situações

```
1   ?- exec(do(A,do(B,s0))), goal(do(A,do(B,s0)))
2   |
3   ?- exec(do(B,s0)), poss(A,do(B,s0)), goal(do(A,do(B,s0)))
4   |
5   ?- exec(s0), poss(B,s0), poss(A,do(B,s0)), goal(do(A,do(B,s0)))
6   |
7   ?- poss(B,s0), poss(A,do(B,s0)), goal(do(A,do(B,s0))) <== ponto de retrocesso
8   |
9   |  B = desempilha(c,a)
10  |
11  ?- poss(A,do(empilha(b,c),s0), goal(do(A,do(desempilha(c,a),s0)))
12  |
13  |  A = empilha(a,b)
14  |
15  ?- goal(do(empilha(a,b),do(desempilha(c,a),s0)))
16  |
17  sucesso
```

Figura 6.6: Mecanismo de instanciação de planos genéricos: sucesso.

e servirá como base para entendermos os planejadores discutidos nas próximas seções. Como o IDDP faz uma busca exaustiva no espaço de estados, o número de planos examinados durante a busca cresce exponencialmente em função do número de ações do domínio, bem como da profundidade alcançada pela busca (i.e., número de passos nos planos construídos). Vale ressaltar que essa ineficiência não é inerente ao Cálculo de Situações, mas sim da busca realizada por planejadores clássicos sem o uso de heurísticas. No entanto, o Cálculo de Situações têm sido usado com muito sucesso em Robótica Cognitiva (através da linguagem Golog), em aplicações tais como coordenação de jogadores robóticos de futebol [Arroz, M et al., 2003] e composição de Serviços Web em *Semantic Web* [McIlraith, S. and Son, T. C., 2001].

6.3 Planejamento no Cálculo de Situações: o planejador WSPBF

Assim como o IDDP, o planejador **WSPBF** (***World's Simplest Breadth-First Planner***) [Reiter, 2001a] raciocina sobre o Cálculo de Situações para sintetizar planos. Proposto por Reiter, um dos criadores da área de Robótica Cognitiva e da linguagem Golog, o WSPBF foi originalmente descrito em Golog.

O domínio e o problema de planejamento também são descritos através de axiomas do Cálculo de Situações como na Figura 6.2. A única diferença na descrição do domínio é que as ações do agente devem ser declaradas através do predicado *primitive* (Figura 6.7).

```
1 % Axiomas de pré-condições:
2 poss(empilha(A,B),S) :- livre(A,S), livre(B,S), sobreMesa(A,S).
3 poss(desempilha(A,B),S) :-  livre(A,S), sobre(A,B,S).
4 % Axiomas de estado sucessor:
5 sobreMesa(A,do(E,S)) :- sobreMesa(A,S), E \= empilha(A,B); E = desempilha(A,B).
6 sobre(A,B,do(E,S)) :- sobre(A,B,S), E \= desempilha(A,B); E = empilha(A,B).
7 livre(A,do(E,S)) :- livre(A,S), E\= empilha(B,A); E = desempilha(B,A).
8 primitive(empilha(_,_)).
9 primitive(desempilha(_,_)).
```

Figura 6.7: Domínio do mundo dos blocos em Cálculo de Situações para o planejador WSPBF.

A Figura 6.8, mostra a implementação do planejador WSPBF em Golog. Dados um conjunto de axiomas de pré-condições e estados sucessores *Dom*; um conjunto de axiomas de estado inicial e estados meta *Prob*, descrevendo um problema de planejamento, e o conjunto de axiomas *Trans* que definem a semântica da linguagem Golog, a tarefa de planejamento consiste em obter uma prova construtiva para o seguinte teorema:

$$Dom \cup Prob \cup Trans \models \exists s\ Do(wspbf(max), s0, s)$$

ou, equivalentemente:

$$Dom \cup Prob \cup Trans \models \exists s\ Do(plans(0, max), s0, s)$$

Na Figura 6.8, o planejador WSPBF é definido por três procedimentos Golog: *wspbf*, *plans* e *actionSequence*. Apesar de seu nome sugerir que a busca realizada é do tipo *busca em Largura*, WSPBF funciona de maneira semelhante ao IDDP, i.e., ele realiza uma busca em profundidade iterativa. Uma peculiaridade do WSPBF é definir um limite máximo *max* do tamanho dos planos a

```
1  proc(wspbf(Max),
2     plans(0,Max)
3  ).

4  proc(plans(N,Max),
5     ?(N =< Max) : (actionSequence(N) : ?(goal) #
6                 pi(N1, ?(N1 is N+1) : plans(N1,Max)))
7  ).

8  proc(actionSequence(N),
9    ?(N=0) #
10   ?(N>0) : pi(a, ?(primitive(a)) : a) :
11            pi(N1, ?(N1 is N-1) : actionSequence(N1))
12 ).
```

Figura 6.8: O algoritmo simplificado do planejador WSPBF.

serem investigados, e caso não encontre um plano solução de tamanho $i < max$, WSPBF falha.

6.3.1 O procedimento *actionSequence* e a instanciação de planos

O procedimento *actionSequence*(N) é um gerador de planos executáveis de tamanho N, totalmente instanciados com ações do domínio.

Uma vez que o operador $\delta_1 \# \delta_2$ define uma escolha alternativa entre δ_1 e δ_2 (como em cláusulas alternativas em programas Prolog), o procedimento *actionSequence* pode ser executado de duas maneiras alternativas: $N = 0$ é uma sequencia de ações nula e o procedimento pára, ou, caso $N > 0$, *actionSequence* executa o teste $?(primitive(a))$ instanciando a com alguma ação primitiva. Através do meta-interpretador Golog ($Trans$), é verificado se é possível executar a ação a. Caso a seja executável, é feita a transição para a situação resultante da execução de a. Caso não seja executável, ocorre *backtracking* na escolha de ação primitiva no teste $?(primitive(a))$. Escolhida uma ação primitiva executável a, chama-se recursivamente *actionSequence*($N - 1$) que irá produzir um plano de $N - 1$ passos, executável a partir da situação resultante da execução de a.

6.3.2 O procedimento *plans* e a estratégia de busca em profundidade iterativa

À semelhança do predicado *plan* do IDDP, o procedimento *plans* controla a busca por planos, verificando todos os planos de tamanho i antes de verificar planos de tamanho $i+1$. Porém, a cada incremento no tamanho de planos, estes são construidos inicialmente vazios, i.e., as seqüências de ações consideradas para os planos de tamanho i, não são mantidas para os de tamanho i+1.

Uma vez que o operador $\delta_1 \# \delta_2$ também é usado em $plans(N, Max)$, há duas maneiras para satisfazer $plans(N, Max)$:

- A primeira tem a função de verificar se um plano de tamanho $N \leq Max$ que leve ao estado meta (especificado através do predicado *goal* na linha 5). Caso o *goal* não seja satisfeito, Golog *backtracks* até $actionSequence(N)$ para que um plano diferente de tamanho N seja encontrado por $actionSequence(N)$.
- A segunda forma de satisfazer $plans(N, Max)$ é através da linha 6. Caso todos os planos de tamanho N falhem no teste $?(goal)$, ocorre um *backtracking* que leva à busca por um plano de tamanho $N+1$ que possivelmente consiga satisfazer o teste $?(goal)$. Através desse mecanismo de *backtracking plans* consegue verificar todos os planos de tamanho i antes de verificar os planos de tamanho $i+1$.

Podemos ver que a chamada a $plans(N+1, Max)$ terá que reconstruir os planos que foram construídos em $plans(N, Max)$. Dessa maneira, podemos concluir que o planejador WSPBF também é um planejador de busca em profundidade iterativa.

6.4 Comparação entre IDDP e WSPBF

Para compararmos os dois planejadores no Cálculo de Situações, IDDP e WSPBF, construimos uma versão em Prolog do WSPBF (Figura 6.4), que chamaremos de SCP (*Situation Calculus Planner*), mantendo a mesma estratégia de construção de planos proposta pelo WSPBF do Reiter. Dessa

forma, temos como resultado um programa que não depende da axiomatização do Golog (axiomas *Trans*). Assim, a tarefa de planejamento em SCP consiste em obter-se uma prova construtiva para o seguinte teorema:

$$Dom \cup Prob \cup SCP \models \exists S\ planning(S)$$

```
1  planning(Max,S) :- plans(0,Max,s0,S).
2  plans(N,Max,S,Sr) :- N =< Max, actionSequence(N,S,Sr), goal(Sr);
3                       N1 is N + 1, plans(N1,Max,S,Sr).
4  actionSequence(0,S,S).
5  actionSequence(N,S,Sr) :- N > 0, primitive(A), poss(A,S),
6                            N1 is n - 1, actionSequence(N1,do(A,S),Sr).
```

Figura 6.9: O planejador WSPBF descrito em Prolog, chamado de SCP (Situation Calculus Planner.

O programa *actionSequence* contrói planos de tamanho N de forma recursiva, inserindo uma ação executável na primeira posição do plano e fazendo uma chamada recursiva para a construção do restante do plano, i.e., um plano de tamanho $N-1$. Para isso, *actionSequence* constrói o plano de tamanho N, com a primera ação do plano unificada com uma ação A possível de ser executada em S ($poss(A,S)$) e chama *actionSequence* para gerar um plano N-1 (chamada recursiva) a partir da situação resultante do(A,S).

Por outro lado, IDDP constrói um plano de tamanho N "vazio", i.e., um plano de tamanho N com a estrutura $do(A_n, do(A_{N-1}, ...(A_1, s_0)...)$, onde as ações A_i são variáveis não instanciadas.

Uma outra diferença é que no planejador SCP, o predicado *plans* não constrói planos incrementalmente como no IDDP (usando s_0 como base da recursão). Ao invés disso, SCP usa símbolos extra-lógicos (aritmética explicita) para incrementar o tamanho dos planos a serem examinados.

Note que, os principais predicados de SCP herdaram a necessidade de dois termos para representar situações do Golog: um para a situação de entrada e outro para a situação de saída, sendo que na

chamada do planejador a situação de entrada é inicializada com a constante s_0. Isso se assemelha à técnica de programação em Prolog do uso de um argumento como entrada e outro como saída para as chamadas recursivas (linha 4 da Figura 6.4). Já no IDDP isso não acontece, o que lhe dá uma característica menos procedimental.

Em resumo, os planejadores IDDP e SCP têm duas características em comum: utilizam o Cálculo de Situações para escolher ações executáveis; e realizam planejamento progressivo através de uma busca em profundidade iterativa. Apesar do IDDP ser mais sintético, não usar símbolos extra-lógicos (de igualdade e aritmético) e além disso, não usar parâmetros (situações) como entrada e saída de chamadas recursivas, os dois planejadores possuem estratégias muito semelhantes para geração de planos de satisfação de metas. No Capítulo 7 será feita uma análise de desempenho desses dois planejadores e o planejador que propomos na próxima seção.

6.5 Tarefas-meta em Golog

[Gabaldon, 2002] interpretou a execução de um programa Golog como planejamento hierárquico mas não fez nenhuma menção às tarefas meta definidas no Capítulo 3. Nessa seção, mostraremos que a implementação de tarefas meta na linguagem Golog permite fazer planejamento para metas em Golog de modo bem mais direto e elegante que os algoritmos IDDP e WSPBF. Além disso, no Capítulo 7 mostramos que essa é uma maneira mais eficiente de planejamento para metas baseada no Cálculo de Situações.

6.5.1 Satisfação como possível conseqüência de programa Golog

Considere o seguinte programa Golog [Levesque, H. J. et al., 1997]:

```
criaTorre(7) : ?(livre(a))
```

Esse programa chama o procedimento *criaTorre*, visto na Figura 4.2, seguido de um teste. A execução deste programa devolve um plano para criar uma torre de sete blocos tal que o bloco a

esteja livre após a execução desse plano. O bloco a não precisa necessariamente fazer parte da torre de sete blocos. O teste $?(livre(a))$ no final desse programa faz com que, de todos os possíveis planos resultantes do processamento de $criaTorre$, sejam devolvidos apenas aqueles em que o bloco a esteja livre. Se mudássemos o teste para $?(sobre(a, b)\ \&\ sobre(b, c))$, o meta-interpretador Golog seria obrigado a devolver apenas os planos resultantes do processamento de $criaTorre(7)$ em que $sobre(a, b)$ e $sobre(b, c)$ fossem verdadeiros.

O fato do meta-interpretador Golog devolver um plano que satisfaz os fluentes de um teste $?(t)$ lembra o planejamento clássico para satisfação de fluentes. entretanto, isso não significa que o meta-interpretador seja capaz de planejar para satisfação de fluentes. O que Golog faz é procurar uma execução de $criaTorre(7)$ particular que também satisfaça o teste seguinte. Caso $criaTorre(7)$ devolva um plano em que o teste falhe, a execução do programa retrocede.

Na Figura 6.10, temos uma situação inicial com três blocos a, b e c e queremos um plano que leve ao estado meta em que a está sobre b que está sobre c. Se tentamos forçar o meta-interpretador Golog a encontrar um processamento de $criaTorre(3)$ que satisfaça $sobre(a, b)$ e $sobre(b, c)$, nenhum plano será encontrado. Existe um plano para satisfazer $sobre(a, b)$ e $sobre(b, c)$, o plano é $desempilha(c, a) \prec empilha(b, c) \prec empilha(a, b)$, mas ele não pode ser obtido a partir de $criaTorre(3)$. Os únicos planos que podem ser obtidos de $criaTorre(3)$ são $empilha(b, c)$ e $move(c, a, b) \prec empilha(a, c)$, que não satisfazem o fluente meta.

Usar $criaTorre(n)$ para devolver um plano que satisfaça uma conjunção de fluentes t nem sempre funciona por um simples motivo: $criaTorre$ não foi especificado para satisfazer fluentes como $sobre(a, b)$, $sobre(b, c)$ ou $livre(a)$. No máximo, podemos dizer que ele foi criado para satisfazer um fluente $existeTorre(n)$, que é verdadeiro quando há uma torre de n blocos. Em algumas ocasiões, $criaTorre$ satisfaz os fluentes de t, mas isso é apenas acidental. Para satisfazer os fluentes $sobre(a, b)$ e $sobre(b, c)$ é necessário usar programas Golog projetados especificamente para isso.

```
% Situação inicial      c
%                       a  b
sobreMesa(b,s0).
sobreMesa(a,s0).
sobre(c,a,s0).
livre(b,s0).
livre(c,s0).

% programa para satisfazer fluentes sobre(a,b) e sobre(b,c)
criaTorre(3) : ?(sobre(a,b) & sobre(b,c))

% Execuções possíveis para criaTorre(3)
do(empilha(b,c),s0)
do(empilha(a,c),do(move(c,a,b),s0))
```

Figura 6.10: Não há execução de *criaTorre*(3) que satisfaça sobre(a,b) e sobre(b,c).

6.5.2 Procedimentos meta

O planejamento hierárquico consegue realizar planejamento orientado para satisfação de metas devido à maneira como as tarefas meta são estruturadas. Procedimentos Golog que reproduzam a estrutura das tarefas meta devem dar ao meta-interpretador a capacidade de realizar o planejamento para satisfação de metas. Chamaremos tais procedimentos de **procedimentos meta**, onde a execução de um procedimento meta *achieve*(f) torna o fluente f verdadeiro.

Assim como uma tarefa meta para satisfazer um fluente f pode ser decomposta por pelo menos dois métodos diferentes (Seção 3.6), o procedimento meta *achieve*(f) pode ser executado de duas maneiras diferentes para realizar a satisfação de f: uma maneira para o caso do fluente f já ser verdadeiro e outra maneira para o caso contrário. Um procedimento meta como *achieve*(*livre*(b)) pode ter a seguinte estrutura:

```
proc(achieve(livre(b)),
    ?(livre(b)) #                                              (i)
    (achieve(livre(a)) || achieve(sobre(a,b))) : desempilha(a,b)    (ii)
).
```

Esse procedimento, descreve duas formas de executar *achieve*(*livre*(b)). A primeira forma (i) é

apenas um teste do fluente $livre(b)$ e funciona como o *método nulo* das tarefas meta. A segunda forma (ii), tenta reproduzir a rede de tarefas da Figura 6.11, que representa a rede do método não-nulo da tarefa meta $achieve(livre(b))$.

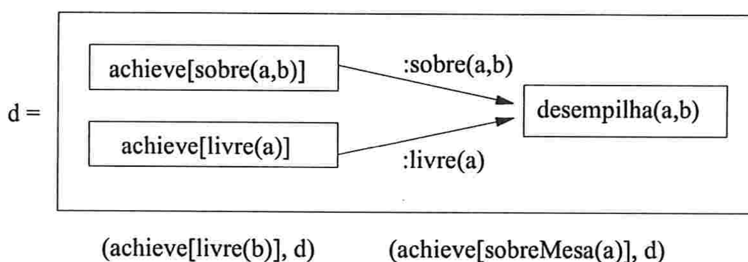

Figura 6.11: Rede de tarefas para $achieve[livre(b)]$ e $achieve[sobreMesa(a)]$.

Lembrando que essa mesma rede também pode servir para satisfazer outros fluentes, por exemplo, o fluente $sobreMesa(a)$, o mesmo programa que representa essa rede pode ser usado em um procedimento $achieve(sobreMesa(a))$, como especificado a seguir:

```
proc(achieve(sobreMesa(a)),
    ?(sobreMesa(a)) #
    (achieve(livre(a)) || achieve(sobre(a,b))) : desempilha(a,b)
).
```

No código (ii), o operador de seqüência de ações (:) consegue representar o fato de que a única tarefa primitiva, $desempilha(a, b)$, é precedida por todas as tarefas meta (do tipo *achieve*) que satisfazem as suas pré-condições. O operador de execução concorrente (||) entre $achieve(livre(a))$ e $achieve(sobre(a, b)))$ representa o fato de não existir relação de ordem entre essas duas tarefas meta.

Apesar de representar bem as restrições de ordem, o programa (ii) não consegue representar as restrições de fluentes $(sobre(a, b), desempilha(a, b))$ e $(livre(a), desempilha(a, b))$, pelos mesmos motivos que já discutimos na seção 5.4.2. Entretanto, isso não impede que seja encontrado um plano para satisfazer o fluente $livre(b)$. Se, por exemplo, $livre(a)$ vier a ter valor falso devido a alguma ação ser intercalada antes de $desempilha(a, b)$, o meta-interpretador avaliará $Trans(desempilha(a, b), S, Pr, Sr)$ como falso, pois $poss(desempilha(a, b), S)$ será falso e isso obrigará o meta-interpretador a fazer

backtracking e desfazer essa intercalação. Apesar de não conseguir manter $livre(a)$ verdadeiro entre $achieve(livre(a))$ e $desempilha(a, b)$, o meta-interpretador consegue garantir que $livre(a)$ seja verdadeiro atrabés de backtrackings (o mesmo vale para o fluente($sobre(a, b)$).

```
proc(achieve(livre(X)),
     ?(livre(X)) #
     pi(y, ?(sobre(y,X)) : achieve(livre(y)) : desempilha(y,X))
).

proc(achieve(sobreMesa(X)),
     ?(sobreMesa(X)) #
     pi(y, ?(sobre(X,y)) : achieve(livre(X)) : desempilha(X,y))
).

proc(achieve(sobre(X,Y)),
     ?(sobre(X,Y)) #
     achieve(livre(X)) : conc(achieve(sobreMesa(X)) , achieve(livre(Y)))
     : empilha(X,Y)
).
```

Figura 6.12: Procedimentos meta para satisfação dos fluentes *livre*, *sobreMesa*, *sobre* do Mundo dos Blocos.

Em resumo, o fato de não ser possível representar fielmente as restrições de fluentes da rede de tarefas da Figura 6.11 não afeta a capacidade do procedimento meta $achieve(livre(b))$ de satisfazer o fluente $livre(b)$).

A Figura 6.12 mostra o programa completo para satisfação dos fluentes *livre*, *sobreMesa*, *sobre* do Mundo dos Blocos. A especificação de desses três procedimentos meta permite que Golog realize planejamento para satisfação de metas como no planejamento hierárquico. Note que essa proposta não requer a definição de nenhum meta-predicado extra, como *plan*, *exec* ou *actionSequence* dos planejadores IDDP e WSPBF. Para criar um plano que atinja um estado meta $\{sobre(a, b)\}$ basta executar o programa

achieve(sobre(a,b)): ?(sobre(a,b))

E para atingir um estado meta $\{sobre(a, b),\ sobre(b, c)\}$, basta executar o programa

(achieve(sobre(a,b)) || achieve(sobre(b,c))) : ?(sobre(a,b) & sobre(b,c))

O uso da concorrência entre $achieve(sobre(a,b))$ e $achieve(sobre(b,c))$ permite que todas as possíveis intercalações entre as redes de tarefas resultantes desses dois procedimentos sejam testadas. O teste $?(sobre(a,b) \& sobre(b,c))$ garante que só será escolhida a intercalação que atinge o estado meta $\{sobre(a,b), sobre(b,c)\}$.

No capítulo seguinte, faremos uma análise empírica dos planejadores para satisfação de metas apresentados nesse capítulo.

Capítulo 7

Análise de desempenho

Para avaliar o desempenho do planejador baseado em procedimentos meta, que chamaremos aqui de **GGPP** (**G**olog **G**oal **P**rocedures based **P**lanner) e dos planejadores IDDP, SCP e WSPBF foram selecionados problemas do Mundo dos Blocos com as ações de empilhar e desempilhar. Para comparar o desempenho entre os quatro tipos de planejamento baseado no Cálculo de Situações, levantamos os seguintes dados: quantidade de planos verificados e o número de inferências (Prolog) realizadas pelos planejadores para encontrar um plano solução.

Pelo que foi dito no Capítulo 6, espera-se que esses testes confirmem que o SCP e o IDDP tenham desempenho muito parecido, i.e., façam o mesmo número de inferências, pois utilizam o mesmo algoritmo e funcionam utilizando apenas a camada do Cálculo de Situações. Para o WSPBF espera-se que seu desempenho seja pior que o do SCP e o do IDDP, pois apesar de utilizar o mesmo algoritmo do SCP, o WSPBF funciona com a camada extra de interpretação da linguagem Golog. Para o GGPP, esperamos que seu desempenho seja superior ao do WSPBF, pois enquanto o WSPBF faz uma busca não orientada, o GGPP, graças aos procedimentos meta, faz uma busca orientada, ou seja, só adiciona passos se esses tiverem a possibilidade de atingir o estado meta.

Em alguns problemas, notamos que o desempenho do IDDP podia ser muito influenciado pela ordem em que as declarações dos axiomas de pré-condição *Poss* das ações *empilha* e *desempilha*

eram feitas em Prolog. No caso do WSPBF, o desempenho podia ser influenciado pela ordem em que eram feitas as declarações *primitive* dessas ações. Por esse motivo, em alguns casos, para um mesmo problema, apresentamos duas quantidades de planos testados e duas quantidades de inferências feitas, obtendo diferentes dados (número de planos, número de inferências) para o mesmo problema resolvido com diferentes ordens das declarações de *Poss*.

A seguir, apresentamos os resultados que obtivemos desses testes.

7.1 Teste 1

No primeiro grupo de testes, foram definidos cinco problemas (Figura 7.1). Nesses problemas a descrição de estado meta é sempre a mesma, $G = \{sobre(a,b), sobre(b,c)\}$ (2 sub-metas), enquanto que o estado inicial é diferente. Note que a disposição dos blocos a, b e c nas difentes situações iniciais permanece a mesma. As diferenças dos problemas estão apenas no número de blocos: o problema $A1$ envolve 3 blocos, $A2$ envolve 7 blocos, $A3$ envolve 9, $A4$ envolve 11 e $A5$ envolve 12 blocos.

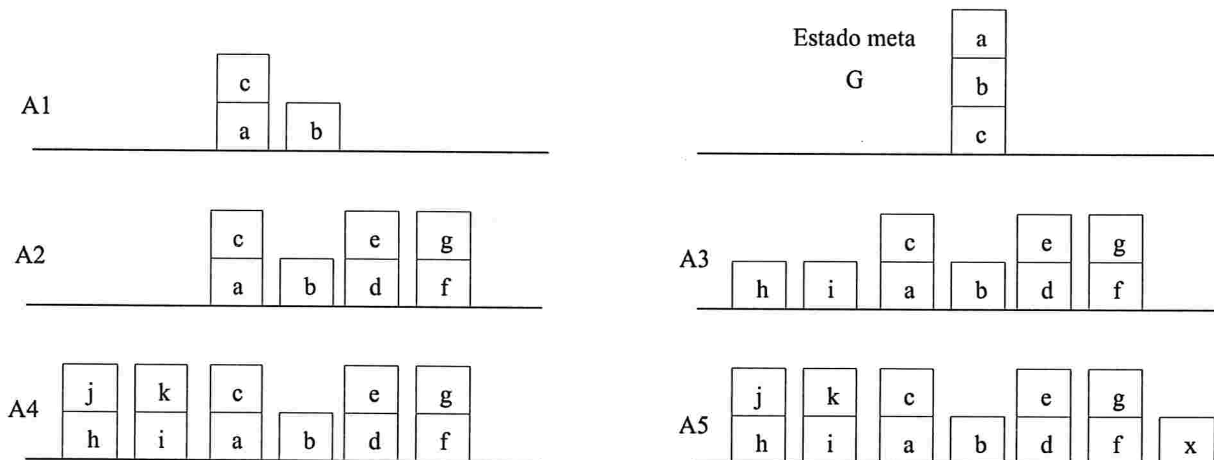

Figura 7.1: Problemas de planejamento para o **Teste 1**. $A1$, $A2$, $A3$, $A4$ e $A5$ são estados iniciais de problemas para o mesmo estado meta G.

O objetivo do Teste 1 é avaliar o desempenho dos planejadores diante de problemas em que

existem muitos objetos no mundo não relacionados com a meta do problema. Os resultados do Teste 1 estão na Tabela 7.1.

	Problema	IDDP		SCP		WSPBF		GGPP	
	Submetas/Tam.	Planos	Infer.	Planos	Infer.	Planos	Infer.	Planos	Infer.
A1	2/3	16	575	16	668	16	207.112	3	104.799
		17	637	17	742	17	237.502	3	104.799
A2	2/3	142	3.843	142	4.448	142	1.184.340	3	104.799
		153	4.625	153	5.300	153	1.385.308	3	104.799
A3	2/3	482	11.651	482	13.638	482	3.724.960	3	104.799
		1.861	56.865	1.861	64.658	1.861	15.338.350	3	104.799
A4	2/3	380	9.687	380	11,220	380	2.940.496	3	104.799
		553	15.845	553	18.090	553	4.470.858	3	104.799
A5	2/3	639	15.529	639	18.096	639	4.842.244	3	104.799
		1.934	56.700	1.934	64,485	1.934	15.366.451	3	104.799

Tabela 7.1: Desempenho com mesma meta e número variável de objetos.

A primeira observação que fizemos é que os problemas resolvidos com o GGPP não eram afetados pela ordem em que eram feitas as declarações *primitive* e *poss*. Nas próximas tabelas, omitiremos os resultados para o GGPP com troca de ordem das declarações *primitive* e *poss*.

Assim, nosso primeiro resultado importante é:

Resultado 1 : A ordem em que são feitas as declarações *primitive* e *poss* não alteram o desempenho do planejador GGPP.

O primeiro fato que chama a atenção do planejador GGPP no Teste 1 é o valor constante de inferências realizadas e o número de planos testados para todos os problemas, de $A1$ a $A5$, para encontrar o plano solução. Não importa a quantidade de blocos a mais que sejam colocados à volta de a, b e c, eles não alteram o número de inferências (104.799) e o número de planos analisados (3) que o GGPP faz na busca pelo plano solução.

Uma explicação para esse comportamento é porque o programa Golog usado para solucionar o problema,

```
conc(achieve(sobre(a,b)), achieve(sobre(b,c))) : ?(sobre(a,b) & sobre(b,c))
```

faz referências apenas aos blocos a, b e c assim como o procedimento *achieve*($sobre(a,b)$) e o procedimento *achieve*($sobre(b,c)$), cujas instâncias mostramos a seguir, também só fazem referência a esses blocos.

```
proc(achieve(sobre(a,b)),
    ?(sobre(a,b)) #
    achieve(livre(a)) : achieve(sobreMesa(a)) : achieve(livre(b)) : empilha(a,b)
).
proc(achieve(sobre(b,c)),
    ?(sobre(b,c)) #
    achieve(livre(b)) : achieve(sobreMesa(b)) : achieve(livre(c)) : empilha(b,c)
).
```

Para o GGPP, blocos que não estão envolvidos na descrição da meta são considerados apenas quando eles interferem na satisfação de novas metas (submetas). Por exemplo, se houvesse um bloco z sobre b, o procedimento *achieve*($sobre(b,c)$) iria chamar *achieve*($livre(b)$) que por sua vez planejaria para a retirada do bloco z de sobre b. A movimentação do bloco z passaria a fazer parte do plano porque o operador de escolha de argumentos *pi*, junto com o teste ?($sobre(y,b)$), descobriria que o bloco z evita a movimentação de b e isso causaria a remoção de z sobre o bloco b.

```
proc(achieve(livre(b)),
    ?(livre(b)) #
    pi(y, ?(sobre(y,b)) : achieve(livre(y)) : desempilha(y,b))
).
```

Dessa forma, nosso segundo resultado importante é:

Resultado 2 : Objetos do domínio que não interagem com as submetas do problema de planejamento não afetam o desempenho do GGPP.

Observando o número de inferências dos planejadores WSPBF e GGPP, notamos que para todos os problemas eles apresentam os piores resultados. Isso pode ser explicado pelo fato deles passarem

pelo tratamento do meta-interpretador Golog, ao contrário dos planejadores IDDP e SCP. Estes dois últimos fazem um número menor de inferências para o Teste 1 por utilizarem apenas o mecanismo de busca do Prolog diretamente com os axiomas do Cálculo de Situações. Como veremos nos próximos testes, isso nem sempre será uma vantagem.

Já os dados do planejador WSPBF, escrito também em Golog, se mostraram bem piores que os resultados do GGPP. Isso pode ser explicado pela ineficiência da busca progressiva implementada em Golog pelo WSPBF quando comparada com a busca para satisfação de metas do GGPP.

A eficiência do GGPP para satisfação de metas pode ser comprovada pelo número pequeno (e constante) de planos verificados.

Resultado 3: Para os problemas do Teste 1 o GGPP verificou o menor número de planos.

Os planejadores IDDP e SCP possuem um comportamento muito semelhante. O número maior de inferências do SCP quando comparado às do IDDP para os mesmos problemas, pode ser justificado pelas operações numéricas que o SCP realiza.

É interessante notar que os três planejadores, IDDP, SCP e WSPBF geraram extamente o mesmo número de planos, o que mostra a equivalência de suas estratégias de busca.

7.2 Teste 2

No segundo grupo de testes, definimos oito problemas, de $B1$ a $B8$ representados na Figura 7.2. Esses problemas têm em comum o mesmo estado inicial I: mesmo número de blocos dispostos da mesma maneira. Os estados meta dos problemas são dados por apenas um fluente e diferem no número de passos necessários para serem atingidos, tal que Bi é solucionado por um plano de i passos. O estado meta de $B1$ é $\{livre(b)\}$, o estado meta de $B2$ é $\{livre(c)\}$, o estado meta de $B3$ é $\{livre(d)\}$ e assim por diante.

Note que, os problemas foram elaborados de forma que as novas submetas que surgem durante o

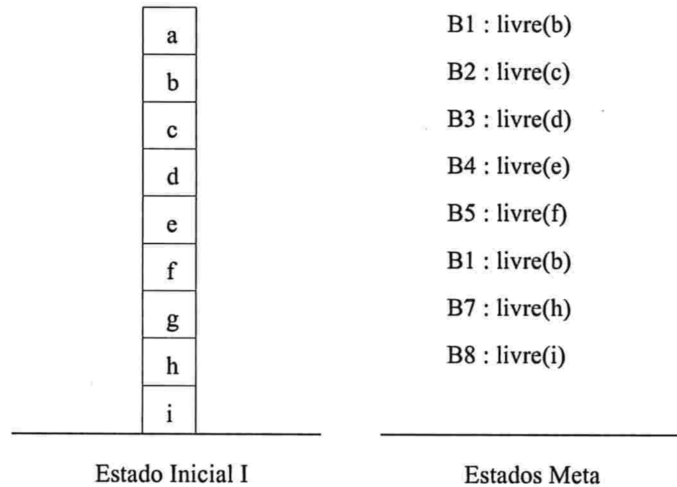

Figura 7.2: Problemas de planejamento para o **Teste 2.** $B1$, $B2$, $B3$, $B4$, $B5$, $B6$, $B7$ e $B8$ descrevem cada uma das diferentes metas dos 8 problemas para o mesmo estado inicial I.

planejamento não interam negativamente entre si, isto é, de forma que a satisfação de uma submeta não desfaz a satisfação de outra submeta. Tentamos evitar interações negativas nesses problemas porque queremos analisar como o desempenho do GGPP é afetado com a variação no tamanho de plano solução sem interações entre ações. Os resultados dos testes estão na Tabela 7.2.

Para todos os problemas, o planejador GGPP verificou apenas um plano.

Resultado 4: problemas de planejamento sem interação negativa de ações, o primeiro plano verificado pelo GGPP é o plano solução.

como era previsto, os desempenhos dos planejadores IDDP, SCP e WSPBF degradaram rapidamente com o tamanho dos planos soluções. O mesmo não ocorreu com o planejador GGPP. Uma explicação para isso deve-se ao fato dos procedimentos meta do tipo *achieve*(f) serem usados para construir planos utilizando apenas as ações que podem levar à satisfação de f, limitando o número de ações a serem consideradas e consequentemente o número de planos a serem verificados. Além disso, como não ocorrem interações entre as submetas, não há o risco de um procedimento meta criar planos em que a satisfação de uma submeta desfaça outra submeta, o que levaria a um *backtracking*

para tentar encontrar outro plano. Assim, no GGPP, o primeiro plano gerado é o plano solução.

Outra conclusão que podemos obter desses testes é que o simples aumento do número de passos do plano solução não produz no planejador GGPP um aumento tão pronunciado no número de inferências como nos outros três planejadores. Novamente, isso se deve ao fato do GGPP implentar uma busca para satisfação de metas enquanto os outros três realizam busca em progressiva (busca em profundidade iterativa).

É interessante notar que para o Teste 2, novamente os três planejadores, IDDP, SCP e WSPBF geraram exatamente o mesmo número de planos, o que mais uma vez confirma a equivalência de suas estratégias de busca. A diferença no número de inferências do WSPBF em relação ao SCP e o IDDP deve-se novamente à camada extra do interpretador Golog.

Problema		**IDDP**		**SCP**		**WSPBF**		**GGPP**	
	Submetas/Tam.	Planos	Infer.	Planos	Infer.	Planos	Infer.	Planos	Infer.
B1	1/1	2	25	2	34	2	17.701	1	6.648
		2	23	2	31	2	19.446		
B2	1/2	4	94	4	117	4	54.146	1	12.497
		3	60	3	77	3	43.694		
B3	1/3	10	358	10	418	10	89.254	1	20.708
		5	164	5	197	5	141.906		
B4	1/4	30	1.498	30	1.672	30	403.001	1	31.881
		11	485	11	557	11	187.896		
B5	1/5	122	7.471	122	8.118	122	1.419.404	1	46.720
		31	1.710	31	1.898	31	459.901		
B6	1/6	610	42.815	610	45.834	610	6.366.545	1	66.033
		123	7.802	123	8.465	123	1.487.248		
B7	1/7	3.586	286.191	3.586	303.125	3.586	34.901.845	1	90.732
		611	43.305	611	46.342	611	6.445.373		
B8	1/8	24.978	2.178.731	24.978	2.292.683	24.978	230.148.900	1	121.833
		3.587	286.886	3.587	303.840	3.587	34.991.703		

Tabela 7.2: Comparação no tamanho do plano solução.

7.3 Teste 3

Os testes anteriores mostraram que o número de objetos e o tamanho dos planos solução não afetam o desempenho do GGPP. O Teste 3 avalia o desempenho dos planejadores para problemas que envolvem ações que interagem. A Figura 7.3 mostra 4 problemas D_i, onde o estado meta possui i submetas. Nesse teste, não nos preocupamos em manter o número de objetos e o número de passos dos planos solução constantes em todos os problemas pois já foi verificado que eles não afetam o desempenho do planejador GGPP.

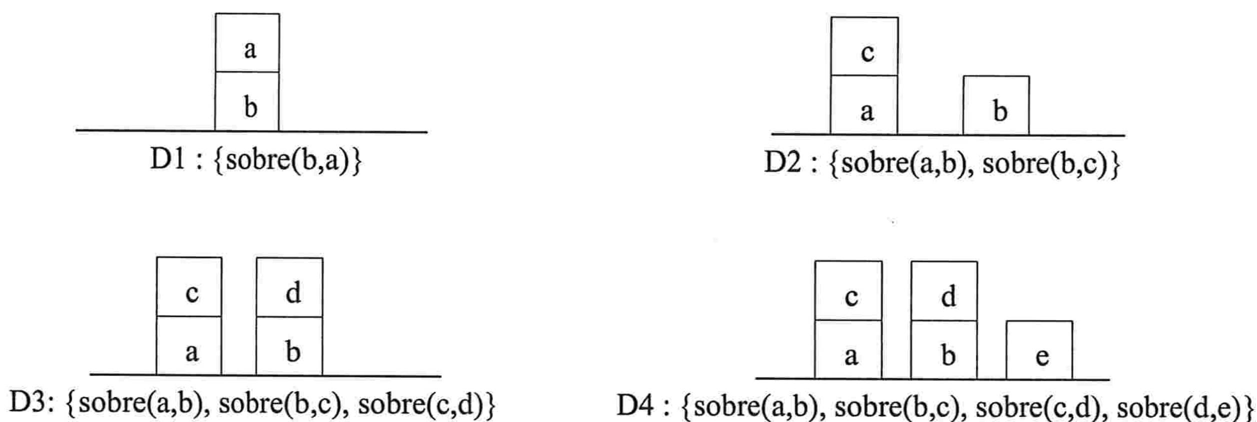

Figura 7.3: Problemas de planejamento do **Teste 3**. As ilustrações representam os estados iniciais e $D1$, $D2$, $D3$, $D4$, $D5$ e $D6$, os estados meta correspondentes.

Na Tabela 7.3, podemos ver que para o Teste 3, o desempenho do planejador GGPP degrada mais rápidamente que o desempenho do IDDP, o SCP e o WSPBF.

Problema		IDDP		SCP		WSPBF		GGPP	
	S./T.	Planos	Infer.	Planos	Infer.	Planos	Infer.	Planos	Infer.
D1	1/2	3	69	3	87	3	45.446	1	18.674
		3	60	3	78	3	45.437		
D2	2/3	16	575	16	659	16	207.103	3	81.802
		17	637	17	733	17	237.493		
D3	3/5	371	21.035	371	22.689	371	3.972.288	134	3.725.549
		482	28.805	482	31.006	482	5.288.166		
D4	4/6	11.145	732.012	11.145	776.157	11.145	104.933.434	55.657	2.091.750.830
		14.008	931.137	14.008	987.249	14.008	133.469.846		

Tabela 7.3: Desempenho em problemas com interação entre submetas.

A causa dessa queda de desempenho tão abrupta do GGPP está na maneira como funciona o operador de execução concorrente do Golog na tentativa de intercalar ações.

Vejamos, novamente, a definição do operador de concorrência feita através do predicado *Trans* do meta-interpretador Golog.

```
trans(conc(P1,P2),S,conc(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).
trans(conc(P1,P2),S,conc(P1,P2r),Sr) :- trans(P2,S,P2r,Sr).
```

Essas cláusulas especificam que a execução concorrente de dois programas $P1$ e $P2$ é uma intercalação das ações do programa $P1$ com as ações do programa $P2$. Por exemplo, em um programa como $a1 \mid\mid a2$, onde $a1$ e $a2$ são ações primitivas, existem duas maneiras de intercalar $a1$ e $a2$, produzindo os planos $a1 \prec a2$ e $a2 \prec a1$. Conforme aumenta o número de ações primitivas sendo executadas concorrentemente, o número de planos possíveis de serem produzidos aumenta exponencialmente, isto é, em um programa $a1 \mid\mid a2 \mid\mid ... \mid\mid a_n$ o número de programas possíveis é $n!$.

Esse número de planos pode crescer ainda mais se o programa conter, além de ações, testes que devem ser intercalados com as ações.

Por exemplo, no programa $(a1 : a2) \mid\mid (a3 : a4)$, os programas $a1 : a2$ e $a3 : a4$ podem ser intercalados de seis maneiras:

$$a1 \prec a2 \prec a3 \prec a4$$
$$a1 \prec a3 \prec a2 \prec a4$$
$$a1 \prec a3 \prec a4 \prec a2$$
$$a3 \prec a1 \prec a2 \prec a4$$
$$a3 \prec a1 \prec a4 \prec a2$$
$$a3 \prec a4 \prec a1 \prec a2$$

Se adicionarmos um teste a $(a1 : a2) \mid\mid (a3 : a4)$, criando $(a1 : a2) \mid\mid (?(f) : a3 : a4)$, o meta-interpretador Golog pode devolver os seguintes planos:

$$a1 \prec a2 \prec a3 \prec a4$$
$$a1 \prec a2 \prec a3 \prec a4$$
$$a1 \prec a3 \prec a2 \prec a4$$
$$a1 \prec a3 \prec a4 \prec a2$$
$$a1 \prec a2 \prec a3 \prec a4$$
$$a1 \prec a3 \prec a2 \prec a4$$
$$a1 \prec a3 \prec a4 \prec a2$$
$$a1 \prec a3 \prec a4 \prec a2$$
$$a3 \prec a1 \prec a4 \prec a2$$
$$a3 \prec a1 \prec a4 \prec a2$$

Notamos que alguns dos planos aparecem mais de uma vez. O motivo da repetição de planos deve-se ao fato de um teste $?(f)$ ser interpretado como se fosse uma ação primitiva como as ações $a1$, $a2$, $a3$ e $a4$, mas uma ação sem efeito e com a única pré-condição de f ser verdadeiro. Como os testes não aparecem, a impressão que se têm é de que intercalações repetidas estão sendo produzidas. A lista de planos a seguir mostra os testes realizados nos diferentes pontos dos planos verificados.

$$a1 \prec a2 \prec ?(f) \prec a3 \prec a4$$
$$a1 \prec ?(f) \prec a2 \prec a3 \prec a4$$
$$a1 \prec ?(f) \prec a3 \prec a2 \prec a4$$

a1 $\prec$?(f) $\prec$ a3 $\prec$ a4 $\prec$ a2

?(f) $\prec$ a1 $\prec$ a2 $\prec$ a3 $\prec$ a4

?(f) $\prec$ a1 $\prec$ a3 $\prec$ a2 $\prec$ a4

?(f) $\prec$ a1 $\prec$ a3 $\prec$ a4 $\prec$ a2

?(f) $\prec$ a3 $\prec$ a1 $\prec$ a2 $\prec$ a4

?(f) $\prec$ a3 $\prec$ a1 $\prec$ a4 $\prec$ a2

?(f) $\prec$ a3 $\prec$ a4 $\prec$ a1 $\prec$ a2

Resultado 5: Em problemas com interação de ações, o desempenho do GGPP cai rapidamente devido ao rápido aumento do número de planos repetidos a serem testados.

7.4 Teste 4

Vimos no teste anterior que as intercalações repetidas produzidas pelos testes no meio dos procedimentos meta são responsáveis pela perda de desempenho do planejador GGPP. Sendo assim, realizamos uma modificação no operador de execução concorrente para eliminar as intercalações de testes, para o mesmo conjunto de problemas do Teste 3, adicionando mais dois problemas extras.

Uma maneira de identificar a ocorrência de um teste em uma transição $Trans(P, S, Pr, Sr)$ é realizar uma comparação das situações S e Sr. Se S é igual a Sr, isso significa que ocorreu um teste na transição. Se S é diferente de Sr, uma ação primitiva foi executada. Usamos essa idéia para modificar o operador de execução concorrente. A seguir, a nova definição do operador de concorrência:

```
trans(conc(P1, P2), S, conc(Pr,P2), Sr) :- transA(P1, S, Pr, Sr).
trans(conc(P1, P2), S, conc(P1,Pr), Sr) :- transA(P2, S, Pr, Sr).

transA(P, S, Ps, Ss) :- trans(P, S, Pr, Sr),
    (final(Pr, Sr) -> (Ps = nil, Ss = Sr);
                (S = Sr -> (transA(Pr, Sr, Prr, Srr), Ps = Prr, Ss = Srr);
```

(Ps = Pr, Ss = Sr))).

Como na definição original, existem duas cláusulas $Trans$ para o operador de concorrência, uma cláusula que executa um passo do programa $P1$ e outra que executa um passo do programa $P2$. Na definição original, *executar um passo de um programa* P significava realizar uma transição de programa e situação. Na nossa definição, executar um passo *real*, significará executar uma ação primitiva. Em lugar de chamar diretamente um resolvente $Trans(P1, S, Pr, Sr)$ ou $Trans(P2, S, Pr, Sr)$, as cláusulas $Trans$ do operador de concorrência chama um resolvente $TransA(P1, S, Pr, Sr)$ ou $TransA(P2, S, Pr, Sr)$. $TransA(P, S, Ps, Ss)$ funciona aplicando sucessivas transições $Trans$ ao programa P até que seja gerada uma situação diferente de S, o que significa que uma ação primitiva foi executada, ou até que o programa tenha chegado a uma configuração final. Em seguida, $TransA$ devolve em Ps e Ss o programa e situação resultantes dessa execução de um passo real do programa P.

Os resultados obtidos com essa nova definição do operador de concorrência são apresentados na Tabela 7.4. Para o WSPBF, em D6, encerramos o teste depois de demorar mais de 30 minutos, o que significa que ele fez mais de um bilhão de inferências até esse momento. Se compararmos com os resultados obtidos utilizando o operador de concorrência original na Tabela 7.3, vamos notar uma significativa melhora. No caso do problema $D4$, por exemplo, o operador original fez algo em torno de dois bilhões de inferências contra menos de dois milhões de inferências com o novo operador. Os testes com os problemas $D5$ e $D6$ puderam ser realizados em um tempo razoável pelo planejador GGPP. O mesmo não pode ser dito do WSPBF que estourou o tempo de trinta minutos para o problema $D6$. Para o planejador WSPBF, não houve diferença nos resultados com o operador de concorrência melhorado pois ele não utiliza esse operador.

Problema		IDDP		SCP		WSPBF		GGPP	
	S./T.	Planos	Infer.	Planos	Infer.	Planos	Infer.	Planos	Infer.
D1	1/2	3	69	3	87	3	45.446	1	20.941
		3	60	3	78	3	45.437		
D2	2/3	16	575	16	659	16	207.103	2	55.287
		17	637	17	733	17	237.493		
D3	3/5	371	21.035	371	22.689	371	3.972.288	8	262.192
		482	28.805	482	31.006	482	5.288.166		
D4	4/6	11.145	732.012	11.145	776.157	11.145	104.933.434	56	2.133.174
		14.008	931.137	14.008	987.249	14.008	133.469.846		
D5	5/7	113.022	7.535.363	113.022	7.952.351	113.022	986.831.233	173	6.214.769
		104.972	6.907.090	104.972	7.295.132	104.972	918.352.037		
D6	6/8	1.194.501	108.023.836	1.194.501	112.468.688		N/D	2.016	79.791.827
		1.888.932	177.002.483	1.888.932	184.037.937				

Tabela 7.4: Problemas com interação entre submetas usando o operador de concorrência melhorado.

Resultado 6: A eliminação dos testes na intercalação de planos através do operador de concorrência melhorado torna o planejador GGPP mais eficiente que os demais planejadores.

Capítulo 8

Conclusões

Golog é uma linguagem de alto nível criada para especificar programas para agentes robóticos, e como acontece em situações do mundo real, Golog deve permitir a programação de agentes capazes de tratar eventos exógenos (externos) imprevistos. Como nem sempre é possível prever todas as contingências do mundo pelo programador, a especificação de tais programas é difícil e pode ocupar muita memória, o que pode ser um problema quando se quer criar agentes robóticos de baixo custo e alto desempenho.

Uma outra possibilidade é o planejamento com execução e monitoramento, i.e., planejar enquanto se executa as ações do plano, tanto de atuação como de sensoriamento do ambiente. Em programas IndiGolog, podemos continuar a execução do programa após uma eventual falha, criando um plano que corrija a falha. Isso pode ser feito por um planejador para satisfação de metas.

Por outro lado, nesse trabalho, foi mostrado que o meta-interpretador Golog realiza planejamento de forma semelhante aos planejadores hierárquicos e que baseado nessa técnica, Golog pode resolver problemas de planejamento para satisfação de metas através de **procedimentos meta**. Chamamos esse tipo de planejamento de **GGPP** (**G**olog **G**oal **P**rocedures based **P**lanner) e para comparar seu desempenho com outros planejadores foram analisados três outros planejadores para satisfação de meta baseados no Cálculo de Situações:

1. o **IDDP** [Pereira, 2002], um planejador não-hierárquico descrito em Prolog, proposto por um dos alunos do grupo de Lógica, Inteligência Artificial e Métodos Formais do IME-USP;
2. o **WSPBF** [Reiter, 2001a], um algoritmo de planejamento não-hierárquico proposto por Reiter para a linguagem Golog;
3. o **SCP**, uma versão em Prolog do planejador WSPBF

A principal contribuição desse trabalho é mostrar que planejamento orientado à satisfação de metas em Golog pode ser visto como a decomposição de **procedimentos meta**. Além disso, foi mostrado que o uso de **procedimentos meta** em Golog pode ser mais eficiente que outros planejadores para satisfação de metas no Cálculo de Situações.

Os testes de desempenho que realizamos mostraram que, apesar da carga extra da interpretação da linguagem Golog, o planejador GGPP pode ser, em muitos casos, mais eficiente que o planejador IDDP e o WSPBF. Os principais resultados dos testes de desempenho realizados com os planejadores para satisfação de metas foram:

- Resultado 1: A ordem em que são feitas as declarações *primitive* e *poss* não alteram o desempenho do planejador GGPP.
- Resultado 2: Objetos do domínio que não interagem com as submetas do problema de planejamento não afetam o desempenho do GGPP.
- Resultado 3: Para os problemas do Teste 1 o GGPP verificou o menor número de planos.
- Resultado 4: problemas de planejamento sem interação negativa de ações, o primeiro plano verificado pelo GGPP é o plano solução.
- Resultado 5: Em problemas com interação de ações, o desempenho do GGPP cai rapidamente devido ao rápido aumento do número de planos repetidos a serem testados.
- Resultado 6: A eliminação dos testes na intercalação de planos através do operador de concorrência melhorado torna o planejador GGPP mais eficiente que os demais planejadores.

Apêndice A

O predicado *badSituations*

No WSPBF (e também no IDDP), o número de planos a serem verificados aumenta exponencialmente conforme aumenta o tamanho dos planos. Esse aumento significa que na maior parte dos domínios, o WSPBF não consegue encontrar, em tempo hábil, um plano solução com mais de algumas dezenas de passos. Uma das razões para tal crescimento deve-se ao fato do WSPBF verificar todos os planos executáveis. Durante a construção de um plano, a escolha da próxima ação a ser adicionada ao plano leva em conta apenas se essa ação pode ser executada naquele ponto do plano, independentemente dessa ação contribuir ou não para o atingimento do estado meta. Se, por exemplo, no estado meta, $livre(a)$ tiver que ser verdadeiro, é razoável supor que uma ação como $comprar(leite)$ seja irrelevante e que não vale a pena verificar planos que contenham essa ação. Como, em muitos domínios, muitas ações são irrelevantes para se atingir o estado meta, descartar a verificação de planos com essas ações pode aumentar substancialmente o tamanho dos problemas que podem ser resolvidos pelo WSPBF.

Na versão completa do WSPBF [Reiter, 2001a], mostrada na Figura A.1, existe um mecanismo de detecção de ações irrelevantes para o atingimento do estado meta. Se compararmos a versão simplificada com a versão completa do WSPBF, veremos que a única diferença entre elas está na linha 10, com a adição de um teste $?(-badSituation)$. Intuitivamente, o predicado $badSituation(S)$ é verdadeiro em uma situação S (S é uma situação ruim) se o plano que leva de s_0 a S contêm ações irrelevantes para o atingimento do estado meta. O teste $?(-badSituation)$ da linha 10 evita

```
1  proc(wspbf(N),
2     plans(0,N)
3  ).

4  proc(plans(M,N),
5     ?(M =< N) : (actionSequence(M) : ?(goal) #
6                  pi(m1, ?(m1 is M+1) : plans(m1,N)))
7  ).

8  proc(actionSequence(N),
9    ?(N=0) #
10   ?(N>0) : pi(a, ?(primitive(a)) : a) : ?(-badSituation) :
11    pi(r, ?(r is N-1) : actionSequence(r))
12 ).
```

Figura A.1: O algoritmo completo do planejador WSPBF.

que o procedimento *actionSequence*(N) devolva um plano de tamanho N com passos irrelevantes e também evita que, na seqüência da linha 11, se tente criar planos de tamanho maior que N a partir de um plano de tamanho N com *badSituation* verdadeiro. Esse teste funciona como um corte no espaço de planos.

Vamos ilustrar o funcionamento de *badSituation* com a Figura A.2, que representa um estado inicial e um estado meta.

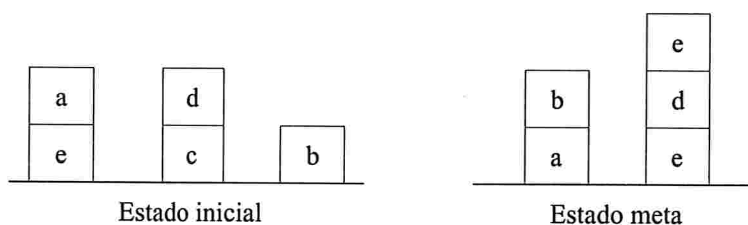

Figura A.2: Uma situação inicial e um estado meta.

Para esses estados inicial e meta, definimos *badSituation* através das três regras (poderíamos definir mais outras regras) da Figura A.3. Todas essas regras dependem de um predicado chamado *boaTorre*(X, S), que, intuitivamente, é verdadeiro na situação S se o bloco X já está em uma posição

que satisfaz as exigências do estado meta. Outro predicado importante é $existeAcaoQueCriaBoaTorre(S)$ que é verdadeiro se existe alguma ação A executável na situação S que modifica a posição de um bloco X ou Y de forma que $boaTorre(do(A, S))$ seja verdadeiro.

A regra da linha 1 determina que se $boaTorre(X, S)$ é verdadeiro, executar a ação de desempilhar X após S leva a uma situação ruim. Como o bloco X já está em sua posição final desejada, não há razão para desempilhá-lo. Nesse caso, desempilhar X irá adicionar pelo menos dois passos desnecessários, $desempilha(X, Y)$ e $empilha(X, Y)$, ao plano solução.

A regra da linha 2 determina que uma situação $do(empilha(X, Y), S)$ é ruim se a ação $empilha(X, Y)$ executada em S não leva a uma $boaTorre(X, do(empilha(X, Y), S))$. Essa regra evita que seja escolhidas ações de empilhamento que destroem ou não criam *boas torres*.

```
1  badSituation(do(desempilha(X,Y),S)) :- boaTorre(X,S).

2  badSituation(do(empilha(X,Y),S)) :- not(boaTorre(X, do(empilha(X,Y),S))).

3  badSituation(do(desempilha(X,Y),S)) :-
4      not(boaTorre(Y,do(desempilha(X,Y),S))), existeAcaoQueCriaBoaTorre(S).

5  existeAcaoQueCriaBoaTorre(S) :-
6      (A = move(X,Y,Z); A = move(Y,X,Z); A = desempilha(X,Y); A = empilha(X,Y)),
7      poss(A,S), boaTorre(X,do(A,S)).

8  boaTorre(a,S) :- sobreMesa(a,S).
9  boaTorre(b,S) :- sobre(b,a,S), livre(b,S), boaTorre(a,S).
10 boaTorre(c,S) :- sobreMesa(c,S).
11 boaTorre(d,S) :- sobre(d,c,S), boaTorre(c,S).
12 boaTorre(e,S) :- sobre(e,d,S), livre(e,S), boaTorre(d,S).
```

Figura A.3: Uma definição de *badSituation*.

A regra da linha 3 é mais sutil. Ela determina que não é uma boa idéia executar a ação $desempilha(X, Y)$, a partir de S, sabendo que isso não irá produzir uma *boa torre* e que ainda existem ações que podem produzir *boas torres*. É importante notar que essa regra não impede a execução de ações que não levam imediatamente a boas torres, mas cujos efeitos podem ser necessários para futuras ações que levam a boas torres. Como não sabemos com muita antecipação se

uma ação será ou não necessária no futuro, executá-la antes de outras ações que são sabidamente necessárias, pode potencialmente adicionar ações desnecessárias.

É importante notar três características dessas regras para *badSituation*: (i) *lookahead* de uma ação, e (ii) definições de *boaTorre* são muito dependentes do estado meta.

(i) **lookahead de uma ação.** As regras de *badSituations* da Figura A.3 enxergam apenas um passo à frente de uma situação para verificar se uma ação deve ou não ser escolhida. Esse *lookahead* não precisa se limitar a apenas um passo. Dependendo do domínio, pode ser possível estabelecer regras de *badSituations* que analisam se seqüências de dois ou mais passos levam à uma situação ruim. Entretanto, o uso de *lookaheads* de mais de um passo deve ser feito com cautela, pois quanto maior o número de passos à frente analizados, maior será o tempo de processamento necessário para avaliar as *badSituations*. Quem especifica o domínio e as *badSituations* deve saber balancear a carga extra de processamento das regras *badSituations* com os ganhos que elas proporcionam com os cortes no espaço de busca.

(ii) **definições de *boaTorre* são muito dependentes do estado meta.** A especificação de *boaTorre* é bem minuciosa, mas não é nem um pouco genérica, isto é, só funciona para o estado meta da Figura A.3. Cada diferente estado meta exige uma diferente definição de *boaTorre*. Assim, para cada diferente estado meta, é necessário que alguém analise o problema e reespecifique *boaTorre*, uma tarefa não trivial. Na prática, quem especifica *boaTorre* e *badSituation* resolve grande parte do problema de planejamento para o WSPBF.

O predicado *badSituation* permite melhorar em muito o desempenho do WSPBF, mas a característica (ii) torna o WSPBF pouco prático, pois torna-o dependente de alguém que reespecifique *boaTorre* e *badSituation* para cada novo estado meta. E especificação do predicado *badSituation* praticamente já dá a solução para o problema de planejamento às custas de muita pré-análise feita por um planejador humano. Além disso, tal especificação de *badSituation* pode ser muito complexa dependendo do problema a ser resolvido. O GGPP, por sua vez, não exige que seja feita uma pré-análise do problema, exige apenas que sejam especificados procedimentos meta facilmente obtidos das definições das ações primitivas.

Apêndice B

Implementação do meta-interpretador Golog.

```
:- dynamic(proc/2).

%===========================================================================
%
:- op(800, xfy, [\&]).   /* Conjunção */
:- op(850, xfy, [v]).   /* Disjunção */
:- op(870, xfy, [=>]).  /* Implicação */
:- op(880, xfy, [<=>]). /* Equivalência */
:- op(950, xfy, [:]).   /* seqüência */
:- op(960, xfy, [\#]).   /* escolha de ações */

%===========================================================================
trans(A,S,nil,do(AS,S)) :-  primitive(A), sub(now,S,A,AS), poss(AS,S).

trans(?(C),S,nil,S) :- holds(C,S).

trans(P1 : P2, S, P2r, Sr) :- final(P1,S), trans(P2,S,P2r,Sr).
trans(P1 : P2, S, P1r : P2, Sr) :- trans(P1,S,P1r,Sr).

trans(P1 \# P2,S,Pr,Sr) :- trans(P1,S,Pr,Sr); trans(P2,S,Pr,Sr).

trans(conc(P1,P2),S,conc(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).
trans(conc(P1,P2),S,conc(P1,P2r),Sr) :- trans(P2,S,P2r,Sr).


trans(if(C,P1,P2),S,Pr,Sr) :- holds(C,S),trans(P1,S,Pr,Sr);
```

```
                         holds(-C,S),trans(P2,S,Pr,Sr).
trans(while(C,P),S,Pr:while(C,P),Sr) :- holds(C,S), trans(P,S,Pr,Sr).

trans(pi(V,P),S,Pr,Sr) :- sub(V,\_,P,PP), trans(PP,S,Pr,Sr).

trans(N,S,Pr,Sr) :- sub(now,S,N,PArgsS), proc(PArgsS,E), trans(E,S,Pr,Sr).

%==========================================================================
final(nil,\_).
final(P1:P2,S) :- final(P1,S), final(P2,S).
final(P1\#P2,S) :- final(P1,S); final(P2,S).
final(conc(P1,P2),S) :- final(P1,S), final(P2,S).
final(pi(V,P),S) :- sub(V, \_,P,PP), final(PP,S).
final(prconc(P1,P2),S) :- final(P1,S), final(P2,S).
final(iter(P),S).
final(iterconc(P),S).
final(if(C,P1,P2),S) :- holds(C,S), final(P1,S);
                        holds(-C,S), final(P2,S).
final(while(C,P),S) :- holds(-C,S); final(P,S).
final(P\_Args,S) :- sub(now,S,P\_Args,P\_ArgsS), proc(P\_ArgsS,P), final(P,S).

%==========================================================================
trans*(P,S,P,S).
trans*(P,S,Pr,Sr) :- trans(P,S,PP,SS), trans*(PP,SS,Pr,Sr).

do(P,S,Sr) :- trans*(P,S,Pr,Sr), final(Pr,Sr).

%==========================================================================
sub(\_, \_, T1, T2) :- var(T1), T2 = T1.
sub(X1, X2, T1, T2) :- not(var(T1)), T1 = X1, T2 = X2.
sub(X1, X2, T1, T2) :- not(T1 = X1), T1 =..[F | L1],
                       sub\_list(X1, X2, L1, L2), T2 =..[F|L2].

sub\_list(\_,\_, [], []).
sub\_list(X1, X2, [T1|L1], [T2|L2]) :- sub(X1, X2, T1, T2),
sub\_list(X1, X2, L1, L2).

%==========================================================================
holds(P \& Q, S) :- holds(P, S), holds(Q, S).
holds(P v Q, S) :- holds(P, S); holds(Q, S).
holds(P => Q, S) :- holds(-P v Q, S).
holds(P <=> Q, S) :- holds((P => Q) \& (Q => P), S).
holds(-(-P), S) :- holds(P, S).
holds(-(P \& Q), S) :- holds(-P v -Q, S).
holds(-(P v Q), S) :- holds(-P \& -Q, S).
holds(-(P => Q), S) :- holds(-(-P v Q), S).
holds(-(P <=> Q), S) :- holds(-((P => Q) \& (Q => P)), S).
holds(-all(V, P), S) :- holds(some(V, -P), S).
holds(some(V, P), S) :- sub(V, \_, P, P1), holds(P1, S).
holds(-some(V, P), S) :- not(holds(some(V, P), S)).
holds(-P, S) :- isAtom(P), not(holds(P, S)).
holds(all(V, P), S) :- holds(-some(V, -P), S).
```

```
holds(some([V1|V2], P), S) :- sub*([V1 | V2], \_, P, P1), holds(P1, S).

holds(A, S) :- restoreSitArg(A, S, F), F;
               not(restoreSitArg(A, S, F)), isAtom(A), A.

isAtom(A) :- not(A = -W; A = (W1 \& W2); A = (W1 => W2); A = (W1 <=> W2);
A = (W1 v W2); A = some(X, W); A = all(X, W)).
```

Referências Bibliográficas

[Arroz, M et al., 2003] Arroz, M, Pires, V., and Custódio, L. (2003). Logic based distribution decision system for a multi-robot team. In *Actas do Encontro Cientifico do Robotica 2003 - Festival Nacional de Robotica.*

[Baral, C. and Son, T. C., 1999] Baral, C. and Son, T. C. (1999). Extending congolog to allow partial ordering. *In Proc. of the sixth International Workshop on Agent Theories, Architetures, and Languages (ATAL-99)*, 1757 of LNCS:188–204.

[Chapman, 1987] Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377.

[Colmerauer, A. et al., 1973] Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. (1973). Un système de communication homme-machine en français. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II.

[Currie, K. and Tate, A.,] Currie, K. and Tate, A. O-plan: The open planning architecture. In *Artificial Intelligence*, pages 52:49–86.

[De Giacomo, G. et al., 2000a] De Giacomo, G., Lespérance, Y., and Levesque, H. (2000a). "congolog, a concurrent programming language based on the situation calculus". *Artificial Intelligence*, 1-2(121):109–169.

[De Giacomo, G. et al., 2002] De Giacomo, G., Lespérance, Y., Levesque, H., and Sardiña, S. (April 2002). On the semantics of deliberation in indigolog - from theory to implementation. In *In .Fensel,*

F. Giunchiglia, D. McGuinness, and M. A. Williams, editors, Proceedings of Eighth International Conference in Principles of Knowledge Representation and Reasoning (KR-2002), pages 603-614, Toulouse, France.

[De Giacomo, G. and Levesque, H. J., 1999] De Giacomo, G. and Levesque, H. J. (1999). An incremental interpreter for high-level programs with sensing. *In Hector J. Levesque and Fiora Pirri, editors, Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter, pages 86-102. Springer, Berlin.*

[De Giacomo, G. et al., 2000b] De Giacomo, G., Lésperance, Y., and Levesque, H. J. (2000b). Congolog a concurrent programming language based on situation calculus: language and implementation.

[De Giacomo, G. et al., 1998] De Giacomo, G., Reiter, R., and Soutchanski, M. (1998). Execution monitoring of high-level robot programs. *In Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98), pages 453-465.*

[Elkan, 1992] Elkan, C. (1992). Reasoning about action in first-order logic. In *Proceeding of the conference of the Canadian Society for Computational Studies of Intelligence. Vancouver, Canada.*

[Erol, 1995] Erol, K. (1995). *Hierarchical Task Network Planning: Formalization, Analysis, and Implementation.* PhD thesis, University of Maryland.

[Erol, K. et al., 1995] Erol, K., Hendler, J., Nau, D. S., and Tsuneto, R. (August 1995). A critical look at critics in htn planning. In *14th International Joint Conference on Artificial Intelligence, Montreal, Canada.*

[Fikes, R. et al., 1972] Fikes, R., Hart, P., and Nilsson, N. J. (1972). *Learning and Executing Generalized Robot Plans*, volume 3, chapter 4, pages 251–288.

[Fikes, R. and Nilsson, N. J., 1971] Fikes, R. and Nilsson, N. J. (1971). *STRIPS: A New Approach to the Application of Theorem Proving to Problem solving*, volume 2, chapter 3/4, pages 189–208.

[Gabaldon, 2002] Gabaldon, A. (April, 2002). Programming hierarchical task networks in the situation calculus. *In AIPS'02 Workshop on On-line Planning and Scheduling, Toulouse, France.*

[Gallab, M et al., 2004] Gallab, M, Nau, D., and Traverso, P. (2004). *Automated Planning.* Morgan Kaufmamm.

[Green, 1969] Green, C. C. (1969). *Theorem proving by resolution as a basis for question-answering systems, vol. 4,* pages 183–205. Edinburg University Press.

[Kambhampati, 1995] Kambhampati, S. (1995). Admissible pruning strategies based on plan minimality for plan-space planning. In *IJCAI*, pages 1627–1635.

[Korf, 1987] Korf, R. (1987). Planning as search: A quantitativa approach. *Artificial Intelligence,* vol. 33:65–88.

[Levesque, H. J. et al., 1997] Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F, and Scherl, R. B. (1997). Golog: A logic programming language for dynamic domains.

[McCarthy, 1963] McCarthy, J. (1963). Situations, actions and causal laws. *Technical report, Stanford University. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, MAss., 1968, pp. 410-417.*

[McCarthy and Hayes, 1969] McCarthy, J. and Hayes, P. J. (1969). *Some Philosophical Problems from the Standpoint of Artificial Intelligence.* in Machine Intelligence 4, ed. D. Michie and B. Meltzer, EdinburghUniversity Press, pp. 463-502. (1,2,7,36,37,42,57).

[McIlraith, S. and Son, T. C., 2001] McIlraith, S. and Son, T. C. (2001). Adapting golog for composition of semantic web services. In *5th Symp. on Logic Formalizations of Commonsense Reasoning (Commonsense 2001).*

[Nau et al., 2004] Nau, D., Au, T., Ilghami, O., U.Kuter, Munoz-Avila, H., Murdock, J., Wu, D., and Yaman, F. (2004). Applications of SHOP and SHOP2. Technical Report CS-TR-4604, University of Maryland.

[Nau, D. S. et al., 2003] Nau, D. S., Au, T. -C., Ilghami, O., Kuter, U., andWu, D., M., and Yaman, F. (2003). Shop2: An htn planning system. In *JAIR, volume 20,* pages 379–404.

[Nau, D. S. et al., 2001] Nau, D. S., Cao, Y., Lotem, A., and Mitchell, S. (2001). Total-order planning with partially ordered subtasks. In *In IJCAI-2001.*

[Newell and Simon, 1961] Newell, A. and Simon, H. (1961). GPS: a program that simulates human thought. *In Lernende Automaten, 279-293, R. Oldenbourg KG. Reprinted in Feigenbaum and Feldman 1963.*

[Newell, A. et al., 1957] Newell, A., Shaw, J., and Simon, H. (1957). Programming the logic theory machine. *In Proceedings of the Western Joint Computer Conference, pages 230-240.*

[Nilsson, 1980] Nilsson, N. J. (1980). *Principles of Artificial Intelligence.* Tioga, Palo Alto.

[Parrod, Y. and Valera, S., 1993] Parrod, Y. and Valera, S. (1993). Optimum-aiv, a planning tool for spacecraft aiv. In Lifschitz, V., editor, *Preparing for the Future, Vol. 3, No. 3*, pages 7–9. European Space Agency.

[Pereira, 2002] Pereira, S. L. (2002). Planejamento abdutivo no cálculo de eventos. Master's thesis, Instituto de Matemática e Estatística - USP.

[Reiter, 1978] Reiter, R. (1978). *On Closed World Data Bases.* in Logic and Databases, ed. H. Gallaire and J. Minker, Plenum Press, pp. 55-76.

[Reiter, 1991] Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. San Diego, CA*, pages 359–380. Academic Press.

[Reiter, 1998] Reiter, R. (1998). Sequential, temporal golog. *In Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98), pages 547-556, Trento, Italy.*

[Reiter, 2001a] Reiter, R. (2001a). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press.

[Reiter, 2001b] Reiter, R. (October 2001b). On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic (TOCL), 2(4):433-457.*

[Roussel, 1975] Roussel, P. (1975). Prolog: Manuel de réference et d'utilisation. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.

[Russell, S. and Norwig, P., 1995] Russell, S. and Norwig, P. (1995). *Artificial Intelligence - A Modern Approach*, chapter 11, pages 355–356. Prentice Hall Series in Artificial Intelligence. Prentice Hall, first edition.

[Sacerdoti, 1975] Sacerdoti, E. (1975). The nonlinear nature of plans. *In Proceedings of the Fourth International Joint Conference on Artificial Intelligence.*

[Shanahan, 1997] Shanahan, M. (1997). *Solving the Frame Problem : a mathematical investigation of the common sense law of inertia.* MIT Press.

[Shapiro, S. et al., 1997] Shapiro, S., Lespérance, Y., and Levesque, H. J. (1997). Specifying communicative multi-agent systems with congolog. *In Working Notes of the AAAI Fall 1997 Symposium on Communicative Action in Humans and Machines, pages 72-82, Cambridge, MA. AAAI Press.*

[Soderland, S. and Weld, D., 1991] Soderland, S. and Weld, D. (1991). Evaluating nonlinear planning. Technical report TR-91-02-03, University of Washington Department of Computer Science and Engineering, Seattle, Washington.

[Tate, 1977] Tate, A. (1977). Generating project networks. *In Proc. IJCAI-77, pp. 888-893.*

[Vere, 1983] Vere, S. A. (1983). Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and MAchine Intteligence, PAMI-5(3):246-247.*

[Weld, 1994] Weld, D. S. (Winter 1994). An introduction to least commitement planning. volume 15(4), pages 27–61.

[Wilkins, 1988] Wilkins, D. (1988). Practical planning: Extending the classical AI planning paradigm. *Morgan Kaufmann, CA.*