Planejamento Não-Determinístico baseado em Verificação de Modelos

Silvio do Lago Pereira

Projeto de tese apresentado ao IME-USP
como requisito parcial para qualificação no curso de
Doutorado em Ciência da Computação

Orientadora: Prof^{a} Dr^{a} Leliane Nunes de Barros

8 de maio de 2007

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Planejamento automatizado

Na vida real, todos os dias, nos deparamos com a necessidade de planejar nossas ações para que nossos objetivos sejam atingidos. De fato, a habilidade de planejar é essencial ao comportamento inteligente e sua implementação é extremamente importante em aplicações práticas tais como robótica, manufatura, logística, *etc.*

Planejamento Automatizado [Ghallab et al., 2004] é um campo da Inteligência Artificial que estuda o processo deliberativo envolvido no planejamento de tarefas, sob um ponto de vista computacional, visando a implementação de planejadores. Um *planejador* é um sistema capaz de sintetizar um *plano* de ações, a partir da análise de uma descrição formal dos objetivos de um *agente* e da dinâmica de seu *ambiente*. Um plano define um padrão de comportamento para o agente: a cada instante, ele observa o ambiente e executa a ação mais apropriada para o estado corrente observado, conforme especificado no plano. Comportando-se dessa forma, o agente deve ser capaz de conduzir a evolução do ambiente, a despeito da ocorrência de eventos exógenos[1], de modo que seus objetivos possam ser atingidos. A interação entre esses componentes pode ser vista na Figura 1.1.

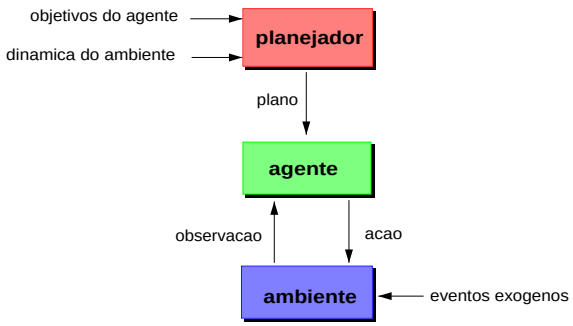


Figura 1.1: Planejamento automatizado.

[1] Eventos sobre os quais o agente não tem controle.

1.2 Planejamento clássico

Visando simplificar o problema de planejamento, a abordagem clássica [Weld, 1994] faz as seguintes suposições: o ambiente evolui deterministicamente; o agente conhece completamente o estado do ambiente; o estado do ambiente muda apenas como efeito das ações do agente; e o objetivo do agente é levar o ambiente a um estado final desejado.

Definição 1.1. *Um problema de planejamento clássico é uma tupla* $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0, S_g \rangle$, *onde:*

- $\mathcal{S} \neq \emptyset$ *é um conjunto finito de estados possíveis do ambiente;*
- $\mathcal{A} \neq \emptyset$ *é um conjunto finito de ações executáveis pelo agente;*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ *é uma função de transição de estados;*
- $s_0 \in \mathcal{S}$ *é o estado inicial do ambiente;*
- $S_g \subseteq \mathcal{S}$ *é um conjunto de estados metas.* ■

Dizemos que uma ação $a \in \mathcal{A}$ é *aplicável* num estado $s \in \mathcal{S}$ se existe um estado $s' \in \mathcal{S}$, tal que $\mathcal{T}(s, a) = s'$. O conjunto de ações aplicáveis num estado s é denotado por $\mathcal{A}(s)$.

No planejamento clássico, a dinâmica de um ambiente (*i.e., domínio de planejamento*) pode ser descrita por meio de um grafo orientado e rotulado, denominado *diagrama de transições*. Nesse diagrama, nós representam estados; arcos rotulados por ações representam transições; e caminhos representam planos. Mais formalmente, um plano para um problema de planejamento clássico $\mathcal{P}$ é uma seqüência de ações $\langle a_0, \ldots, a_{n-1} \rangle$, tal que $a_i \in \mathcal{A}(s_i)$, para $0 \leq i < n$, e $\mathcal{T}(s_{n-1}, a_{n-1}) \in S_g$.

Embora as suposições feitas na abordagem clássica realmente simplifiquem o problema de planejamento, ainda assim, planejamento clássico pode chegar a ser PSPACE-completo [Bylander, 1994]. Além disso, em várias situações práticas interessantes, nem sempre essas suposições podem ser feitas. Dependendo da dinâmica do ambiente, pode ser que:

- ocorram falhas ou eventos exógenos que tornem incertos os efeitos das ações;
- seja necessário sensoriamento para se determinar o estado corrente do ambiente;
- seja necessário satisfazer restrições durante toda a execução do plano.

1.2.1 Determinismo

Em conseqüência da suposição de determinismo, dados um estado inicial e uma seqüência de ações a serem executadas a partir desse estado, a evolução do ambiente pode ser descrita por uma única seqüência de estados completamente especificados. Entretanto, tal suposição vale somente em contextos muito restritos. Quando o ambiente é modelado de forma mais realista, a solução para um problema de planejamento deve levar em conta que as ações do agente podem ter efeitos não-determinísticos.

Há pelo menos três razões que justificam o não-determinismo observado em situações reais: a primeira delas é que a descrição da dinâmica do ambiente, geralmente incompleta, não é capaz de justificar todos os fatos observados no mundo real (*e.g.* problemas de *qualificação* e *ramificação* [Shanahan, 1997]); a segunda é que a ocorrência de eventos exógenos, fora do controle do agente, podem interferir nos efeitos de suas ações; a terceira razão é que algumas ações realmente parecem ter natureza não-determinística (*e.g.* lançar uma moeda). De qualquer forma, mesmo considerando a possibilidade de que o mundo real seja completamente determinístico, uma vez que desconhecemos *todas* as causas dos fatos observados, é razoável considerá-lo como não-determinístico.

1.2.2 Observabilidade

Num ambiente determinístico, não há necessidade de observações: a cada ação executada, há um único estado sucessor possível. Por outro lado, em ambientes não-determinísticos, mesmo conhecendo o estado corrente do ambiente, o agente não tem como prever o estado sucessor resultante da execução de uma ação. A cada ação executada, a única maneira de distinguir o estado corrente do ambiente é observá-lo. Quanto mais informações forem obtidas com essas observações, mais precisamente o agente poderá determinar o estado corrente. Se esse estado pode ser determinado sem ambigüidade, dizemos que o agente tem *observabilidade completa*; senão, dizemos que tem *observabilidade parcial*.

1.2.3 Metas

Uma *meta de planejamento* é uma especificação formal dos objetivos que o agente pretende atingir ao planejar suas ações. No planejamento clássico, as metas são *simples*, ou seja, expressam condições que devem ser satisfeitas apenas no estado final, atingido pela execução de um plano. Numa abordagem de planejamento mais abragente, entretanto, devemos considerar também a possibilidade de metas *estendidas*, *i.e.*, metas estabelecendo restrições que devem ser satisfeitas durante toda a execução do plano, e não apenas no estado final atingido por ele.

1.3 Planejamento não-determinístico

Recentemente, um grande interesse tem surgido com relação a planejamento em ambientes não-determinísticos [Dolgov and Durfee, 2004], [Bryce and Kambhampati, 2004], [Jensen et al., 2004], [Brafman and Hoffmann, 2004], [Bertoli and Pistore, 2004]. Entre as abordagens mais promissoras para esse tipo de planejamento, estão aquelas baseadas em *processos de decisão markovianos*, *satisfazibilidade* e *verificação de modelos*. As duas primeiras abordagens serão apresentadas nas seções 1.3.1 e 1.3.2; a terceira abordagem, que será adotada nesse trabalho, será apresentada no Capítulo 3.

1.3.1 Processos de decisão markovianos

Um processo *estocástico* é uma seqüência de variáveis aleatórias $\langle S_t \rangle_{t=0}^{\infty}$, representando estados, que induz uma função de transição probabilística da forma $Pr[S_{t+1} = s_{t+1} \mid S_0 = s_0, \ldots, S_t = s_t]$; ou seja, a probabilidade de transição para um estado futuro s_{t+1} depende do passado do processo $s_0, \ldots, s_t$. Um processo estocástico é *markoviano* se o estado corrente resume o passado de forma compacta, sem descartar informações necessárias para prever o estado futuro; ou seja, um processo markoviano induz uma função de transição probabilística da forma $Pr[S_{t+1} = s_{t+1} \mid S_t = s_t]$.

Um processo de decisão markoviano (PDM) modela uma interação síncrona entre um agente e um ambiente não-determinístico: a cada instante t, o agente observa o estado corrente s_t e decide executar uma ação a_t; essa ação afeta o estado corrente, produzindo um estado sucessor s_{t+1} e um ganho g_{t+1}. O objetivo do agente é maximizar seu ganho acumulado com o passar do tempo [Boutilier et al., 1999].

Definição 1.2. *Um problema de planejamento markoviano é uma tupla* $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, p, r, c \rangle$, *onde:*

- $\mathcal{S} \neq \emptyset$ *é um conjunto finito de estados possíveis do ambiente;*
- $\mathcal{A} \neq \emptyset$ *é um conjunto finito de ações executáveis pelo agente;*
- $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ *é uma função de transição de estados probabilística;*
- $r : \mathcal{S} \mapsto \mathbb{R}_+$ *é uma função que associa uma recompensa a cada estado;*
- $c : \mathcal{A} \mapsto \mathbb{R}_+$ *é uma função que associa um custo a cada ação.* ■

Na Figura 1.2, temos um modelo de um ambiente probabilístico em que a função de transição p define uma distribuição de probabilidades sobre $\mathcal{S}$; *i.e.*, $\sum_{s' \in \mathcal{S}} p(s, a, s') = 1$, onde $p(s, a, s')$ é a probabilidade de transição para o estado s', dado que a ação a foi executada no estado s. Nesse modelo, temos $r(s_5) = 1$ e $r(s_i) = 0$, para $0 \leq i \leq 4$; e $c(a) = 1$, para toda ação $a \in \mathcal{A}$. Note que, em termos de planejamento, isso equivale a definir s_5 como um estado meta; pois, como o agente visa maximizar seu ganho, ele será "naturalmente atraído" para esse estado. Estados de *absorção*, marcados com círculos e modelados pela transição reflexiva *nop*, denotam estados metas (dos quais o agente não deve sair, *e.g.* s_5) ou becos (dos quais o agente não pode sair, *e.g.* s_4).

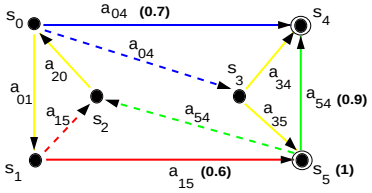


Figura 1.2: Modelo de um ambiente probabilístico.

Como as transições são probabilísticas, o estado s', resultante da execução de uma ação a num estado s, não é *previsível*, mas apenas *observável*. Assim, diferentemente do que ocorre no planejamento clássico, a solução para um problema de planejamento markoviano não é uma simples seqüência de ações, mas sim uma função $\pi : \mathcal{S} \mapsto \mathcal{A}$, que mapeia estados em ações. Essa função π, denominada *política*, descreve o comportamento do agente; especificando, para cada estado s do ambiente, que ação $\pi(s)$ deve ser executada por ele. Quando as ações especificadas pela política são sempre as mesmas, para cada estado, independentemente do instante de tempo, dizemos que a política é *estacionária*; do contrário, dizemos que é *não-estacionária*.

Avaliação de política. Para avaliarmos uma política, precisamos definir por quanto tempo ela será seguida pelo agente. Há duas possibilidades:

- *Horizonte finito:* o agente age sob a política por um número finito de passos. Nesse caso, a maneira como o agente se comporta costuma mudar, à medida em que ele se aproxima de seus últimos passos. Assim, quando o tempo de vida do agente é finito, geralmente, a política é não-estacionária. Sejam π uma política não-estacionária (que associa o estado s à ação $\pi_n(s)$, quando ainda restam n passos ao agente) e $v_n^\pi(s)$ a soma esperada dos ganhos obtidos, a partir do estado s, seguindo-se a política π por n passos. Definimos indutivamente o valor de π como:

$$v_n^\pi(s) = \begin{cases} r(s) & \text{se } n = 0 \\ r(s) - c(\pi_n(s)) + \sum_{s' \in \mathcal{S}} p(s, \pi_n(s), s') v_{n-1}^\pi(s') & \text{se } n > 0 \end{cases} \quad (1.1)$$

- *Horizonte infinito:* o agente age sob a política por um número infinito de passos. Agora, como o agente tem sempre um número infinito de passos restantes, não há porque mudar o seu modo de agir com o passar do tempo. Então, é mais razoável considerar π uma política estacionária. Além disso, para garantir que seu valor seja finito (apesar de ser dado pela soma de infinitos termos), podemos usar uma *fator de desconto* $0 < \gamma < 1$. Nesse caso, podemos definir o valor de π como:

$$v_t^\pi(s) = r(s) - c(\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s, \pi(s), s') v_{t+1}^\pi(s') \quad (1.2)$$

Obtenção de política ótima. Uma política π é *ótima* se, para toda política π' e todo estado s, temos $v^\pi(s) \geq v^{\pi'}(s)$; ou seja, uma política é ótima se maximiza a função v. Seja v^* a função valor de uma política ótima. Uma *política gulosa* com relação a v^*, denotada por π^*, é uma política ótima. Essa política é definida por:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \{r(s) - c(a) + \gamma \sum_{s' \in \mathcal{S}} p(s, a, s') v^*(s')\} \quad (1.3)$$

Uma política ótima pode ser obtida utilizando-se o método de *iteração de valor*. Esse método calcula o valor v^* de uma política ótima e, simultaneamente, constrói uma política ótima π^*. Em cada iteração n do processo, consideramos que restam apenas n passos até o final da vida do agente e escolhemos uma ação que maximiza seu ganho esperado nesse ponto. À medida em que essas ações vão sendo escolhidas, uma política ótima vai sendo construída. No caso de horizonte finito ($\gamma = 1$), as ações escolhidas em cada iteração n formam uma política ótima não-estacionária π_n^*. Já no caso de horizonte infinito ($0 < \gamma < 1$), após um número finito de iterações k, o processo converge para a função v^*. Então, as ações escolhidas na k-ésima etapa desse processo constituem uma política ótima estacionária [Boutilier et al., 1999].

Em cada iteração n, o *ganho esperado* pela execução de uma ação a num estado s, denotado por $g_n(s, a)$, é definido como:

$$g_n(s, a) = r(s) - c(a) + \gamma \sum_{s' \in \mathcal{S}} p(s, a, s') v_{n-1}(s'), \qquad \text{para } n > 0. \quad (1.4)$$

O *valor esperado* de um estado s, denotado por valor $v_n(s)$, é definido como:

$$v_n(s) = \begin{cases} r(s) & \text{se } n = 0 \\ \max_{a \in \mathcal{A}} \{g_n(s, a)\} & \text{se } n > 0 \end{cases} \quad (1.5)$$

E, finalmente, a ação $\pi_n(s)$, que maximiza $v_n(s)$, é

$$\pi_n(s) = \arg\max_{a \in \mathcal{A}} \{g_n(s, a)\}, \qquad \text{para } n > 0. \quad (1.6)$$

O método de iteração de valor é implementado pelo algoritmo IterVal, apresentado a seguir. Esse algoritmo recebe como entrada um processo de decisão markoviano $\mathcal{P}$ e devolve como saída uma política estacionária ótima π^*.

ITERVAL($\mathcal{P}$)
1 **para** $\forall s \in \mathcal{S}$ **faça**
2 $\quad v_0(s) \leftarrow r(s)$
3 $n \leftarrow 0$
4 **repita**
5 $\quad n \leftarrow n + 1$
6 $\quad$ **para** $\forall s \in \mathcal{S}$ **faça**
7 $\quad\quad$ **para** $\forall a \in \mathcal{A}$ **faça**
8 $\quad\quad\quad g_n(s,a) \leftarrow r(s) - c(a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') v_{n-1}(s')$
9 $\quad\quad v_n(s) \leftarrow \max_{a \in \mathcal{A}} \{g_n(s,a)\}$
10 $\quad\quad \pi_n(s) \leftarrow \arg\max_{a \in \mathcal{A}} \{g_n(s,a)\}$
11 **até** $|v_n(s) - v_{n-1}(s)| < \epsilon, \forall s \in \mathcal{S}$
12 devolva π_n

Por exemplo, com $\gamma = 0.5$ e $\epsilon = 10^{-3}$, ITERVAL gera a seguinte política ótima para o modelo da Figura 1.2: $\pi^* = \{(s_0, a_{01}), (s_1, a_{15}), (s_2, a_{20}), (s_3, a_{35}), (s_4, nop), (s_5, nop)\}$.

Considerações. A principal vantagem da abordagem baseada em PDM é o seu grande poder expressivo, que permite associar recompensas aos estados, custos às ações e probabilidades aos efeitos incertos. Paradoxalmente, tal expressividade é também uma fraqueza dessa abordagem; já que, em muitas situações práticas, as distribuições de probabilidades não são conhecidas. Ademais, como as políticas são funções totais de $\mathcal{S}$, essa abordagem exige o uso de técnicas sofisticadas para suportar escalabilidade [Boutilier et al., 1999].

1.3.2 satisfazibilidade

Seja Φ uma fórmula proposicional, cujas proposições são $p_1, \ldots, p_n$. Uma *valoração* de Φ é uma atribuição de valores-verdade ($\top$ ou $\bot$) às proposições $p_1, \ldots, p_n$. Dizemos que Φ é *satisfazível* se e só se existe uma valoração μ que torna Φ verdadeira. Uma tal valoração μ é denominada *modelo* de Φ.

Dada uma fórmula proposicional Φ, o *problema de satisfazibilidade* (SAT) consiste em responder à seguinte questão: Φ é satisfazível? Uma forma de responder afirmativamente a essa questão é apresentar um modelo para Φ.

Usando satisfazibilidade, podemos resolver problemas de *planejamento conformante*, *i.e.*, decidir se existe uma seqüência linear de ações que atinge um estado meta, a partir do estado inicial, a despeito do não-determinismo existente na dinâmica do ambiente.

Definição 1.3. *Um problema de planejamento conformante é uma tupla* $\mathcal{P} = \langle \mathbb{P}, \mathcal{S}, \mathcal{A}, \mathcal{L}, \mathcal{T}, s_0, S_g \rangle$*, onde:*

- $\mathbb{P} \neq \emptyset$ *é um conjunto finito de proposições que descrevem estados do ambiente;*
- $\mathcal{S} \neq \emptyset$ *é um conjunto finito de estados possíveis do ambiente;*
- $\mathcal{A} \neq \emptyset$ *é um conjunto finito de ações executáveis pelo agente;*
- $\mathcal{L} : \mathcal{S} \mapsto 2^{\mathbb{P}}$ *é uma função que associa um estado* s *a um conjunto* $\mathcal{L}(s) \in 2^{\mathbb{P}}$*;*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto 2^{\mathcal{S}}$ *é uma função de transição de estados;*
- $s_0 \in \mathcal{S}$ *é um estado inicial;*
- $S_g \subseteq \mathcal{S}$ *é um conjunto de estados metas;* ■

Seja π uma seqüência de ações $\langle a_0, \ldots, a_{n-1} \rangle$. Uma *trajetória* de π é uma seqüência $\langle s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n \rangle$, tal que $s_{i+1} \in \mathcal{T}(s_i, a_i)$, para $0 \leq i < n$. Dada uma seqüência de ações π e um problema de planejamento conformante $\mathcal{P}$, dizemos que π é um *plano conformante fraco* (*forte*) para $\mathcal{P}$ se alguma (toda) trajetória de π parte do estado inicial e atinge um estado meta de $\mathcal{P}$. Por exemplo, para o modelo apresentado na Figura 1.3, tomando s_0 como estado inicial e s_5 como estado meta, a seqüência $\langle a_{01}, a_{15} \rangle$ é um plano conformante fraco (note que não existe plano conformante forte para esse problema).

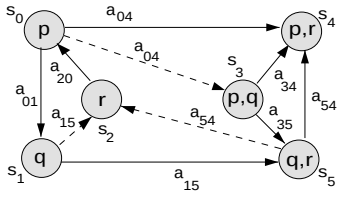


Figura 1.3: Modelo de um ambiente não-determinístico com estados simbólicos.

Dado um problema de planejamento conformante $\mathcal{P}$, a solução baseada em satisfazibilidade [Kautz and Selman, 1992] consiste em:

- codificar o problema $\mathcal{P}$ como uma fórmula proposicional $\xi(\mathcal{P})$;
- encontrar um modelo μ para $\xi(\mathcal{P})$, usando um procedimento SAT;
- e, a partir do modelo μ, extrair um plano para o problema $\mathcal{P}$.

Estados e transições. A codificação de um estado s é uma fórmula proposicional $\xi(s)$, que representa o conjunto de proposições que valem no estado s. Por exemplo, na Figura 1.3, temos um modelo onde o conjunto de proposições que descrevem os estados é $\mathbb{P} = \{p, q, r\}$. Nesse modelo, o estado s_0 pode ser codificado pela fórmula $\xi(s_0) = p \wedge \neg q \wedge \neg r$.

Um conjunto de estados $S \subseteq \mathcal{S}$ também pode ser codificado por uma fórmula proposicional. Por exemplo, se a_{04} for executada no estado s_0 (Figura 1.3), como essa ação é não-determinística, pode ocorrer uma transição para s_3 ou s_4. O conjunto de estados sucessores de s_0, via a_{04}, pode ser codificado como $\xi(\mathcal{T}(s_0, a_{04})) = (p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge r)$.

Note, porém, que as fórmulas $\xi(s_0)$ e $\xi(\mathcal{T}(s_0, a_{04}))$ não podem representar o fato de que o ambiente está evoluindo do estado s_0 para um estado $s \in \mathcal{T}(s_0, a_{04})$, via a_{04}. Para codificar uma transição, precisamos de variáveis diferentes para denotar proposições que valem antes e depois dessa transição[2]: $a_{04} \wedge (p \wedge \neg q \wedge \neg r) \wedge ((p' \wedge q' \wedge \neg r') \vee (p' \wedge \neg q' \wedge r'))$.

Codificação do problema. A codificação proposicional de um problema de planejamento $\mathcal{P}$ é baseada em duas idéias principais:

- restringir o problema de planejamento a encontrar um plano de tamanho n, para algum n fixo (iniciamos com $n = 0$ e o incrementamos a cada passo);
- transformar o problema de planejamento restrito num problema de satisfatibilidade (cada proposição p dá origem às proposições $p^0, \ldots, p^n$, necessárias para descrever estados e transições em cada passo de planejamento).

[2] As fórmulas proposicionais para transições são definidas sobre o conjunto $\mathbb{P} \cup \mathcal{A}$.

Um estado s, num passo i, é codificado como:

$$\xi^i(s) = \left(\bigwedge_{p \in \mathcal{L}(s)} p^i \right) \wedge \left(\bigwedge_{p \notin \mathcal{L}(s)} \neg p^i \right) \tag{1.7}$$

O conjunto de estados iniciais é codificado como:

$$\xi(S_0) = \bigvee_{s \in S_0} \xi^0(s) \tag{1.8}$$

O conjunto de estados metas é codificado como:

$$\xi(S_g) = \bigvee_{s \in S_0} \xi^n(s) \tag{1.9}$$

Um transição da forma $s \xrightarrow{a} s'$ é codificada como:

$$\xi(a) = a^i \wedge \xi^i(s) \wedge \xi^{i+1}(s') \tag{1.10}$$

Sendo a representação de estados *completa*, não há necessidade de axiomas de persistência temporal. Entretanto, precisamos de *axiomas de exclusão*, estabelecendo que apenas uma ação pode ser escolhida em cada passo de planejamento: para cada duas ações distintas a_j e a_k, devemos ter um axioma da forma $\neg a_j^i \vee \neg a_k^i$, para $0 \leq i < n$.

A codificação de um problema de planejamento $\mathcal{P}$, restrito para n passos, é dada pela conjunção das fórmulas (1.7) a (1.10), denotada por $\xi^n(\mathcal{P})$.

Busca de um modelo. Seja Φ a fórmula resultante da conversão de $\xi^n(\mathcal{P})$ para a forma normal conjuntiva. Usando o algoritmo DavisPutnam, podemos construir um modelo para Φ. Esse algoritmo, correto e completo, realiza uma busca em profundidade no espaço de valorações de Φ, até encontrar um modelo para Φ ou até concluir que não existe tal modelo.

```
DavisPutnam(Φ, μ)
1 se ∅ ∈ Φ então devolva ∅
2 se ∅ = Φ então devolva μ
3 PropagaçãoUnitária(Φ, μ)
4 selecione uma variável p ocorrendo em Φ
5 μ' ← DavisPutnam(Φ ∪ {p}, μ)
6 se μ' = ∅ então μ' ← DavisPutnam(Φ ∪ {¬p}, μ)
7 devolva μ'
```

O procedimento PropagaçãoUnitária simplifica a fórmula Φ, definindo $\mu(p) = \top$ para cada cláusula unitária $\{p\} \in \Phi$. (Analogamente, para cada cláusula unitária $\{\neg p\} \in \Phi$, o procedimento define $\mu(p) = \bot$).

```
PropagaçãoUnitária(Φ, μ)
1 enquanto existir uma cláusula unitária λ ∈ Φ faça
2     μ ← μ ∪ {λ}
3     para cada cláusula C ∈ Φ faça
4       se λ ∈ C então Φ ← Φ − {C}
5       senão se ¬λ ∈ C então Φ ← Φ − {C} ∪ {C − {¬λ}}
```

Extração do plano. Dado um modelo μ para a fórmula $\xi^n(\mathcal{P})$, um plano para o problema de planejamento conformante $\mathcal{P}$ pode ser obtido da seguinte forma:

- *Caso determinístico:* Tomamos uma seqüência de variáveis proposicionais $\langle a_0^0, \ldots, a_{n-1}^{n-1} \rangle$, onde a_j^i denota a ação a_j executada no passo i, tal que $\mu(a_j^i) = \top$, para $0 \leq i < n$. O plano extraído é justamente a seqüência de ações $\langle a_0, \ldots, a_{n-1} \rangle$.
- *Caso não-determinístico:* Claramente, qualquer plano extraído conforme descrito no caso acima é um plano conformante fraco para problemas não-determinísticos. Para encontrar um plano conformante forte para $\mathcal{P}$, basta gerar planos fracos e verificar se um delas é forte. Um algoritmo trivial para essa validação consiste em simular a execução de um plano fraco π, verificando se, para cada par de ações consecutivas a_i e a_{i+1} em π, a ação a_{i+1} é executável em todo estado $s \in \mathcal{T}(s_i, a_i)$. Um algoritmo mais sofisticado para esse problema pode ser encontrado em [Ghallab et al., 2004].

Considerações. A principal vantagem da abordagem baseada em SAT é que ela permite resolver problemas de planejamento em ambientes não-determinísticos com observabilidade nula (= planejamento conformante). Esse tipo de planejamento é muito importante em aplicações em que o agente não dispõe de sensores (ou em que o custo de sensoriar o ambiente é extremamente alto). A desvantagem dessa abordagem é que, para muitos problemas, em diversos domínios, não existem planos conformantes fortes e planos conformantes fracos não são aceitáveis.

1.4 Organização

Nesse capítulo, introduzimos *planejamento determinístico* e ressaltamos sua inadequação a aplicações realmente práticas de planejamento. Em seguida, apresentamos duas abordagens para *planejamento não-determinístico*: a primeira delas (PDM) exige conhecimento sobre as probabilidades das transições, informação que nem sempre está disponível; e a segunda (SAT), só extrai planos conformantes, que nem sempre existem.

No capítulo 2, introduzimos os principais conceitos relativos à *verificação de modelos*. Com base nesses conceitos, no Capítulo 3, apresentamos uma terceira abordagem para planejamento não-determinístico: *planejamento baseado em verificação de modelos*.

Finalmente, no Capítulo 4, apresentamos a proposta de pesquisa para a tese.

Capítulo 2

Verificação de Modelos

2.1 Modelo computacional

Verificação de modelos (VM) é uma técnica para validação automática de sistemas. Formalmente, verificação de modelos consiste em resolver o problema $\mathcal{K} \models \varphi$, onde $\mathcal{K}$ é um modelo formal do sistema e φ é uma descrição da propriedade a ser verificada.

Essencialmente, um *verificador* (Figura 2.1) é um algoritmo que recebe como entrada um par $\langle \mathcal{K}, \varphi \rangle$ e percorre os estados do modelo $\mathcal{K}$, verificando a validade da propriedade φ. Se todos os estados percorridos satisfazem a propriedade, o verificador devolve *sucesso*; senão, ele devolve um *contra-exemplo* (*e.g.*, estado onde a propriedade φ é violada).

Figura 2.1: Verificador de modelos.

2.1.1 Estruturas de Kripke

O comportamento de um sistema pode ser representado por um diagrama de transições (cujos arcos são rotulados por ações). Em VM, porém, é comum abstrair a identidade das transições, sendo relevantes apenas as propriedades que são satisfeitas nos estados representados pelos nós do diagrama. Por esse motivo, o modelo formal de um sistema em VM é representado por uma estrutura de Kripke (cujos nós são rotulados por proposições).

Definição 2.1. *Seja $\mathbb{P} \neq \emptyset$ um conjunto finito de proposições atômicas. Uma estrutura de Kripke sobre $\mathbb{P}$ é uma tupla $\mathcal{K} = \langle \mathcal{S}, \mathcal{T}, \mathcal{L} \rangle$, onde:*

- *$\mathcal{S} \neq \emptyset$ é um conjunto finito de estados;*
- *$\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ é uma relação de acessibilidade (ou transição);*
- *$\mathcal{L} : \mathcal{S} \mapsto 2^{\mathbb{P}}$ é uma função de interpretação de estados.* ■

Intuitivamente, as proposições em $\mathbb{P}$ representam propriedades de estados e $\mathcal{L}(s)$ denota o conjunto de proposições que descrevem um estado s. Ademais, assumimos que $\mathbb{P}$ sempre contém as proposições $\top$ e $\bot$ e que, para todo $s \in \mathcal{S}$, temos $\top \in \mathcal{L}(s)$ e $\bot \notin \mathcal{L}(s)$.

Seja $\mathcal{K}$ uma estrutura de Kripke. Dizemos que $\mathcal{K}$ é *total* se, para todo $s \in \mathcal{S}$, existe um $s' \in \mathcal{S}$ tal que $(s, s') \in \mathcal{T}$; caso contrário, dizemos que é *parcial*.

2.1.2 Lógica modal proposicional

Estruturas de Kripke foram originalmente propostas como um modelo semântico para lógicas modais, cujas fórmulas utilizam a modalidade $\Box$ para expressar *necessidade* (e a modalidade dual $\Diamond$ para expressar *possibilidade*). Nesse contexto, os estados da estrutura de Kripke correspondem a "mundos possíveis", em que diferentes proposições são verdadeiras, e as transições correspondem a acessos que possibilitam transitar entre mundos. Assim, dizer que um fato é *necessário* num determinado mundo s significa que tal fato é observado em todos os mundos acessíveis a partir de s. Por outro lado, dizer que um fato é *possível* num determinado mundo s significa que, partindo de s, existe um acesso que leva a um mundo onde esse fato é observado.

A linguagem da lógica modal proposicional é constituída de um conjunto de proposições atômicas $\mathbb{P} \supset \{\top, \bot\}$, o conectivo $\rightarrow$ (ou qualquer outro conjunto funcionalmente completo de conectivos proposicionais clássicos) e da modalidade $\Box$. O conjunto de fórmulas da lógica modal proposicional, denotado por $\mathfrak{L}_m$, é definido indutivamente como:

$\varphi \doteq p \in \mathbb{P} \mid \varphi_1 \rightarrow \varphi_2 \mid \Box\varphi$

Os demais conectivos da lógica proposicional e a modalidade dual são definidos como:

- $\neg\varphi \doteq \varphi \rightarrow \bot$
- $\varphi_1 \wedge \varphi_2 \doteq (\varphi_1 \rightarrow (\varphi_2 \rightarrow \bot)) \rightarrow \bot$
- $\varphi_1 \vee \varphi_2 \doteq (\varphi_1 \rightarrow \bot) \rightarrow \varphi_2$
- $\Diamond\varphi \doteq \neg\Box\neg\varphi$

Definição 2.2. *Um modelo de Kripke é um par $\langle \mathcal{K}, \models \rangle$, onde $\mathcal{K} = \langle \mathcal{S}, \mathcal{T}, \mathcal{L} \rangle$ é uma estrutura de Kripke e $\models$ é uma relação entre um estado $s \in \mathcal{S}$ e uma fórmula modal $\varphi \in \mathfrak{L}_m$, tal que:*

- $(\mathcal{K}, s) \models p$ *se e só* $p \in \mathcal{L}(s)$*;*
- $(\mathcal{K}, s) \models \varphi_1 \rightarrow \varphi_2$ *se e só se* $(\mathcal{K}, s) \not\models \varphi_1$ *ou* $(\mathcal{K}, s) \models \varphi_2$*;*
- $(\mathcal{K}, s) \models \Box\varphi$ *se e só se* $(\mathcal{K}, s') \models \varphi$ *para todo* $s' \in \mathcal{S}$ *tal que* $(s, s') \in \mathcal{T}$*.* ■

A relação $\models$ é denominada *relação de satisfação* e a notação $(\mathcal{K}, s) \models \varphi$ significa que o estado s, da estrutura de Kripke $\mathcal{K}$, satisfaz a fórmula φ. Quando a estrutura $\mathcal{K}$ está clara no contexto, podemos escrever apenas $s \models \varphi$.

2.1.3 Árvores de computação

Quando um estado particular numa estrutura de Kripke é designado como *estado inicial*, a estrutura pode ser desdobrada numa árvore infinita, enraizada nesse estado. Caso a relação de transição seja parcial, consideramos que cada estado terminal tem uma transição reflexiva, conforme ilustrado na Figura 2.2. A justificativa para essas transições reflexivas é que a árvore infinita obtida é um *modelo temporal* para a estrutura de Kripke desdobrada e estados terminais nessa estrutura devem persistir infinitamente no tempo.

A árvore resultante do desdobramento de uma estrutura de Kripke representa todos os caminhos possíveis de computação do sistema modelado e, por esse motivo, tal árvore é também denominada *árvore de computação*.

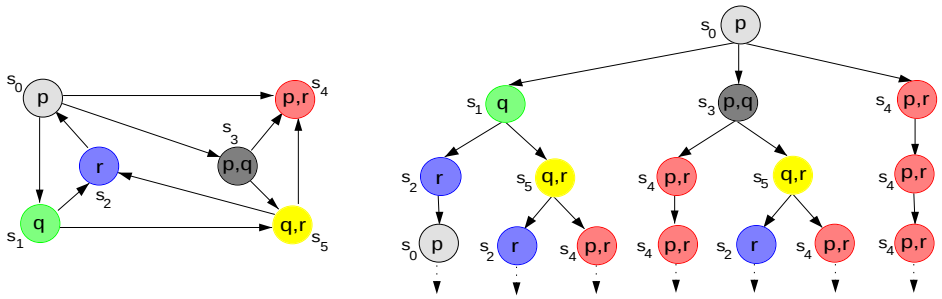


Figura 2.2: Estrutura de Kripke e árvore de computação correspondente.

A árvore de computação obtida pelo desdobramento de uma estrutura de Kripke $\mathcal{K}$, usando um estado s como raiz, é denotada por $\Upsilon^s_{\mathcal{K}}$. Também usamos $\Upsilon^s_{\mathcal{K}}$ para denotar o conjunto de todos os caminhos que partem de um estado s numa estrutura de Kripke $\mathcal{K}$.

Um *caminho* numa árvore de computação $\Upsilon^{s_0}_{\mathcal{K}}$ é uma seqüência infinita de estados $\pi = \langle s_0, s_1, \ldots\rangle \in \mathcal{S}^\omega$, tal que cada par de estados consecutivos $(s_i, s_{i+1}) \in \pi$ é um elemento da relação de transição $\mathcal{T}$. Dado um caminho $\pi = \langle s_0, s_1, \ldots\rangle$, usamos π_i para denotar um estado s_i de π e $\pi_{[i\ldots]}$ para denotar um sufixo $\langle s_i, s_{i+1}, \ldots\rangle$ de π.

2.2 Lógicas temporais

Em geral, para especificar as propriedades a serem verificadas, VM utiliza lógicas temporais, *i.e.*, lógicas modais cujas modalidades expressam aspectos temporais. Nessas lógicas, os estados na árvore de computação (modelo temporal do sistema) representam instantes no tempo e o valor de uma proposição depende do instante de tempo considerado.

Há basicamente dois tipos de lógicas temporais: aquelas de tempo linear e aquelas de tempo ramificado. Essencialmente, o que as distingue é modo como elas lidam com as ramificações na árvore de computação: enquanto as lógicas de tempo linear consideram que cada instante no tempo tem um único futuro possível (semântica baseada em caminhos), as lógicas de tempo ramificado consideram que, para cada instante no tempo, pode haver vários futuros possíveis (semântica baseada em estados).

2.2.1 A lógica de tempo linear LTL

LTL (*Linear Time Logic*) [Pnueli, 1977] é uma lógica proposicional de tempo linear, cujas fórmulas permitem especificar propriedades sobre caminhos numa árvore de computação. As fórmulas LTL são construídas a partir dos seguintes operadores temporais:

- $\odot$ (no *próximo* estado futuro)
- $\Box$ (*invariantemente*, em todos os estados futuros)
- $\Diamond$ (*finalmente*, em algum estado futuro)
- $\cup$ (*até* em algum estado futuro)

As fórmulas LTL são sempre avaliadas com relação a um caminho particular π numa árvore de computação: a fórmula $\odot\varphi$ requer que φ seja satisfeita no estado π_1; a fórmula

$\Box\varphi$ requer que φ seja satisfeita em todos os estados de π; a fórmula $\Diamond\varphi$ requer que a propriedade seja satisfeita em algum estado π_i e; finalmente, a fórmula $\varphi_1 \cup \varphi_2$ requer que φ_2 seja satisfeita em algum estado π_j e que φ_1 seja satisfeita em todo estado π_i tal que $i < j$. A modalidade $\Box$ funciona como um quantificador universal sobre estados de um caminho, enquanto $\Diamond$ funciona como uma espécie de quantificador existencial.

O conjunto de fórmulas LTL é definido indutivamente como:

$\varphi \doteq p \in \mathbb{P} \mid \varphi_1 \rightarrow \varphi_2 \mid \odot\varphi \mid \varphi_1 \cup \varphi_2$

Os demais conectivos lógicos são definidos em termos de $\bot$ e $\rightarrow$, da maneira usual (seção 2.1.2). Os demais operadores temporais são definidos como:

- $\Diamond\varphi \doteq \top \cup \varphi$
- $\Box\varphi \doteq \neg\Diamond\neg\varphi$

Definição 2.3. *Sejam $\mathcal{K}$ uma estrutura de Kripke, s um estado dessa estrutura e φ uma fórmula* LTL. *A semântica das fórmulas* LTL *é definida como segue:*

- $s \models \varphi$ *se e só se* $\pi \models \varphi$, *para todo* $\pi \in \Upsilon^s_{\mathcal{K}}$;
- $\pi \models p$ *se e só se* $p \in \mathcal{L}(\pi_0)$;
- $\pi \models \varphi_1 \rightarrow \varphi_2$ *se e só se* $\pi \not\models \varphi_1$ *ou* $\pi \models \varphi_2$;
- $\pi \models \odot\varphi$ *se e só se* $\pi_1 \models \varphi$;
- $\pi \models \varphi_1 \cup \varphi_2$ *se e só se existe* $j \geq 0$ *tal que* $\pi_j \models \varphi_2$ *e, para todo* $i < j$, $\pi_i \models \varphi_1$. ■

2.2.2 A lógica de tempo ramificado CTL

CTL (*Computation Tree Logic*) [Clarke and Emerson, 1982] é uma lógica proposicional de tempo ramificado, cujas fórmulas permitem especificar propriedades quantificadas sobre caminhos numa árvore de computação. Na lógica CTL, os operadores temporais da lógica LTL devem ser imediatamente precedidos por um quantificador de caminhos ($\forall$ ou $\exists$).

Os operadores temporais em CTL são os seguintes:

- $\forall\odot$ (em *todos* os *próximos* estados)
- $\forall\Box$ (em *todos* os estados futuros, *invariantemente*)
- $\forall\Diamond$ (em *todos* os estados futuros, *finalmente*)
- $\forall\cup$ (em *todos* os estados futuros, *até*)
- $\exists\odot$ (em *algum* dos *próximos* estados)
- $\exists\Box$ (em *algum* dos estados futuros, *invariantemente*)
- $\exists\Diamond$ (em *algum* dos estados futuros, *finalmente*)
- $\exists\cup$ (em *algum* dos estados futuros, *até*)

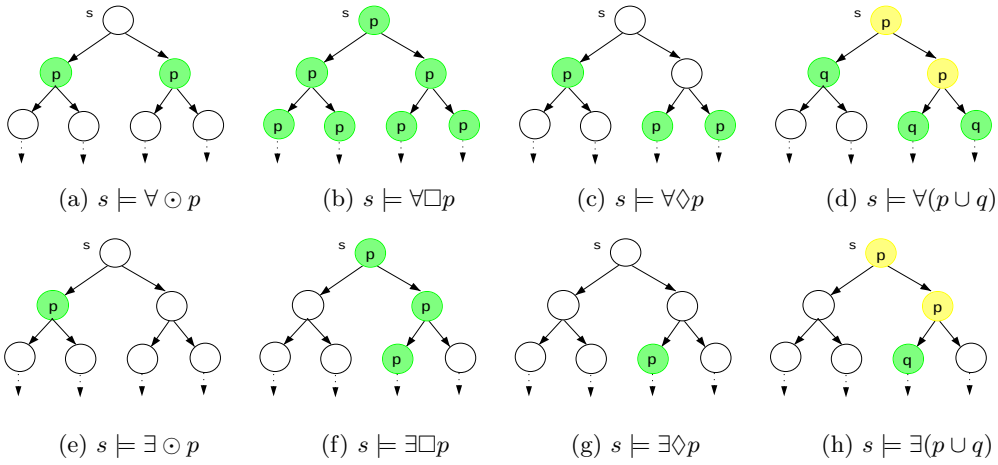
O conjunto de fórmulas CTL é definido indutivamente como:

$\varphi \doteq p \in \mathbb{P} \mid \varphi_1 \rightarrow \varphi_2 \mid \forall \odot \varphi \mid \forall(\varphi_1 \cup \varphi_2) \mid \exists(\varphi_1 \cup \varphi_2)$

Os demais conectivos lógicos são definidos em termos de $\bot$ e $\rightarrow$, da maneira usual (seção 2.1.2). Os demais operadores temporais são definidos conforme a seguir[1]:

- $\forall\Box\varphi \doteq \neg\exists(\top \cup \neg\varphi)$
- $\forall\Diamond\varphi \doteq \forall(\top \cup \varphi)$

[1] Note que $\forall\cup$ e $\exists\cup$ não são interdefiníveis.



(a) $s \models \forall \odot p$ (b) $s \models \forall \Box p$ (c) $s \models \forall \Diamond p$ (d) $s \models \forall (p \cup q)$

(e) $s \models \exists \odot p$ (f) $s \models \exists \Box p$ (g) $s \models \exists \Diamond p$ (h) $s \models \exists (p \cup q)$

Figura 2.3: Semântica dos operadores temporais em CTL.

- $\exists \odot \varphi \doteq \neg \forall \odot \neg \varphi$
- $\exists \Box \varphi \doteq \neg \forall (\top \cup \neg \varphi)$
- $\exists \Diamond \varphi \doteq \exists (\top \cup \varphi)$

Definição 2.4. *Sejam $\mathcal{K}$ uma estrutura de Kripke, s um estado dessa estrutura e φ uma fórmula* CTL. *A semântica das fórmulas* CTL *é definida como segue:*

- $s \models p$ *se e só se* $p \in \mathcal{L}(s)$;
- $s \models \varphi_1 \rightarrow \varphi_2$ *se e só se* $s \not\models \varphi_1$ *ou* $s \models \varphi_2$;
- $s \models \forall \odot \varphi$ *se e só se, para todo caminho* $\pi \in \Upsilon^s_{\mathcal{K}}$, $\pi_1 \models \varphi$;
- $s \models \forall (\varphi_1 \cup \varphi_2)$ *se e só se para todo caminho* $\pi \in \Upsilon^s_{\mathcal{K}}$, *existe* $j \geq 0$ *tal que* $\pi_j \models \varphi_2$ *e, para todo* $i < j$, $\pi_i \models \varphi_1$;
- $s \models \exists (\varphi_1 \cup \varphi_2)$ *se e só se para algum caminho* $\pi \in \Upsilon^s_{\mathcal{K}}$, *existe* $j \geq 0$ *tal que* $\pi_j \models \varphi_2$ *e, para todo* $i < j$, $\pi_i \models \varphi_1$. ■

2.2.3 Considerações sobre as lógicas LTL e CTL

Geralmente, lógicas são comparadas sob dois aspectos: expressividade e complexidade.

Expressividade. Com relação à expressividade, as lógicas LTL e CTL são incomparáveis[2] [Emerson and Halpern, 1986], [Vardi, 1998b].

Devido à forma como as semânticas dessas lógicas são definidas, CTL é capaz de distinguir situações que são consideradas idênticas em LTL. Por exemplo, considere os sistemas de venda automática apresentados na Figura 2.4: no primeiro, a inserção de uma moeda causa uma transição para um estado onde pode-se escolher entre café ou chá; no segundo, a inserção de uma moeda causa uma transição não-determinística para um estado a partir

[2]Essas lógicas são fragmentos de uma lógica mais expressiva (CTL* [Emerson and Halpern, 1986]), que combina tempo linear e ramificado num mesmo formalismo.

do qual não há escolha. Embora esses sistemas sejam realmente distintos, a semântica baseada em caminhos da lógica LTL não é capaz de distingüi-los; já que todos os caminhos que iniciam em s_0 são iguais, em ambos os sistemas.

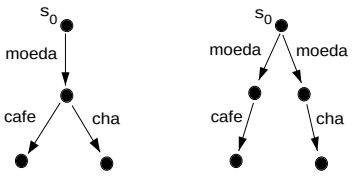


Figura 2.4: Sistemas distintos em CTL, mas idênticos em LTL.

Ademais, devido à ausência de quantificadores em LTL, não há como especificar propriedades existenciais nessa lógica. Porém, isso não significa que LTL seja menos expressiva que CTL. Também há propriedades que podem ser especificadas em LTL, mas não em CTL. Por exemplo, a propriedade "*em todo caminho, finalmente p torna-se invariante*" pode ser escrita em LTL como $\Diamond\Box p$. Em CTL, devido à restrição imposta na sintaxe, essa fórmula teria que ser escrita como $\forall\Diamond\forall\Box p$, que não expressa a propriedade desejada. De fato, essa fórmula CTL especifica que "*em todo caminho, finalmente, há um estado a partir do qual, em todos os caminhos, a propriedade p é invariante*". Na Figura 2.5, temos o modelo de um sistema que apresenta (e satisfaz) a propriedade especificada pela fórmula LTL, mas que não pode satisfazer a fórmula CTL: quando os caminhos são avaliados independentemente (semântica LTL), em todos eles há um ponto onde finalmente p torna-se invariante; por outro lado, quando os caminhos são avaliados simultaneamente (semântica CTL), como o estado s_0 sempre tem um estado sucessor onde p é falso, não existe um ponto a partir do qual p torna-se invariante em todos os caminhos.

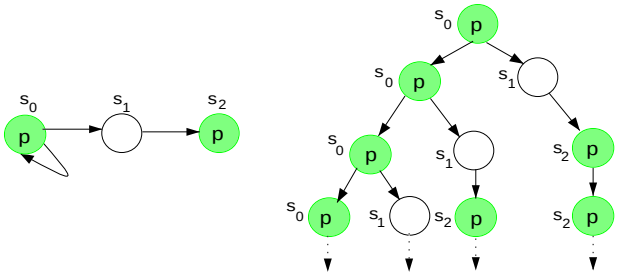


Figura 2.5: Modelo de sistema com uma propriedade que não pode ser expressa em CTL.

Complexidade. Devido à complexidade relativa dos algoritmos, verificação de modelos baseada em CTL é muito mais comum que verificação baseada em LTL.

Considere um modelo de tamanho m e uma fórmula temporal de tamanho n. Enquanto a complexidade de tempo dos algoritmos de verificação baseados em CTL é $O(m \times n)$, a complexidade dos algoritmos baseados em LTL é $m \times 2^{O(n)}$. Ademais, como os algoritmos para LTL são PSPACE-completos, é improvável que esse limite superior de tempo possa ser reduzido [Vardi, 1998a], [Schnoebelen, 2002].

Essa comparação tradicional entre LTL e CTL é feita em termos de complexidade de pior caso. Porém, como essas lógicas têm expressividades incomparáveis, uma comparação mais significativa deveria levar em conta apenas o fragmento comum dessas lógicas, *i.e.*, o conjunto de propriedades que podem ser representadas tanto em LTL quanto em CTL.

2.3 Verificação de modelos usando a lógica CTL

Nas duas seções anteriores, apresentamos as ferramentas utilizadas em VM para representar o modelo do sistema e as propriedades a serem verificadas. Nessa seção, apresentamos os algoritmos que efetivamente percorrem o modelo, verificando as propriedades.

2.3.1 Caracterização de ponto fixo para operadores CTL

A caracterização de ponto fixo para os operadores CTL [Clarke and Emerson, 1982], permite a criação de um algoritmo padrão para o problema de verificação de modelos, que associa a cada fórmula φ o conjunto de estados de $\mathcal{K}$ que a satisfazem.

Sejam $\mathcal{K} = \langle \mathcal{S}, \mathcal{T}, \mathcal{L} \rangle$ uma estrutura de Kripke e $2^{\mathcal{S}}$ o conjunto potência de $\mathcal{S}$. Então, $2^{\mathcal{S}}$ forma um reticulado sob união e intersecção. Esse reticulado é ordenado por inclusão de conjuntos, onde $A \subseteq B$ se e só se $A \cup B = B$. Um *funcional* $\tau[Y]$ é uma fórmula com um símbolo proposicional Y não-interpretado. Isso define uma função $\tau : 2^{\mathcal{S}} \mapsto 2^{\mathcal{S}}$, onde $\tau(A)$ é obtido substituindo-se Y por A em τ. Por definição:

- τ é *monotônica* quando $A \subseteq B$ implica $\tau(A) \subseteq \tau(A)$.
- τ é $\cup$-*contínua* quando $A_1 \subseteq A_2 \subseteq \ldots$ implica $\tau(\cup_i A_i) = \cup_i \tau(A_i)$.
- τ é $\cap$-*contínua* quando $A_1 \supseteq A_2 \supseteq \ldots$ implica $\tau(\cap_i A_i) = \cap_i \tau(A_i)$.

Quando o conjunto $\mathcal{S}$ é finito, toda seqüência crescente de conjuntos tem um elemento máximo e toda seqüência decrescente de conjuntos tem um elemento mínimo. Assim, no caso finito, monotonicidade implica $\cup$-continuidade e $\cap$-continuidade.

Um *ponto fixo* de τ é qualquer A tal que $\tau(A) = A$. Conforme [Tarski, 1955], um funcional monotônico, com relação à ordem de inclusão, sempre tem um ponto fixo mínimo e um ponto fixo máximo.

Teorema 2.1. *(Tarski-Knaster) Se $\tau[Y]$ é monotônica, ela tem um ponto fixo mínimo $\mu Y.\tau[Y]$ e um ponto fixo máximo $\nu Y.\tau[Y]$. Se $\tau[Y]$ é também $\cup$-contínua, $\mu Y.\tau[Y] = \cup_{i \geq 0} \tau^i(\bot)$. Se $\tau[Y]$ é também $\cap$-contínua, $\nu Y.\tau[Y] = \cap_{i \geq 0} \tau^i(\top)$.* ■

Teorema 2.2. *(Clarke-Emerson) Se $\mathcal{S}$ é finito, operadores CTL são caracterizados por:*

- $\exists \Diamond \varphi = \mu Y.(\varphi \vee \exists \odot Y)$
- $\exists \Box \varphi = \nu Y.(\varphi \wedge \exists \odot Y)$
- $\exists(\varphi_1 \cup \varphi_2) = \mu Y.(\varphi_2 \vee (\varphi_1 \wedge \exists \odot Y))$ ■

2.3.2 Algoritmo padrão para verificação de fórmulas CTL

O algoritmo padrão para encontrar um ponto fixo mínimo (máximo) para um funcional $\tau[Y]$ consiste em iniciar com $\bot$ ($\top$) e iterar esse funcional até que um ponto fixo seja atingido. Para $\mathcal{S}$ finito, esse algoritmo termina em no máximo $|\mathcal{S}| + 1$ iterações.

Na Figura 2.6, podemos ver o funcionamento do algoritmo padrão para computar o conjunto de estados que satisfazem a fórmula $\exists\Diamond p$, usando o funcional $\tau[Y] = p \vee \exists \odot Y$. Na primeira iteração, temos $\tau^1[\bot] = p \vee \exists \odot \bot = \{s_1\}$; na segunda iteração, temos $\tau^2[\bot] = p \vee \exists \odot p = \{s_0, s_1\}$; e, finalmente, na terceira iteração, temos $\tau^3[\bot] = p \vee \exists \odot (p \vee \exists \odot p) = \{s_0, s_1, s_2\}$. Note que , $\tau^3[\bot]$ já é o ponto fixo mínimo pois, na quarta iteração, $\tau^4[\bot]$ produz o mesmo resultado que $\tau^3[\bot]$.

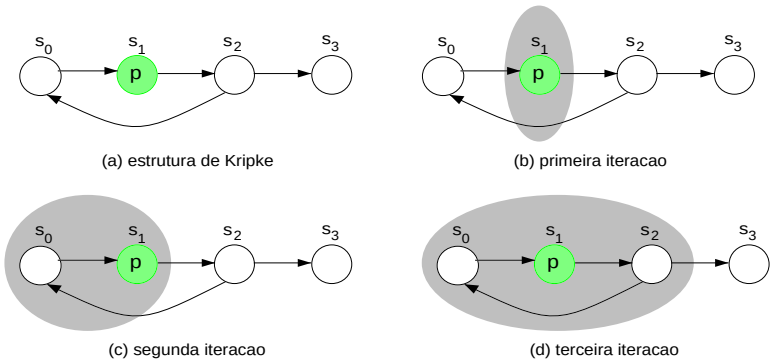


Figura 2.6: Funcionamento do algoritmo de ponto fixo mínimo para $\exists\Diamond p$.

Na Figura 2.7, usamos o funcional $\tau[Y] = p \ \wedge \exists \odot Y$ para computar o conjunto de estados que satisfazem a fórmula $\exists\Box p$. Na primeira iteração, temos $\tau^1[\top] = p \wedge \exists \odot \top = \{s_0, s_1, s_2\}$; na segunda iteração, temos $\tau^2[\top] = p \wedge \exists \odot p = \{s_0, s_1\}$; e, finalmente, na terceira iteração, temos $\tau^3[\top] = p \wedge \exists \odot (p \wedge \exists \odot p) = \{s_0\}$, que já é o ponto fixo máximo.

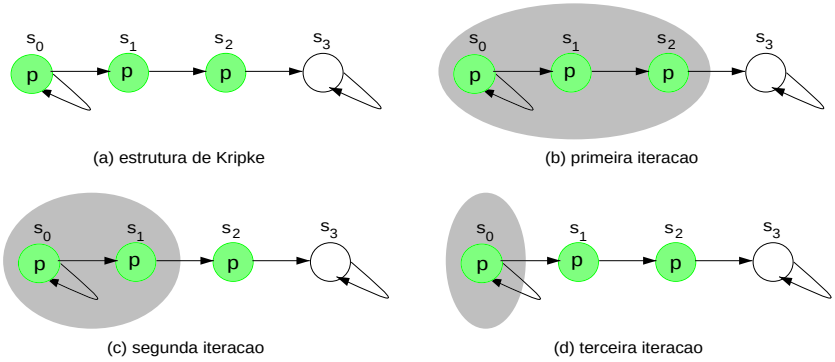


Figura 2.7: Funcionamento do algoritmo de ponto fixo máximo para $\exists\Box p$.

A seguir, apresentamos uma versão básica do algoritmo VM, que verifica modelos com relação a propriedades especificadas em CTL. Uma versão mais completa pode ser obtida simplesmente utilizando-se das definições dos demais operadores temporais de CTL.

```
VM(K, φ)
1 se φ ∈ P então devolva {s : p ∈ L(s)}
2 se φ = φ1 → φ2 então devolva (S−VM(K, φ1)) ∪ VM(K, φ2)
3 se φ = ∀ ⊙ φ então devolva {s : ∅ ≠ T(s) ⊆ VM(K, φ)}
4 se φ = ∀(φ1 ∪ φ2) então devolva VM∀∪(K, φ1, φ2)
5 se φ = ∃(φ1 ∪ φ2) então devolva VM∃∪(K, φ1, φ2)
```

```
VM∀∪(K, φ1, φ2)
1 A ← VM(φ2)
2 B ← ∅
3 enquanto A ≠ B faça
4    B ← A
5    A ← A ∪ ({s : ∅ ≠ T(s) ⊆ A} ∩ VM(φ1))
6 devolva A
```

```
VM∃∪(K, φ1, φ2)
1 A ← VM(φ2)
2 B ← ∅
3 enquanto A ≠ B faça
4    B ← A
5    A ← A ∪ ({s : T(s) ∩ A ≠ ∅} ∩ VM(φ1))
6 devolva A
```

Claramente, a caracterização de ponto fixo permite a criação de um algoritmo efetivo para o problema de verificação de modelos, baseado em busca regressiva e enumeração explícita de estados. De fato, existe um algoritmo ainda mais eficiente, baseado em busca em largura e identificação de componentes fortemente conexos no modelo de Kripke. Entretanto, ambos os algoritmos sofrem do problema de explosão de estados; já que ambos precisam construir completamente o grafo que representa o modelo do sistema, para que os conjuntos de estados possam ser computados.

2.4 Verificação de modelos simbólicos

Verificação de modelos simbólicos é uma abordagem utilizada para controlar o problema da explosão de estados nos sistemas verificados. Nessa abordagem, em vez de representar e manipular conjuntos de estados explicitamente, os algoritmos representam conjuntos de estados por meio de diagramas de decisão, obtidos a partir de fórmulas proposicionais, e toda manipulação de conjuntos é feita por meio de operações sobre esses diagramas.

2.4.1 Representação simbólica de estados e transições

Seja $\mathcal{K} = \langle \mathcal{S}, \mathcal{T}, \mathcal{L} \rangle$ uma estrutura de Kripke sobre um conjunto finito de proposições atômicas $\mathbb{P}$. A representação simbólica para um estado $s \in \mathcal{S}$ é a fórmula proposicional

$$\xi(s) = \bigwedge_{p \in \mathcal{L}(s)} p \ \wedge \bigwedge_{p \notin \mathcal{L}(s)} \neg p \tag{2.1}$$

e a representação para um conjunto de estados $S \subseteq \mathcal{S}$ é a fórmula proposicional

$$\xi(S) = \bigvee_{s \in \mathcal{S}} \xi(s). \tag{2.2}$$

Para representar transições, precisamos de variáveis distintas para denotar propriedades que valem antes e depois das transições. Usaremos a notação $[\varphi]'$ para denotar a fórmula proposicional obtida por renomeação das variáveis de φ, de modo que cada variável p em φ é substituída por outra variável p' correspondente. Essa função de renomeação deve ser bijetora, já que os símbolos p e p' representam a mesma proposição, porém, em instantes consecutivos de tempo.

A representação para uma transição $(s_i, s_j) \in \mathcal{T}$ é a fórmula proposicional

$$\xi((s_i, s_j)) = \xi(s_i) \wedge [\xi(s_j)]' \tag{2.3}$$

e a representação simbólica para a relação de transições $\mathcal{T}$ é

$$\xi(\mathcal{T}) = \bigvee_{t \in \mathcal{T}} \xi(t). \tag{2.4}$$

2.4.2 Diagramas de decisão binária

Sejam φ um fórmula proposicional e $\varphi\,|_{p \leftarrow c}$ a fórmula obtida pela substituição de toda ocorrência da variável proposicional p em φ pela constante $c \in \{\top, \bot\}$. A *expansão de Shannon* da fórmula φ, com relação à variável p, é dada pela seguinte equivalência:

$$\varphi \equiv \varphi\,|_{p \leftarrow \top} \vee \varphi\,|_{p \leftarrow \bot}. \tag{2.5}$$

Com base nessa equivalência, definimos o operador condicional *ite*[3] do seguinte modo:

$$ite(p, \alpha, \beta) \doteq (p \wedge \alpha) \vee (\neg p \wedge \beta) \tag{2.6}$$

Na expressão $ite(p, \alpha, \beta)$, a proposição p é denominada *teste*. Intuitivamente, se essa variável é verdadeira, o valor da expressão é dado por α; senão, é dado por β.

Todos os conectivos clássicos podem ser expressos por meio do operador *ite*, veja:

- $\neg p = ite(p, \bot, \top)$
- $p_1 \wedge p_2 = ite(p_1, \top \wedge p_2, \bot \wedge p_2) = ite(p_1, p_2, \bot) = ite(p_1, ite(p_2, \top, \bot), \bot)$
- $p_1 \vee p_2 = ite(p_1, \top \vee p_2, \bot \vee p_2) = ite(p_1, \top, p_2) = ite(p_1, \top, ite(x_2, \top, \bot))$
- $p_1 \rightarrow p_2 = ite(p_1, \top \rightarrow p_2, \bot \rightarrow p_2) = ite(p_1, p_2, \top) = ite(p_1, ite(p_2, \top, \bot), \top)$

Um fórmula proposicional está na *forma normal condicional* se e só se contém apenas *ite* como operador, todos os testes são realizados sobre variáveis e todas as variáveis ocorrem apenas como testes [Andersen, 1997].

Toda fórmula proposicional φ pode ser convertida para a forma normal condicional. A conversão é feita do seguinte modo: enquanto houver uma variável proposicional $p \in \varphi$ que não seja teste, reescreva φ como $ite(p, \varphi\,|_{p \leftarrow \top}, \varphi\,|_{p \leftarrow \bot})$. Como exemplo, veja a conversão da fórmula $(p_1 \vee p_2) \wedge (p_2 \vee p_3)$ para a forma normal condicional:

[3] *if-then-else*

$(p_1 \vee p_2) \wedge (p_2 \vee p_3)$
$= ite(p_1, (\top \vee p_2) \wedge (p_2 \vee p_3), (\bot \vee p_2) \wedge (p_2 \vee p_3))$
$= ite(p_1, p_2 \vee p_3, p_2)$
$= ite(p_1, ite(p_2, \top \vee p_3, \bot \vee p_3, ite(p_2, \top, \bot))$
$= ite(p_1, ite(p_2, \top, p_3, ite(p_2, \top, \bot))$
$= ite(p_1, ite(p_2, \top, ite(p_3, \top, \bot), ite(p_2, \top, \bot))$

Árvore de decisão binária. A forma normal condicional de uma fórmula proposicional descreve um grafo, denominado *árvore de decisão binária*, que define o valor dessa fórmula sob toda interpretação possível de suas variáveis. Por exemplo, a árvore de decisão binária correspondente à fórmula $(p_1 \vee p_2) \wedge (p_2 \vee p_3)$, cuja forma normal condicional é $ite(p_1, ite(p_2, \top, ite(p_3, \top, \bot), ite(p_2, \top, \bot))$, pode ser vista na Figura 2.8(a).

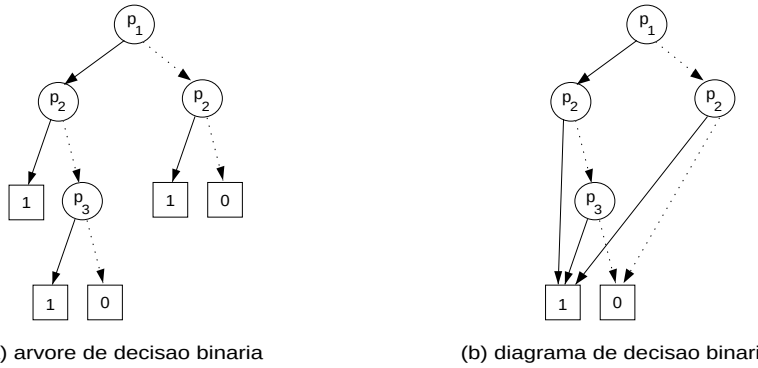


Figura 2.8: Árvore e diagrama de decisão binária para a fórmula $(p_1 \vee p_2) \wedge (p_2 \vee p_3)$.

Numa árvore de decisão binária, as folhas são rotuladas com as constantes 0 e 1 (*aka.* $\bot$ e $\top$) e os nós internos são rotulados com variáveis proposicionais. Cada nó interno p_i tem um filho esquerdo (para $p_i = \top$), representado por uma linha contínua, e um filho direito (para $p_i = \bot$), representado por uma linha pontilhada.

Diagrama de decisão binária. Embora as árvores de decisão sejam muito úteis para representar fórmulas proposicionais, freqüentemente, elas apresentam muita redundância. Parte dessa redundância pode ser eliminada por meio do compartilhamento de subgrafos isomorfos. Quando esse compartilhamento é feito, a árvore é transformada num grafo orientado acíclico, denominado *diagrama de decisão binária* (DDB) [Bryant, 1992]. Por exemplo, o resultado do compartilhamento de subgrafos isomorfos na árvore de decisão para a fórmula $(p_1 \vee p_2) \wedge (p_2 \vee p_3)$ pode ser visto na Figura 2.8.

Particularmente, quando a ordem das variáveis de teste em todos os caminhos que levam da raiz até uma folha da árvore é sempre a mesma, o grafo obtido pelo compartilhamento é denominado *diagrama de decisão binária ordenado* (DDBO).

Às vezes, após o compartilhamento, alguns testes tornam-se redundantes. Para eliminar um teste redundante, basta excluir o nó que o representa e redirecionar todo arco de entrada desse nó para o seu filho (Figura 2.9). Quando todos os testes redundantes num diagrama de decisão binária ordenado são eliminados, o grafo resultante é denominado *diagrama de decisão binária ordenado reduzido* (DDBOR).

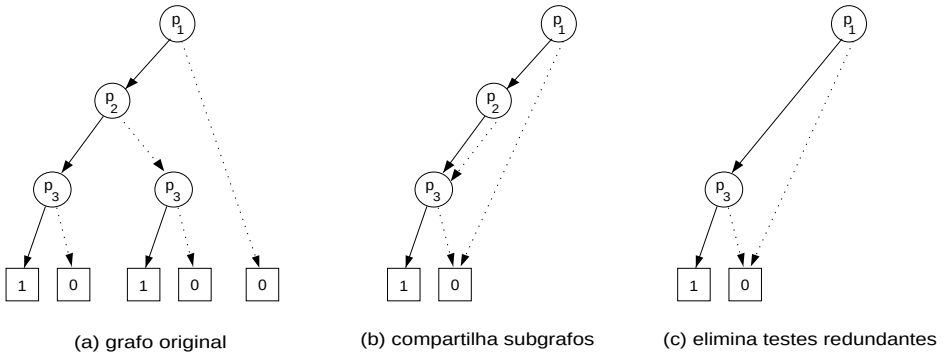


Figura 2.9: Compartilhamento de grafos isomorfos e eliminação de testes redundantes.

Os DDBOR's apresentam algumas propriedades importantes [Andersen, 1997]:

- para toda fórmula proposicional há um único DDBOR correspondente;
- proporcionam uma representação compacta para fórmulas proposicionais;
- possibilitam algoritmos muito eficientes para manipulação das fórmulas.

Na literatura, é comum o uso do termo "*diagrama de decisão binária*" (DDB) como sinônimo de *diagrama de decisão binária ordenado e reduzido* (DDBOR).

Algoritmos para manipulação de DDBOR's. Em termos de estruturas de dados, um DDBOR é uma tabela $T : n \mapsto \langle v, t, f \rangle$, que associa a cada identificador n um nó com variável de teste v, filho esquerdo t e filho direito f. Ademais, devido ao compartilhamento de subgrafos, a tabela T tem uma inversa $T^{-1} : \langle v, t, f \rangle \mapsto n$, mapeando nós em identificadores, que será utilizada para garantir que os diagramas sejam reduzidos. Também assumiremos que $T(n) = T^{-1}(\langle v, t, f \rangle) = nil$, sempre que $(n, \langle v, t, f \rangle) \notin T$. Nessas tabelas, os identificadores são $0, 1, 2, \ldots$ (sendo 0 e 1 reservados para os nós terminais), as variáveis proposicionais $p_1, p_2, \ldots$ são representadas pelos índices $1, 2, \ldots$ e a ordem em que as variáveis são testadas é definida pelos seus índices.

Para facilitar a representação dos algoritmos, trataremos a tabela T como uma variável global e escreveremos $|T|$ para denotar o número de entradas existentes na tabela T.

- *Iniciação da estrutura:* O algoritmo INICIA recebe uma entrada m, indicando o número de variáveis proposicionais existentes na fórmula a ser representada, e inicia a tabela T com duas tuplas especiais, representando os nós terminais 0 e 1. Para um tratamento uniforme, esses nós terminais são associados à variável p_{m+1}.

  INICIA$[T](m)$
  1 $T \leftarrow \{(0, \langle m+1, nil, nil \rangle), (1, \langle m+1, nil, nil \rangle)\}$

- *Inserção de nós:* Para inserir um nó na tabela, usamos o algoritmo INSERE. Caso o nó seja redundante, o algoritmo simplesmente devolve o identificador de seu filho; caso o nó já tenha sido criado anteriormente, o algoritmo devolve seu identificador; finalmente, caso o nó seja novo, o algoritmo o cria e devolve seu identificador.

INSERE$[T](v,t,f)$
1 **se** $t = f$ **então devolva** t
2 $n \leftarrow T^{-1}(\langle v,t,f\rangle)$
3 **se** $n = nil$ **então**
5 $n \leftarrow |T|$
6 $T \leftarrow T \cup \{(n,\langle v,t,f\rangle)\}$
7 **devolva** n

- *Construção do diagrama:* O algoritmo CONSTRÓI recebe uma fórmula $ite(v,\varphi_t,\varphi_f)$, na forma normal condicional, cria uma tabela com os nós do diagrama de decisão para essa fórmula e devolve o identificador para o nó raiz desse diagrama.

CONSTRÓI$(ite(v,\varphi_t,\varphi_f))$
1 **se** $\varphi_t,\varphi_f \in \{0,1\}$ **então devolva** INSERE(v,φ_t,φ_f)
2 **se** $\varphi_f \in \{0,1\}$ **então devolva** INSERE$(v$,CONSTRÓI$(\varphi_t),\varphi_f)$
3 **se** $\varphi_t \in \{0,1\}$ **então devolva** INSERE$(v,\varphi_t$,CONSTRÓI$(\varphi_f))$
4 **devolva** INSERE$(v$,CONSTRÓI(φ_t),CONSTRÓI$(\varphi_f))$

- *Operações lógicas:* As operações lógicas entre fórmulas são implementadas pelo algoritmo genérico APLICA, que baseia-se na seguinte equivalência:

$$ite(v,\varphi_t,\varphi_f) \star ite(v,\varphi'_t,\varphi'_f) \equiv ite(v,\varphi_t \star \varphi'_t,\varphi_f \star \varphi'_f),\ \text{para} \star \in \{\wedge,\vee,\rightarrow\}. \quad (2.7)$$

Assim, para efetuarmos uma operação lógica entre duas fórmulas, basta aplicarmos essa transformação, recursivamente, a partir das raízes dos diagramas para essas fórmulas. Para maior eficiência, o algoritmo apresentado a seguir utiliza a técnica de *memoização*[4], implementada por meio da tabela M.

APLICA(op,r_1,r_2)
1 $M \leftarrow \emptyset$
2 **devolva** APLICA$'(op,r_1,r_2)$

APLICA$'[T,M](op,r_1,r_2)$
1 **se** $M(\langle r_1,r_2\rangle) \neq nil$ **então devolva** $M(\langle r_1,r_2\rangle)$
2 $\langle v_1,t_1,f_1\rangle \leftarrow T(r_1)$
3 $\langle v_2,t_2,f_2\rangle \leftarrow T(r_2)$
4 **se** $r_1,r_2 \in \{0,1\}$ **então** $r \leftarrow op(r_1,r_2)$
5 **senão se** $v_1 = v_2$ **então** $r \leftarrow$ INSERE$(v_1$,APLICA$'(op,t_1,t_2)$,APLICA$'(op,f_1,f_2))$
6 **senão se** $v_1 < v_2$ **então** $r \leftarrow$ INSERE$(v_1$,APLICA$'(op,t_1,r_2)$,APLICA$'(op,f_1,r_2))$
7 **senão se** $v_1 > v_2$ **então** $r \leftarrow$ INSERE$(v_2$,APLICA$'(op,r_1,t_2)$,APLICA$'(op,r_1,f_2))$
8 $M \leftarrow M \cup \{(\langle r_1,r_2\rangle,r)\}$
9 **devolva** r

2.4.3 Algorimos para verificação de modelos simbólicos

A idéia básica da verificação de modelos simbólicos [McMillan, 1992] é usar DDBOR's para representar *funções características* de conjuntos de estados e da relação de transição.

Se ρ é uma relação n-ária sobre $\{0,1\}$, então ρ pode ser representada pelo DDBOR de sua função característica:

$$f_\rho(x_1,\ldots,x_n) = 1 \Leftrightarrow (x_1,\ldots,x_n) \in \rho \quad (2.8)$$

[4]Programação dinâmica sob demanda, isso permite que o algoritmo consuma tempo polinomial.

Para que relações complexas possam ser construídas, é conveniente estender a lógica proposicional de modo a permitir quantificação sobre os valores das proposições:

$$\forall p\ \varphi \equiv \varphi\,|_{p\leftarrow\top} \wedge \varphi\,|_{p\leftarrow\bot} \tag{2.9}$$

$$\exists p\ \varphi \equiv \varphi\,|_{p\leftarrow\top} \vee \varphi\,|_{p\leftarrow\bot} \tag{2.10}$$

Fórmulas proposicionais quantificadas têm o mesmo poder expressivo das fórmulas proposicionais ordinárias; porém, são mais concisas. Toda fórmula proposicional quantificada determina uma relação n-ária sobre $\{\top, \bot\}$. Dado um diagrama de decisão para uma fórmula proposicional φ, é fácil construir os diagramas para as fórmulas $\forall p\ \varphi$ e $\exists p\ \varphi$, usando as equivalências 2.9 e 2.10. Na prática, porém, algoritmos especiais são necessários para manipular quantificadores eficientemente.

Produto relacional. Em algoritmos para verificação de modelos simbólicos, quantificadores ocorrem mais freqüentemente em *produtos relacionais*, que têm a seguinte forma:

$$\exists\, \vec{p}\ [\varphi_1(\vec{p}) \wedge \varphi_2(\vec{p})], \tag{2.11}$$

onde $\vec{p}$ é o conjunto de variáveis proposicionais.

O algoritmo ProdRel, apresentado a seguir, realiza essa operação sobre os diagramas das fórmulas $\varphi_1(\vec{p})$ e $\varphi_2(\vec{p})$ (cujas raízes são representadas por r_1 e r_2), sem ter que construir o diagrama para a fórmula $\varphi_1(\vec{p}) \wedge \varphi_2(\vec{p})$.

```
ProdRel[T](p⃗, r1, r2)
1 M ← ∅
2 devolva ProdRel'(p⃗, r1, r2)

ProdRel'[T, M](p⃗, r1, r2)
 1 se r1 = r2 = 0 então devolva 0
 2 se r1 = r2 = 1 então devolva 1
 3 se M(⟨p⃗, r1, r2⟩) ≠ nil então devolva M(⟨p⃗, r1, r2⟩)
 4 ⟨v1, t1, f1⟩ ← T(r1)
 5 ⟨v2, t2, f2⟩ ← T(r2)
 6 v3 ← min{v1, v2}
 7 t3 ← ProdRel'(p⃗, φ1 |v3←⊤, φ2 |v3←⊤)
 8 f3 ← ProdRel'(p⃗, φ1 |v3←⊥, φ2 |v3←⊥)
 9 se v3 ∈ p⃗ então r ←Aplica(∨, t3, f3)
10 senão r ←Insere(v3, t3, f3)
11 M ← M ∪ {(⟨p⃗, r1, r2⟩, r)}
12 devolva r
```

Estados e transições como ddbor's. Dado um conjunto A, sua função característica $A(x) = x \in A$ permite identificar todo elemento x de A. Um estado pode ser representado por um vetor $\vec{p} = (p_1, \ldots, p_k)$, onde k é o número de proposições que descrevem os estados na estrutura de Kripke. Assim, um conjunto de estados S pode ser representado por uma função característica $S(\vec{p})$, codificada como um ddbor. Analogamente, a função $T(\vec{p}, \vec{p}')$ pode ser usada para representar a função característica da relação de transição.

Por exemplo, considere um modelo cujos estados são s_1, s_2 e s_3, tal que $\xi(s_1) = p_1 \wedge p_2$, $\xi(s_2) = p_1 \wedge \neg p_2$ e $\xi(s_3) = \neg p_1 \wedge p_2$ e cujas transições são (s_1, s_2), (s_1, s_3) e (s_2, s_3). Então, a relação de transição para esse modelo pode ser representada pela seguinte função característica:

$$\begin{aligned} T(\vec{p},\vec{p}\,') = (p_1 \wedge p_2 \ \ \wedge \ \ p_1' \wedge \neg p_2') \ \ \vee \\ (p_1 \wedge p_2 \ \ \wedge \ \ \neg p_1' \wedge p_2') \ \ \vee \\ (p_1 \wedge \neg p_2 \ \ \wedge \ \ \neg p_1' \wedge p_2') \end{aligned}$$

Usando produto relacional, podemos obter o conjunto de estados sucessores de s_1 da seguinte maneira:

$$\begin{aligned} \mathcal{T}(s_1) &= (\exists \vec{p}.(\xi(s_1) \ \ \wedge \ \ T(\vec{p},\vec{p}\,'))[\vec{p}/\vec{p}\,'] \\ &= (\exists \vec{p}.((p_1 \wedge p_2) \ \ \wedge \ \ T(\vec{p},\vec{p}\,'))[\vec{p}/\vec{p}\,'] \\ &= (\exists \vec{p}.((p_1 \wedge p_2 \ \ \wedge \ \ p_1' \wedge \neg p_2') \ \ \vee \ \ (p_1 \wedge p_2 \ \ \wedge \ \ \neg p_1' \wedge p_2')))[\vec{p}/\vec{p}\,'] \\ &= (p_1' \wedge \neg p_2' \ \ \vee \ \ \neg p_1' \wedge p_2'))[\vec{p}/\vec{p}\,'] \\ &= (p_1 \wedge \neg p_2 \ \ \vee \ \ \neg p_1 \wedge p_2)) \\ &= \{s_2, s_3\} \end{aligned}$$

Note que a quantificação existencial abstrai o estado origem da transição e a renomeação restaura os nomes das variáveis proposicionais que descrevem os estados. Assim, para obter estados predecessores, basta quantificar existencialmente as variáveis $\vec{p}\,'$.

O algoritmo VMS**.** O algoritmo VMS recebe como entrada um DDBOR (que deixaremos como parâmetro implícito), representando a relação de transição de um sistema, e uma fórmula CTL φ, especificando a propriedade a ser verificada nesse sistema. Como saída, o algoritmo devolve um DDBOR representando o conjunto de estados do sistema que satisfazem essa fórmula. Esse algoritmo é definido indutivamente sobre a estrutura da fórmula φ. Se φ é uma proposição p_i, $\text{VMS}(r, \varphi)$ simplesmente devolve o DDBOR para p_i. Para tratar operadores temporais, o algoritmo utiliza as seguintes funções auxiliares:

- $\text{VMS}(\exists \odot \varphi) = \text{VMS}_{\exists\odot}(\text{VM}(\varphi))$
- $\text{VMS}(\exists\Box\varphi) = \text{VMS}_{\exists\Box}(\text{VM}(\varphi))$
- $\text{VMS}(\exists(\varphi_1 \cup \varphi_2)) = \text{VMS}_{\exists\cup}(\text{VMS}(\varphi_1),\text{VMS}(\varphi_2))$

Essas funções recebem fórmulas proposicionais como argumentos, enquanto o algoritmo principal recebe uma fórmula CTL. Os casos de fórmulas proposicionais com operadores lógicos usuais são tratados pelo algoritmo APLICA. A função $\text{VMS}_{\exists\odot}$ é implementada diretamente como um produto relacional, sendo suficiente encontrar os estados predecessores daqueles onde a fórmula sendo verificada é satisfeita:

$$\text{VMS}_{\exists\odot}(\ \varphi(\vec{p})\ ) = \ \exists\vec{p}\,'[\varphi(\vec{p}\,') \wedge r((\vec{p}),(\vec{p}\,'))] \tag{2.12}$$

As outras duas funções são baseadas na caracterização de ponto fixo dos operadores CTL, além do produto relacional:

$$\text{VMS}_{\exists\Box}(\ \varphi(\vec{p})\ ) = \ \nu Y(\vec{p}).[\varphi(\vec{p}) \wedge \text{VMS}_{\exists\odot}(Y(\vec{p}))] \tag{2.13}$$

$$\text{VMS}_{\exists\cup}(\ \varphi_1(\vec{p}),\ \varphi_2(\vec{p})\ ) = \ \mu Y(\vec{p}).[\varphi_2(\vec{p}) \vee (\varphi_1(\vec{p}) \wedge \text{VMS}_{\exists\odot}(Y(\vec{p})))] \tag{2.14}$$

2.5 Sumário

Nesse capítulo, apresentamos o modelo computacional utilizado para verificação de modelos, definimos um algoritmo para verificação de modelos baseado na caracterização de ponto-fixo para os operadores da lógica temporal CTL e mostramos como esse algoritmo pode ser implementado eficientemente, em termos de operações simbólicas sobre diagramas de decisão binária ordenados e reduzidos.

No próximo capítulo, mostraremos como planejamento não-determinístico pode ser implementado dentro da abordagem de verificação de modelos (simbólicos).

Capítulo 3

Planejamento baseado em Verificação de Modelos

3.1 Introdução

Quando o arcabouço de verificação de modelos é empregado no contexto de planejamento, o modelo $\mathcal{K}$ descreve a dinâmica de um ambiente, enquanto a propriedade φ descreve os objetivos do agente nesse ambiente. Nesse caso, além de $\mathcal{K}$ e φ, o planejador recebe também o estado inicial s_0 do ambiente. Se $(\mathcal{K}, s_0) \models \varphi$, o planejador devolve um *plano*, *i.e.*, uma política de comportamento que permite ao agente atingir seus objetivos; senão, o planejador devolve *falha* (Figura 3.1).

Figura 3.1: Planejador baseado em verificação de modelos.

Note, porém, que há uma diferença significativa entre um verificador e um planejador: enquanto o verificador precisa verificar a validade da propriedade em todos os estados do modelo, o planejador precisa garantir a validade da propriedade apenas na estrutura de Kripke induzida pelas ações incluídas no plano sintetizado.

Planejamento baseado em verificação de modelos é uma extensão bastante genérica de planejamento clássico, que permite resolver problemas com metas simples ou estendidas, em ambientes determinísticos ou não-determinísticos, com observabilidade completa, nula ou parcial, usando representação explícita ou simbólica de estados [Ghallab et al., 2004]. Nesse trabalho, porém, consideramos apenas observabilidade completa.

3.2 Domínios de planejamento

Formalmente, um domínio de planejamento é especificado por um diagrama de transições cujos nós representam estados e cujos arcos representam transições entre estados.

Definição 3.1. *Um domínio de planejamento não-determinístico com estados simbólicos é uma tupla* $\mathcal{D} = \langle \mathbb{P}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rangle$*, onde:*

- $\mathbb{P} \neq \emptyset$ *é um conjunto de proposições que descrevem o ambiente;*
- $\mathcal{S} \subseteq 2^{\mathbb{P}}$ *é um conjunto finito de estados possíveis do ambiente;*
- $\mathcal{A} \neq \emptyset$ *é um conjunto finito de ações executáveis pelo agente;*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto 2^{\mathcal{S}}$ *é uma função de transição de estados.* ■

Em aplicações práticas, entretanto, especificar domínios por meio de diagramas de transições não é muito comum. Como o número de estados num ambiente é exponencial no número de proposições usadas na descrição desse ambiente, em geral, é praticamente impossível enumerar todos os estados e transições explicitamente. Ademais, é muito mais simples e intuitivo especificar ações por meio de precondições e efeitos do que por meio da enumeração das triplas em $\mathcal{T}$. Assim, domínios de planejamento são normalmente especificados em linguagens de alto-nível, especializadas para descrição de ações.

3.2.1 Linguagens $\mathcal{AR}$

$\mathcal{AR}$ [Giunchiglia et al., 1997] é uma família de linguagens para descrição de ações. Essas linguagens têm uma sintaxe simples e uma semântica bem-definida, que incorpora uma solução para o problema de persistência temporal[1] na presença de efeitos incertos e indiretos (*i.e.*, não-determinismo e ramificação).

Numa linguagem $\mathcal{AR}$, as proposições que denotam propriedades dos estados do ambiente são denominadas *fluentes*. Dizemos que um fluente é *inerte* se seu valor pode ser modificado apenas como conseqüência da execução de ações. Fluentes cujos valores podem se modificar independentemente da ação do agente, ou seja, como conseqüência da ocorrência de eventos exógenos, são denominados *não-inertes*.

Uma linguagem $\mathcal{AR}$ é caracterizada por um conjunto de *fluentes*, parte dos quais devem ser *inertes*, e por um conjunto de *ações*.

Sintaxe. Uma *fórmula atômica* em $\mathcal{AR}$ tem a forma (p `is` x), onde p é um fluente e $x \in Dom(p)$. Quando $Dom(p) = \{\top, \bot\}$, a fórmula (p `is` $\top$) pode ser escrita como p e (p `is` $\bot$), como $\neg p$. Uma *fórmula* é uma combinação proposicional de fórmulas atômicas.

Uma linguagem $\mathcal{AR}$ permite sentenças das seguintes formas (onde a é uma ação, p é um fluente inerte e φ é uma fórmula proposicional):

$$a \texttt{ causes } \varphi' \texttt{ if } \varphi \tag{3.1}$$

$$a \texttt{ possibly changes } p \texttt{ if } \varphi \tag{3.2}$$

$$\texttt{always } \varphi \tag{3.3}$$

Intuitivamente, (3.1) estabelece que a execução da ação a, num estado que satisfaz φ, causa *necessariamente* uma transição para um estado que satisfaz φ'; (3.2) estabelece que a execução da ação a, num estado que satisfaz φ, causa *possivelmente* uma transição para um estado onde o valor do fluente p é modificado; e, finalmente, (3.3) estabelece que a fórmula φ deve ser satisfeita em todo estado.

[1] *Frame problem* [Shanahan, 1997].

Uma sentença da forma (a `causes` φ' `if` $\top$) pode ser abreviada como (a `causes` φ'). Analogamente, uma sentença da forma (a `causes` $\bot$ `if` $\neg\varphi$), pode ser abreviada como (a `has preconditions` φ), estabelecendo que a ação a só pode ser executada em estados que satisfazem a sua precondição φ.

Um conjunto de sentenças das formas (3.1), (3.2) e (3.3) é denominado *descrição de domínio* de planejamento.

Semântica. Uma *valoração* é uma função que associa a cada fluente p um elemento $x \in Dom(p)$. Uma valoração σ é estendida para fórmulas atômicas da seguinte maneira:

$$\sigma(p \texttt{ is } x) = \begin{cases} \top & \text{se } \sigma(p) = x \\ \bot & \text{caso contrário} \end{cases} \tag{3.4}$$

A extensão de σ para fórmulas arbitrárias pode ser feita diretamente.

Seja $\mathcal{D}$ uma descrição de domínio. Uma valoração σ é um *estado* em $\mathcal{D}$ se, para toda sentença da forma (`always` φ) em $\mathcal{D}$, temos $\sigma(\varphi) = \top$.

A semântica para uma descrição de domínio $\mathcal{D}$ é um diagrama de transições, consistindo de um conjunto de estados $\mathcal{S}$ e de uma função de transição $\mathcal{T}$, que mapeia um estado e uma ação num conjunto de estados. Intuitivamente, $\mathcal{T}(s, a)$ denota o conjunto de estados sucessores que podem ser atingidos pela execução da ação a no estado s.

Seja $\mathcal{T}_0(s, a)$ o conjunto de estados s' tais que, para cada sentença da forma (a `causes` φ' `if` φ) em $\mathcal{D}$, temos $s'(\varphi') = \top$ sempre que $s(\varphi) = \top$. O conjunto $\mathcal{T}(s, a)$ é definido como o subconjunto de $\mathcal{T}_0(s, a)$ que contém apenas estados "bem próximos" de s.

Para tornar essa definição mais precisa, para uma ação a e estados s e s', escreveremos $New^a(s, s')$ para denotar o conjunto de fórmulas da forma

$$p \texttt{ is } s'(p) \tag{3.5}$$

tais que:

- p é um fluente inerte e $s'(p) \neq s(p)$; ou
- existe em $\mathcal{D}$ uma sentença da forma (a `possibly changes` p `if` φ) e $s(\varphi) = \top$.

A condição $s'(p) \neq s(p)$, nessa definição, expressa que (3.5) é um fato que torna-se verdadeiro quando a execução da ação a no estado s resulta no estado s'. O conjunto $New^a(s, s')$ inclui esses "novos fatos" para todos os fluentes inertes do domínio. Ademais, se alguma sentença de efeito não-determinístico permite p mudar, tratamos seu valor em s' como "novo", mesmo se acontecer dele coincidir com o valor de p no estado s.

Agora, podemos definir a função de transição como segue: $\mathcal{T}(s, a)$ é o conjunto de estados $s' \in \mathcal{T}_0(s, a)$, para os quais $New^a(s, s')$ é minimal com relação à inclusão de conjuntos, ou seja, para os quais não existe $s'' \in \mathcal{T}_0(s, a)$, tal que $New^a(s, s'')$ seja um subconjunto próprio de $New^a(s, s')$. Essa condição de minimalidade na semântica de $\mathcal{AR}$ é que garante a solução do problema de persistência temporal.

3.2.2 Descrevendo um domínio em $\mathcal{AR}$

Considere a descrição de domínio a seguir, em que *ligar*, *desligar* e *trocar* são as ações que o agente é capaz de executar e *ligado*, *queimada* e *acesa* são os fluentes que descrevem o ambiente desse agente:

$$
\begin{array}{l}
\mathit{ligar}\ \texttt{has preconditions}\ \neg \mathit{ligado} \\
\mathit{ligar}\ \texttt{causes}\ \mathit{ligado} \\
\mathit{ligar}\ \texttt{possibly changes}\ \mathit{queimada}\ \texttt{if}\ \neg \mathit{queimada} \\
\mathit{desligar}\ \texttt{has preconditions}\ \mathit{ligado} \\
\mathit{desligar}\ \texttt{causes}\ \neg \mathit{ligado} \\
\mathit{trocar}\ \texttt{has preconditions}\ \neg \mathit{ligado} \\
\mathit{trocar}\ \texttt{causes}\ \neg \mathit{queimada}\ \texttt{if}\ \mathit{queimada} \\
\texttt{always}\ \mathit{ligado} \wedge \neg \mathit{queimada} \leftrightarrow \mathit{acesa}
\end{array}
\tag{3.6}
$$

Nessa descrição, todos os fluentes são inertes. Porém, numa descrição mais realista, não bastaria ter o interruptor ligado e a lâmpada não-queimada para que tivéssemos a lâmpada acesa; seria necessário que houvesse *energia*. Esse é um exemplo de fluente *não-inerte*, cujo valor não está sujeito à *Lei da Inércia*: haver ou não energia independe das ações executadas pelo agente (*ligar*, *desligar* ou *trocar*) e o valor do fluente *energia* pode se modificar espontaneamente (*i.e.* como conseqüência de um evento exógeno).

Estados. Há quatro estados no domínio descrito em (3.6), cada um deles correspondendo a uma valoração[2] que satisfaz a restrição (`always` $\mathit{ligado} \wedge \neg \mathit{queimado} \leftrightarrow \mathit{acesa}$):

$$\{\, L\, Q\, \overline{A},\ L\, \overline{Q}\, A,\ \overline{L}\, Q\, \overline{A},\ \overline{L}\, \overline{Q}\, \overline{A}\} \tag{3.7}$$

Transições. Considere a ação *ligar*, descrita em (3.6), cuja precondição é $\neg \mathit{ligado}$ ($\overline{L}$). Claramente, há apenas dois estados do domínio em que essa ação pode ser executada:

- Para o primeiro estado ($\overline{L}\, Q\, \overline{A}$), temos:
  - $\mathcal{T}_0(\overline{L}\, Q\, \overline{A},\ \mathit{ligar}) = \{L\, Q\, \overline{A},\ L\, \overline{Q}\, A\}$
  - $New^{\mathit{ligar}}(\overline{L}\, Q\, \overline{A},\ L\, Q\, \overline{A}) = \{L\}$
  - $New^{\mathit{ligar}}(\overline{L}\, Q\, \overline{A},\ L\, \overline{Q}\, A) = \{L,\ \overline{Q}, A\}$

  Então, como o conjunto $New^{\mathit{ligar}}(\overline{L}\, Q\, \overline{A},\ L\, \overline{Q}\, A)$ não é minimal, concluímos que:
  - $\mathcal{T}(\overline{L}\, Q\, \overline{A},\ \mathit{ligar}) = \{L\, Q\, \overline{A}\}$.

  De fato, a transição do estado $\overline{L}\, Q\, \overline{A}$ para o estado $L\, \overline{Q}\, A$ não faz mesmo sentido, já que a ação *ligar* não pode transformar uma lâmpada queimada em não-queimada.

- Para o segundo estado ($\overline{L}\, \overline{Q}\, \overline{A}$), temos.
  - $\mathcal{T}_0(\overline{L}\, \overline{Q}\, \overline{A},\ \mathit{ligar}) = \{L\, Q\, \overline{A},\ L\, \overline{Q}\, A\}$
  - $New^{\mathit{ligar}}(\overline{L}\, \overline{Q}\, \overline{A},\ L\, Q\, \overline{A}) = \{L, Q\}$
  - $New^{\mathit{ligar}}(\overline{L}\, \overline{Q}\, \overline{A},\ L\, \overline{Q}\, A) = \{L, \overline{Q}, A\}$

  Agora, os dois conjuntos New^{ligar} são minimais ($Q \neq \overline{Q}$) e, portanto:
  - $\mathcal{T}(\overline{L}\, \overline{Q}\, \overline{A},\ \mathit{ligar}) = \{L\, Q\, \overline{A},\ L\, \overline{Q}\, A\}$.

[2]Para facilitar a exposição, escreveremos L para denotar (ligado `is` $\top$) e $\overline{L}$ para denotar (ligado `is` $\bot$). Analogamente, para os demais fluentes do domínio (3.6), utilizaremos as abreviaturas Q, $\overline{Q}$, A e $\overline{A}$.

De fato, quando a lâmpada não está queimada, a ação *ligar* pode queimá-la, já que esse é um efeito não-determinístico dessa ação.

O diagrama de transições para o domínio em (3.6) pode ser visto na figura 3.2.

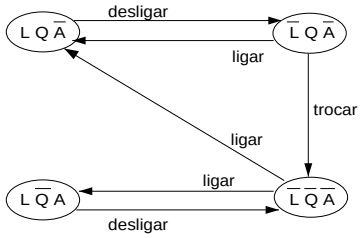


Figura 3.2: Diagrama de transições para o domínio descrito em (3.6).

3.2.3 Descrevendo um problema em $\mathcal{AR}$

Para descrever um problema de planejamento em $\mathcal{AR}$, precisamos estender sua sintaxe com mais dois tipos de sentenças:

$$\texttt{initially}\ \varphi \tag{3.8}$$

$$\texttt{goal}\ \varphi \tag{3.9}$$

Intuitivamente, uma sentença da forma (3.8) estabelece que a fórmula φ é satisfeita no estado inicial, enquanto uma sentença da forma (3.9) estabelece que os estados metas são aqueles que satisfazem a fórmula φ.

Uma descrição de domínio, juntamente com um conjunto de fórmulas das formas (3.8) e (3.9), é denominado *descrição de problema* de planejamento.

Por exemplo, um problema de planejamento pode ser obtido estendendo-se a descrição de domínio em (3.6) com as seguintes sentenças:

$$\begin{array}{l}\texttt{initially}\ \neg ligado \wedge \neg acesa \\ \texttt{goal}\ acesa\end{array} \tag{3.10}$$

Note que uma sentença da forma (`initially` φ) define um conjunto de estados iniciais possíveis. De fato, o estado inicial definido em (3.10) pode ser qualquer estado onde o interruptor está desligado e a lâmpada está apagada (*i.e.*, $\overline{L}\,Q\overline{A}$ ou $\overline{L}\,\overline{Q}\,\overline{A}$). Sentenças da forma (`goal` φ) também podem definir mais que um estado meta, embora a sentença em (3.6) defina apenas o estado $L\overline{Q}A$ como meta.

3.2.4 Construção de autômatos simbólicos

Dada uma descrição de problema de planejamento $\mathcal{P}$, numa linguagem $\mathcal{AR}$, podemos construir um autômato finito simbólico correspondente. Os estados desse autômato podem ser codificados como segue:

$$\mathcal{S} \doteq \bigwedge_{(\texttt{always}\ \varphi) \in \mathcal{P}} \varphi \tag{3.11}$$

$$S_0 \doteq \mathcal{S} \wedge \bigwedge_{(\texttt{initially}\ \varphi) \in \mathcal{P}} \varphi \tag{3.12}$$

$$S_g \doteq \mathcal{S} \wedge \bigwedge_{(\texttt{goal}\ \varphi) \in \mathcal{P}} \varphi \tag{3.13}$$

Para codificar a relação de transição do autômato, introduzimos uma variável *act*, sobre o conjunto de ações $\mathcal{A}$. Para representar os valores dos fluentes antes e após a execução de uma ação, introduzimos uma variável p' para cada fluente p do domínio. A relação de transição é representada por uma fórmula $\mathcal{T}$ definida sobre as variáveis *act*, $\vec{p}$ e $\vec{p}\,'$. Atribuições a essas variáveis, satisfazendo $\mathcal{T}$, representam transições no autômato.

$$\mathcal{T}_0 \doteq \mathcal{S} \wedge \mathcal{S}[\vec{p}\,'/\vec{p}\,] \ \wedge \bigwedge_{(a\ \texttt{causes}\ \varphi'\ \texttt{if}\ \varphi) \in \mathcal{P}} ((act = a \wedge \varphi) \rightarrow \varphi'[\vec{p}\,'/\vec{p}\,]) \tag{3.14}$$

Intuitivamente, o conjunto $\mathcal{S}[\vec{p}\,'/\vec{p}\,]$ garante que as valorações para estados sucessores sejam estados do autômato; enquanto $\varphi'[\vec{p}\,'/\vec{p}\,]$ impõe que os efeitos da ação devem valer nos estados sucessores, sempre que a condição φ for satisfeita no estado corrente.

Note que a fórmula $\mathcal{T}_0$ identifica tudo que deve mudar nos estados sucessores, como conseqüência da execução de uma ação particular, mas não diz nada a respeito do que deve persistir. Podemos obter $\mathcal{T}$ minimizando as mudanças[3] em $\mathcal{T}_0$, ou seja, eliminando todas as valorações onde variações nos fluentes não são necessárias.

Na fórmula a seguir, assumimos que $p_1, \ldots, p_m$ ($p_{m+1}, \ldots, p_n$) são fluentes inertes (não-inertes) de $\mathcal{P}$, listados de acordo com alguma enumeração fixa:

$$\begin{aligned}
\mathcal{T} \doteq\ & \mathcal{T}_0 \wedge \neg\exists v_1 \ldots v_n.(\\
& \mathcal{T}_0[v_1/p'_1, \ldots, v_n/p'_n] \wedge \\
& \bigwedge_{(a\ \texttt{possibly}\,\texttt{changes}\ p_j\ \texttt{if}\ \varphi) \in \mathcal{P}} ((act = a \wedge \varphi) \rightarrow p'_j = v_j) \wedge \\
& \bigwedge_{i \in [1,\ldots,m]} (p_i = v_i \vee p'_i = v_i) \ \wedge \bigvee_{i \in [1,\ldots,m]} (p'_i \neq v_i))
\end{aligned} \tag{3.15}$$

Intuitivamente, essa definição estabelece que, dada uma ação a, uma valoração para as variáveis $\vec{p}\,'$ é compatível com uma valoração para as variáveis $\vec{p}$ se e só se satisfaz as condições de efeitos (*i.e.* $\mathcal{T}_0$) e não existe outra valoração ($\neg\exists v_1 \ldots v_n$) que também satisfaça as condições de efeitos ($\mathcal{T}_0[v_1/p'_1, \ldots, v_n/p'_n]$), que seja compatível com a valoração em $\vec{p}$ com relação aos fluentes afetados por $((act = a \wedge \varphi) \rightarrow p'_j = v_j)$ e que seja mais próxima do estado corrente (conjunção e disjunção iterativas no final da fórmula).

Daqui em diante, como os algoritmos para verificação de modelos podem operar diretamente sobre o autômato derivado de uma descrição de problema em $\mathcal{AR}$, vamos abstrair a fase de compilação de descrições de ações e supor que o autômato está disponível.

[3] Idéia semelhante àquela da circunscrição em lógica de predicados não-monotônica [Shanahan, 1997].

3.3 Planejamento para metas simples

Nessa seção, apresentamos os algoritmos de planejamento para metas simples, encontrados na literatura da área, baseados na seguinte definição de problema de planejamento:

Definição 3.2. *Um problema de planejamento não-determinístico para metas simples, com estados explícitos, é uma tupla* $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, S_0, S_g \rangle$*, onde:*

- $\mathcal{S} \neq \emptyset$ *é um conjunto finito de estados possíveis do ambiente;*
- $\mathcal{A} \neq \emptyset$ *é um conjunto finito de ações executáveis pelo agente;*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto 2^{\mathcal{S}}$ *é uma função de transição de estados;*
- $S_0 \subseteq \mathcal{S}$ *é um conjunto de estados iniciais possíveis;*
- $S_g \subseteq \mathcal{S}$ *é um conjunto de estados metas.* ■

Dados um estado s e uma ação a, $\mathcal{T}(s, a)$ é o conjunto de estados que podem ser alcançados pela execução de a em s. Uma ação a é *aplicável* a um estado s se e só se $\mathcal{T}(s, a) \neq \emptyset$. O conjunto de ações aplicáveis num estado s é denotado por $\mathcal{A}(s)$.

Uma solução para um problema de planejamento é um *plano*, também denominado *política*, que descreve um comportamento condicional e iterativo. Uma política π para um problema de planejamento $\mathcal{P}$ é um conjunto de pares (s, a) tais que $s \in \mathcal{S}$ e $a \in \mathcal{A}(s)$. Para cada estado s deve existir no máximo uma ação a tal que $(s, a) \in \pi$. O conjunto de estados de uma política π é $\mathcal{S}_\pi = \{s : (s, a) \in \pi\}$. Como podemos notar, políticas em planejamento baseado em verificação de modelo são muito semelhantes àquelas em planejamento baseado em PDM. A diferença é que agora a política é uma função *parcial* de $\mathcal{S}$, ou seja, $\mathcal{S}_\pi \subseteq \mathcal{S}$. Por exemplo, considerando o diagrama na Figura 3.3, uma política para o problema com estado inicial s_0 e estado meta s_5 é $\pi = \{(s_0, a_{01}), (s_1, a_{15}), (s_2, a_{20}\}$. O comportamento condicional deve-se ao fato da escolha de ações depender do estado corrente do mundo e o comportamento iterativo, ao fato do sistema poder visitar um mesmo estado um número arbitrário de vezes (*e.g.* executando a ação a_{20} no estado s_2).

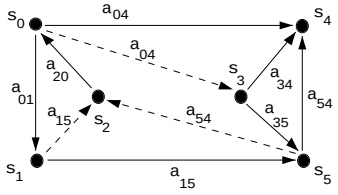


Figura 3.3: Modelo de um ambiente não-determinístico com estados explícitos.

3.3.1 Classes de soluções

Seja π uma política para um problema de planejamento $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, S_0, S_g \rangle$. Seja $\mathcal{D}_\pi = (\mathcal{S}_\pi, \mathcal{T}_\pi)$ o diagrama de transições induzido por π, onde $\mathcal{S}_\pi \subseteq \mathcal{S}$ e $\mathcal{T}_\pi \subseteq \mathcal{T}$. Podemos distinguir três classes de soluções [Ghallab et al., 2004]:

- *Soluções fracas* são planos que podem atingir a meta, mas não garantidamente. Formalmente, um plano π é uma solução fraca para $\mathcal{P}$ se e só se, para cada estado em S_0, existe um caminho que leva a um estado em S_g que é terminal em $\mathcal{D}_\pi$.

- *Soluções fortes* são planos que atingem a meta, garantidamente, a despeito do não-determinismo, ou seja, todos os caminhos no diagrama $\mathcal{D}_\pi$ são finitos e atingem a meta. Formalmente, um plano π é uma solução *forte* para $\mathcal{P}$ se e só se $\mathcal{D}_\pi$ é acíclico e todo terminal em $\mathcal{D}_\pi$ está em $\mathcal{S}_g$.
- *Soluções fortes cíclicas* são planos que atingem a meta, supondo-se que os ciclos sejam eventualmente interrompidos. Formalmente, um plano π é uma solução *forte cíclica* para $\mathcal{P}$ se e só se a partir de cada estado em $\mathcal{S}_\pi$ existe um caminho até um terminal em $\mathcal{D}_\pi$ e todo terminal em $\mathcal{D}_\pi$ está em $\mathcal{S}_g$.

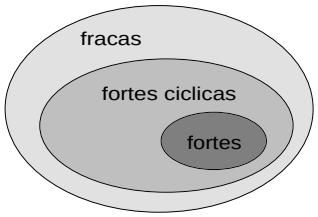


Figura 3.4: Classes de soluções *fraca*, *forte* e *forte cíclica*.

Como podemos observar na Figura 3.4, toda solução forte é também uma solução forte cíclica e toda solução forte cíclica é também uma solução fraca. Soluções estritamente fracas e soluções fortes correspondem a dois casos extremos de satisfação de metas simples. Intuitivamente, soluções estritamente fracas correspondem a "planos otimistas"e soluções fortes correspondem a "planos seguros". Entretanto, na prática, há casos em que soluções estritamente fracas não são aceitáveis e soluções fortes não existem. Nesses casos, soluções estritamente fortes cíclicas podem ser uma alternativa viável.

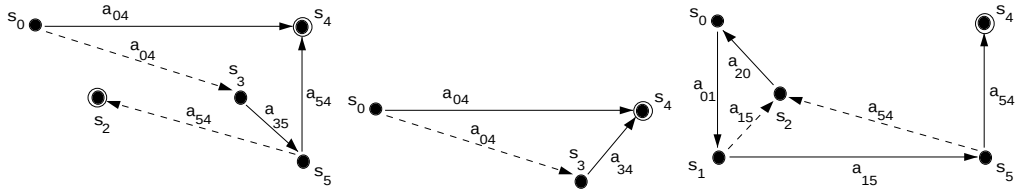


Figura 3.5: Diagramas de transições induzidos pelas políticas π_1, π_2 e π_3.

Considere as políticas a seguir, cujos diagramas induzidos são exibidos na Figura 3.5:
$\pi_1 = \{(s_0, a_{04}), (s_3, a_{35}), (s_5, a_{54})\}$
$\pi_2 = \{(s_0, a_{04}), (s_3, a_{34})\}$
$\pi_3 = \{(s_0, a_{01}), (s_1, a_{15}), (s_2, a_{20}), (s_5, a_{54})\}$

Essas políticas são diferentes estratégias para atingir o estado s_4, a partir do estado s_0. A política π_1 é uma solução fraca: se o agente chega ao estado s_2, ele não consegue mais atingir a meta. A política π_2 é uma solução forte: independentemente do não-determinismo da ação a_{04}, o agente sempre atinge a meta. Finalmente, π_3 é uma solução forte cíclica. Note que essa última política descreve um comportamento iterativo, já que o agente pode ter que repetir várias vezes uma mesma ação antes de atingir a meta. Enquanto as políticas π_1 e π_2, garantidamente, atingem um estado terminal (não necessariamente meta), a política π_3 pode ficar executando infinitamente

3.3.2 Planejamento forte

O algoritmo de planejamento FORTE [Cimatti et al., 1998] recebe como entrada um problema de planejamento $\mathcal{P}$ e devolve uma solução forte (se existir) ou `falha`. Para encontrar uma solução forte, esse algoritmo realiza uma busca regressiva em largura, de $\mathcal{S}_g$ até $\mathcal{S}_0$, através da aplicação iterativa da função PREIMAGEMFORTE ao conjunto corrente de estados atingíveis $\mathcal{S}'$, definida como:

$$\text{PREIMAGEMFORTE}(\mathcal{S}') = \{(s,a) : \mathcal{T}(s,a) \neq \emptyset \wedge \mathcal{T}(s,a) \subseteq \mathcal{S}'\} \tag{3.16}$$

Note que essa função resulta num conjunto de estados[4] cujos sucessores estão todos no conjunto $\mathcal{S}'$. Como a regressão inicia com $\mathcal{S}' = \mathcal{S}_g$, podemos garantir que todo caminho iniciando num estado em PREIMAGEMFORTE($\mathcal{S}'$) leva a um estado meta; ademais, como cada novo estado no conjunto PREIMAGEMFORTE($\mathcal{S}'$) só tem sucessores em $\mathcal{S}'$, não haverá ciclos nesses caminhos. Portanto, podemos garantir que o algoritmo FORTE termina em um dos casos: quando todos os estados iniciais estiverem no conjunto $\mathcal{S}'$ (nesse caso, π_i é uma solução forte para $\mathcal{P}$); ou quando um ponto fixo mínimo, a partir do qual o conjunto $\mathcal{S}'$ não pode mais ser estendido, for atingido (nesse caso, não existe solução forte para $\mathcal{P}$).

FORTE($\mathcal{P}$)
1 $i \leftarrow 0$
2 $\pi_0 \leftarrow \emptyset$
3 `repita`
4 $\quad \mathcal{S}' \leftarrow S_g \cup \mathcal{S}_{\pi_i}$
5 $\quad$ `se` $S_0 \subseteq \mathcal{S}'$ `então devolva` DETERM(π_i)
6 $\quad \pi_{i+1} \leftarrow \pi_i \cup$ PODA(PREIMAGEMFORTE($\mathcal{S}'$), $\mathcal{S}'$)
7 $\quad i \leftarrow i + 1$
8 `até` $\pi_i = \pi_{i-1}$
9 `devolva falha`

Para eliminar da pré-imagem forte de $\mathcal{S}'$ os estados que já estavam em $\mathcal{S}'$ (aos quais já foram associadas ações em iterações anteriores), o algoritmo de planejamento FORTE utiliza a função PODA, definida como:

$$\text{PODA}(\pi, \mathcal{S}') = \{(s,a) \in \pi : s \notin \mathcal{S}'\} \tag{3.17}$$

Finalmente, para garantir que o plano encontrado seja determinístico, *i.e.*, que associe no máximo uma ação a cada estado, o algoritmo FORTE utiliza a função DETERM(π), que devolve uma política $\pi' \subseteq \pi$ tal que $\mathcal{S}_{\pi'} = \mathcal{S}_\pi$ e π' satisfaz a condição de que para cada estado $s \in \mathcal{S}_{\pi'}$ exista uma única ação associada.

Considere o exemplo apresentado na Figura 3.6, onde $S_0 = \{s_0\}$ e $S_g = \{s_4\}$. Na primeira iteração do algoritmo FORTE, temos:

$\mathcal{S}' = \{s_4\}$
PREIMAGEMFORTE($\mathcal{S}'$) = $\{(s_3, a_{34})\}$
PODA(PREIMAGEMFORTE($\mathcal{S}'$), $\mathcal{S}'$) = $\{(s_3, a_{34})\}$
$\pi_1 = \{(s_3, a_{34})\}$

[4] Na verdade PREIMAGEMFORTE resulta numa política, ou seja, um conjunto de pares (s,a), mas estamos nos referindo apenas aos estados mapeados por essa política.

Na segunda iteração, temos:

$\mathcal{S}' = \{s_3, s_4\}$
PreImagemForte$(\mathcal{S}') = \{(s_0, a_{04}), (s_3, a_{34})\}$
Poda(PreImagemForte$(\mathcal{S}'), \mathcal{S}') = \{(s_0, a_{04})\}$
$\pi_2 = \{(s_0, a_{04}), (s_3, a_{34})\}$

Finalmente, como na terceira iteração a condição $S_0 \subseteq (S_g \cup S_{\pi_2})$ é satisfeita, o algoritmo Forte devolve π_2 como resposta.

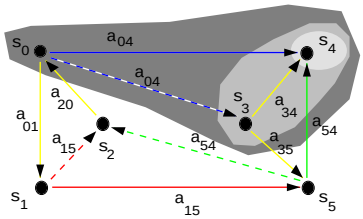


Figura 3.6: Funcionamento do algoritmo Forte, com $S_0 = \{s_0\}$ e $S_g = \{s_4\}$.

3.3.3 Planejamento fraco

O algoritmo de planejamento Fraco [Cimatti et al., 1997] é similar ao algoritmo de planejamento Forte, mas usa a função de pré-imagem fraca:

$$\text{PreImagemFraca}(\mathcal{S}') = \{(s, a) : \mathcal{T}(s, a) \cap \mathcal{S}' \neq \emptyset\} \tag{3.18}$$

Essa função resulta num conjunto de estados a partir dos quais é possível atingir estados em $\mathcal{S}'$; porém, não há garantia de que todas as transições realizadas a partir desses estados realmente levem a estados em $\mathcal{S}'$. É por esse motivo que estados terminais que não são estados metas (*e.g.*, estado s_2 na Figura 3.7) podem surgir no diagrama de transições induzido pelo plano encontrado.

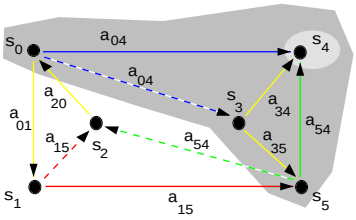


Figura 3.7: Funcionamento do algoritmo Fraco, com $S_0 = \{s_0\}$ e $S_g = \{s_4\}$.

3.3.4 Planejamento forte cíclico

O algoritmo de planejamento ForteCíclico [Daniele et al., 1999] inicia com uma política universal $\pi_0 = \{(s, a) : s \in \mathcal{S} \text{ e } a \in \mathcal{A}(s)\}$, contendo todos os pares *estado-ação* possíveis

no domínio considerado. Em seguida, iterativamente, vai eliminando os pares que levam a estados fora dessa política. A eliminação é implementada pelas funções PodaBeco, que remove todos os pares $(s, a) \in \pi_i$ a partir dos quais é impossível atingir um estado meta, e PodaFora, que remove todos os pares $(s, a) \in \pi_i$ que levam a estados fora de $S_g \cup \mathcal{S}_{\pi_i}$. Como PodaFora pode causar o surgimento de becos e PodaBeco pode causar o surgimento de transições para fora do conjunto $S_g \cup \mathcal{S}_{\pi_{i+1}}$, o algoritmo prossegue até que a convergência seja atingida. Nesse ponto, caso a política obtida tenha algum estado inicial para o qual não exista um caminho até um estado meta, o algoritmo devolve `falha`; caso contrário, as transições que geram ciclos não-triviais são removidas por PodaRetrocesso e a política determinística resultante é devolvida como solução.

```
ForteCíclico(P)
1  i ← 0
2  π_0 ← {(s, a) : s ∈ S e a ∈ A(s)}
3  repita
4    π_{i+1} ← PodaFora(PodaBeco(π_i, S_g), S_g)
5    i ← i + 1
7  até π_i = π_{i−1}
8  se S_0 ⊆ (S_g ∪ S_{π_i}) então devolva Determ(PodaRetrocesso(π_i, S_g))
9  devolva falha

PodaBeco(π, S_g)
1  i ← 0
2  π_0 ← ∅
3  repita
4    π_{i+1} ← π ∩ PreImagemFraca(S_g ∪ S_{π_i})
5    i ← i + 1
7  até π_i = π_{i−1}
8  devolva π_i

PodaFora(π, S_g)
1  devolva π − {(s, a) ∈ π : T(s, a) ⊈ S_g ∪ S_π}

PodaRetrocesso(π, S_g)
1  i ← 0
2  π_0 ← ∅
3  repita
4    S' ← S_g ∪ S_{π_i}
4    π_{i+1} ← π_i ∪ Poda(π ∩ PreImagemFraca(S'), S')
5    i ← i + 1
7  até π_i = π_{i−1}
8  devolva π_i
```

Na Figura 3.8, vemos o funcionamento de ForteCiclico para $S_0 = \{s_0\}$ e $S_g = \{s_5\}$. Primeiro o beco s_4, para o qual não há ação executável, é removido. Em seguida, todas as transições que levam a estados becos são eliminadas. Como esse problema é muito simples, o algoritmo termina na primeira iteração. Em problemas mais complexos, a eliminação de transições poderia causar o surgimento de novos becos e mais iterações seriam necessárias. Note que esse problema, apesar de simples, não tem solução forte.

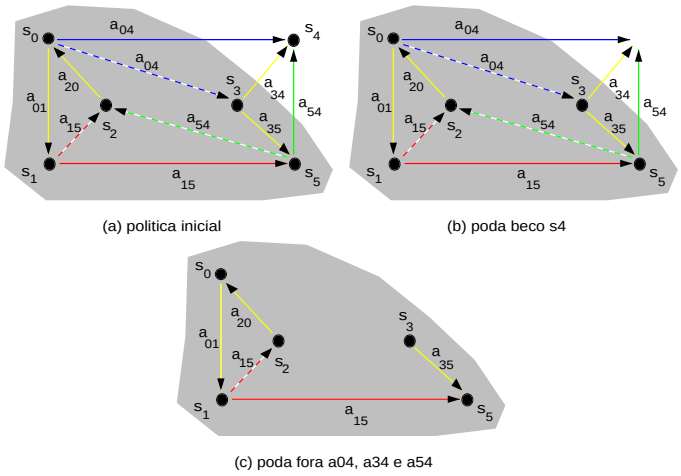


Figura 3.8: Funcionamento do algoritmo ForteCíclico, com $S_0 = \{s_0\}$ e $S_g = \{s_5\}$.

Considerações. Diferentemente do planejamento baseado em PDM, planejamento baseado em VM não necessita das distribuições de probabilidades associadas aos efeitos das ações não-determinísticas (que nem sempre estão disponíveis). Também, diferentemente do planejamento baseado em SAT, planejamento baseado em VM não está restrito a planejamento conformante (que nem sempre tem solução). De fato, planejamento baseado em VM tem se mostrado uma abordagem bastante eficiente para aplicações práticas em domínios onde há incerteza. Como os algoritmos de planejamento baseado em VM operam sobre conjuntos de estados e conjuntos de transições, em vez de estados únicos e transições únicas, eles levam vantagem sobre planejadores construídos dentro das abordagens PDM e SAT: quanto mais incerteza no domínio, mais rápidos eles ficam [Ghallab et al., 2004].

3.4 Planejamento para metas estendidas

Quando as ações têm efeitos incertos, a cada vez que um plano é executado, um caminho distinto pode ser seguido. Embora não possamos evitar que isso aconteça, podemos impor restrições sobre esses caminhos. Por exemplo, considere um problema em que um robô móvel precisa se deslocar até um certo local do seu ambiente. Evidentemente, qualquer plano que permita ao robô atingir o local desejado é uma solução para o problema. Agora, suponha que no ambiente existam locais de risco para o robô. Nesse caso, além de alcançar o local desejado, precisamos evitar que o robô seja danificado. Assim, uma solução para o problema seria um plano que garantisse ao robô alcançar o local desejado, sem nunca passar por locais de risco. Em várias situações práticas, porém, esse requisito poderia ser forte demais para ser satisfeito. Uma alternativa plausível seria exigir apenas que o plano permitisse ao robô alcançar o local desejado, sem garantia disso, mas com a garantia de que os locais de risco seriam evitados; ou ainda, que o robô alcançasse o local desejado, tentando evitar locais de risco sempre que possível.

Metas estendidas permitem especificar requisitos sobre os possíveis caminhos de um plano e são extremamente importantes em aplicações práticas de planejamento.

3.4.1 Planos para metas estendidas

Considere o cenário ilustrado na Figura 3.9. Suponha que a meta do robô seja "entregar um produto na loja-1, voltar ao depósito e entregar um outro produto na loja-2". Então, como ao estado "depósito" devem ser associadas duas ações distintas ("entregar na loja-1" e "entregar na loja-2"), não há política da forma $\pi : \mathcal{S} \mapsto \mathcal{A}$ que possa satisfazer essa meta. Um outro exemplo similar, porém mais interessante, é a meta "iniciando no depósito, mantenha-se entregando produtos nas lojas 1 e 2, alternadamente". Agora, a cada vez que o robô sai do depósito, ele precisa lembrar em que loja ele fez a última entregar e se dirigir para a outra. Esses dois exemplos mostram que, diferentemente do que acontece com os planos para metas simples (que decidem a ação em função apenas do estado corrente do ambiente), planos para metas estendidas também devem levar em conta o *contexto de execução* corrente, *i.e.* o estado interno do agente.

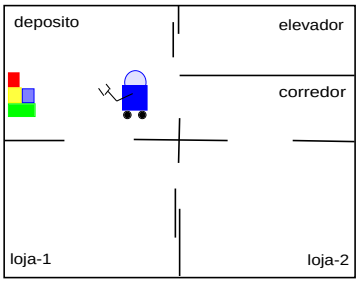


Figura 3.9: Cenário para o domínio do robô móvel.

Em geral, um plano para uma meta estendida pode ser definido em termos de uma *função de ação* que, dado um estado e um contexto de execução, codificando o estado interno do agente, especifica a ação a ser executada, e em termos de uma *função de contexto* que, dependendo do efeito da ação, especifica o próximo contexto de execução.

Definição 3.3. *Um plano estendido, sobre um domínio de planejamento $\mathcal{D}$, é uma tupla $\Pi = \langle \mathcal{C}, c_0, act, ctxt \rangle$, onde:*

- *$\mathcal{C}$ é um conjunto de contextos de execução;*
- *$c_o \in \mathcal{C}$ é o contexto inicial;*
- *$act : \mathcal{S} \times \mathcal{C} \mapsto \mathcal{A}$ é a função de ação;*
- *$ctxt : \mathcal{S} \times \mathcal{C} \times \mathcal{S} \mapsto \mathcal{C}$ é a função de contexto.* ■

Se estamos num estado s e o contexto de execução é c, então $act(s, c)$ devolve a ação que deve ser executada pelo agente, enquanto $ctxt(s, c, s')$ associa a cada estado s' atingido o novo contexto de execução. Isso permite associar diferentes ações a um mesmo estado, dependendo do contexto considerado. As funções *act* e *ctxt* são parciais, já que alguns estados e contextos nunca são atingidos durante a execução do plano. Por exemplo, um plano estendido para a meta "iniciando no depósito, mantenha-se entregando produtos nas lojas 1 e 2, alternadamente" poderia ser o seguinte:

estado	*contexto*	*ação*	*próximo estado*	*próximo contexto*
depósito	c_1	*entregar na loja 1*	*loja 1*	c_2
depósito	c_2	*entregar na loja 2*	*loja 2*	c_1
loja 1	c_2	*voltar*	*depósito*	c_2
loja 2	c_1	*voltar*	*depósito*	c_1

Dizemos que um plano Π é *executável* se, sempre que $act(s,c) = a$ e $ctxt(s,a,s') = c'$, temos $s' \in \mathcal{T}(s,a)$. Dizemos que um plano é *completo* se, sempre que $act(s,a) = a$ e $s' \in \mathcal{T}(s,a)$, existe um contexto c' tal que $ctxt(s,c,s') = c'$ e $act(s',c')$ está definida. Intuitivamente, um plano completo sempre especifica como proceder, em qualquer um dos possíveis estados sucessores que podem ser atingidos por qualquer uma das ações no plano. Soluções para metas estendidas são planos executáveis e completos.

A execução de um plano estendido resulta em mudanças de estados e contextos. Assim, podemos descrevê-la em termos de transições entre pares *estado-contexto*. Formalmente, dado um domínio $\mathcal{D}$ e um plano Π, uma transição de Π em $\mathcal{D}$ é uma tupla $(s,c) \xrightarrow{a} (s',c')$ tal que $act(s,c) = a$, $s' \in \mathcal{T}(s,a)$ e $ctxt(s,c,s') = c'$. Uma *trajetória* de um plano Π, a partir de s_0, é uma seqüência infinita $(s_0,c_0) \xrightarrow{a_0} (s_1,c_1) \xrightarrow{a_1} (s_2,c_2) \xrightarrow{a_0} (s_3,c_3)\ldots$, em que $(s_i,c_i) \xrightarrow{a_i} (s_{i+1},c_{i+1})$ são transições.

Devido ao não-determinismo, um plano pode ter várias trajetórias. O conjunto dessas trajetórias pode ser representado pela estrutura de Kripke induzida pelo plano, ou seja, a estrutura de Kripke cujo conjunto de estados é o conjunto de pares *estado-contexto* e cujo conjunto de transições corresponde às transições das trajetórias. Tal estrutura de Kripke é também denominada *estrutura de execução* do plano Π.

Definição 3.4. *A estrutura de execução de um plano Π num domínio $\mathcal{D}$, a partir do estado s_0, é a estrutura de Kripke $\mathcal{K}_\Pi = \langle \mathcal{S}_\Pi, \mathcal{T}_\Pi, \mathcal{L}_\Pi \rangle$, onde:*

- $\mathcal{S}_\Pi = \{(s,c) : act(s,c) \textit{é definida}\}$;
- $\mathcal{T}_\Pi = \{((s,c),(s',c')) : (s,c) \xrightarrow{a} (s',c') \textit{para alguma ação } a\}$;
- $\mathcal{L}_\Pi(s,c) = \{p : p \in \mathcal{L}(s)\}$. ■

Definimos quando uma meta φ é satisfeita em (s,c), denotado por $(\mathcal{K}_\Pi,(s,c)) \models \varphi$, usando a semântica padrão para fórmulas CTL sobre a estrutura de Kripke $\mathcal{K}_\Pi$ (seção 2.2.2). Por exemplo, $(\mathcal{K}_\Pi,(s_0,c_0)) \models \forall(\varphi_1 \cup \varphi_2)$ se, para todo caminho $\langle (s_0,c_0) = (s,c)_0,(s,c)_1,(s,c)_2,\ldots\rangle$, existe $j \geq 0$ tal que $(\mathcal{K}_\Pi,(s,c)_j) \models \varphi_2$ e, para todo $i < j$, $(\mathcal{K}_\Pi,(s,c)_i) \models \varphi_1$.

Definição 3.5. *Sejam $\mathcal{D}$ um domínio de planejamento e φ uma meta em $\mathcal{D}$. Sejam Π um plano e $\mathcal{K}_\Pi$ a estrutura de execução desse plano. O plano Π satisfaz a meta φ, a partir do estado inicial s_0, denotado por $(\Pi,s_0) \models \varphi$, se $(\mathcal{K}_\Pi,(s_0,c_0)) \models \varphi$. O plano Π satisfaz a meta φ, a partir do conjunto de possíveis estados iniciais S_0, denotado por $(\Pi,S_0) \models \varphi$, se $(\mathcal{K}_\Pi,(s,c_0) \models \varphi$, para cada $s \in S_0$.* ■

3.4.2 Metas estendidas temporais

Metas estendidas temporais são metas especificadas em CTL, que impõem restrições sobre os possíveis caminhos de uma política.

Planejamento para metas estendidas temporais pode ser implementado diretamente como uma busca progressiva no diagrama de transições do domínio. Por exemplo, se a meta for $\forall(p \cup q)$, podemos progredi-la para $q \vee \forall \odot (p \wedge \forall(p \cup q))$. Caso o estado corrente não satisfaça q, selecionamos uma ação aplicável ao estado corrente e, para cada estado sucessor obtido pela aplicação dessa ação, verificamos p e chamamos o procedimento recursivamente. Porém, busca progressiva com estados explícitos não é muito eficiente na prática [Ghallab et al., 2004]. Uma alternativa melhor é usar técnicas de verificação de modelos simbólicos, que manipulam conjuntos em vez de estados únicos.

O algoritmo apresentado em [Pistore and Traverso, 2001] utiliza a idéia de *autômato de controle* para dirigir a busca no espaço de estados do problema de planejamento. Esse autômato é construído a partir da progressão da meta especificada em CTL.

Definição 3.6. *A progressão de uma fórmula* CTL φ*, a partir de um estado* s*, denotada por* $progr(\varphi, s)$*, é definida como:*

- $progr(c, s) = c$ *para* $c \in \{\top, \bot\}$*;*
- $progr(p, s) = p$*;*
- $progr(\neg p, s) = \neg p$*;*
- $progr(\varphi_1 \wedge \varphi_2, s) = progr(\varphi_1, s) \wedge progr(\varphi_2, s)$*;*
- $progr(\varphi_1 \vee \varphi_2, s) = progr(\varphi_1, s) \vee progr(\varphi_2, s)$*;*
- $progr(\forall \odot \varphi, s) = \forall \odot \varphi$*;*
- $progr(\exists \odot \varphi, s) = \exists \odot \varphi$*;*
- $progr(\forall(\varphi_1 \cup \varphi_2)) = (progr(\varphi_1, s) \wedge \forall \odot \forall(\varphi_1 \cup \varphi_2)) \vee progr(\varphi_2, s)$*;*
- $progr(\exists(\varphi_1 \cup \varphi_2)) = (progr(\varphi_1, s) \wedge \exists \odot \exists(\varphi_1 \cup \varphi_2)) \vee progr(\varphi_2, s)$. ■

Uma meta especificada em CTL estabelece condições sobre o estado corrente, bem como sobre estados futuros. Intuitivamente, se φ deve ser satisfeita num estado s, então algumas condições devem sem projetadas aos estados sucessores de s. Obtemos essas condições "progredindo" a fórmula φ. No autômato de controle construído a partir da progressão da meta, cada estado corresponde a uma submeta a ser atingida, que torna-se um contexto no plano. Arcos entre estados do autômato de controle determinam quando a busca deve comutar de um contexto para outro.

O único artigo que encontramos na literatura da área que descreve planejamento para metas estendidas temporais em CTL [Pistore and Traverso, 2001] apresenta o seguinte esquema de planejador:

```
PLANEJADOR(φ)
1  aut ←CONSTRÓIAUTÔMATO(φ)
2  assoc ←ASSOCIAESTADOS(aut)
3  π ←EXTRAIPLANO(aut, assoc)
4  devolva π
```

O algoritmo CONSTRÓIAUTÔMATO cria um autômato, usando a idéia de progressão, que irá controlar a busca. O algoritmo ASSOCIAESTADOS explora o espaço de busca, associando um conjunto de estados do espaço de busca a cada estado do autômato. Finalmente, o algoritmo EXTRAIPLANO contrói um plano explorando as informações sobre os estados associados aos contextos representados pelos estados do autômato.

Infelizmente, o artigo não mostra muitos detalhes sobre a implementação desses algoritmos. [Ghallab et al., 2004] também cita esses algoritmos, mas apresenta apenas um exemplo de construção do autômato para uma fórmula específica.

3.4.3 Metas estendidas procedimentais

Considere a seguinte meta de planejamento: *tente alcançar p; se falhar, alcance q.* Não há fórmula CTL (nem LTL) capaz de expressar essa meta [Ghallab et al., 2004]. Um plano que satisfaça a fórmula $\exists\Diamond p$ não garante que o agente fará o possível para atingir

p, mas apenas que ele terá uma chance de atingir p. Além disso, não há como expressar o significado de "*se falhar*". Uma fórmula que tentasse expressar essa idéia usando disjunção (implicação) não poderia impedir que o planejador encontrasse uma solução que ignorasse completamente a primeira parte da meta, satisfizendo apenas a segunda.

A linguagem EaGle [Lago et al., 2002] visa justamente expressar metas estendidas procedimentais. Essa linguagem permite as seguintes construções:

- Alcance: `DoReach` p, `TryReach` p
- Manutenção: `DoMaintain` p, `TryMaintain` p
- Conjunção: g `And` g'
- Falha: g `Fail` g'
- Controle: g `Then` g', `Repeat` g

A meta "`DoReach` p" requer um plano que garanta alcançar p, enquanto a meta "`TryReach` p" requer um plano que faça o máximo para alcançar p. Analogamente, a meta "`DoMaintain` p" requer um plano que garanta manter p, enquanto a meta "`TryReach` p" requer um plano que faça o máximo para manter p. A meta "p `Fail` q" serve para lidar com casos de falha e recuperação, especificando p como submeta preferencial. Por exemplo, para a meta "`TryReach` p `Fail DoReach` q", o planejador deve satisfazer "`DoReach` q" apenas nos estados em que "`TryReach` p" falha. A meta "p `And` q" requer um plano que satisfaça simultaneamente p e q; a meta "p `Then` q" requer um plano que satisfaça p e, em seguida, q; finalmente, a meta "`Repeat` p" requer um plano que satisfaça p ciclicamente.

A exemplo do que ocorre com o algoritmo para metas estendidas temporais em CTL, o algoritmo para metas estendidas procedimentais em EaGle também utiliza um autômato de controle e sua implementação não é detalhada no artigo.

3.5 Planejadores baseados em verificação de modelos

Nessa seção, a título de exemplo, apresentamos três planejadores baseados em verificação de modelos bem sucedidos.

3.5.1 MBP

MBP (*Model Based Planner*) [Cimatti et al., 1997] é um planejador para domínios não-determinísticos, cuja principal contribuição está em proporcionar um arcabouço bastante genérico para planejamento baseado em verificação de modelos, que nos permite tratar problemas de diversos tipos.

Através de uma extensão da linguagem PDDL [McDermott, 1998], denominada NuPDDL, o MBP nos permite modelar incerteza sobre o estado inicial, sobre os efeitos das ações e sobre os estados nos quais as ações podem ser executadas. Para isso, NuPDDL introduz as palavras-chaves `oneof`, `unknown` e `observation`. Por exemplo, uma cláusula da forma (`oneof` $p_1\ p_2 \ldots p_n$) especifica que uma das proposições p_i deve ser verdadeira, uma cláusula da forma (`unknown` p) especifica que a proposição p tem valor desconhecido e uma cláusula da forma (: `observation` p − `boolean`) especifica que o valor da proposição p depende de observação do estado corrente do ambiente. NuPDDL nos permite ainda especificar a qualidade do plano a ser obtido, através do uso das seguintes palavras-chaves:

- `weakgoal`: plano fraco, com observabilidade completa.

- `stronggoal`: plano forte, com observabilidade completa.
- `strongcyclicgoal`: plano forte cíclico, com observabilidade completa.
- `postronggoal`: plano forte com observabilidade parcial.
- `conformantgoal`: plano conformante (observabilidade nula).
- `ctlgoal`: plano satisfazendo uma meta especificada em lógica CTL.

Além disso, NuPDDL inclui também um conjunto de palavras-chaves (e.g. `sequence`, `case` e `while`) que possibilitam a representação de planos seqüenciais, condicionais e iterativos [Bertoli et al., 2001].

3.5.2 MIPS

MIPS (*Model-checking Integrated Planning System*) [Edelkamp and Reffel, 1999] é um planejador baseado em verificação de modelos, capaz de resolver problemas de planejamento em domínios determinísticos. Foi um dos primeiros planejadores dentro dessa abordagem a se mostrar competitivo com planejadores baseados em satisfatibilidade. Sendo que sua eficiência deve-se, principalmente, à sua fase de preprocessamento, capaz de criar representações de estados extremamente concisas, a partir de inferências realizadas sobre o conhecimento implícito na descrição dos domínios de planejamento.

Na sua versão original, o planejador MIPS era capaz de tratar apenas o subconjunto STRIPS [Fikes and Nilsson, 1990] da linguagem PDDL. Tendo sido, mais tarde, estendido para tratar também algumas características adicionais da linguagem ADL [Pednault, 1989], tais como precondições negativas e efeitos condicionais. Na sua versão atual, MIPS inclui ainda algoritmos de busca heurística e é capaz de tratar problemas com recursos numéricos, ações durativas e otimização de objetivos [Edelkamp and Helmert, 2001].

3.5.3 UMOP

UMOP (*Universal Multi-agent Obdd-based Planner*) [Jensen and Veloso, 1999] é um planejador capaz de tratar problemas em domínios não-determinísticos e multi-agentes. Dado um problema de planejamento especificado em NADL, a linguagem de entrada do UMOP, o planejador o transforma em um problema de busca numa estrutura de Kripke. Tal estrutura é representada por uma relação de transição particionada e codificada simbolicamente por um conjunto de diagramas de decisão binária [Bryant, 1992], o que torna o processo extremamente eficiente.

O planejador UMOP implementa cinco algoritmos de planejamento: forte, forte cíclico [Cimatti et al., 1998], otimista [Jensen and Veloso, 2000], adversarial forte cíclico e adversarial otimista [Jensen et al., 2001]. Todos esses algoritmos baseiam-se numa busca regressiva em largura, usando eficientes operações para geração da pré-imagem de um dado conjunto de estados. Alem disso, devido à busca em largura, todos os planos obtidos pelo UMOP são ótimos com relação ao número de passos esperados.

3.6 Sumário

Nesse capítulo, apresentamos planejamento não-determinístico dentro a abordagem de verificação de modelos (simbólicos). Também descrevemos os principais algoritmos encontrados na literatura da área, tanto para metas simples quanto estendidas (temporais

e procedimentais). Finalizamos descrevendo três dos mais bem sucedidos planejadores baseados em verificação de modelos, sendo que apenas dois deles consideram ambientes não-determinísticos.

Capítulo 4

Proposta da Tese

4.1 Validação *vs.* síntese

Sejam $\mathcal{D}$ o diagrama de transições de um domínio de planejamento (Figura 4.1) e $\mathcal{K}$ a estrutura de Kripke correspondente. Considere a fórmula $\forall(p \cup q)$ especificando uma meta de planejamento nesse domínio. Então, como a semântica de CTL não leva em conta a identidade das transições, temos que $(\mathcal{K}, s_0) \not\models \forall(p \cup q)$; ou seja, nem todo caminho partindo de s_0 em $\mathcal{K}$ mantém p até atingir q.

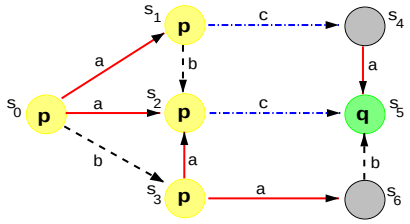


Figura 4.1: Diagrama de transições para um domínio de planejamento.

Considere agora a política $\pi = \{(s_0, a), (s_1, b), (s_2, c)\}$, para o domínio na Figura 4.1. Sejam $\mathcal{D}_\pi \subset \mathcal{D}$ o diagrama de transições induzido por π (Figura 4.2) e $\mathcal{K}_\pi$ a estrutura de Kripke correspondente. Claramente, temos que $(\mathcal{K}_\pi, s_0) \models \forall(p \cup q)$; ou seja, todo caminho partindo de s_0 em $\mathcal{K}_\pi$ mantém p até atingir q.

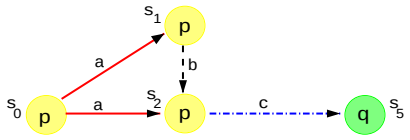


Figura 4.2: Diagrama de transições induzido pela política π.

Como podemos notar, a semântica de CTL permite validar a propriedade $\forall(p \cup q)$ no modelo $\mathcal{D}_\pi$; mas não no modelo $\mathcal{D}$, do qual $\mathcal{D}_\pi$ é um subgrafo. Por esse motivo, sob um

ponto de vista formal, podemos dizer que os trabalhos na área de planejamento baseado em VM utilizam a lógica CTL apenas para *validar planos* e não para *sintetizar planos*, que é o principal objetivo da área de planejamento automatizado.

4.2 A lógica temporal α-CTL

Nesse trabalho, desejamos estender a sintaxe e a semântica de CTL, de modo que possamos usar lógica temporal não apenas para formalizar a validação de planos, como temos observado na literatura ([Cimatti et al., 1997] [Cimatti et al., 1998] [Daniele et al., 1999]), mas, principalmente, para formalizar a sua síntese.

Para tanto, propomos uma lógica temporal que denominamos α-CTL. Essencialmente, essa lógica estende CTL incluindo *quantificadores qualificados*, denotados por $\forall_\alpha$ e $\exists_\alpha$. Como estamos interessados numa semântica que permita sintetizar planos como efeito colateral da verificação de um modelo, as ações que qualificam esses quantificadores devem ser existenciais. Intuitivamente, a fórmula $\forall_\alpha \odot p$ vale num estado s se existe uma ação α que, quando executada em s, sempre leva a um estado sucessor em que p vale; enquanto a fórmula $\exists_\alpha \odot p$ vale num estado s se existe uma ação α que, quando executada em s, às vezes, leva a um estado sucessor em que p vale. Esses novos quantificadores devem ser interpretados com base numa *estrutura de Kripke estendida*, cujas transições são rotuladas por ações.

Para manter a consistência da semântica desejada para os quantificadores qualificados, precisamos tratar a negação em α-CTL com um cuidado especial. Observe que, enquanto as fórmulas CTL $\forall \odot p$ e $\neg\exists \odot \neg p$ são equivalentes, as fórmulas α-CTL $\forall_\alpha \odot p$ e $\neg\exists_\alpha \odot \neg p$ não o são. Por exemplo, o estado s na estrutura de Kripke estendida apresentada na Figura 4.3 satisfaz a fórmula $\forall_\alpha \odot p$ (pois a ação a sempre leva a um estado em que p vale), mas não satisfaz $\neg\exists_\alpha \odot \neg p$ (pois existem ações que, às vezes, levam a estados em que p não vale). Por esse motivo, em α-CTL, a negação deve ser restrita às proposições atômicas e devemos usar o operador $\mathbin{\mathbb{W}}$ (*até fraco*) como dual do operador $\cup$ (*até forte*). Assim, por exemplo, a fórmula $\forall_\alpha \Box p$ poderá ser escrita como $\forall_\alpha(p \mathbin{\mathbb{W}} \bot)$, mas não como $\neg\exists_\alpha \Diamond \neg p$ (que equivale a $\neg\exists_\alpha(\top \cup \neg p)$). Parte do trabalho na definição da nova semântica será verificar que equivalências de CTL são mantidas em α-CTL.

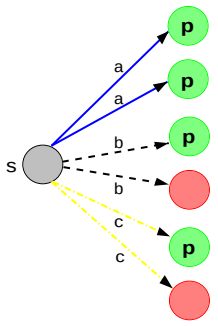


Figura 4.3: Estrutura de Kripke estendida que satisfaz $\forall_\alpha \odot p$, mas não $\neg\exists_\alpha \odot \neg p$.

4.3 Tratamento uniforme de metas

Também constatamos que os trabalhos na área de planejamento baseado em verificação de modelos distinguem metas simples e estendidas, dando a cada um desses tipos de meta um tratamento bastante diferenciado [Ghallab et al., 2004].

Em geral, metas simples são tratadas por algoritmos que usam *pré-imagem* para conduzir uma busca regressiva no espaço de estados do problema (*model-theoretic*); por outro lado, metas estendidas são tratadas por algoritmos que constróem autômatos, a partir de fórmulas em lógica temporal, e os utilizam para controlar uma busca progressiva no espaço de estados do problema (*automata-theoretic*).

Além disso, metas simples são especificadas por fórmulas proposicionais; enquanto metas estendidas são especificadas por fórmulas temporais (LTL ou CTL), ou ainda por sentenças em EAGLE [Lago et al., 2002], uma linguagem procedimental que estende a lógica CTL de uma forma "extra-lógica".

Definindo planejamento como verificação de modelos em α-CTL, acreditamos que poderemos dar um tratamento mais uniforme a metas simples e estendidas (não apenas usando a mesma linguagem para especificação dessas metas, mas também empregando o mesmo formalismo e as mesmas técnicas para sintetizar planos que as satisfaçam). Por exemplo, poderemos usar as fórmulas $\forall_\alpha \Diamond q$ (*alcance* q) e $\exists_\alpha \Diamond q$ (*tente alcançar* q) para especificar *metas de alcance*; bem como as fórmulas $\forall_\alpha \Box p$ (*mantenha* p) e $\exists_\alpha \Box p$ (*tente manter* p) para especificar *metas de manutenção*. Ou ainda, usando os operadores $\cup$ e $\mathbin{\mathbb{W}}$, poderemos escrever $\forall_\alpha (p \cup q)$ (*mantenha* p *até alcançar* q), $\exists_\alpha (\top \cup q)$ (*tente alcançar* q) e $\forall_\alpha (p \mathbin{\mathbb{W}} \bot)$ (*mantenha* p).

4.4 Algoritmo para verificação de fórmulas α-CTL

Uma vez que tenhamos definido precisamente a sintaxe e a semântica de α-CTL, vamos criar os algoritmos necessários para verificação de modelos com relação a propriedades especificadas nessa lógica. Para isso, podemos nos inspirar nos algoritmos existentes para verificação de fórmulas em CTL. Também vamos definir os algoritmos para verificação de fórmulas em α-CTL em termos de operações com diagramas de decisão binária ordenados e reduzidos, isso possibilitará uma implementação mais eficiente dos algoritmos propostos.

A partir daí, mostraremos que planos podem ser sintetizados como resultado da verificação de modelos usando α-CTL e que a validade desses planos é conseqüência imediata de um processo de síntese bem formalizado.

4.5 Objetivos e cronograma da pesquisa

O principal objetivo da tese será formalizar planejamento em termos de verificação de modelos, com relação a propriedades especificadas em α-CTL. Isso significa empregar o arcabouço de verificação de modelos não apenas para validação de planos mas, sobretudo, para síntese de planos. Como conseqüência dessa formalização, obteremos algoritmos capazes de tratar, de maneira uniforme e consistente, metas simples e estendidas.

Essa abordagem formal e uniforme que propomos tem pelo menos duas vantagens importantes com relação aos trabalhos encontrados na literatura:

- permite tratar meta simples como caso especial de meta estendida (ou seja, podemos especificá-las na mesma linguagem e resolvê-las com os mesmos algoritmos);
- permite que o mesmo formalismo usado para a validação de planos seja também usado para a síntese (essa tem sido a maior deficiência dos trabalhos encontrados na literatura, que propõem métodos *ad-hoc* para a síntese de planos e usam lógica temporal apenas para validá-los); de fato, com um processo de síntese bem formalizado, a validação dos planos torna-se desnecessária.

Como objetivos secundários da tese, exploraremos ainda os seguintes aspectos:

- como estender o formalismo proposto para tratar problemas de planejamento não-determinístico em domínios cujas ações têm duração e cujas submetas têm prazos? é possível tratar produção e consumo de recursos discretos? que tipos de otimização podem ser realizadas?
- um conjunto de axiomas de restrição de estados (dado juntamente com a descrição de domínio ou dela extraído de forma automática) poderia tornar o planejamento mais eficiente? analisando as descrições de meta e de domínio, para um dado problema de planejamento, seria possível obter um codificação de estado (com menos proposições) que reduzisse o espaço de estados explorado durante a busca?

A seguir, apresentamos o cronograma com as atividades necessárias para atingir os objetivos da tese:

- *Relação das atividades*:

  1. Definir precisamente a sintaxe e a semântica da lógica α-CTL.
  2. Obter uma caracterização de ponto-fixo para os operadores de α-CTL.
  3. Criar algoritmos para verificação de propriedades especificadas em α-CTL.
  4. Adaptar os algoritmos de verificação para α-CTL para sintetizar planos.
  5. Implementar os algoritmos usando técnicas simbólicas (DDBOR).
  6. Criar um domínio de planejamento para testar metas simples e estendidas.
  7. Redação da tese.
  8. Defesa da tese.

- *Plano de trabalho*:

Atividade	Jul	Ago	Set	Out	Nov	Dez	Jan	Fev	Mar	Abr	Mai	Jun
1	x	x	x									
2		x	x	x								
3			x	x	x							
4				x	x	x	x					
5					x	x	x	x	x			
6			x	x	x	x						
7						x	x	x	x	x	x	
8												x

Referências Bibliográficas

[Andersen, 1997] Andersen, H. R. (1997). Introduction to binary decision diagrams.

[Bertoli et al., 2001] Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., and Traverso, P. (2001). Mbp: a model based planner.

[Bertoli and Pistore, 2004] Bertoli, P. and Pistore, M. (2004). Planning with extended goals and partial observability. In *ICAPS*, pages 270–278.

[Boutilier et al., 1999] Boutilier, C., Dean, T., and Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94.

[Brafman and Hoffmann, 2004] Brafman, R. I. and Hoffmann, J. (2004). Conformant planning via heuristic forward search: A new approach. In *ICAPS*, pages 355–364.

[Bryant, 1992] Bryant, R. E. (1992). Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.

[Bryce and Kambhampati, 2004] Bryce, D. and Kambhampati, S. (2004). Heuristic guidance measures for conformant planning. In *ICAPS*, pages 365–375.

[Bylander, 1994] Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204.

[Cimatti et al., 1997] Cimatti, A., Giunchiglia, F., Giunchiglia, E., and Traverso, P. (1997). Planning via model checking: A decision procedure for $\mathcal{AR}$. In *ECP*, pages 130–142.

[Cimatti et al., 1998] Cimatti, A., Roveri, M., and Traverso, P. (1998). Strong planning in non-deterministic domains via model checking. In *Artificial Intelligence Planning Systems*, pages 36–43.

[Clarke and Emerson, 1982] Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK. Springer-Verlag.

[Daniele et al., 1999] Daniele, M., Traverso, P., and Vardi, M. Y. (1999). Strong cyclic planning revisited. In *ECP*, pages 35–48.

[Dolgov and Durfee, 2004] Dolgov, D. A. and Durfee, E. H. (2004). Optimal resource allocation and policy formulation in loosely-coupled markov decision processes. In *ICAPS*, pages 315–324.

[Edelkamp and Helmert, 2001] Edelkamp, S. and Helmert, M. (2001). The model checking integrated planning system MIPS.

[Edelkamp and Reffel, 1999] Edelkamp, S. and Reffel, F. (1999). Deterministic state space planning with BDDs. Technical report.

[Emerson and Halpern, 1986] Emerson, E. A. and Halpern, J. Y. (1986). "sometimes" and "not never" revisited: On branching versus linear time temporal logic. In *Journal of the ACM*, volume 33(1).

[Fikes and Nilsson, 1990] Fikes, R. E. and Nilsson, N. J. (1990). Strips: A new approach to the application of theorem proving to problem solving. In Allen, J., Hendler, J., and Tate, A., editors, *Readings in Planning*, pages 88–97. Kaufmann, San Mateo, CA.

[Ghallab et al., 2004] Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Giunchiglia et al., 1997] Giunchiglia, E., Kartha, G. N., and Lifschitz, V. (1997). Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95(2):409–438.

[Jensen and Veloso, 2000] Jensen, R. and Veloso, M. (2000). OBDD-based universal planning for multiple synchronized agents in non-deterministic domains,. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-00)*, pages 167–176. AAAI Press.

[Jensen and Veloso, 1999] Jensen, R. M. and Veloso, M. M. (1999). OBDD-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. *Lecture Notes in Computer Science*, 1600:213–??

[Jensen et al., 2001] Jensen, R. M., Veloso, M. M., and Bowling, M. H. (2001). Obdd-based optimistic and strong cyclic adversarial planning. In *Proceedings of the 6th European Conference on Planning (ECP-01)*.

[Jensen et al., 2004] Jensen, R. M., Veloso, M. M., and Bryant, R. E. (2004). Fault tolerant planning: Toward probabilistic uncertainty models in symbolic non-deterministic planning. In *ICAPS*, pages 335–344.

[Kautz and Selman, 1992] Kautz, H. and Selman, B. (1992). Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363.

[Lago et al., 2002] Lago, U. D., Pistore, M., and Traverso, P. (2002). Planning with a language for extended goals. In *Eighteenth national conference on Artificial intelligence*, pages 447–454, Menlo Park, CA, USA. American Association for Artificial Intelligence.

[McDermott, 1998] McDermott, D. (1998). Pddl — the planning domain definition language.

[McMillan, 1992] McMillan, K. L. (1992). Symbolic model checking: An approach to the state explosion problem. In *PhD thesis, CMU*.

[Pednault, 1989] Pednault, E. P. D. (1989). Adl: Exploring the middle ground between strips and the situation calculus. In *KR*, pages 324–332.

[Pistore and Traverso, 2001] Pistore, M. and Traverso, P. (2001). Planning as model checking for extended goals in non-deterministic domains. In *IJCAI*, pages 479–486.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *Proc. 18th Symp. Foundations of Computer Science*, pages 46–57.

[Schnoebelen, 2002] Schnoebelen, P. (2002). The complexity of temporal logic model checking.

[Shanahan, 1997] Shanahan, M. (1997). *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT Press, Cambridge, MA, USA.

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. In *Pacific J. Math.*, volume 5, pages 285–309.

[Vardi, 1998a] Vardi, M. Y. (1998a). Linear vs. branching time: A complexity-theoretic perspective. In *Logic in Computer Science*, pages 394–405.

[Vardi, 1998b] Vardi, M. Y. (1998b). Sometimes and not never re-revisited: On branching versus linear time. In *International Conference on Concurrency Theory*, pages 1–17.

[Weld, 1994] Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4):27–61.

Índice Remissivo