UNIVERSIDADE DO MINHO

Escola de Engenharia

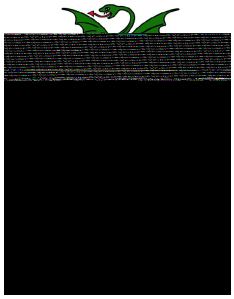
Departamento de Informática

Programação com Funções de Utilidade

Provas de Aptidão Pedagógica e Capacidade Científica

de

Manuel Alcino Pereira da Cunha



Componente Científica

Braga, 2001

UNIVERSIDADE DO MINHO

Escola de Engenharia

Departamento de Informática

Programação com Funções de Utilidade

Provas de Aptidão Pedagógica e Capacidade Científica

de

Manuel Alcino Pereira da Cunha

Prova submetida à Universidade do Minho, para acesso à categoria de Assistente, elaborada sob a orientação do Doutor José Bernardo dos Santos Monteiro Vieira Barros.

Componente Científica

Braga, 2001

Resumo

Nos últimos anos tem-se notado na área da inteligência artificial um interesse crescente pelo conceito de agente inteligente. Uma classe muito particular de agentes inteligentes são os denominados agentes orientados por uma função utilidade, que são entidades que agem com o objectivo de maximizar o seu "bem-estar", avaliado explicitamente através de uma relação de preferência definida sobre os seus estados. Esta tese apresenta um estudo sobre a possibilidade de se desenvolver uma linguagem de programação de alto nível para implementar este tipo de agentes. A estratégia seguida consistiu em adaptar uma linguagem já existente para especificação de programas concorrentes, permitindo a parametrização de cada processo com uma função definida sobre as suas variáveis locais com o objectivo de representar as suas preferências. Grande parte da tese é dedicada ao estudo de técnicas para transformar programas sequenciais escritos nesta linguagem em programas com um comportamento equivalente escritos na linguagem original, onde poderiam ser compilados e executados usando técnicas habituais. Esta transformação levanta alguns problemas interessantes, na sua maioria derivados da necessidade de se comparar a utilidade de execuções infinitas. Os diversos modelos e técnicas que inspiraram a solução obtida são originários de áreas tão diversas como a investigação operacional, os métodos formais e, naturalmente, a inteligência artificial. Como complemento deste estudo foi concebida uma pequena aplicação onde são testadas as técnicas desenvolvidas.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Na área da inteligência artificial não tem existido um consenso quanto aos objectivos que se devem procurar atingir ao desenvolver um sistema dito inteligente. Nomeadamente, é possível identificar quatro grandes categorias para estes objectivos [RN95]: desenvolver sistemas que pensam como humanos, sistemas que agem como humanos, sistemas que pensam racionalmente ou sistemas que agem racionalmente.

A diferença entre uma abordagem centrada no comportamento humano ou num comportamento dito racional, é que na segunda é possível definir com clareza o que se entende por comportamento ou forma de pensar ideal, enquanto que na primeira a noção de comportamento ideal comporta muitas vezes uma certa dose de "irracionalidade" e varia muito de acordo com o indivíduo. Por esta razão, a comunidade da inteligência artificial está cada vez mais relutante em tentar compreender e reproduzir um comportamento ou forma de pensar muitas vezes imperfeito e irracional, estando a investigação centrada no comportamento humano cada vez mais relegada para áreas específicas, tais como a ciência cognitiva ou o processamento de linguagem natural. Esta especialização também se tem verificado na investigação do pensamento racional, que cada vez tem sido mais relacionada com o estudo da lógica.

Actualmente, grande parte dos investigadores preocupam-se em desenvolver agentes ditos inteligentes, que são entidades que actuam num determinado ambiente por forma a atingir os seus objectivos. Muito deste trabalho está enquadrado na última categoria referida no início, a dos sistemas que agem racionalmente, pois os objectivos dos agentes estão normalmente especificados de forma muito clara e precisa. Obviamente, estas áreas não são estanques e todo o restante trabalho contribui fortemente para o desenvolvimento destes agentes. No entanto, o objectivo já não é tanto perceber como se processam as inferências lógicas ou a forma de pensar humana, mas usar todo esse conhecimento no desenvolvimento de uma entidade de software útil, cujo comportamento possa ser avaliado com exactidão.

Existem muitas formas de definir com exactidão os objectivos de um agente inteligente. A mais usual consiste em definir um ou vários estados do mundo que o agente deve procurar atingir. Embora poderoso, por vezes este método não é suficientemente flexível para originar um comportamento de qualidade.

Os objectivos expressos desta forma apenas distinguem os estados entre bons e maus, sendo por vezes necessário ter uma graduação de valores mais ampla. Por exemplo, normalmente existem muitas formas de conduzir um automóvel até a um destino desejado. Alguns caminhos poderão ser rápidos, mas outros mais económicos ou mais aprazíveis. Todos nos levam ao nosso objectivo, mas a satisfação que retiramos de cada um deles poderá ser diferente. Normalmente, não os dividimos apenas entre os que gostamos e os que não gostamos, mas estabelecemos uma ordem de preferências entre eles. Esta ordem permite-nos escolher o mais satisfatório para prosseguir, mas se este estiver impedido também facilita a escolha de outro, proporcionando simultaneamente um decréscimo gradual do nível de satisfação.

Este tipo de raciocínio pode ser conseguido se os objectivos forem especificados através de uma função de utilidade. Esta função atribui a cada estado do mundo um valor numérico, que representa o nível de satisfação que esse estado proporciona ao agente. Os agentes cujos objectivos são representados por uma função de utilidade serão denominados agentes orientados por utilidade. Definidos os objectivos desta forma, o mecanismo de decisão do agente deverá, em cada momento, escolher a acção que lhe permita atingir a maior utilidade. Dito desta forma, o mecanismo de decisão parece trivial. No entanto, um agente é uma entidade reactiva que interage continuamente com o seu ambiente: ao longo da sua existência passa por uma sequência de estados do mundo, e deverá de alguma forma usar a função de utilidade, que por definição apenas compara estados isolados, para valorizar essas sequências. Esta particularidade complica bastante os mecanismos de decisão, sendo essa complicação ainda maior quando os agentes são concebidos com um horizonte de vida ilimitado.

Nos últimos tempos tem havido algum esforço no desenvolvimento de linguagens especialmente concebidas para implementar agentes inteligentes. Neste contexto, o objectivo desta tese pode precisamente ser visto como o desenvolvimento de uma linguagem para implementação de agentes orientados por funções de utilidade. Muito sinteticamente, a metodologia proposta para atingir este objectivo consistiu em, primeiro, adaptar uma linguagem já existente para especificação de programas concorrentes com primitivas que permitam definir, de forma explícita, a função de utilidade associada a cada processo. A necessidade de uma linguagem concorrente prende-se com o facto de um agente estar inserido num ambiente onde coexistem outros agentes, sendo normalmente necessário modelar o comportamento dessas entidades para que o mecanismo de decisão possa prever a melhor forma de agir. Cada processo desta nova linguagem irá, evidentemente, representar um agente da comunidade. As instruções presentes nos processos irão representar o papel das acções que cada agente tem a possibilidade de executar em cada instante.

Depois de definida a sintaxe desta linguagem, será necessário apresentar um mecanismo para compilar o código de um dos agentes modelados, por forma a se obter um programa cujo comportamento seja racional, no sentido de procurar maximizar a função de utilidade explicitada. A forma como o agente modelado irá actuar deverá ficar pré-definida em tempo de compilação. Esta opção só é possível porque o ambiente onde ele irá actuar se encontra modelado na linguagem. Se quisermos ser um pouco mais filosóficos para justificar esta

opção podemos parafrasear Savage, dizendo que um agente racional ideal *"só tem uma decisão a tomar em toda a sua vida. Ele deve, nomeadamente, decidir como viver, podendo isto, em princípio, ser feito de uma vez por todas."*

Como já foi referido, este processo de compilação não é trivial, porque os processos podem apresentar ciclos intermináveis que levam à necessidade de comparar a utilidade de sequências infinitas de estados. No entanto, existe uma complicação adicional derivada da existência de vários agentes, que consiste na incerteza que cada um dos agentes possui quanto ao estado global da computação. Para facilitar um pouco esta investigação, decidiu-se começar por abordar os programas em que apenas um agente é modelado, ou seja, onde só é permitido um processo estritamente sequencial, ficando a versão concorrente para trabalho futuro. Será essa versão restrita da linguagem que irá ser apresentada nesta tese.

1.1 Estrutura da tese

Excluindo este capítulo de introdução e motivação para o trabalho desenvolvido, esta tese encontra-se estruturada da seguinte forma:

- No próximo capítulo apresenta-se a sintaxe e a semântica da linguagem desenvolvida para programação sequencial com funções de utilidade. Neste capítulo pretende-se também motivar a existência de uma linguagem com estas características através da apresentação de alguns exemplos de programas.

- No capítulo seguinte será feita uma síntese de possíveis modelos e técnicas que poderiam ser utilizados para compilar programas escritos nesta linguagem. Esta síntese engloba trabalhos de diversas áreas, nomeadamente, da investigação operacional, dos métodos formais e da inteligência artificial.

- O capítulo 4 apresenta um possível método para compilar programas com funções de utilidade. Genericamente, este método passa pela tradução dos programas para a linguagem de programação original (como já foi referido, a nova linguagem derivou de uma linguagem já existente para especificação de programas concorrentes). Esta tradução procura preservar o comportamento esperado do programa original (i.é, maximização da utilidade) num sentido que será formalmente descrito. Neste capítulo também é apresentado o princípio de uma teoria de refinamento para esta linguagem.

- Finalmente, no capítulo 5 são apresentadas as conclusões desta tese e desenvolvidas algumas ideias para o trabalho futuro. Dado que esta tese apresenta os frutos de um projecto de investigação que ainda está no seu início, a secção de trabalho futuro é relativamente extensa, pois são discutidas detalhadamente algumas hipóteses para a sua continuação, nomeadamente no que toca à extensão da linguagem com primitivas de concorrência.

- Em anexo são incluídos um estudo preliminar sobre a caracterização formal das computações válidas de um programa, uma apresentação sucinta da lógica temporal, que é frequentemente utilizada ao longo da tese, alguns exemplos adicionais para melhor compreensão do processo de compilação, e uma breve descrição do protótipo concebido para testar a linguagem.

Capítulo 2

A linguagem USPL

A sintaxe da linguagem que foi utilizada para desenvolver programas baseados em funções de utilidade baseia-se num subconjunto da *Simple Programming Language* (SPL), introduzida por Manna e Pnueli para especificar programas concorrentes[MP95].

A única alteração introduzida a esta linguagem consistirá na introdução de uma primitiva que permite parametrizar cada processo com uma função de utilidade definida sobre as suas variáveis. Esta função irá representar as preferências desse processo. O nome *Utility-based Simple Programming Language* (USPL) deriva precisamente desta alteração.

Embora o objectivo a longo prazo deste trabalho de investigação consista em estudar programação concorrente baseada em funções de utilidade, esta tese foca essencialmente o problema da programação sequencial. Assim, embora a linguagem SPL permita especificar programas concorrentes, no presente trabalho apenas foi considerado um seu subconjunto puramente sequencial.

As razões para se optar pelo SPL como ponto de partida para a linguagem desenvolvida foram as seguintes:

- Dado que o objectivo a longo prazo deste trabalho de investigação consiste em estudar programação concorrente baseada em funções de utilidade, é conveniente começar a utilizar desde o início uma linguagem com primitivas de paralelismo.
- O SPL acomoda vários paradigmas de programação concorrente, permitindo comunicação quer por passagem de mensagens, quer através de memória partilhada, aglomerando uma amostra significativa das diversas primitivas existentes em várias linguagens de programação concorrente.
- Existem ferramentas para analisar propriedades temporais de programas escritos em SPL, como por exemplo o STeP [BBC+97], que serão úteis para provar propriedades derivadas do refinamento de programas USPL.

2.1 Sintaxe

O subconjunto da linguagem SPL que será utilizado será essencialmente constituído pelas seguintes primitivas:

- Declaração de variáveis. Por razões que serão apresentadas no capítulo 4 apenas será permitido o tipo gama de valores. Cada variável deve ser classificada num dos seguintes "modos":
  - `in`, quando é uma variável de entrada do programa e cujo valor não pode ser alterado durante a sua execução;
  - `local`, quando é uma variável local que não é visível fora do contexto do programa; ou
  - `out`, quando contem um resultado do programa.

  É possível impor restrições nos valores iniciais das variáveis através de uma asserção lógica. No caso das variáveis locais e de saída esta asserção serve para as inicializar. No caso das variáveis de entrada serve para definir a gama de valores que o programa aceita.

- Sequenciação (`;`).

- Escolha não determinística (`or`). Escolhe aleatoriamente uma instrução que esteja pronta para executar de entre um determinado conjunto de instruções.

- Ciclo enquanto (`while`).

- Atribuição guardada (`guard`). Efectua a atribuição de um valor a uma variável quando a condição de guarda especificada for verdadeira.

- Espera condicional (`await`). Introduz uma espera no programa até que uma condição de guarda seja verificada.

A escolha do subconjunto de instruções a utilizar baseou-se na facilidade de reconstrução de programas a partir do modelo semântico. Essa reconstrução será necessária para permitir obter um programa SPL a partir de um autómato optimizado correspondente a um programa USPL. Este problema será o assunto da secção 4.3, onde também procuraremos justificar mais fundamentadamente esta decisão.

A sintaxe precisa do subconjunto da linguagem SPL que será considerado neste trabalho é apresentada (em notação EBNF) na figura 2.1. Os símbolos terminais são apresentados em `typefont` e os não terminais em *itálico*. As alternativas encontram-se separadas por `|`, as partes opcionais estão delimitadas por [] e a existência de zero ou mais ocorrências de uma produção encontra-se assinalada por {}.

Embora reduzido, o subconjunto de instruções incluído é suficientemente expressivo para representar a maior parte das instruções utilizadas para programação sequencial em SPL. Este facto pode ser comprovado pelas seguintes equivalências:

programa ::= [*declarações*] *processo*
declarações ::= *declaração* {`;` *declaração*}
declaração ::= *modo variável* `:` `[` *int* `..` *int* `]` [`where` *expbool*]
modo ::= `in` | `out` | `local`
processo ::= *id* `::` `[` *instrução* `]`
instrução ::= `[` *instrução* `]`
| *instrução* `;` *instrução*
| *instrução* `or` *instrução*
| `while` *expbool* `do` *instrução*
| `guard` *expbool* `do` *id* `:=` *expint*
| `await` *expbool*
expbool ::= `false`
| `true`
| `(` *expbool* `\/` *expbool* `)`
| `(` *expbool* `/\` *expbool* `)`
| `!` `(` *expbool* `)`
| *expint* `=` *expint*
expint ::= *int*
| *variável*
| `(` *expint* `-` *expint* `)`
| `(` *expint* `+` *expint* `)`
| `abs` `(` *expint* `)`
int ::= *digito* { *digito* }
variável ::= *id*
id ::= *alpha* { *alpha* | *digito* | `_` }
digito ::= `0` | ... | `9`
alpha ::= `a` | ... | `z`
| `A` | ... | `Z`

Figura 2.1: Sintaxe de um subconjunto da linguagem SPL

`if` c `then` i `else` j	$\equiv$	`[ await` c `;` i `] or [ await ( !` c `) ;` j `]`
`if` c `then` i	$\equiv$	`[ await` c `;` i `] or await ( !` c `)`
`when` c `do` i	$\equiv$	`await` c `;` i
`skip`	$\equiv$	`await true`
x `:=` v	$\equiv$	`guard true do` x `:=` v
`repeat` i `until` c	$\equiv$	i `; while` c `do` i

As maiores limitações desta linguagem em relação ao subconjunto do SPL puramente sequencial são:

- apenas existe o tipo correspondente a uma gama de inteiros;
- só é utilizado um conjunto muito restrito de operadores matemáticos;
- não são permitidas instruções agrupadas;
- não existem atribuições simultâneas; e
- não é possível implementar a instrução `loop forever do` com a semântica usual do SPL.

A única diferença da linguagem USPL em relação ao SPL consiste na possibilidade de parametrizar o único processo existente com uma função de utilidade definida sobre as suas variáveis. Esta função de utilidade vai determinar uma relação de preferência sobre os estados do processo e condicionar o seu comportamento. No entanto, não é suficiente especificar apenas a função de utilidade devido à necessidade, já referida na introdução, de se compararem sequências de estados em vez de estados isolados. Não existe uma forma universalmente aceite de extrapolar uma relação de preferências sobre estados para uma relação de preferências sobre sequências. A forma usual de resolver este problema, tal como se verá no capítulo 3, consiste em especificar para além da função de utilidade qual o critério de decisão que deverá ser usado para realizar essa extrapolação.

Assim, a alteração que é necessário efectuar à sintaxe apresentada na figura 2.1 consiste em substituir a definição do símbolo não terminal *processo* pela seguinte definição:

processo	::=	*id* `(` *expint* `,` *critério*`) :: [` *instrução* `]`
critério	::=	`look-at-end`
	|	`look-ahead` *int*
	|	`discounted 0.`*int*

A expressão inteira que aparece dentro do parêntesis depois do identificador do processo é definida sobre as variáveis locais do processo e determina explicitamente a sua função de utilidade. No estado actual deste projecto são permitidos três critérios de decisão no USPL[1]:

- `look-at-end`. Este critério dita que a utilidade de uma sequência é a utilidade do seu último estado.

[1] A descrição aqui apresentada é muito informal. Na secção 2.4.2 será apresentada uma definição formal de cada um deles.

- `look-at-ahead`. Se este critério for adoptado, quando num determinado instante se escolhe qual a instrução a executar deverá ser comparada a utilidade que se vai atingir n passos adiante, sendo n o inteiro especificado. Quando $n = 1$ temos o caso mais conhecido de uma estratégia *greedy*, em que se procura maximizar a recompensa imediata.

- `discounted`. Neste caso, a utilidade total é a soma de todas as diferenças de utilidade da sequência, descontadas de acordo com o factor especificado (denotado por γ). O incremento de utilidade do instante t é pesado pelo factor γ^t. A justificação para a existência deste critério será apresentada na secção 3.4.1.

A primitiva `or` é a única cujo comportamento é afectado por esta alteração. Esta primitiva passa a descrever quais as instruções entre as quais um processo pode optar, deixando de representar uma escolha não-determinística. A escolha de qual a instrução a executar deixa de ser feita aleatoriamente entre todas as instruções prontas a executar, sendo em vez disso escolhida a instrução que permite ao processo maximizar a sua utilidade de acordo com o critério de decisão adoptado. Caso haja mais do que uma instrução nestas condições então será escolhida uma delas não deterministicamente.

2.2 Exemplos de programas em USPL

Um exemplo muito simples de um programa escrito em USPL é apresentado na figura 2.2. Neste exemplo temos duas variáveis de entrada `a` e `b` cujos valores apenas serão conhecidos aquando de uma execução do programa. Este deverá atribuir à variável de saída `x` o valor de uma das variáveis de entrada.

```
out x : [0..1] where x=0
in  a : [0..1]
in  b : [0..1]

P(x, look-at-end) :: [
    x := a
  or
    x := b
]
```

Figura 2.2: Exemplo de programa em USPL

Caso este programa não fosse escrito em USPL, mas sim em SPL (i.é, sem a função de utilidade a parametrizar o processo `P`), o comportamento consistia simplesmente em retornar aleatoriamente um dos valores de entrada. Tendo em consideração o comportamento induzido pela função de utilidade o resultado esperado passa a ser ligeiramente diferente. Como processo deverá procurar maximizar o valor final da sua função de utilidade, que neste caso é simplesmente `x`, na instrução de escolha deverá procurar atribuir à variável `x` a variável

de entrada cujo valor seja mais elevado. Ou seja, com esta função de utilidade este processo passa a comportar-se como uma função que calcula o máximo de dois valores. Se quisermos fazer um processo que calcule o mínimo de dois valores bastaria alterar a função de utilidade para a expressão `-x`.

É sempre possível num programa USPL reproduzir o comportamento normal esperado do programa SPL correspondente. Para tal basta utilizar como função de utilidade uma expressão constante qualquer, o que faz com que o processo passe a considerar todos os estados igualmente preferíveis. Por exemplo, se no programa anterior se utilizasse a expressão `0` como função de utilidade, o processo voltaria a retornar aleatoriamente um dos valores de entrada. Este facto será traduzido numa regra de refinamento na secção 4.4.

No capítulo 4 será apresentado um mecanismo que permite transformar programas USPL em programas SPL com um comportamento equivalente. Por exemplo, o programa SPL que corresponde ao programa da figura 2.2 pode ser consultado na figura 4.5 (página 71).

Embora este pequeno exemplo possa ser elucidativo quanto ao comportamento esperado de um programa escrito em USPL, não permite apreciar as suas verdadeiras potencialidades. Para as demonstrar iremos agora apresentar um problema um pouco mais complexo e apresentar a sua solução escrita nesta linguagem.

Suponha-se que se pretende escrever um programa para destrocar dinheiro em moedas de 5$00, 3$00 ou 2$00. Pretende-se que, para além de destrocar correctamente sempre que possível, se minimize o número de moedas devolvidas. A solução para este problema numa linguagem convencional, embora não seja complicada, não é trivial, pois é necessário ter em conta que, para valores superiores a 5$00, a melhor solução não é sempre começar por devolver moedas de 5$00, pois pode-se chegar a um estado em que não se consegue destrocar correctamente o valor pedido (por exemplo, se pretendermos destrocar 6$00, não se deve começar por devolver uma moeda de 5$00 porque depois não existem moedas de 1$00).

Em USPL a solução é relativamente trivial, como se pode ver na figura 2.3[2]. Neste programa assume-se que o valor que se pretende destrocar se encontra na variável `dinheiro`, sendo o acto de devolver uma moeda representado por um decremento nesta variável[3]. São utilizadas duas variáveis auxiliares, uma para contabilizar o número de moedas devolvido e outra para controlar a terminação do ciclo. O corpo do processo apenas se limita a explicitar quais as opções que existem para destrocar o dinheiro, sendo toda complexidade da programação passada para a especificação da função de utilidade. Note-se que mesmo a decisão de terminar o ciclo deixa de ser feita pelo programador.

A função de utilidade reflecte os dois objectivos principais do programa: o primeiro, e mais importante, é que se deve tentar que a variável `dinheiro` no final da execução tenha o valor `0` ou o mais próximo possível de `0`, e o segun-

[2]Por questões de simplificação este programa utiliza alguns operadores que não foram introduzidos na sintaxe da linguagem apresentada na figura 2.1, como o `*` ou o `>=`, mas que poderiam ser reescritos facilmente nessa sintaxe.

[3]Devido ao facto de se pretender alterar o seu valor esta foi declarada como sendo do tipo `local` e não `in`.

```
local dinheiro : [0..10]
local moedas   : [0..10] where moedas=0
local continua : [0..1]  where continua=1

P(0-2*dinheiro-moedas, discounted 0.95) :: [
  while (continua=1) do
    [
        [
          guard (dinheiro>=5) do dinheiro := (dinheiro-5)
        or
          guard (dinheiro>=3) do dinheiro := (dinheiro-3)
        or
          guard (dinheiro>=2) do dinheiro := (dinheiro-2)
        ];
        guard !(moedas=10) do moedas := (moedas + 1);
        [
          continua := 0
        or
          continua := 1
        ]
    ]
]
```

Figura 2.3: Máquina de destrocar moedas em USPL

do, que a variável `moedas` tenha o menor valor possível. Este dois objectivos são representados explicitamente na função de utilidade, aparecendo as duas variáveis com factores negativos. A variável `dinheiro` aparece afectada por um peso maior por forma a reflectir a maior importância de se conseguir destrocar correctamente o valor pedido. Por exemplo, se assim não fosse, era indiferente para o processo ao destrocar 6$00 devolver uma moeda de 5$00 e ficar com 1$00 ou devolver duas moedas de 3$00. Neste caso o critério de decisão é o da soma total da utilidade descontada pelo factor 0.95.

2.3 Tipos de funções de utilidade

Um dos problemas que normalmente se coloca quando se tenta desenvolver um agente orientado por utilidade consiste em saber que tipo de função de utilidade deve ser usado. Embora teoricamente qualquer função possa ser considerada existem algumas classes de funções que abrangem grande parte das aplicações desejadas. Por exemplo, Rosenschein e Zlotkin definem três classes de funções [RZ94]:

- **Conjunto de objectivos parciais**. Um agente pode ter um conjunto de distintos objectivos parciais (ou tarefas) $\{g_k \mid k = 1 \ldots n\}$ que pretende

atingir.[4] Cada objectivo parcial g_k vale para ele w_k. O seu objectivo global pode ser definido como a conjunção $g = \wedge_{k=1}^{n}(g_k)$. A função de utilidade pode ser definida como[5]

$$u = \lambda\ s\ .\ \sum_{s \Vdash g_k} (w_k)$$

Podem existir domínios onde os agentes recebem uma penalização por não atingirem alguns dos objectivos parciais. Sendo p_k a penalização recebida por não se atingir o objectivo g_k (um valor negativo), então a função de utilidade seria definida da seguinte maneira:

$$u = \lambda\ s\ .\ \sum_{s \Vdash g_k} (w_k) + \sum_{s \nVdash g_k} (p_k)$$

- **Distância entre estados**. Por vezes é conveniente pensar que um agente tem o objectivo de atingir g pelo qual receberá a recompensa w, recebendo uma recompensa cada vez menor conforme atingir estados cada vez mais afastados dos estados onde se verifica g. Para definir a função de utilidade nestes casos é conveniente definir o conceito de distância entre dois estados s e f como $d(s, f) = c(s \rightarrow f)$, onde $c(s \rightarrow f)$ é o custo do melhor plano que faz mover o sistema do estado s para o estado f. Dadas estas definições a função de utilidade define-se como:

$$u = \lambda\ s\ .\ w - \min_{f \Vdash g}(d(s, f))$$

- **Distância probabilística**. Esta classe assemelha-se bastante à anterior, mas em vez de se usar como distância entre dois estados o custo do melhor plano para transitar de um para outro, usa-se distribuição de probabilidade condicional que diz, para cada estado, qual é a probabilidade de se atingir um qualquer estado. Esta classe de funções de utilidade pode ser aplicada em domínios onde existe uma diferença entre os estados finais do sistema e os estados intermédios onde um plano pode acabar. Dado um conjunto de estados finais F e uma função de utilidade definida nesses estados, a função de utilidade define-se nos estados intermédios $s \notin F$ como:

$$u = \lambda\ s\ .\ \sum_{f \in F} p(f \mid s) \times (u\ f)$$

Outro mecanismo muito utilizado na especificação de funções de utilidade baseia-se na teoria da utilidade multi-atributo. Nesta teoria as características relevantes dos estados resultantes de tomar uma decisão são designadas atributos. O objectivo desta teoria é estudar em que condições é possível, dado um conjunto de funções de utilidade definidas sobre subconjuntos de atributos,

[4] Uma tarefa é aqui representada por uma fórmula que deverá ser válida num estado onde essa tarefa esteja concluída.

[5] $s \Vdash g$ se o objectivo g é atingido no estado s.

compor essas funções de utilidade numa única função de utilidade que caracterize correctamente as preferências do agente de decisão. A grande vantagem de se usarem estas técnicas deriva da maior facilidade de se especificar funções de utilidade parciais.

2.4 Semântica

A semântica da linguagem USPL é muito semelhante à semântica denotacional baseada em *Fair Transitions Systems* da linguagem SPL introduzida em [MP95].

Definição 2.1 *Um* Fair Transitions System *(FTS) é definido por um tuplo* $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$*, onde:*

- V *é um conjunto finito de variáveis de sistema. Estas variáveis tanto podem ser variáveis de dados, quando são manipuladas explicitamente pelo programa, como variáveis de controlo, quando representam a localização do controlo em cada um dos processos. Assume-se que cada variável possui um tipo, que define o seu domínio, sendo um estado neste sistema de transição definido como uma possível interpretação (consistente quanto aos tipos) de* V*.* Σ *representa o conjunto de todos os estados possíveis.*

- $\Theta : \Sigma \rightarrow Bool$ *é um predicado que determina os estados iniciais de execução do programa.*

- $\mathcal{T}$ *é um conjunto finito de transições. Cada transição* $\tau \in \mathcal{T}$ *é uma função com o tipo* $\Sigma \rightarrow 2^{\Sigma}$*, que mapeia cada estado* $s \in \Sigma$ *num conjunto (possivelmente vazio) de estados* $\tau(s) \subseteq \Sigma$*. Uma transição diz-se activa num estado* s *se* $\tau(s) \neq \emptyset$*.*

- $\mathcal{J} \in \mathcal{T}$ *é um conjunto de transições justas. Se uma transição* τ *for justa então não serão permitidas computações em que* τ *está continuamente activa sem nunca ser executada a partir de um determinado estado.*

- $\mathcal{C} \in \mathcal{T}$ *é um conjunto de transições compassivas. Se uma transição* τ *for compassiva então não serão permitidas computações em que* τ *está activa um número infinito de vezes sendo apenas executada um número finito de vezes.*

Cada transição $\tau \in \mathcal{T}$ é representada por uma fórmula de primeira ordem $\rho_\tau(V, V')$, denominada *relação de transição*, que determina a relação existente entre um estado s e qualquer dos seus sucessores $s' \in \tau(s)$. Nesta fórmula, uma variável sem apóstrofe denota o valor dessa variável em s e uma variável com apóstrofe em s'. Sendo assim, um estado s' é um sucessor de um estado s se a fórmula $\rho_\tau(V, V')$ tiver o valor verdadeiro quando se interpretar cada $x \in V$ como $s[x]$ e cada $x' \in V'$ como $s'[x]$. O valor de uma relação de transição ρ_τ calculado num par de estados $\langle s, s' \rangle$ desta forma será representado por

$$\langle s, s' \rangle [\rho_\tau]$$

Dada uma relação de transição ρ_τ é possível expressar a asserção de actividade da transição τ através da seguinte fórmula:

$$En(\tau) \equiv \exists V' \cdot \rho_\tau(V, V')$$

Uma das transições contidas em $\mathcal{T}$ deve ser a transição nula τ_I cuja relação de transição é definida por $\rho_I \equiv (V = V')$. Esta transição mantém inalterados os valores de todas as variáveis de sistema (incluindo as variáveis de controlo). Todas as restantes transições alteram pelo menos o valor de uma variável e designam-se transições diligentes.

Finalmente, é necessário definir a noção de computação válida para um programa $\mathcal{P}$. Uma computação é uma sequência infinita de estados

$$\sigma \equiv s_0, s_1, s_2, \ldots, s_k, \ldots$$

Uma transição $\tau \in \mathcal{T}$ está activa numa posição k de σ se a sua asserção de actividade for válida nesse estado, ou seja,[6]

$$s_k \Vdash En(\tau)$$

A transição τ ocorre nessa posição se $s_{(k+1)}$ for um sucessor de s_k por τ, ou seja, $s_{(k+1)} \in \tau(s_k)$ ou, visto de outra forma se for válida a seguinte asserção:

$$\models \langle s_k, s_{(k+1)} \rangle [\rho_\tau]$$

Definição 2.2 *Uma sequência σ é uma computação válida de um programa $\mathcal{P}$ (designada por $\mathcal{P}$-computação) se satisfizer os seguintes requisitos:*

- *O estado s_0 é um estado inicial, i.é,*

$$s_0 \Vdash \Theta$$

- *Para a toda a posição de σ existe pelo menos uma transição $\tau \in \mathcal{T}$ que ocorre nessa posição.*

- *σ não viola os critérios de justiça e compaixão impostos pelos conjuntos $\mathcal{J}$ e $\mathcal{C}$.*

Uma sequência de estados que só satisfaz os dois primeiros critérios denomina-se uma execução.

[6] Se p é uma fórmula em lógica de 1ª ordem onde podem intervir variáveis flexíveis (i.é, variáveis pertencentes ao conjunto V não quantificadas) e $s \in \Sigma$ um estado de um FTS, designaremos por $s[p]$ a fórmula da lógica de 1ª ordem que se obtém depois de interpretar nesse estado todas as variáveis flexíveis contidas em p. Neste caso, p será válida num estado s se $s[p]$ for uma tautologia em lógica de 1ª ordem. Esta noção de validade de uma fórmula num estado será capturada pela relação $\Vdash$ que, para uma fórmula de estado p, se define da seguinte forma: $s \Vdash p \equiv \models s[p]$

2.4.1 *Utility-based Fair Transition Systems*

A semântica de um programa USPL será baseada num modelo que deriva directamente de um FTS. Analogamente ao nome da linguagem, o novo modelo será designado *Utility-based Fair Transition System* (UFTS). As diferenças mais óbvias de um UFTS em relação a um FTS são as seguintes:

- Dado que, exceptuando a transição nula, todas as transições são justas e que os aspectos de compassividade não são relevantes devido ao facto de não existirem primitivas de concorrência, os conjuntos $\mathcal{J}$ e $\mathcal{C}$ deixam de ser necessários.
- Como só existe um processo, o conjunto de variáveis V apenas contem uma variável de controlo, que será designada π.
- Será introduzida uma função de utilidade definida sobre os estados do sistema de transição para representar as preferências do único processo existente.
- Também será introduzido o critério de decisão que será usado para extrapolar a função de utilidade para sequências de estados.

A alteração menos óbvia aos FTSs deriva da necessidade de se alterar a noção de computação válida. Esta deve reflectir as preferências do processo expressas através da função de utilidade e do critério de decisão. Este será representado através de uma função que, dada a função de utilidade e um conjunto de execuções, decide quais as execuções preferidas.

Definição 2.3 *Formalmente, um UFTS é definido por um tuplo* $\langle V, \Theta, \mathcal{T}, u, \rho \rangle$*, onde:*

- V*,* Θ *e* $\mathcal{T}$ *têm o significado já apresentado na definição de FTS, com a única diferença de* V *apenas possuir a variável de controlo* π*.*
- $u : \Sigma \to \mathbb{R}$ *é uma função de utilidade que determina uma relação de preferência do processo sobre os estados.*
- $\rho : (\Sigma \to \mathbb{R}) \to 2^{\Sigma^\omega} \to 2^{\Sigma^\omega}$ *representa o seu critério de decisão. Esta função tem as seguintes restrições:*
  - *Um critério de decisão deve optar apenas entre as execuções que lhe são fornecidas:*
    $$\forall u, \Omega \cdot \rho\ u\ \Omega \subseteq \Omega$$
  - *Sempre que possível deve fazer alguma escolha:*
    $$\forall u, \Omega \neq \emptyset \cdot \rho\ u\ \Omega \neq \emptyset$$

Definição 2.4 *Considerando* Ω *como o conjunto de todas as execuções possíveis de um UFTS que cumprem os critérios de justiça, então uma execução* $\sigma \in \Omega$ *é uma computação válida desse UFTS se pertencer ao conjunto* $\rho\ u\ \Omega$*.*

2.4.2 Determinação de um UFTS a partir de um programa

Naturalmente, a cada programa USPL corresponde um UFTS. O restante desta secção será dedicado a mostrar como se calcula um UFTS a partir de um programa.

O conjunto das variáveis V é constituído pelas variáveis declaradas no programa mais a variável de controlo π. Esta variável está definida sobre uma gama de inteiros, correspondendo cada valor a uma localização no programa.

A condição inicial Θ define-se como

$$(\pi = 0) \wedge \varphi$$

sendo φ a conjunção de todas as asserções que aparecem na cláusula `where` das declarações do programa.

Para se definir as relações de transição associadas a cada instrução do USPL, é necessário saber qual o valor da variável de controlo antes e depois de cada instrução num programa. Para determinar estes valores assume-se que cada instrução do programa está decorada com uma marca antes e outra depois e define-se uma função *pos*, que dada uma marca dá a posição que lhe corresponde. Esta função *pos* deve, no máximo, identificar as seguintes marcas:

- Num processo do tipo *id* `(` *expint* `) :: [` l: *instrução* $\widehat{l}$: `]`

$$pos(l) = 0$$

- Numa instrução do tipo l: `[` m: *instrução* $\widehat{m}$: `]` $\widehat{l}$:

$$pos(l) = pos(m) \wedge pos(\widehat{l}) = pos(\widehat{m})$$

- Numa instrução do tipo l: m: *instrução* $\widehat{m}$: `;` n: *instrução* $\widehat{n}$: $\widehat{l}$:

$$pos(l) = pos(m) \wedge pos(\widehat{m}) = pos(n) \wedge pos(\widehat{n}) = pos(\widehat{l})$$

- Numa instrução do tipo l: m: *instrução* $\widehat{m}$: `or` n: *instrução* $\widehat{n}$: $\widehat{l}$:

$$pos(l) = pos(m) = pos(n) \wedge pos(\widehat{l}) = pos(\widehat{m}) = pos(\widehat{n})$$

- Numa instrução do tipo l: `while` *expbool* `do` m: *instrução* $\widehat{m}$: $\widehat{l}$:

$$pos(l) = pos(\widehat{m})$$

Usando esta função *pos* é possível recuperar a relação de equivalência sobre marcas $\sim_L$ definida em [MP95] através da seguinte equivalência:

$$l \sim_L m \quad \text{sse} \quad pos(l) = pos(m)$$

Devido ao facto de se usarem explicitamente marcas depois das instruções não é necessário definir a função *post* referida em [MP95], servindo a função

pos para determinar o valor da variável de controlo antes e depois da execução de uma instrução.

Dado que em cada relação de transição normalmente apenas são modificadas algumas variáveis do conjunto V, define-se, para um conjunto $U \subseteq V$, a seguinte abreviatura:

$$pres(U) \equiv \bigwedge_{u \in U} (u' = u)$$

Finalmente define-se qual a relação de transição que corresponde a cada instrução da linguagem da seguinte forma:

- *Guard* A uma instrução do tipo l: `guard` c `do` u `:=` e $\widehat{l}$: corresponde a relação de transição:

  $$c \wedge \pi = pos(l) \wedge \pi' = pos(\widehat{l}) \wedge u' = e \wedge pres(V - \{\pi, u\})$$

  Como podemos reparar, esta relação de transição só pode ocorrer quando c se verificar no estado de origem. As únicas variáveis que mudam de valor são π, que vai passar a conter a nova posição de controlo, e u, a variável à qual é atribuída a expressão e.

- *Await* A uma instrução do tipo l: `await` c $\widehat{l}$: corresponde a relação de transição:

  $$c \wedge \pi = pos(l) \wedge \pi' = pos(\widehat{l}) \wedge pres(V - \{\pi\})$$

  Este caso é quase igual ao anterior, exceptuando no facto de apenas π mudar de valor.

- *While* A uma instrução do tipo l: `while` c `do` m: *instrução* $\widehat{m}$: $\widehat{l}$: correspondem as seguintes relações de transição:

  $$c \wedge \pi = pos(l) \wedge \pi' = pos(m) \wedge pres(V - \{\pi\})$$
  $$\neg c \wedge \pi = pos(l) \wedge \pi' = pos(\widehat{l}) \wedge pres(V - \{\pi\})$$

  Neste caso temos duas relações de transição, a primeira para representar as transições que dão origem a uma nova iteração do ciclo, quando c é verdadeiro, e a segunda para os casos de paragem, quando c é falso. Mais uma vez, a única variável que muda de valor é π.

- *Or*
  Esta instrução não dá origem a nenhuma relação de transição.

- *Sequenciação*
  Idem.

A função de utilidade é definida pela valoração da expressão indicada a seguir ao nome do programa. Para especificar formalmente o critério de decisão, vamos começar por definir qual a relação de preferência sobre execuções induzida em cada um dos casos, sendo depois as computações escolhidas definidas como o conjunto das execuções que não têm nenhuma execução mais preferida.

Dada uma relação de preferência $\succ$, esse conjunto será denotado por $\uparrow_{\succ} \Omega$ e define-se formalmente como:

$$\sigma \in \uparrow_{\succ} \Omega \,\text{sse}\, \sigma \in \Omega \wedge \forall\, \sigma' \in \Omega \cdot \neg(\sigma' \succ \sigma)$$

Como os critérios devem ignorar as transições nulas sempre que outras transições estejam disponíveis, cada uma das execuções a seguir apresentadas não considera essas transições e representa todo o conjunto de transições que se obtém dela por repetição arbitrárias dos seus estados[7]. Ou seja, assumindo que a relação de preferência é representada por $\succ$, se $\ldots, s, \ldots \succ \ldots, t, \ldots$ então $\ldots, s, \ldots \succ \ldots, t, t, \ldots$ e $\ldots, s, s, \ldots \succ \ldots, t, \ldots$. Também é de notar que apenas são comparáveis sequências que se iniciam com o mesmo estado, dado que o estado inicial da computação é controlado por factores externos ao programa.

Começando pelo critério `look-at-end`, começamos por notar que, como as execuções são infinitas, este critério só pode ser aplicado se todas as execuções terminarem com uma repetição infinita do mesmo estado (que será denotado por s^{ω}). Dada uma função de utilidade u, será definida a relação de preferência $\multimap_u$ da seguinte forma:

$$s_0, \ldots, s^{\omega} \multimap_u t_0, \ldots, t^{\omega} \text{ sse } s_0 = t_0 \wedge u(s^{\omega}) > u(t^{\omega})$$

Para o critério `look-ahead`, dada uma função de utilidade u e um horizonte n, será definida a relação de preferência $\curvearrowright_u^n$ da seguinte forma:

$$s_0, s_1, \ldots \curvearrowright_u^n t_0, t_1, \ldots \text{ sse } \forall j, k \in \mathbb{N}_0 \cdot s_j = t_k \supset u(s_{j+n}) > u(t_{k+n})$$

No caso do critério `discounted`, dada uma função de utilidade u e um factor de desconto γ, a relação de preferência $\rightsquigarrow_u^{\gamma}$ define-se da seguinte forma:

$$s_0, s_1, \ldots \rightsquigarrow_u^{\gamma} t_0, t_1, \ldots \text{ sse } s_0 = t_0 \wedge \sum_k^{\infty} \gamma^k (u(s_{k+1}) - u(s_k)) > \sum_k^{\infty} \gamma^k (u(t_{k+1}) - u(t_k))$$

Como exemplo de um modelo semântico, apresentamos o seguinte UFTS correspondente ao programa da figura 2.2:

$$\begin{aligned}
V &= \{x, a, b, \pi\} \\
\Theta &= (\pi = 0) \wedge (x = 0) \\
\mathcal{T} &= \{\tau_1, \tau_2, \tau_I\} \\
\tau_1 &= (\pi = 0) \wedge (\pi' = 1) \wedge (x' = a) \wedge (a' = a) \wedge (b' = b) \\
\tau_2 &= (\pi = 0) \wedge (\pi' = 1) \wedge (x' = b) \wedge (a' = a) \wedge (b' = b) \\
u &= \lambda\ s\ .\ s(x) \\
\rho &= \lambda\ u\ \Omega\ .\ \uparrow_{\multimap_u}\ \Omega
\end{aligned}$$

[7] Isto significa que o único estado que pode aparecer repetido é o último.

Capítulo 3

Síntese de estratégias para entidades com preferências

Embora a linguagem USPL tenha sido apresentada como uma linguagem de especificação de agentes orientados por funções de utilidade, também pode ser vista de forma mais genérica como uma linguagem de especificação de problemas de decisão. Ambas as perspectivas podem ser úteis na contextualização deste trabalho, com o objectivo final de identificar um método de compilação para esta linguagem. Seguindo a pista da especificação de agentes, ou mais genericamente, da especificação de programas reactivos, serão analisados diversos trabalhos na área da inteligência artificial e dos métodos formais, e, mais concretamente, na área da síntese de agentes ou programas a partir de especificações lógicas. Na perspectiva dos problemas de decisão serão analisados os processos de decisão de Markov, cujo modelo semântico é o que mais se aproxima do modelo semântico do USPL.

Como se verá, a maior parte destes problemas resolve-se através da síntese de estratégias. Uma estratégia é, essencialmente, uma função que nos indica em cada momento qual a melhor acção para executar. No capítulo seguinte veremos que uma estratégia pode ser vista como um programa sequencial, facilmente compilado ou executado usando técnicas convencionais. A tese fundamental deste trabalho é precisamente a equivalência entre o problema de compilação de programas USPL e a síntese de estratégias num problema de decisão.

Antes de partir para a análise de trabalhos específicos, vamos começar por situar com clareza o modelo semântico do USPL no contexto geral dos problemas de decisão. Esta primeira parte tem por objectivo identificar formalmente quais os componentes do modelo de decisão a adoptar. Após esta contextualização serão então abordadas as técnicas antes referidas. O capítulo termina com uma pequena síntese das técnicas apresentadas e com a escolha da técnica a adoptar para o caso do USPL.

3.1 Contextualização do modelo de decisão

A contextualização do modelo de decisão de um UFTS vai incidir fundamentalmente em três partes: primeiro, será analisado o modelo de computação sem

considerar as preferências do processo; depois iremos enquadrar o modelo de representação das suas preferências; e, finalmente, será introduzida a noção particular de estratégia que procurará capturar o comportamento óptimo de um processo USPL, abrindo caminho para a sua concretização num programa sequencial convencional.

3.1.1 Transições

Tradicionalmente, os problemas de decisão são classificados de acordo com o conhecimento que os agentes de decisão possuem acerca do estado do meio ambiente que os envolve. Assim, podemos dividir estes problemas nas seguintes categorias [Fre93]:

- *Decisões com certeza.* Nesta categoria assume-se que o agente sabe qual é o verdadeiro estado do ambiente antes de tomar uma decisão, ou seja, consegue prever a consequência das suas acções com total certeza.

- *Decisões com risco.* Nesta categoria o agente de decisão não sabe qual é o verdadeiro estado do ambiente, mas consegue quantificar a sua incerteza através uma distribuição de probabilidades sobre os possíveis estados.

- *Decisões com total ignorância.* Aqui o agente também não sabe qual é o verdadeiro estado do ambiente nem consegue quantificar a sua incerteza, apenas conseguindo enumerar quais os possíveis estados.

Também é frequente dividir os modelos sobre os quais se desenvolve a teoria da decisão em quantitativos ou qualitativos dependendo do facto de haver ou não quantificação da incerteza usando distribuições de probabilidade. Assim, a segunda categoria apresentada anteriormente assenta num modelo quantitativo e a terceira num modelo qualitativo. No caso da primeira categoria esta classificação não faz sentido pois não existe incerteza.

A teoria da decisão clássica é muito baseada em modelos quantitativos probabilísticos. A justificação para este facto pode ser consultada em [Fre93]. No entanto, existem várias razões epistemológicas e computacionais para que se usem modelos qualitativos em detrimento dos quantitativos. A justificação mais frequente (ver, por exemplo, [BG96, BT96, BT00]) é a extrema dificuldade e, por vezes, impossibilidade de obter as funções de utilidade e probabilidades necessárias nestes modelos. Nomeadamente, os seres humanos expressam normalmente as suas crenças, objectivos e preferências usando modelos qualitativos e tem alguma dificuldade em expressa-los quantitativamente. Brafman e Tennenholtz apontam também a esperança de se obterem algoritmos de decisão mais eficientes devido à maior simplicidade dos modelos [BT96, BT00].

Werner vai mais longe na caracterização da incerteza sobre a consequência de executar uma determinada acção. Esta incerteza poderá dever-se a uma das seguintes causas [Wer96]:

- Incerteza inerente. O ambiente onde a acção é executada pode ser naturalmente não-determinístico, podendo uma acção ter várias consequências possíveis.

- Incerteza subjectiva. Quando a entidade que executa uma acção determinística não possui capacidade computacional suficiente para prever a sua consequência.
- Incerteza derivada da incerteza quanto ao estado actual.
- Incerteza derivada da coexistência de outras entidade que podem influenciar as consequências de uma acção.

Na fase actual deste trabalho de investigação apenas foi considerado um subconjunto puramente sequencial do SPL, levando a que não ocorra nenhuma das causas anteriores:

- Todas as instruções da linguagem utilizada tem uma semântica perfeitamente determinística, não existindo incerteza inerente.
- Como a resolução dos problemas de decisão não será feita em tempo de decisão, mas à priori num processo semelhante à compilação, existe tempo suficiente para determinar o estado resultante de executar uma acção. De qualquer forma, todas as instruções são suficientemente simples para que a sua consequência seja de fácil determinação.
- Como por enquanto apenas existe uma entidade, não existe incerteza quanto ao estado actual. Todas as variáveis do programa são visíveis para o único processo existente e, como os estados são possíveis valorações para essas variáveis, é sempre possível determinar com certeza o estado actual.
- Obviamente, o último tipo de incerteza também não existe.

Assim, o problema de decisão com que nos deparamos encaixa na classe dos problemas de decisão com certeza, não fazendo sentido falar em problemas qualitativos ou quantitativos em consequência desse facto. Como se verá nas secções seguintes, em todos os modelos de decisão normalmente existe uma noção de estado em que o agente se encontra, e de um conjunto de acções, entre as quais o agente deverá escolher a "melhor", e cuja execução condiciona de alguma forma o próximo estado do mundo. Neste capítulo iremos, sempre que possível, denominar por S o conjunto de estados e por A o conjunto de acções.

Formalmente, para o caso do USPL é suficiente que o modelo de decisão a adoptar possa lidar com relações de transição do tipo $S \times A \hookrightarrow S$. Aparentemente este tipo para a relação de transição não permite representar as transições de um FTS tal como foram apresentadas na definição 2.1: um conjunto de funções com o tipo $S \to 2^S$. No entanto, se atentarmos ao método de determinar um UFTS apresentado na secção 2.4.2, conclui-se que cada instrução do programa dá origem a uma relação de transição e, indirectamente, a uma destas funções[1]. Tendo em atenção apenas as instruções do SPL consideradas, verifica-se, tal como já foi referido anteriormente, que todas são determinísticas, ou seja, caso sejam possíveis de executar num estado apenas existe um e um só estado

[1]Para esta análise ser correcta, devemos considerar a instrução `While` como sendo composta por duas sub-instruções, dando origem a duas funções no conjunto de transições.

seguinte possível de alcançar com a sua execução. Sendo assim, para o subconjunto da linguagem SPL considerado, seria suficiente representar as transições por um conjunto de funções com o tipo $S \hookrightarrow S$.

Considerando um programa USPL como a especificação de uma agente orientado por uma função de utilidade, as suas instruções iriam representar o papel das acções de entre as quais este deve escolher as que permitem maximizar a sua utilidade (no sentido referido pelo critério de decisão). Identificando o conjunto das relações de transição $\mathcal{T}$ com o conjunto de acções A, a representação de todo o conjunto de funções de transição por uma única função com o tipo referido passa então a ser trivial.

3.1.2 Preferências

Os problemas de decisão podem também ser classificados quanto ao horizonte da decisão em três tipos:

- *One-shot*, quando o problema apenas consiste na escolha da melhor acção para uma situação ocasional.
- *Multi-stage*, quando já é necessário escolher uma sequência de acções para resolver um problema que se extende ao longo do tempo, mas que irá terminar num prazo limitado.
- *Infinite*, quando o horizonte de decisão pode ser ilimitado ou, no caso dos humanos, tão longo que em termos de modelo de decisão pode ser considerado como equivalente a um horizonte infinito.

A maior parte dos trabalhos e técnicas desenvolvidos nas áreas da teoria da decisão e da teoria dos jogos apenas dizem respeito às duas primeiras classes. Mesmo no caso do horizonte infinito, normalmente só é considerada uma classe muito particular que consiste na repetição infinita do mesmo problema de decisão. No entanto, no caso do USPL, devido à possibilidade de existirem ciclos, é necessário considerar um horizonte infinito. Por esta razão é que não é suficiente caracterizar as preferências apenas pela função de utilidade. No caso dos horizontes *one-shot* e *multi-stage*, como não existe incerteza, poderia usar-se um critério aceite mais ou menos universalmente de calcular a utilidade de uma estratégia como sendo a utilidade final de executar essa estratégia. Neste caso, a decisão é bastante simples, bastando escolher a estratégia com maior utilidade. Quando o horizonte é ilimitado já não existe nenhum critério universal, pois a noção de estado final desaparece, passando a ser necessário incluir no modelo qual o critério de decisão a utilizar.

Embora as preferências sobre os estados sejam representadas por uma função com tipo $S \to \mathbb{R}$, quando consideramos a representação conjunta com o critério de decisão, tal como definido em 2.3, obtém-se uma outra representação das preferências, agora definida sobre as computações, com o tipo $S^\omega \to \mathbb{B}$. Embora ambas as representações sejam equivalentes, optamos por incluir na linguagem a definição baseada na função de utilidade em conjunto com o critério de decisão devido, por um lado, à maior facilidade sintáctica para as representar e também devido ao facto de ser mais natural para o programador representar

as preferências desta forma. Como se verá na secção 3.2, para representar as preferências directamente de acordo com o segundo modelo, seria possível utilizar uma especificação em lógica temporal. No entanto é muito difícil traduzir numa especificação temporal umas preferências tão simples e naturais como, por exemplo, tentar maximizar o retorno imediato de utilidade (facilmente traduzível numa função de utilidade mais o critério `look-ahead 1`). Mesmo que esta tradução seja possível (facto que, para os critérios de decisão utilizados, não se garante que seja sequer possível), seria muito complicado para o programador expressar as preferências dessa forma. No entanto, nada impede que essa tradução se faça mais tarde automaticamente no processo de compilação.

3.1.3 Estratégias

Normalmente, qualquer tentativa de implementar um agente orientado por funções de utilidade passa pela determinação da sua estratégia de actuação óptima. Aliás este é praticamente o ponto comum a todos as técnicas analisadas nas próximas secções. Uma estratégia é uma função que, em cada circunstância, irá indicar ao agente qual a melhor acção ou acções para executar. Tradicionalmente, as estratégias são classificadas quanto ao seu domínio em dois tipos:

- Estratégias sem memória, quando a acção a executar apenas depende do estado actual da computação.
- Estratégias com memória, quando o agente escolhe a acção a executar em função de toda a sua história passada, ou mais concretamente, da sequência de estados por onde passou desde o início da execução.

Quanto ao contra-domínio podemos encontrar os seguintes tipos:

- Determinísticas, quando apenas é escolhida uma acção.
- Não-determinísticas, quando é escolhido um conjunto de acções cuja execução é igualmente satisfatória para o agente. Neste caso, a acção concreta a executar deverá ser escolhida não-deterministicamente de entre o conjunto determinado.
- Probabilísticas, quando é definida uma função de distribuição de probabilidades sobre o conjunto de acções, que quantifica com precisão a relação de preferência entre estas no momento da decisão. Este tipo de estratégias é frequentemente adoptada em conjunto com modelos de computação quantitativos.

De seguida tentar-se-á justificar abreviadamente o tipo de estratégias pretendidas no caso do USPL. Quanto ao contradomínio, pretendem-se estratégias não-deterministas. Dado que o objectivo é refinar o programa USPL o menos possível[2], em cada estado devem ser enumeradas todas as acções que permitem

[2]I.é, perdendo o menor número de computações válidas, ou, no contexto da definição 2.3, pretende-se obter todas as computações em $\rho\ u\ \Omega$.

atingir computações óptimas. Caso contrário, apenas uma das computações seria escolhida. Também não é conveniente utilizar estratégias probabilísticas, porque isso obrigaria ao desenvolvimento de uma linguagem especial para a sua implementação. Se nos mantivermos nas estratégias não-determinísticas, a sua implementação poderá ser feita facilmente na própria linguagem SPL, tal como se verá no capítulo 4.

Quanto ao contra-domínio, optamos por utilizar estratégias sem memória. No entanto, a justificação para este facto já é um pouco mais subtil. Para se implementar uma estratégia com memória no programa SPL resultante poderíamos adoptar uma das seguintes estratégias:

- Modificar o espaço de estados e de transições, introduzindo pelo menos mais uma variável para armazenar os estados por onde se passou e acrescentar instruções para actualizar esta variável. Com a noção de computação válida actual esta hipótese não é viável pois as computações obtidas não seriam exactamente as computações do programa original. No entanto, como se verá na secção de trabalho futuro, com uma noção de computação válida diferente talvez fosse possível implementar este tipo de estratégias.

- Acrescentar um processo que estaria encarregue de executar a estratégia obtida e que poderia controlar o processo original através de um canal de comunicações. Embora o conjunto dos dois processos também possua computações muito diferentes do programa original, a restrição das computações ao conjunto de variáveis locais ao processo principal seria "quase" um subconjunto das computações originais. O "quase" deriva de um pormenor técnico do SPL que não permite, mesmo usando comunicação por mensagens entre os dois processos, testar o valor da mensagem recebida sem primeiro a guardar numa variável local.

Embora teoricamente nenhuma das estratégias seja aceitável pois origina computações diferentes das originais, seria interessante explorar a sua aplicação pois, como se verá nas secções seguintes, nem sempre é possível obter comportamentos óptimos em qualquer situação com estratégias sem memória. No entanto, nesta fase do projecto a complexidade introduzida por esta abordagem seria bastante grande, ficando a sua exploração relegada para trabalho futuro. Para concluir, apenas referir que formalmente se pretenderá sintetizar estratégias não determinísticas sem memória com o tipo $S \to 2^A$.

3.2 Síntese de programas a partir de especificações lógicas temporais

Uma das áreas com um interesse potencial para esta tese é a da síntese de programas a partir de especificações lógicas, especialmente se o modelo de programação utilizado se aproximar do modelo aqui utilizado, i.é, se se tratar de programas reactivos e concorrentes, e se a lógica de especificação utilizada for a lógica temporal ou similar. Se for possível capturar numa especificação lógica as

preferências e comportamento possível do processo, então poderia, em princípio, ser utilizado o trabalho desenvolvido nesta área para efectuar a compilação de um programa USPL através da sua tradução para um programa equivalente, mas escrito numa linguagem tradicional. Outra grande vantagem desta abordagem seria a possibilidade de raciocinar directamente sobre o comportamento do programa ao nível da especificação usando as ferramentas já existentes para a lógica temporal. No entanto, existem dois grandes problemas que dificultam a aplicação imediata desta abordagem: a obtenção da especificação lógica e a existência de técnicas suficientemente poderosas para sintetizar programas para essa especificação.

O primeiro destes problemas encontra-se parcialmente resolvido. Em [KMP94] mostra-se que, para todo o sistema de transição justo S, existe uma fórmula em lógica temporal linear $comp_S$, designada semântica temporal de S, que especifica o conjunto de todas as computações de S. Esta fórmula é importante porque, tal como se viu na secção 2.4, todas as computações do programa resultante deverão pertencer a Ω, o conjunto de todas as execuções que cumprem os critérios de justiça. Naturalmente, esta fórmula deverá pertencer à especificação final para se garantir que o programa sintetizado não possui computações não previstas no programa original.

Dado um FTS S expresso pelo tuplo $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, $comp_S$ define-se como

$$\Theta \wedge \Box \bigvee_{\tau \in \mathcal{T}} taken(\tau) \wedge \bigwedge_{\tau \in \mathcal{J}} just(\tau) \wedge \bigwedge_{\tau \in \mathcal{C}} compassionate(\tau) \qquad (3.1)$$

onde

- $taken(\tau) \equiv \rho_\tau(V, V^+)$ verifica-se numa posição j de uma computação se s_{j+1} for um sucessor de s_j por τ, ou seja, se se verificar $\langle s_j, s_{j+1} \rangle [\rho_\tau]$. De notar, que nesta fórmula os autores utilizam x^+ para referir o próximo valor de x em cada instante.

- $just(\tau) \equiv \Diamond\Box En(\tau) \supset \Box\Diamond taken(\tau)$ requer que, caso uma transição esteja continuamente activa a partir de uma determinada posição, ela seja executada um número infinito de vezes. Naturalmente, qualquer computação que verifica esta fórmula é justa em relaçao a τ.

- $compassionate(\tau) \equiv \Box\Diamond En(\tau) \supset \Box\Diamond taken(\tau)$ é semelhante à anterior, bastando agora que a transição esteja activa um número infinito de vezes.

Quanto às preferências temos um problema um pouco mais complicado. Devido à existência de um critério de decisão é possível determinar o conjunto das computações preferidas. É trivial caracterizar uma computação por uma fórmula temporal de comprimento infinito envolvendo o operador $\bigcirc$, e, consequentemente, seria possível caracterizar as computações preferidas por uma fórmula temporal. O grande problema é garantir a existência de uma fórmula equivalente, mas com comprimento tratável. Na prática, como muitos trabalhos de síntese traduzem a especificação temporal num autómato finito para reconhecimento de frases não-finitas (segundo [VW94] essa tradução é sempre

possível), seria suficiente traduzir as preferências para expressões ω-regulares[3] que descrevem a classe de linguagens reconhecidas por estes autómatos.

A fórmula (3.1) realça a importância do segundo problema referido anteriormente. A necessidade de expressar os critérios de justiça obriga a que a técnica de síntese utilizada seja suficientemente poderosa por forma a lidar com propriedades de resposta e reactividade. Existem vários trabalhos sobre síntese de programas a partir de especificações em lógica temporal linear, como por exemplo, [Tho95], [AMP95] ou [WTD91]. Uma das características interessantes do problema de síntese é que quase sempre é resolvido através da modelação do sistema recorrendo à teoria dos jogos, reduzindo-se a síntese ao cálculo de uma estratégia vencedora, no sentido em que permite tornar a fórmula de especificação sempre verdadeira. Na secção seguinte será apresentado sucintamente o primeiro destes trabalhos, e como se verá, com propriedades de resposta e reactividade a síntese nem sempre é trivial ou permite obter programas com as características desejadas. Um dos factos mais relevantes que se conclui dessa apresentação é que só para casos muito particulares de fórmulas temporais se conseguem determinar estratégias sem memória.

Também pode ser produtivo explorar a aplicabilidade da lógica temporal com tempo ramificado, como por exemplo o CTL* [ES88], pois a sua expressividade pode facilitar a especificação das preferências do processo. A definição das preferências obriga normalmente à comparação entre execuções, e como esta lógica é interpretada sobre árvores de execuções em vez de execuções isoladas, talvez essa comparação seja mais simples ou possível de expressar. Quanto à complexidade do problema de síntese, verifica-se que, no contexto dos sistemas reactivos, ambos os tipos de lógicas temporais são equivalentes [Var01, KMTV00], não existindo por isso qualquer penalização por se usar lógica temporal com tempo ramificado[4]. Para este tipo de lógica também existem alguns trabalhos de síntese, como por exemplo, [KMTV00] ou [MT98].

O trabalho apresentado em [AMP95] é bastante interessante porque os estados e as transições são caracterizados de forma simbólica, através de fórmulas lógicas, permitindo uma redução substancial no espaço de procura. O modo como se representam as transições é igual ao definido por Manna e Pnueli nos FTSs (aliás, este último também é co-autor desse trabalho), utilizando-se fórmulas definidas sobre variáveis simples e variáveis com apóstrofe, posteriormente interpretadas sobre estados adjacentes. Estas fórmulas são fortalecidas por um processo iterativo até que se chegue a um sistema de transição cujas execuções validem sempre a especificação. Esta semelhança com a semântica utilizada no USPL e o facto das estratégias sintetizadas não possuírem memória, torna este trabalho facilmente adaptável ao processo de compilação do USPL. O seu grande inconveniente é só lidar com fórmulas de invariância, do tipo $\Box p$, restringindo muito a possibilidade de se traduzir as preferências numa especificação em lógica temporal.

[3]Expressões do tipo $\cup_i \alpha_i(\beta_i)^\omega$, onde a união é finita, α e β são expressões regulares clássicas, e ω representa a repetição contável.

[4]Na realidade este facto só é totalmente verdade para uma sub-lógica do CTL* designada apenas por CTL.

3.2.1 Uma abordagem à síntese de estratégias em jogos infinitos

Outro problema interessante dentro desta área é o da síntese de estratégias vencedoras em jogos infinitos, quando as preferências são representadas através de predicados lógicos. Como já foi referido, a teoria dos jogos captura fielmente as interacções entre entidades com interesses diferenciados representados através de uma relação de preferência, sendo, como tal, um modelo alternativo muito interessante para representar os programas USPL. No entanto para capturar a noção de programa reactivo é necessário que os jogos sejam infinitos[5].

Uma possível abordagem a este problema foi desenvolvida por Wolfgang Thomas e apresentada em [Tho95]. Neste trabalho um jogo G com dois jogadores (um para modelar o programa e o outro o ambiente) é representado através de um tuplo

$$\langle S, S_0, S_1, A, \delta, \Omega \rangle$$

onde

- S é um conjunto finito ou contável de estados;
- S_0 e S_1 definem uma partição de S, contendo o conjunto S_i os estados onde é a vez do jogador i executar uma acção;
- A é um conjunto finito de acções;
- $\delta : S \times A \hookrightarrow S$ é uma função de transição parcial com as seguintes restrições:
  - o grafo é bipartido em relação às transições, ou seja, com algum abuso da notação, $\delta(S_i \times A) \subseteq S_{1-i}$ para $i \in \{0, 1\}$;
  - para todo $s \in S$ tem que existir $a \in A$ onde $\delta(s, a)$ esteja definida.
- Ω é a "componente de aceitação", utilizada para caracterizar as jogadas vencedoras como se verá mais tarde, que pode ser de uma das seguintes formas:
  - um conjunto de estados $F \subseteq S$;
  - uma colecção finita $\mathcal{F} = \{F_1, \ldots, F_m\}$ de conjuntos de estados de S;
  - uma sequência $\langle E_1, F_1, \ldots, E_m, F_m \rangle$ de conjuntos de estados de S.

Por vezes um estado $s \in S$ é designado estado inicial, passando o jogo a designar-se G_s. Uma jogada neste jogo é uma sequência $\gamma \in S^\omega$ tal que para quaisquer dois estados consecutivos, $\gamma(i)$ e $\gamma(i+1)$, existe uma acção a que verifica $\delta(\gamma(i), a) = \gamma(i+1)$, e $\gamma(0) = s$.

Num jogo G com estados S a decisão de quem ganha é fixada por um subconjunto $\mathcal{C}$ de S^ω, que será designado o predicado vitorioso. O facto de $\gamma \in S^\omega$ pertencer a $\mathcal{C}$ será representado por $\mathcal{C}(\gamma)$. Este predicado será definido por uma fórmula envolvendo fórmulas atómicas do tipo $s \in F$, sendo F um

[5] O conceito de jogo infinito é mais genérico que o de jogo repetido infinitamente. Basicamente, não se assume que a árvore de jogo tem uma profundidade limitada.

conjunto da componente de aceitação Ω. As fórmulas mais simples aparecem quando $\Omega = F$, sendo definidas da seguinte forma:

$$\begin{aligned}
\mathcal{C}(\gamma) &\equiv \exists i \cdot \gamma(i) \in F \\
\mathcal{C}(\gamma) &\equiv \forall i \cdot \gamma(i) \in F \\
\mathcal{C}(\gamma) &\equiv \exists j \cdot \forall i \geq j \cdot \gamma(i) \in F \\
\mathcal{C}(\gamma) &\equiv \forall j \cdot \exists i \geq j \cdot \gamma(i) \in F
\end{aligned}$$

Se tentarmos traduzir estas fórmulas para lógica temporal linear no espírito de [MP95] (ver anexo A), e considerando que o conjunto F representa uma fórmula de estado, temos, respectivamente, as seguintes condições:

$$\begin{aligned}
\mathcal{C}(\gamma) &\equiv \gamma \models \Diamond F \\
\mathcal{C}(\gamma) &\equiv \gamma \models \Box F \\
\mathcal{C}(\gamma) &\equiv \gamma \models \Diamond \Box F \\
\mathcal{C}(\gamma) &\equiv \gamma \models \Box \Diamond F
\end{aligned}$$

Quando Ω é $\mathcal{F} = \{F_1, \ldots, F_m\}$ foram estudadas fórmulas do tipo

$$\mathcal{C}(\gamma) \equiv \{f | \forall j \cdot \exists i \geq j \cdot \gamma(i) = f\} \in \mathcal{F}$$

designadas condições de Muller, e quando $\Omega = \langle E_1, F_1, \ldots, E_m, F_m \rangle$ fórmulas do tipo

$$\mathcal{C}(\gamma) \equiv \bigvee_{k=1}^{m} (\exists j \cdot \forall i \geq j \cdot \gamma(i) \notin E_k \wedge \forall j \cdot \exists i \geq j \cdot \gamma(i) \in F_k)$$

designadas condições de Rabin. Estas fórmulas podem ser reescritas em lógica temporal, respectivamente, como:

$$\mathcal{C}(\gamma) \equiv \gamma \models \exists i (\Box \Diamond F_i \wedge \forall j \neq i \cdot \neg \Box \Diamond F_j)$$

e

$$\mathcal{C}(\gamma) \equiv \gamma \models \bigvee_{k=1}^{m} (\neg \Box \Diamond E_k \wedge \Box \Diamond F_k)$$

As condições de Muller e Rabin são particularmente importantes porque são suficientemente expressivas para representar propriedades relevantes em sistemas concorrentes, como por exemplo propriedades de justiça.

Neste trabalho, uma estratégia π para o jogador i é uma função que decide, para cada segmento inicial de uma jogada terminado num estado em s_i, qual a acção a ser executada. Uma estratégia de um determinado jogador é vencedora se, para toda a jogada γ não excluída por ela, se verificar $\mathcal{C}(\gamma)$. Um resultado importante acerca da existência de estratégias vencedoras diz-nos que todo o jogo onde o predicado vitorioso for Borel[6] é determinado, ou seja, pelo menos

[6]Um predicado Borel pode ser qualquer fórmula de primeira ordem, onde as fórmulas atómicas são do tipo $R(i_1, \ldots, i_n)$ para qualquer relação numérica R, e não apenas fórmulas do tipo $\gamma(i) \in F$ (que pode ser traduzida numa igualdade simples induzindo uma coloração dos estados orientada por F).

um dos jogadores tem uma estratégia vencedora, sendo este o caso de todos os predicados apresentados anteriormente. No entanto, Wolfgang Thomas tem outras preocupações para além da existência, ou não, de estratégias vencedoras. Dado que o seu objectivo é sintetizar programas com um comportamento correcto e, sendo estes modelados pelo jogador 0 (o outro jogador é o ambiente), a sua investigação recai em questões um pouco mais específicas:

- Será que o jogador 0 tem uma estratégia vencedora?
- Será possível construir de forma eficiente uma estratégia vencedora para o jogador 0?
- Será possível construir uma estratégia para o jogador 0 cuja execução seja de baixa complexidade computacional?

O resultado mais importante de [Tho95] refere que, dado um jogo com um conjunto finito de estados e uma condição de Muller como predicado vitorioso, a partição do espaço de estados nos conjuntos de estados a partir dos quais cada um dos jogadores é vitorioso é computável (sendo apresentado um algoritmo que faz essa partição), sendo as estratégias vencedoras obtidas computáveis através de um autómato finito. Para chegar a este resultado, o autor também demonstrou que é possível obter estratégias vitoriosas sem memória quando os predicados vitoriosos são do tipo $\Diamond F$, $\Box\Diamond F$ ou um caso muito particular da condição de Rabin em que Ω verifica $E_1 \subset F_1 \subset \ldots \subset E_m \subset F_m$.

3.2.2 A linguagem de programação concorrente METATEM

Uma das analogias interessantes aplicáveis ao problema da síntese é que podemos considerar a lógica de especificação como uma vulgar linguagem de programação de entidades reactivas, sendo o mecanismo de síntese o processo de compilação dessa linguagem. Esta ideia foi o ponto de partida para uma linguagem de programação concorrente designada METATEM, que tem sido desenvolvida nos últimos anos por Michael Fisher [Fis94, Fis95]. Neste caso em vez de um mecanismo de síntese feito à priori, temos uma linguagem interpretada baseada na noção de objectos comunicantes e concorrentes com as seguintes premissas:

- O mecanismo básico de comunicação entre objectos consiste na difusão de mensagens.
- O comportamento dos objectos não é despoletado apenas pelas mensagens. A execução começa no momento em que são criados e continua mesmo que não haja troca de mensagens.
- Cada objecto tem associado um conjunto de fórmulas temporais que especifica o seu comportamento.
- Os objectos executam assincronamente.

A especificação do comportamento interno de cada objecto é feita numa fórmula em lógica temporal linear do tipo

$$\Box \bigwedge_{i=1}^{q} R_i$$

onde cada R_i representa uma regra do seu comportamento do tipo "fórmula passada e presente" implica "fórmula presente ou futura". Mais especificamente, as regras que são directamente executadas devem estar numa forma normal que engloba os seguintes tipos de fórmulas:

$$\begin{aligned}
\mathbf{start} &\supset \bigvee_{j=1}^{r} m_j \quad (\Box - \text{inicial}) \\
\circledcirc \bigwedge_{i=1}^{q} k_i &\supset \bigvee_{j=1}^{r} m_j \quad (\Box - \text{global}) \\
\mathbf{start} &\supset \Diamond l \quad (\Diamond - \text{inicial}) \\
\circledcirc \bigwedge_{i=1}^{q} k_i &\supset \Diamond l \quad (\Diamond - \text{global})
\end{aligned}$$

onde **start** é um predicado que apenas é válido no instante inicial, $\circledcirc\varphi$ é uma fórmula temporal que é válida num determinado instante se φ for válida no instante imediatamente anterior, e cada m_j, k_i e l são predicados atómicos. Note-se que grande parte das fórmulas da lógica linear podem ser convertidas para fórmulas deste tipo. Essencialmente, as regras $\Box$ são instruções de escolha não determinística que indicam as possíveis evoluções do estado interno do objecto (representado pelo conjunto dos predicados válidos), enquanto que as regras $\Diamond$ permitem representar os objectivos do agente que, na prática, tentarão ser atingidos o mais breve possível.

As mensagens comunicadas entre os agentes são predicados atómicos, possuindo cada objecto uma interface que define quais os predicados que este reconhece e quais os predicados que são visíveis para o exterior. Por exemplo, uma possível interface para um objecto que represente uma stack poderia ser

```
stack(pop,push)[popped,full]
```

sendo `pop` e `push` os predicados reconhecidos como mensagens de entrada e `popped` e `full` as mensagens produzidas pelo objecto. Se um possível programa para este objecto fosse[7]

$$\Box(\mathbf{start} \supset \texttt{popped}) \wedge (\circledcirc\texttt{pop} \supset \Diamond\texttt{popped}) \wedge (\circledcirc\texttt{push} \supset \texttt{full} \vee \texttt{popped})$$

o objecto começaria por difundir a mensagem `popped` e caso recebesse a mensagem `pop` comprometeria-se tentar enviar a mensagem `popped` o mais rápido possível (o que, neste caso, só seria possível caso recebesse a mensagem `push`).

Sem entrar em grandes detalhes, a execução das fórmulas temporais passa pelo seguinte ciclo interminável:

[7]Este programa não faz nada de útil servindo apenas de ilustração.

1. Actualizar a história passada do objecto adicionando-lhe as mensagens recebidas.

2. Verificar quais as regras activas, verificando os antecedentes que são válidos.

3. Executar conjuntamente as regras activas em conjunto com os compromissos transportados de ciclos anteriores (derivados de regras $\Diamond$). Isto implica coleccionar todos os consequentes das regras activas e os compromissos anteriores e tentar construir um novo estado que satisfaça todas as restrições. Como estas são constituídas por uma disjunção é necessário escolher de entre as várias possibilidades de evolução. Caso as restrições originem uma contradição é possível realizar *backtracking* por forma a explorar alternativas diferentes, com a limitação de não afectar comunicações previamente realizadas.

4. Todos os consequentes de regras $\Diamond$ que não possam ser satisfeitos neste ciclo são propagados para os seguintes sobre a forma de compromissos.

5. Voltar ao passo inicial.

Já existem muitas extensões a esta linguagem que permitem a sua aplicação num conjunto muito alargado de domínios. Nomeadamente, pode ser permitida a existência de interfaces dinâmicas, execução síncrona, comunicação ponto-a-ponto ou comunicação por grupos. Embora a lógica que especifica o comportamento de cada objecto seja baseada num modelo de tempo linear e discreto, como os objectos executam assincronamente, a semântica de um programa concorrente em METATEM teve de ser baseada num modelo de tempo denso, estando já previstas a maior parte das extensões referidas. Um dos resultados mais interessantes desta linguagem diz respeito à sua versão sequencial, e afirma que o procedimento de execução é correcto no sentido em que, caso exista algum modelo que satisfaça a especificação, então o procedimento de execução é capaz de gerar um desses modelos.

3.3 Síntese de agentes intencionais

Actualmente, a área da Inteligência Artificial tem sido cada vez mais identificada com o desenvolvimento de agentes inteligentes. Uma das definições de agente que se tem revelado mais construtiva, sendo por isso frequentemente utilizada, consiste na identificação do conceito de agente com um *sistema intencional* [WJ95]. Este termo foi "cunhado" pelo filosofo Daniel Dennett para identificar

> *"... entidades cujo comportamento pode ser previsto pelo método de atribuição de crenças, desejos e comportamento racional ..."* [Den87, p49].

A teoria dos sistemas intencionais tem origem na psicologia do senso comum: normalmente, os humanos explicam o comportamento de outros humanos, ou outros seres complexos, através da atribuição de atitudes intencionais, como

crenças e desejos, e determinando depois qual a acção mais provável de ocorrer dadas essas atitudes. A maior parte das previsões que fazemos baseiam-se neste estratégia e raramente são incorrectas[8].

Uma das combinações de atitude intencionais mais usadas para modelar agentes inteligentes é a que junta as crenças, desejos e intenções, originando o denominado paradigma BDI (*Beliefs, Desires and Intentions*). Neste paradigma os agentes são modelados de acordo com estas atitudes mentais, que permitem representar, respectivamente, o estado informacional, motivacional e deliberativo de um agente racional. Rao e Georgeff argumentam que estas três atitudes são necessárias para representar agentes que actuam em domínios com as seguintes características [RG95a, RG95b]:

1. O ambiente é não-determinístico.
2. Existem muitas acções ou procedimentos diferentes que podem ser executados em cada instante.
3. São estabelecidos diversos objectivos ao agente, podendo estes não ser simultaneamente alcançáveis.
4. As acções ou procedimentos que permitem atingir melhor os diversos objectivos são dependentes do estado do ambiente (ou contexto) e independentes do estado interno do agente.
5. As capacidades sensoriais são limitadas.
6. A frequência a que podem ser efectuadas as computações e as acções é comparável à frequência a que muda o ambiente.

Dada a característica 4, é essencial que o sistema possua informação sobre o estado do ambiente. Como esta não pode ser obtida instantaneamente sempre que necessária (devido às características 1 e 5), é necessário que exista um componente do sistema onde essa informação é armazenada, sendo actualizada após cada acto sensorial. Esse componente representa as crenças do agente. Também é necessário que o agente possua informação sobre os objectivos a ser atingidos ou, mais genericamente, sobre quais as prioridades e recompensas associadas a cada um dos objectivos definidos (características 3 e 4). Esta informação representa os desejos do agente. Devido à característica 6, o ambiente pode mudar significativamente durante a execução da acção determinada pelo processo de decisão do agente, ou mesmo durante o próprio de decisão. Para contornar este problema é necessário encontrar a estratégia de compromisso que constitua um balanço entre reconsiderar a decisão tomada a cada instante, com o consequente custo computacional, ou nunca reconsiderar, fazendo com que por vezes o agente não consiga atingir os seus objectivos. Para que este balanço possa ser encontrado é necessário que o agente possua um componente onde estão armazenados os resultados de decisões recentes. Este componente corresponde às intenções do agente.

[8]Por exemplo "A Sara pegou no seu guarda-chuva porque *acreditava* que ia chover" ou "O Alcino trabalhou arduamente porque *queria* obter o grau de Doutor".

Embora, tradicionalmente na IA, a investigação sobre a formalização de agentes racionais e sobre a sua implementação tenha procedido em paralelo com poucas contactos entre si, alguns autores começam já a abordar o problema da síntese de uma implementação concreta a partir de uma especificação do agente no paradigma BDI. Se virmos o USPL como uma linguagem de especificação para agentes intencionais e o processo de compilação, a apresentar no capítulo 4, como um mecanismo de obtenção de uma implementação concreta a partir de uma especificação, então esta área pode ser muito relevante para esta tese sendo importante apresentar uma pequena síntese de alguns dos seus trabalhos mais relevantes. De notar que no nosso caso não faz sentido falar em crenças pois todo o estado é acessível ao agente. Na versão concorrente do USPL essa atitude intencional já será relevante.

3.3.1 A abordagem de Rao e Georgeff

No Instituto Australiano de Inteligência Artificial, Anand Rao e Michael Georgeff desenvolveram desde o início dos anos 90 um dos mais trabalhos representativos sobre a utilização de lógicas multi-modais para a formalização de agentes segundo o paradigma BDI [RG95a, RG95b, Rao96]. A abordagem seguida pelos autores segue a tradição, usual neste paradigma, de utilizar uma semântica baseada em mundos possíveis, com relações de acessibilidade diferentes para modelar cada uma das atitudes intencionais.

O trabalho de Rao e Georgeff difere de outras formalizações de agentes usando lógicas multi-modais, na medida em que pretende resolver alguns dos problemas que estas apresentam, nomeadamente:

- Normalmente, não é claro qual o significado concreto da semântica associada aos diversos operadores modais, nem qual a relação entre estas e os métodos mais tradicionais de analisar a racionalidade, como a teoria da decisão.

- Embora sejam postulados vários axiomas para capturar as relações existentes entre as diferentes modalidades, não são normalmente propostas axiomatizações completas e consistentes para estas lógicas.

- Dado o poder expressivo destas lógicas, não são usualmente apresentados algoritmos para verificar a validade ou satisfabilidade de uma proposição, e, quando o são, nem sempre é analisada a sua complexidade computacional.

Para resolver o primeiro destes problemas, Rao e Georgeff começam por descrever como é que uma árvore de decisão clássica pode ser usada para representar os diversos estados intencionais de um agente, indicando depois como é que esta pode ser transformada num modelo quantitativo baseado em mundos possíveis. Quanto ao segundo, baseando-se no modelo semântico obtido, é apresentada uma família de sistemas axiomáticos consistentes e completos para descrever sistemas baseados em crenças, desejos e intenções. Os autores apresentam também uma taxonomia para classificação destes sistemas baseada nas inter-relações entre as três atitudes intencionais, catalogando as propriedades

que correspondem a essas inter-relações. Esta catalogação permite que a escolha do melhor sistema axiomático a usar seja baseada nas propriedades desejadas para os agentes a estudar. Finalmente, o terceiro problema é abordado através da utilização de algoritmos baseados em *tableaux*, sendo a sua complexidade analisada.

O ponto de partida para o modelo semântico proposto por Rao e Georgeff foi o estudo da possibilidade de representação do comportamento de um agente através de uma árvore de decisão. Uma árvore de decisão é uma árvore que caracteriza a evolução do comportamento de um sistema. Cada nodo desta árvore representa um estado do mundo e cada transição uma acção primitiva desencadeada pelo sistema, um evento primitivo que ocorre no ambiente, ou ambos. Existem três tipos de nodos: *nodos de decisão*, que representam um estado do mundo onde o sistema é chamado a escolher uma alternativa, representando as várias transições que deles saem as opções disponíveis; *nodos aleatórios*, que procuram modelar a incerteza e não-determinismo do ambiente, representando cada transição uma possível evolução do ambiente não controlada pelo sistema (estas transições podem ser etiquetadas pela respectiva probabilidade de ocorrência); *nodos terminais*, que representam os possíveis resultados da execução do sistema, estando cada um deles etiquetado pela utilidade que o sistema obtém caso a execução termine nesse nodo. Sobre árvores de decisão podem ser definidas *estratégias*, que essencialmente são funções definidas sobre os nodos de decisão e que indicam qual a acção que o sistema irá realizar nesse nodo. Se atentarmos nas características dos domínios onde Rao e Georgeff pretendem aplicar a sua teoria, facilmente se verifica que uma árvore de decisão modela o comportamento típico de um agente baseado em crenças, desejos e intenções. As crenças são modeladas através do conhecimento da estrutura da árvore, nomeadamente das probabilidades associadas aos resultados de acções não-determinísticas. Os desejos são modelados através das utilidades associadas aos nodos terminais e as intenções através de estratégias.

Embora fosse concebível a utilização directa de árvores de decisão para modelar agentes baseados em crenças, desejos e intenções, Rao e Georgeff optaram por transformar esse modelo num equivalente que representa explicitamente as crenças, desejos e intenções como relações de acessibilidade sobre conjuntos de mundos possíveis. Segundo os autores esta transformação proporciona uma melhor modelação em situações em que existe insuficiente informação sobre as probabilidades e recompensas, e também modela melhor os aspectos dinâmicos do domínio do problema.

A formalização proposta por Rao e Georgeff constitui uma extensão da *"Computation Tree Logic"* (nas variantes CTL e CTL*) que tem sido muito usada para descrever e raciocinar sobre sistemas concorrentes [ES88]. A essas lógicas foram acrescentados operadores modais para descrever as crenças, desejos e intenções dos agentes. Para além das regras usuais de formação de fórmulas de estado e de caminho existentes nestas lógicas, os autores acrescentaram a seguinte: se ϕ é uma fórmula de estado então BEL(ϕ), DES(ϕ) e INTEND(ϕ) são fórmulas de estado, sendo o seu significado informal, respectivamente, que o agente acredita em ϕ, deseja ϕ e tem a intenção de ϕ. Estas novas lógicas foram denominadas, respectivamente, BDI_{CTL} e $\text{BDI}_{\text{CTL}^*}$.

Posteriormente, estes autores desenvolveram um interpretador cujo objectivo seria executar as especificações de agentes segundo esta lógica [RG95a]. No entanto, o grande poder expressivo da mesma e a necessidade de gerar o comportamento óptimo em tempo de execução invalidaram a utilização de técnicas de prova na sua implementação. Em vez disso, os autores representaram cada uma das atitudes mentais por uma estrutura de dados concreta e, para que o sistema fosse funcional, restringiram bastante o poder expressivo da linguagem de especificação. Muito sucintamente, as características principais do interpretador são as seguintes: as crenças sobre o mundo só podem ser representadas por conjuntos de fórmulas atómicas; existe uma biblioteca de planos pré-definida que permite ao agente saber como forçar certos estados do mundo a ocorrer dadas as necessárias pré-condições; e as intenções são representadas por pilhas de planos hierarquicamente relacionados. Este método de especificação foi mais tarde consolidado numa linguagem de programação de agentes designada AgentSpeak(L), fortemente influenciada pelas linguagens de programação lógica [Rao96].

O grande problema desta abordagem é que, embora exista uma teoria de especificação baseada numa lógica multi-modal muito poderosa e bem desenvolvida, a distância que a separa da implementação final é tão grande que se torna pouco credível raciocinar sobre o comportamento final dos agentes partindo de possíveis resultados obtidos ao nível teórico. Este facto foi mesmo reconhecido por um dos autores aquando da apresentação da linguagem AgentSpeak(L) [Rao96]. Em vez de partir da especificação em lógica multi-modal para a implementação, este autor propõe, baseado numa semântica operacional para esta linguagem, um processo de atribuir à posteriori as atitudes mentais aos agentes, possibilitando raciocinar sobre o seu comportamento de forma mais realista.

Um dos pontos fortes deste trabalho é que uma especificação baseada directamente em atitudes mentais pode ser mais facilmente dominada por utilizadores finais do sistema (tipicamente os detentores do conhecimento sobre um determinado domínio de aplicação) e não apenas por programadores experimentados. Pelo contrário, especificar agentes em USPL implica um certo domínio de linguagens de programação imperativas convencionais, mas como o nível de especificação é menos elevado existem perspectivas de existir uma maior correcção formal na passagem para a implementação.

3.3.2 A abordagem de Wooldridge

Michael Wooldridge desenvolveu um formalismo para especificar agentes intencionais cujo objectivo seria potenciar a síntese de implementações concretas a partir da respectiva especificação [Woo95b]. Este formalismo diverge um pouco da abordagem tradicional no paradigma BDI estando baseado numa lógica do tipo *branching* com as conectivas modais *knowledge* e *seeing to it that* (stit). A estas conectivas foi dada uma semântica concreta em termos dos estados e acções que englobam o modelo de um agente, tornando possível estabelecer um relacionamento preciso entre a linguagem de especificação e os autómatos determinísticos, abrindo o caminho para a síntese de agentes a partir de especificações lógicas.

As fórmulas da lógica de especificação (designada $\mathcal{L}$) são interpretadas sobre estruturas tipo árvore, que representam todas as possíveis evoluções do mundo. Um caminho sobre esta estrutura está associado com a noção de escolha por parte do agente: em cada estado ao longo de um caminho, o agente deve seleccionar a acção a executar dentro de um leque de alternativas possíveis; a execução da acção escolhida limita a evolução do mundo a um subconjunto dos estados acessíveis a partir desse estado. Formalmente, $\mathcal{L}$ é uma versão proposicional da lógica CTL*, à qual foram acrescentadas as conectivas modais K (*knowledge*) e S (*stit*). A fórmula $K\varphi$ representa o facto de que a informação φ está implícita no estado do agente: φ não precisa de estar representada dentro do agente e não existe o sentido de a informação estar imediatamente disponível para o agente (o conhecimento é uma noção externa ao agente). A fórmula $S\varphi$ representa o facto de φ ser satisfeita como uma consequência necessária de uma escolha que o agente tenha feito.

Quanto à semântica, um modelo M para $\mathcal{L}$ é um tuplo $\langle G, R, A, Act, \pi\rangle$ onde

- $\langle G, R, Act\rangle$ é uma estrutura do tipo *branching* com acções associadas às transições, sendo $R \subseteq G \times G$ uma relação de transição que representa as possíveis execuções do sistema e $Act : R \to A$ uma função que determina a acção envolvida em cada transição;

- $\pi : \Phi \times G \to \{T, F\}$ é uma função de valoração que determina a validade de todas as proposições $p \in \Phi$ no conjunto de estados globais G.

Este formalismo pretende representar as propriedades de sistemas contendo uma parte passiva correspondente ao *ambiente* e uma parte activa, correspondente a um *agente* autónomo, que é capaz de mudar o estado do ambiente através da execução de acções. Assim, o estado global do sistema é formado pelo produto dos estados do ambiente e dos estados locais do agente. Os primeiros são denominados por $E = \{e, e', \ldots\}$, os segundos por $L = \{l, l', \ldots\}$, e o conjunto dos estados globais por $G = E \times L = \{g, g', \ldots\}$. Sobre os estados globais estão definidas as respectivas projecções $es : L \to E$ e $ls : G \to L$.

O conhecimento do agente é modelado por uma relação de equivalência $\sim \subseteq G \times G$ que define quais os estados globais que lhe são indistinguíveis. Dois estados globais são indistinguíveis para um agente se o seu estado interno for igual para os dois, ou seja, $g \sim g'$ sse $ls(g) = ls(g')$. A definição da semântica do operador *stit* implica a definição da relação de equivalência $\equiv \subseteq G \times G$, representando $g \equiv g'$ o facto de que g e g' são possíveis resultados de uma decisão tomada pelo agente. Assim, $g \equiv g'$ sse $\exists g'' \in G \cdot \{(g'', g), (g'', g')\} \subseteq R \wedge Act(g'', g) = Act(g'', g')$.

A semântica da lógica $\mathcal{L}$ é na sua maior parte a usual para a lógica temporal CTL*, sendo aqui omitida por questões de espaço. A semântica dos dois operadores modais é dada pelas seguintes regras, onde M representa um modelo e g um estado (as fórmulas do CTL* estão divididas em fórmulas de estado e fórmulas de caminho, e se φ é uma fórmula de estado então $K\varphi$ e $S\varphi$ também o são):

$$(M,g) \models K\varphi \quad \text{sse} \quad \forall g' \in G \cdot g \sim g' \Rightarrow (M,g') \models \varphi$$
$$(M,g) \models S\varphi \quad \text{sse} \quad \forall g' \in G \cdot g \equiv g' \Rightarrow (M,g') \models \varphi \wedge \exists g' \in G \cdot (M,g') \not\models \varphi$$

O objectivo final deste trabalho seria conseguir sintetizar automaticamente a implementação a partir de uma especificação feita em $\mathcal{L}$ (uma especificação deve ser entendida como uma conjunção de fórmulas de $\mathcal{L}$, representando cada um dos termos uma propriedade desejada para o sistema). Tendo em vista tal objectivo, Wooldridge começa por formalizar a noção de agente como sendo uma máquina sequencial de Moore. Assim, um agente α é um tuplo $\langle L, E, A, ns, c, l_0 \rangle$ onde

- $L = \{l, l', \ldots\}$ é um conjunto de estados locais ao agente;
- $E = \{e, e', \ldots\}$ é um conjunto de estados do ambiente, que aqui devem ser entendidos como estímulos para o agente;
- $A = \{a, a', \ldots\}$ representa o reportório de acções disponíveis para o agente;
- $ns : E \times L \rightarrow L$ é uma função que determina o próximo estado de um agente dado o seu estado actual e um estímulo vindo do exterior;
- $c : L \rightarrow A$ é uma função de escolha que determina a acção a tomar em cada estado; e
- $l_0 \in L$ representa o estado inicial do agente.

Dado um agente $\alpha \equiv \langle L, E, A, ns, c, l_0 \rangle$ e uma estrutura do tipo branching $\langle G, R, Act \rangle$ é agora pertinente saber se essa estrutura representa de facto todas as execuções possíveis de um sistema composto por α. Assim, diz-se que $\langle G, R, Act \rangle$ *representa* α sse:

1. $ls(root(R)) = l_0$;
2. $\forall g, g' \in G \cdot (g, g') \in R \Rightarrow Act(g, g') = c(ls(g))$;
3. $\forall g, g' \in G \cdot (g, g') \in R \Rightarrow ns(g) = ls(g')$.

Esta relação *representa* pode ser estendida a modelos de $\mathcal{L}$, escrevendo-se $representa(M, \alpha)$ se a estrutura *branching* contida em M representar o agente α. Um agente α satisfaz uma especificação φ (facto denotado por $\{\alpha\} \models \varphi$) sse, para todos os modelos determinísticos M, $representa(M, \alpha)$ implica $M \models \alpha$.

Dado um modelo determinístico M é possível sintetizar automaticamente um agente $agente(M)$ tal que $representa(M, agente(M))$. Para tal são necessárias duas funções auxiliares, $\hat{c}$ e $\widehat{ns}$, que dado um caminho p sobre uma estrutura *branching* determinística $\langle G, R, Act \rangle$ extraem, respectivamente, a função de escolha e a função de próximo estado associadas com p. A primeira define-se como

$$\hat{c} \stackrel{def}{=} \{l \mapsto a \mid \exists u \in \mathcal{N} \cdot ls(p(u)) = l \wedge Act(p(u), p(u+1)) = a\}$$

e a segundo como

$$\widehat{ns} \stackrel{def}{=} \{g \mapsto l \mid \exists u \in \mathcal{N} \cdot p(u) = g \wedge ls(p(u+1)) = l\}$$

Dado um modelo determinístico $M \equiv \langle G, R, A, Act, \pi \rangle$ então é possível sintetizar o $agente(M) \equiv \langle L, E, A, ns, c, l_0 \rangle$, onde:

$$\begin{array}{rcl} L & \stackrel{def}{=} & \{ls(g) \mid g \in G\} \\ E & \stackrel{def}{=} & \{es(g) \mid g \in G\} \\ A & \stackrel{def}{=} & \mathrm{ran} Act \\ ns & \stackrel{def}{=} & \bigcup\{\widehat{ns}(p) \mid p \in fpaths(G,R)\} \\ c & \stackrel{def}{=} & \bigcup\{\hat{c}(p) \mid p \in fpaths(G,R)\} \\ l_0 & \stackrel{def}{=} & ls(root(R)) \end{array}$$

De notar que o método proposto por Wooldridge para extrair agentes de modelos apenas funciona para modelos determinísticos e para modelos com um número finito de estados. Para completar o ciclo de síntese falta no entanto esclarecer como é que, dada uma especificação φ, se determina o modelo M tal que $M \models \varphi$. Wooldridge não apresenta em [Woo95b] nenhum método para o fazer, mas sugere que tal poderá ser feito recorrendo a técnicas construtivas para prova de teoremas.

Para finalizar a apresentação deste formalismo, apresentam-se três justificações adiantadas por Wooldridge em favor da utilização da modalidade *stit* em detrimento das mais comuns *crenças*, *desejos* ou *intenções*:

- Primeiro, esta modalidade proporciona um elevado nível de abstracção para a especificação de agentes, visto permitir uma separação desejável entre *o que* se quer que um agente faça e *como* é que o agente o irá fazer. Quando se especifica que um agente *stit* φ, não há a preocupação de saber como é que um agente irá originar φ, apenas surgindo essa preocupação na altura da implementação.

- Segundo, ao contrário das *crenças*, *desejos* e *intenções*, é relativamente fácil dar uma semântica concreta ao nível do modelo para a modalidade *stit*.

- Terceiro, *stit* é apenas uma simples ferramenta de abstracção e não um construtor intencional como os *desejos* ou *intenções*. Sendo assim, não é necessário haver preocupação com a relação entre a modalidade S e a estrutura cognitiva dos agentes.

3.3.3 A abordagem de Brafman e Tennenholtz

Brafman e Tennenholtz desenvolveram um trabalho na área da modelação de agentes usando técnicas qualitativas [BT97] que, embora difira do nosso nos seus objectivos finais, partilha algumas opções ao nível dos modelos, sendo por

isso aqui descrito brevemente. A ideia fundamental deste trabalho consiste em, dada uma implementação concreta de um agente, obter um modelo do mesmo em termos de atitudes mentais ou intencionais. Este trabalho é de certa forma semelhante ao que alguns autores, como por exemplo Wooldridge [Woo95a], tem tentado fazer em relação às lógicas multi-modais tentando dar-lhes uma semântica em termos de execuções reais de agentes. A ideia é poder raciocinar sobre o comportamento dos agentes usando uma lógica multi-modal semelhante à desenvolvida por Rao e Georgeff, mas com garantias de que os resultados obtidos tem correspondência no comportamento real dos agentes.

O modelo concreto para agentes definido por estes autores é baseado numa máquina de estados, com um conjunto de estados locais, um conjunto de acções possíveis e um programa definido como uma estratégia. Um agente existe num ambiente, que também é modelado como uma máquina de estados e que serve para representar tudo o que lhe é exterior (incluindo possivelmente outros agentes). O não-determinismo só existe no contexto local do agente, sendo o resultado de executar uma acção uma função determinística do estado global do sistema. Formalmente, um agente $\mathcal{A}$ é definido pelo tuplo

$$\langle \mathcal{L}_{\mathcal{A}}, A_{\mathcal{A}}, \mathcal{P}_{\mathcal{A}} \rangle$$

onde $\mathcal{L}_{\mathcal{A}}$ é o conjunto de estados locais do agente, $A_{\mathcal{A}}$ é o conjunto de acções que $\mathcal{A}$ pode realizar e $\mathcal{P}_{\mathcal{A}} : \mathcal{L}_{\mathcal{A}} \hookrightarrow \mathcal{A}_{\mathcal{A}}$ é o seu protocolo (é uma função parcial porque podem existir estados onde não está especificada a acção a realizar). O conjunto dos estados globais do sistema é definido por $\mathcal{L}_{\mathcal{A}} \times \mathcal{L}_{\mathcal{E}}$, onde $\mathcal{L}_{\mathcal{E}}$ é o conjunto de estados do ambiente. O conjunto de estados iniciais será denotado por $I \subseteq \mathcal{L}_{\mathcal{A}} \times \mathcal{L}_{\mathcal{E}}$ e a função de transição do sistema definida como $\tau : (\mathcal{L}_{\mathcal{A}} \times \mathcal{L}_{\mathcal{E}}) \times A_{\mathcal{A}} \rightarrow (\mathcal{L}_{\mathcal{A}} \times \mathcal{L}_{\mathcal{E}})$.

Para se compreender o modelo a nível mental é necessário introduzir primeiro alguns conceitos auxiliares. Uma execução é uma sequência de estados globais $s_0, s_1, s_2, \ldots$ e diz-se possível para um agente $\mathcal{A}$ sse $\forall k > 0 \cdot \exists a \in A_{\mathcal{A}} \cdot \tau(s_{k-1}, a) = s_k$. O conjunto de todas as execuções possíveis é denotado por $\mathcal{R}$ e conjunto de todos os sufixos de execuções possíveis por $\mathcal{R}_{suf}$. Uma execução diz-se consistente com a estratégia de $\mathcal{A}$ sse for possível para esse agente e a acção escolhida em cada estado for a definida pela estratégia ou esta não se encontrar definida nesse estado. Dado um estado local l, $PW_I(l) = \{s_0 \in I \mid s_0, s_1, \ldots \textit{ é uma execução consistente de } \mathcal{A} \textit{ e } \exists n \geq 0, e \in \mathcal{L}_{\mathcal{E}} \cdot s_n = (l, e)\}$ é o conjunto de estados iniciais possíveis em l.

O modelo a nível mental de um agente $\mathcal{A}$ é determinado pelo tuplo

$$\langle \mathcal{L}_{\mathcal{A}}, A_{\mathcal{A}}, B, u, \rho \rangle$$

onde $B : \mathcal{L}_{\mathcal{A}} \rightarrow (2^I - \emptyset)$, tal que $\forall l \in \mathcal{L}_{\mathcal{A}} \cdot B(l) \subseteq PW_I(l)$, representa as suas crenças (implicitamente, esta função define o conjunto de execuções que o agente acha plausível terem-no conduzido ao seu estado actual, se assumirmos que a estratégia é conhecido pelo menos até esse estado), $u : \mathcal{R}_{suf} \rightarrow \mathbb{R}$ é a função de utilidade que representa as suas preferências e $\rho : \cup_{n \in \mathbb{N}} 2^{\mathbb{R}^n} \rightarrow (\cup_{n \in \mathbb{N}} 2^{\mathbb{R}^n} - \emptyset)$ (i.é, uma função de e para conjuntos de tuplos de reais de comprimento idêntico), tal que $\forall \mathcal{U} \in \cup_{n \in \mathbb{N}} 2^{\mathbb{R}^n} \cdot \rho(\mathcal{U}) \subseteq \mathcal{U}$, representa o seu critério de decisão. Aparte a

noção de crença, que no nosso caso não faz muito sentido devido à não existência de incerteza, este modelo é, nos seus princípios, muito semelhante ao de um UFTS. A principal diferença é a definição das preferências sobre sufixos de execuções ao contrário de apenas sobre estados. Esta diferença leva também a uma alteração na definição do critério de decisão que, neste caso apenas necessita de escolher entre conjuntos de utilidades, enquanto que num UFTS é necessário escolher entre conjuntos de sequências infinitas de utilidades. O resultado final acaba por ser o mesmo, só que neste caso os autores optaram por contornar o problema das execuções infinitas logo na definição da função de utilidade, enquanto que nós relegamos essa solução para o critério de decisão.

Dado não ser relevante para esta tese o processo de determinação do modelo a nível mental a partir do modelo concreto não será aqui apresentado. No entanto, embora esse não seja o objectivo principal dos autores, neste trabalho é indirectamente referidos um facto relevante sobre o mecanismo de decisão que permitiria obter um protocolo a partir de um modelo mental. Como as preferências estão definidas sobre sufixos é necessário fazer a escolha entre protocolos e não apenas entre acções, pois apenas um protocolo origina a sequência de estados à qual é possível aplicar a função de utilidade. Este facto leva à necessidade de enumerar os protocolos possíveis no processo de escolha. No entanto, este conjunto cresce exponencialmente com o número de passos de decisão, tornando este processo proibitivo para casos reais. A alternativa seria tornar possível escolher uma acção de cada vez. Para isso seria necessário converter a função de utilidade definida sobre sufixos para uma valoração equivalente definida apenas sobre estados (aliás, um processo semelhante ao usado nos MDPs). Os autores demonstraram que essa conversão é possível quando se verificarem as seguintes condições:

- Todas as execuções devem ter um comprimento limitado, no sentido em que todas tem um sufixo onde todos os estados são iguais.

- As preferências devem obedecer a uma condição de admissibilidade que diz o seguinte: dados dois estados locais sucessivos l e l', se $PW_I(l') \cap B(l) \neq \emptyset$ então $B_I(l') = PW_I(l') \cap B_I(l)$ senão $B_I(l')$ pode ser qualquer subconjunto de $PW_I(l')$.

- O critério de decisão deve obedecer ao principio "sure-thing" que diz o seguinte: sejam (w, w') e (v, v') pares de vectores de números reais, tal que $|w| = |v|$ e $|w'| = |v'|$, então $v \circ w$ é pelo menos tão preferido quanto $v' \circ w'$ sempre que v for pelo menos tão preferido quanto v' e w for pelo menos tão preferido quanto w'.

Ou seja, para se utilizar este modelo para implementar agentes seria necessário restringir muito a classe de entidades implementáveis pois a primeira restrição limita todo o tipo de comportamento infinito. Como se verá na análise dos problemas de decisão de Markov, é possível lidar com uma maior variedade de modelos partindo de uma representação diferente para as preferências.

3.4 Implementação de agentes orientados por funções de utilidade

Como se depreende desta análise dos trabalhos sobre a programação de agentes intencionais, poucos trabalhos representam os desejos explicitamente como uma função de utilidade. A abordagem mais típica consiste em tratá-los apenas como objectivos, ou seja, dividindo os estados do mundo entre os desejados e os não desejados. Esta é por exemplo a abordagem presente nas formalizações baseadas em lógica multi-modal, da qual o trabalho de Rao e Georgeff representa uma solução típica. No trabalho de Brafman e Tennenholtz já se usa uma representação dos desejos como uma função de utilidade, mas dados os objectivos dos autores não se adianta muito sobre o mecanismo de decisão dos agentes.

A abordagem mais elegante e bem sucedida para a implementação de agentes orientados por funções de utilidade foi sintetizada por Russell e Norvig em [RN95]. Basicamente, estes autores identificam esta classe de agentes com os problemas de decisão sequencial tratados à muito tempo na área da investigação operacional. Quando o estado do ambiente é completamente acessível ao agente, estes problemas são conhecidos como problemas de decisão de Markov e possuem técnicas de resolução mesmo para alguns casos em que a sequência de decisões é ilimitada.

3.4.1 Problemas de decisão de Markov

Diz-se que um sistema de decisão possui a propriedade de Markov se as transições que partem de um determinado estado dependem apenas desse estado e não da sequência de estados passados. Um processo de decisão de Markov (MDP - *Markov Decision Process*) apresenta-se com um modelo para um agente que interage sincronamente com o seu ambiente[9]. Os MDPs só permitem representar agentes que não possuem incerteza quanto ao estado em que se encontram. No entanto, poderá existir incerteza quanto ao resultado de executar uma acção.

Definição 3.1 *Um MDP pode ser descrito formalmente como um tuplo* $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ *onde:*

- $\mathcal{S}$ *é um conjunto finito de possíveis estados do mundo;*
- $\mathcal{A}$ *é um conjunto finito de acções que o agente pode executar;*
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ *é uma função de transição de estados, que associa a cada possível estado e acção uma distribuição de probabilidade sobre estados; e*
- $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ *é uma função de recompensa, que captura o benefício que o agente tem por executar cada uma das acções possíveis num estado.*

[9] Estes modelos são descritos em muitos livros na área da investigação operacional, sendo a descrição aqui apresentada baseada em [Put90] e [KLC95].

Critérios de optimização

O objectivo destes modelos é permitir descobrir uma estratégia para o agente que maximize alguma medida que envolva as recompensas recebidas ao longo de uma execução. Quando o horizonte de execução é finito, a medida é normalmente a soma total das recompensas recebidas. Supondo que se pretende optimizar num horizonte de k passos e que r_t representa a recompensa recebida no passo t, então a medida a maximizar é

$$\sum_{t=0}^{k-1} r_t$$

Quando o horizonte é ilimitado não é tão óbvio qual a medida que se deverá utilizar para maximizar. Normalmente usa-se uma das seguintes:

- Recompensa total descontada. Usando um factor de desconto $0 \leq \gamma < 1$, deve-se maximizar

$$\sum_{t=0}^{\infty} \gamma^t r_t \tag{3.2}$$

  Esta medida valoriza de diferente forma as recompensas recebidas em diferentes instantes de tempo, sendo mais relevantes as recompensas recebidas num futuro próximo

- Limite da soma das recompensas. A medida a maximizar é

$$\lim_{N \to \infty} \sum_{t=0}^{N} r_t$$

  Este critério valoriza todas as recompensas recebidas em instantes diferentes de igual forma, sendo sensível a mudanças na recompensa de um período.

- Limite da média das recompensas. A medida a maximizar é

$$\lim_{N \to \infty} \frac{\sum_{t=0}^{N} r_t}{N}$$

  Este critério também valoriza todas as recompensas recebidas em instantes diferentes de igual forma, sendo, no entanto, insensível a diferenças de recompensa em períodos finitos.

A utilização de factores de desconto é muito frequente devido aos seguintes factores[RN95, Put90]:

- É uma boa técnica para não ter que lidar com infinitos. É muito difícil tomar decisões quando um agente tem a possibilidade de executar indefinidamente devido à existência de ciclos. A recompensa total recebida ao longo de execução infinita pode também ser ilimitada e obrigaria à comparação, nem sempre fácil, entre valores infinitos para se poder determinar a melhor estratégia. Dado que $0 \leq \gamma < 1$ e que as recompensas não variam ao longo do tempo a soma (3.2) converge para um valor finito, facilitando a comparação entre as recompensas totais.

- Constitui um modelo adequado para o comportamento verificado nos seres humanos. Devido à sua vida muito longa também possuem os mesmos problemas a quando da tomada de decisões, optando muitas vezes implicitamente por uma estratégia semelhante, valorizando mais as recompensas recebidas num futuro próximo do que num futuro longínquo.

- Deriva de um principio de estacionariedade bastante razoável de aceitar aquando da comparação de sequências de recompensas. Este principio diz o seguinte: se duas sequências de recompensas $R_1, R_2, R_3, \ldots$ e $S_1, S_2, S_3, \ldots$ começam com a mesma recompensa (i.e., $R_1 = S_1$), então o resultado de comparar as duas sequências deve ser igual ao resultado de comparar as sequências $R_2, R_3, \ldots$ e $S_2, S_3, \ldots$.

- Economicamente faz sentido devido à taxa de desvalorização da moeda.

Vejamos alguns exemplos por forma a ilustrar melhor os três critérios de maximização apresentados para horizontes ilimitados. Usando um factor de desconto qualquer, a sequência de recompensas $1, -1, 0, 0, \ldots$ é preferível à sequência $0, 0, \ldots$, sendo, no entanto, estas sequências indiferentes de acordo com os outros dois critérios. A sequência $-1, 2, 0, 0, \ldots$ é preferível a $0, 0, \ldots$ pelo critério do limite das somas das recompensas, mas são indiferentes de acordo com o limite das médias. A sequência $0, \ldots, 0, 1, 1, \ldots$ é preferível a $1, 0, 0, \ldots$ de acordo com o limite das médias, mas existe sempre pelo menos um factor de desconto que torna a segunda mais preferível que a primeira.

Cálculo de estratégias óptimas

Uma estratégia especifica qual a regra de decisão que deve ser usada em cada instante do tempo. Uma regra de decisão especifica qual a acção que deve ser escolhida em cada estado, podendo ser classificada da seguinte forma:

- *Markoviana* ou dependente do historial. No primeiro caso, as regras de decisão apenas dependem do estado actual do sistema, enquanto que no segundo caso podem depender também de todo o seu passado.

- Pura ou não-determinística. Uma regra de decisão pura especifica uma única acção para cada estado, enquanto que uma regra não-determinística especifica uma distribuição de probabilidades sobre as acções permitidas em cada estado.

Por sua vez, as estratégias podem ser de dois tipos:

- Estacionárias, quando especificam a mesma regra de decisão para todos os estados de tempo. Iremos assumir que, neste caso, π representa a regra de decisão escolhida.

- Não estacionárias. Uma estratégia não estacionária consiste numa família de regras de decisão indexada pelo instante de tempo. π_t representa a regra de decisão que deverá ser usada para escolher a acção a executar quando faltarem t passos de execução.

para $s \in \mathcal{S}$ **do**
    $V_1(s) := 0$
od
$t := 1$
repetir
    $t := t + 1$
    para $s \in \mathcal{S}, a \in \mathcal{A}$ **do**
        $Q_t^a(s) := R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V_{t-1}(s')$
        $V_t(s) := \max_a Q_t^a(s)$
    od
ate $\forall s \in S \cdot |V_t(s) - V_{t-1}(s)| < \epsilon$

Figura 3.1: O algoritmo *value iteration*.

Na apresentação que se segue apenas consideramos regras de decisão *Markovianas* e puras, ou seja, cujo tipo é $\mathcal{S} \to \mathcal{A}$.

Na optimização usando um horizonte finito a melhor estratégia é frequentemente não estacionária. Howard demonstrou que usando factores de desconto e um horizonte infinito existe sempre uma estratégia óptima estacionária[How60]. Para determinar esta estratégia óptima com este critério, começa-se por calcular uma função de valoração que associa a cada estado a recompensa recebida se se executar essa estratégia a partir desse estado. Esta função de valoração V^* é a única solução do seguinte conjunto de equações (uma por cada estado):

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V^*(s') \right]$$

Se $0 \leq \gamma < 1$ e for possível calcular o máximo então esta equação tem solução. Em [Put90] são apresentadas algumas condições para que este máximo exista e para a existência de estratégias óptimas estacionárias. Depois de determinado V^*, a estratégia óptima π^* determina-se através do seguinte conjunto de equações (essencialmente, limita-se a escolher, para cada estado, a acção que maximiza a recompensa imediata mais a média pesada das recompensas recebida a partir dos possíveis estados destino ajustada pelo factor de desconto):

$$\pi^*(s) = \operatorname*{argmax}_a \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V^*(s') \right] \tag{3.3}$$

Existem vários algoritmos para determinar estratégias óptimas para MDPs com o critério da recompensa total descontada, de entre os quais se destaca o *value iteration*, cujos fundamentos foram apresentados pela primeira vez em [Bel57]. Este algoritmo determina o valor de V^* por aproximações sucessivas até que a diferença do seu valor entre duas iterações consecutivas seja inferior a um determinado ϵ. Este algoritmo pode ser consultado na figura 3.1.

Quando o algoritmo *value iteration* termina V_t não difere de V^* mais do que $(2\epsilon\gamma)/(1-\gamma)$. Este resultado permite quantificar com precisão a qualidade

da solução obtida, embora não haja garantias de que a estratégia obtida é a estratégia óptima.

Frequentemente, a estratégia óptima π^* não é muito sensível ao valor da função de valoração V^* devido à natureza da equação (3.3). Este facto sugere que se poderia usar um algoritmo ligeiramente diferente do *value iteration* para determinar a estratégia óptima, onde em vez de se tentar chegar iterativamente à função de valoração óptima se procura descobrir iterativamente a estratégia óptima. Neste algoritmo começa-se por escolher uma estratégia qualquer, calculando-se depois a função de valoração para cada estado dada essa estratégia. Depois actualiza-se a estratégia por forma a reflectir a nova função de valoração e procede-se iterativamente até que a estratégia estabilize, ou seja, prescreva as mesmas acções em todos os estados em duas iterações consecutivas. Analogamente, este algoritmo introduzido em [How60] chama-se *policy iteration*.

Os *Markov Decision Problems* cuja medida de recompensa é a soma total das recompensas descontadas ou o limite da média das recompensas podem ser resolvidos usando técnicas de programação linear num número de operações aritméticas polinomial em $|\mathcal{S}|$ (o número de estados), em $|\mathcal{A}|$ (o número de acções), e no número de bits necessários para codificar as recompensas imediatas e as probabilidades de transição como números racionais (este parâmetro será designado por B)[10]. A prova desta complexidade obtém-se formulando este problema como um programa linear. Para além de polinomial, este problema é também P-completo. O algoritmo *value iteration* apresentado anteriormente tem um número de iterações polinomial em $|\mathcal{S}|$, $|\mathcal{A}|$, B e em $1/(1-\gamma)$, tendo cada iteração uma complexidade $O(MN^2)$.

Quando a função de transição é determinística (i.é, T devolve sempre 0 ou 1), o problema pode ser resolvido num número de operações polinomial em $|\mathcal{S}|$ e em $|\mathcal{A}|$, pertencendo à classe NC[11][PT87]. Este resultado sugere que a natureza estocástica dos MDPs tem uma grande influência na complexidade da sua resolução, devendo a sua estrutura ser mais aproveitada por forma a se encontrarem algoritmos mais eficientes para determinadas classes de problemas.

Em [Put90] é apresentada uma síntese de alguns métodos que podem ser utilizados para resolver MDPs com critérios de optimização diferentes da recompensa total descontada. No entanto, estes métodos apenas são aplicáveis em condições muito particulares, sendo a sua utilização limitada a classes muito particulares de MDPs.

Técnicas para redução do espaço de estados

Um dos problemas de se modelar problemas de decisão como MDPs é a complexidade do cálculo da melhor estratégia quando o número de estados é relativamente grande. Para minimizar este problema existem várias técnicas que

[10] Os resultados de complexidade aqui apresentados devem-se a vários autores, sendo [LDK95] uma excelente síntese dos mesmos.

[11] Classe de todos os problemas resolúveis em tempo paralelo polilogarítmico e com um esforço total polinomial. Para mais informações sobre esta classe consultar, por exemplo, [Pap94].

permitem determinar MDPs abstractos equivalentes mas com um conjunto de estados mais pequeno, cuja resolução se pretende equivalente à resolução do MDP original. Nesta secção vamos apresentar brevemente duas técnicas que permitem efectuar esta optimização.

A primeira deve-se a Thomas Dean e Robert Givan [DG97] e baseia-se no conceito de *stochastic bissimulation homogeneity* para partir $\mathcal{S}$ em conjuntos de estados com um comportamento idêntico. Seja $P = \{B_1, \ldots, B_n\}$ uma partição de $\mathcal{S}$. P apresenta a propriedade de *stochastic bissimulation homogeneity* (ou, para simplificar, é homogénea) em relação a um determinado MDP sse se verificar a seguinte propriedade:

$$\forall B, C \in P, \alpha \in \mathcal{A},\, p, q \in B \cdot \sum_{r \in C} T(p, \alpha, r) = \sum_{r \in C} T(q, \alpha, r)$$

Quando uma partição P é homogénea então todos os seus conjuntos são estáveis. Um conjunto $B \in P$ é estável em relação a um conjunto $C \in P$ e uma acção $\alpha \in \mathcal{A}$ sse em todos os estados de B existe a mesma probabilidade de se transitar para o conjunto C através da acção α. Uma partição Q é um refinamento de P sse todo o conjunto de Q é um subconjunto de um conjunto de P. Quando isso se verifica diz-se também que P tem uma maior granularidade que Q. Os autores apresentam um algoritmo para, dada uma partição inicial, determinar a partição homogénea com maior granularidade que a refina. Basicamente, este algoritmo verifica para cada par de conjuntos da partição se se verifica a condição de estabilidade, dividindo os conjuntos não estáveis conforme forem sendo descobertos.

Para determinar um MDP reduzido cuja solução seja exactamente a solução do MDP original a partição inicial usada no algoritmo anterior deverá ser coerente quando às recompensas. Formalmente, essa partição P deve verificar a seguinte propriedade:

$$\forall B \in P,\, p, q \in B, \alpha \in \mathcal{A} \cdot R(p, \alpha) = R(q, \alpha)$$

Considerando que P^* é a partição homogénea com maior granularidade que refina esta partição inicial, então obtemos o MDP reduzido da seguinte forma:

- O conjunto de estados é $\mathcal{S}' = P^*$, passando os estados a ser conjuntos de estados do MDP original.
- O conjunto de acções é idêntico: $\mathcal{A}' = \mathcal{A}$.
- A nova função de transição de estados T' passa a definir-se como

  $$T'(B, \alpha, C) = \sum_{r \in C} T(p, \alpha, r)$$

  onde p é um estado qualquer de B.
- A função de recompensa define-se como

  $$R'(B, \alpha) = R(p, \alpha)$$

  onde, mais uma vez, p é um estado qualquer de B.

A complexidade deste algoritmo de optimização depende muito do mecanismo utilizado para representar as partições, devendo este ser suficientemente expressivo para permitir representar qualquer partição de $\mathcal{S}$. No entanto, é possível afirmar que o número de operações de divisão cresce linearmente em $\mathcal{S}$ e o número de operações de verificação de estabilidade cresce quadraticamente. O ganho de eficiência em relação a utilização do MDP original varia muito de caso para caso, dependendo do facto de se obterem partições homogéneas com maior ou menor granularidade. No entanto, uma das grandes vantagens desta técnica em relação a outras técnicas de optimização (incluindo a que se vai apresentar a seguir) é a garantia de que a solução do MDP reduzido é exactamente a mesma do MDP original, ou seja, o possível ganho de eficiência não é compensado com uma perda de qualidade na solução.

A segunda técnica aqui apresentada foi introduzida por vários autores em [HMK+98]. O conceito chave desta técnica é o de macro-acção. Uma macro-acção é uma estratégia localizada, definida para uma determinada região do espaço de estados, que deixa de ser utilizada quando se sai dessa região.

Os autores dividem o problema de determinar o MDP reduzido em duas partes: primeiro, como construir o MDP dada uma partição do espaço de estados e um conjunto de macro-acções; e segundo, como determinar automaticamente um conjunto aceitável de macro-acções. Começando pelo primeiro problema, vamos assumir que o conjunto de estados $\mathcal{S}$ foi dividido em várias regiões segundo uma partição $\Pi = \{S_1, \ldots, S_n\}$. Para cada região S_i encontra-se definida uma periferia de saída como

$$XPer(S_i) = \{t \in S - S_i | \exists s \in S_i, \alpha \in \mathcal{A} \cdot T(s, \alpha, t) > 0\}$$

e uma periferia de entrada como

$$EPer(S_i) = \{t \in S_i | \exists s \in S - S_i, \alpha \in \mathcal{A} \cdot T(s, \alpha, t) > 0\}$$

Os elementos de $XPer(S_i)$ denominam-se estados de saída para S_i e os elementos de $EPer(S_i)$ estados de entrada. O conjunto de todos os estados periféricos para uma partição Π define-se como

$$Per_\Pi(\mathcal{S}) = \cup_i EPer(S_i) = \cup_i XPer(S_i)$$

Neste contexto, uma macro-acção não mais do que uma estratégia para uma região S_i, ou seja, $\pi_i : S_i \rightarrow \mathcal{A}$.

Dada uma partição $\Pi = \{S_1, \ldots, S_n\}$ e um colecção de conjuntos de macro-acções $\mathbf{A} = \{A_i | i \leq n\}$, onde $A_i = \{\pi_i^1, \ldots, \pi_i^{n_i}\}$ é um conjunto de macro-acções para a região S_i, o MDP reduzido constrói-se da seguinte forma:

- $\mathcal{S}' = Per_\Pi(S)$.
- $\mathcal{A}' = \cup_i A_i$, sendo que $\pi_i^k \in A_i$ só é possível de executar nos estados $s \in EPer(S_i)$.

- $T'(s, \pi_i^k, t)$ é dado pelo modelo de transições descontado[12] para π_i^k, quando $s \in EPer(S_i)$ e $t \in XPer(S_i)$ e $T'(s, \pi_i^k, t) = 0$ quando $t \notin XPer(S_i)$.

- $R'(s, \pi_i^k)$ é dado pelo modelo de recompensas descontado[13] para π_i^k, quando $s \in EPer(S_i)$.

Para determinar qual o conjunto de macro-acções que serão utilizadas para cada região, os autores desenvolveram um método heurístico que, partindo de uma estimativa do valor de se atingir os estados da periferia de saída, define um MDP para essa região, cuja resolução resulta numa macro-acção para incluir no modelo reduzido. Obviamente, a qualidade das macro-acções geradas vai depender muito da estimativa do valor dos estados de saída. Para minimizar o erro associado a esta estimativa pode ser utilizado um método iterativo: começa-se por determinar as macro-acções associadas a uma determinada estimativa; depois resolve-se o MDP reduzido com essas macro-acções; finalmente utiliza-se a função de valoração associada à solução desse MDP para determinar a nova estimativa do valor dos estados de saída e reinicia-se o processo.

Embora a redução do número de iterações necessárias para calcular a solução do MDP reduzido possa ser substancial, esta técnica apresenta algumas desvantagens:

- Mesmo que as macro-acções sejam automaticamente geradas, é sempre necessário definir à partida a partição do conjunto de estados.

- A estratégia obtida como solução do MDP reduzido pode não ser óptima em relação ao MDP original, pois o MDP reduzido apenas contem macro-acções, podendo certos comportamentos não estar abrangidos.

- O tempo necessário para gerar as macro-acções pode ser muito elevado, podendo mesmo exceder o tempo de resolver o MDP original. No entanto, esta desvantagem pode ser minimizada caso se possa reutilizar macro-acções na resolução de outros MDPs.

[12] Seja π_i uma macro-acção definida sobre S_i. A probabilidade de transição descontada $T_i(s, \pi_i, t)$ para $s \in S_i$ e $t \in XPer(S_i)$ satisfaz a seguinte equação:

$$T_i(s, \pi_i, t) = T(s, \pi_i(s), t) + \gamma \sum_{r \in S_i} T(s, \pi_i(s), r) T_i(r, \pi_i, t)$$

Esta equação equação dá origem a $|XPer(S_i)|$ sistemas de equações lineares, sendo cada sistema constituído por $|S_i|$ equações com $|S_i|$ incógnitas. Estes podem ser resolvidos directamente ou com métodos iterativos, originando uma complexidade total da ordem $O(|XPer(S_i)||S_i|^3)$ por macro-acção.

[13] Analogamente, $R_i(s, \pi_i)$, a recompensa esperada e descontada por se adoptar a estratégia π_i começando no estado $s \in S_i$, satisfaz a seguinte equação:

$$R_i(s, \pi_i) = R(s, \pi_i(s)) + \gamma \sum_{r \in S_i} T(s, \pi_i(s), r) R_i(r, \pi_i)$$

Esta equação dá origem a $|S_i|$ equações com $|S_i|$ incógnitas, originando uma complexidade total da ordem $O(|S_i|^3)$ por macro-acção.

3.4.2 Algumas exemplos de aplicação

Existem vários trabalhos na área da inteligência artificial resolvidos com técnicas derivadas dos problemas de decisão de Markov. Por exemplo, em [TR94] Jonathan Tash e Stuart Russell aplicam estas técnicas para resolver um problema clássico de planeamento em tempo real, levando em consideração o esforço de processamento para determinar as estratégias óptimas, por forma a alcançar um equilíbrio entre tempo de computação e qualidade da solução. Sendo um trabalho na área do planeamento, os objectivos são definidos através de uma função de utilidade que apenas atribui valor positivo ao conjunto de estados que se pretende atingir e só são consideradas estratégias que permitam ao agente permanecer indefinidamente num destes estados mal seja atingido. Restringindo desta forma o domínio de aplicação garante-se a convergência com o critério do limite da soma total das recompensas. As extensões propostas neste trabalho em relação às técnicas tradicionais de resolução de MDPs não se aplicam ao USPL, pois esta linguagem será compilada e não interpretada, sendo irrelevante para a execução o tempo necessário para calcular as melhores estratégias. Técnicas semelhantes às usadas neste trabalhos foram utilizadas noutros trabalhos com objectivos idênticos, como por exemplo, em [DKKN95].

Um exemplo de aplicação dos MDPs que já se aproxima mais do trabalho apresentado nesta tese foi desenvolvido por Shieu-Hong Lin e Thomas Dean [LD95]. Este trabalho baseia-se na premissa de que frequentemente é possível dividir um processo de decisão complexo em pequenos planos, posteriormente combinados num plano de alto nível usando uma linguagem de programação muito simples (com condicionais, ciclos e sequenciação), onde esses planos fazem o papel de instruções primitivas. Um plano de alto nível descrito desta forma define um processo de decisão de Markov que poderia ser resolvido de forma convencional. No entanto, para optimizar o cálculo da estratégia óptima os autores apresentam uma técnica que tira partido da estrutura localizada embebida num desses planos: normalmente, os efeitos de uma acção dependem apenas de um pequeno subconjunto de variáveis e afectam apenas um pequeno conjunto de variáveis; e os planos primitivos não são executados com uma frequência uniforme, existindo a tendência para entrar em ciclo num pequeno fragmento do plano durante um certo período de tempo e depois seguir para outro fragmento. Embora sejam permitidos ciclos na linguagem de combinação de planos, os autores assumem que o programa termina sempre, i.é, atinge um estado no qual permanece indefinidamente. Esta premissa possibilita, tal como no exemplo anterior, a utilização do limite da soma total das recompensas como critério de optimização mais adequado.

Os planos simples apresentados neste trabalho correspondem essencialmente a escolhas não determinísticas de atribuições guardadas (instrução `guard` do USPL). No conjunto das instruções primitivas temos, para além de planos simples, nodos de controlo que apenas podem ser ultrapassados quando uma condição for satisfeita. Estes nodos correspondem a uma instrução de espera activa (instrução `await` do USPL). Dadas estas correspondências, as únicas diferenças introduzidas pelo USPL são as seguintes:

- A possibilidade de tratar programas que não terminem, permitindo a

utilização de outros critérios de optimização.

- A definição concreta da linguagem de programação e do processo de obter um problema de decisão de Markov a partir da representação textual de um programa. Neste caso os autores partem logo do modelo semântico de um programa, não completando o ciclo de desenvolvimento de programas.
- A descrição das estratégias óptimas numa linguagem de programação convencional.

Essencialmente, as diferenças entre este trabalho e o USPL derivam de diferenças de objectivos: enquanto que estes autores colocam uma maior ênfase na optimização do processo de cálculo das estratégias óptimas, no USPL o objectivo principal é desenvolver uma linguagem de programação com todo o ciclo de desenvolvimento bem especificado. Naturalmente, as semelhanças entre ambos abrem a perspectiva de num futuro próximo incorporar estas optimizações no USPL.

3.5 Resumo

Neste capítulo foi apresentada uma síntese dos modelos e técnicas analisados, com o objectivo de escolher a que será mais adequada para aplicar na compilação de programas USPL. Quanto ao modelo de computação praticamente todos os trabalhos se adequam ao modelo do USPL. Efectivamente, a maior parte é mesmo mais flexível, permitindo representar incerteza quanto ao resultado de executar uma acção ou mesmo modelar o ambiente que rodeia o agente. O modelo de computação do USPL é muito simples e poderia facilmente ser representado pela maior parte dos modelos referidos. Os grandes problemas começam na representação das preferências e das estratégias.

Começando pela síntese de programas a partir de especificações temporais, neste momento não é claro como é que se poderia traduzir de forma sistemática as preferências representadas como uma função de utilidade mais um critério de decisão para uma fórmula em lógica temporal. Sem esta tradução, a utilização destas técnicas é logo à partida inviabilizada. No entanto, existem ainda outro grande problema: caso se pretendesse utilizar um conjunto minimamente expressivo de operadores temporais nessa especificação nunca seria possível sintetizar estratégias sem memória. Como se verá no capítulo seguinte, esta é mesmo uma das grandes limitações impostas pela abordagem actual à noção de computação válida, pois é claro que no caso geral este tipo de estratégias não é suficientemente expressivo para sintetizar comportamentos complexos.

No caso da síntese de agentes intencionais o problema da especificação das preferências volta a surgir, sendo no entanto minimizado quando se verifica que as lógicas de especificação utilizadas são na sua maioria extensões à lógica temporal com operadores específicos para representar atitudes intencionais como, por exemplo, os desejos. No entanto, continua a não ser muito claro o processo de tradução das preferências para essas lógicas. Para além deste problema, o mais grave é a falta generalizada de sistematização no processo de tradução da especificação para uma implementação. Na maior parte dos casos fica-se com a

ideia que alguns trabalhos estão ainda numa fase embrionária, não transmitindo muita confiança na sua utilização. Infelizmente, o único trabalho analisado que apresenta algumas garantias de correcção (secção 3.3.2), não utiliza um operador intencional para representar os desejos.

Finalmente, resta a área dos problemas de decisão de Markov, que como facilmente se apercebe apresenta um mapeamento quase perfeito com o nosso trabalho, sendo as técnicas de resolução destes problemas as utilizadas na compilação de programas USPL. Para além das estratégias sintetizadas não possuírem memória e de também utilizarem a noção de critério de decisão, a tradução da função de utilidade para uma função de recompensa é trivial (a recompensa por se transitar entre dois estados é a diferença de utilidade entre eles). A juntar a estes factos, temos uma teoria muito bem desenvolvida (já com quase cinquenta anos de maturação), que proporciona garantias de correcção na utilização dos algoritmos de optimização. Aliás, este relacionamento é tão estreito que o USPL poderia praticamente ser apresentado como uma linguagem de especificação para problemas de decisão de Markov determinísticos. Como se verá no capítulo seguinte, os únicos problemas interessantes que sobram para ser resolvidos com a utilização destas técnicas derivam das condições de justiça existentes num UFTS, e do objectivo de representar as estratégias sintetizadas como um programa sequencial.

Capítulo 4

Tradução de programas USPL para SPL

A linguagem USPL por si só, tal como foi apresentada no capítulo 2, não é muito útil. Primeiro, não pode ser utilizada como linguagem de programação porque não foi apresentado nenhum mecanismo que possibilite a execução dos seus programas. Segundo, também não pode ser utilizada apenas como linguagem de especificação porque, ao contrário do SPL, não possui nenhuma teoria nem ferramentas desenvolvidas que permitam provar propriedades sobre os seus programas. A transposição directa da teoria desenvolvida por Manna e Pnueli sobre o SPL não é possível devido, fundamentalmente, à alteração substancial na noção de computação válida.

A solução para lidar com ambos os inconvenientes consistiu em desenvolver um mecanismo de compilação por tradução, que permite transformar programas escritos em USPL em programas escritos em SPL com um comportamento equivalente.[1] Formalmente, o objectivo desta transformação é que uma computação σ seja válida no modelo de um programa USPL se e só se for válida no modelo do programa SPL gerado. Se considerarmos $\mathcal{P}'$ como o programa SPL cujo conjunto de computações válidas é exactamente Ω, o conjunto das execuções de um programa USPL $\mathcal{P}$ que verificam os critérios de justiça[2], então este processo de tradução vai tentar retirar de forma controlada algum do não-determinismo de $\mathcal{P}'$ por forma a chegar ao programa SPL desejado $\mathcal{P}^*$. Naturalmente, a redução do não-determinismo será controlada pela função de utilidade do processo e pelo respectivo critério de decisão, devendo o programa resultante "incorporar" de forma implícita estas estruturas. O programa $\mathcal{P}^*$ será um programa que simula $\mathcal{P}'$, de acordo com a definição de simulação de programas SPL apresentada em [KMP94], pois toda a computação de $\mathcal{P}^*$ é uma computação de $\mathcal{P}'$, devendo $\mathcal{P}^*$ validar a fórmula que define a semântica temporal de $\mathcal{P}'$ (apresentada na secção 3.2), ou seja, $\mathcal{P}^* \models comp_{\mathcal{P}'}$. Nesta pers-

[1] Note-se que, embora o SPL não tenha sido desenvolvido como uma linguagem de programação, julgo que seria relativamente trivial, pelo menos para o subconjunto de instruções utilizado, desenvolver um compilador ou interpretador que permita executar os seus programas.

[2] Este programa obtém-se de $\mathcal{P}$ retirando-lhe simplesmente a função de utilidade e o critério de decisão.

pectiva clássica de refinamento, o programa USPL $\mathcal{P}$ pode ser visto como uma especificação abstracta do comportamento de um sistema, cuja implementação concreta será $\mathcal{P}^*$.

Na perspectiva do USPL como linguagem de especificação de um agente inteligente orientado por uma função de utilidade, o que se pretende com esta transformação é determinar qual a melhor estratégia para actuar no ambiente que o rodeia. O programa SPL resultante pode ser visto como uma codificação bastante eficiente dessa estratégia. Com notam Brafman e Tennenholtz em [BT96], existem várias formas de representar a estratégia de um agente, sendo a representação explícita através de um mapeamento de estados para acções altamente ineficiente em termos de espaço. Alternativamente, estes autores sugerem a utilização de representações implícitas que permitam reduzir os custos de armazenamento e transmissão das estratégias. Com a utilização de programas SPL como mecanismo de representação implícita esse objectivo pode ser satisfatoriamente atingido.

A tradução de um programa USPL para um programa SPL equivalente faz-se segundo as seguintes fases:

1. Primeiro, começa-se por determinar um modelo para o programa USPL. Esta transformação é feita de acordo com as regras referidas na secção 2.4.2, sendo depois o UFTS obtido convertido para um sistema de transição onde as arestas são representadas de forma explícita, depois da expansão das relações de transição. Esta fase será descrita na secção 4.1.

2. Depois, será determinada a melhor estratégia de actuação utilizando técnicas de resolução de MDPs. Esta é a operação fundamental de todo o processo e será descrita com mais detalhe na secção 4.2.

3. Finalmente, é determinado um FTS cujas relações de transição reflectem a estratégia óptima determinada no passo anterior, e calculado um programa SPL que possa ser modelado por este FTS. Estas duas últimas operações serão descritas na secção 4.3.

Ao longo deste capítulo, as técnicas apresentadas são acompanhadas com o exemplo da sua aplicação ao programa da figura 2.2. Dado que este programa é demasiado simples, no anexo B é apresentado um exemplo um pouco mais interessante, que já possui um ciclo com possibilidade de execução infinita, e cuja consulta pode contribuir para uma compreensão mais profunda de todo o processo.

O capítulo termina, na secção 4.4, com os resultados de uma investigação bastante preliminar sobre a possibilidade de se efectuar esta tradução por refinamento gradual do programa USPL.

4.1 Obtenção de um sistema de transição a partir de um programa USPL

A primeira tarefa a executar no processo de transformação de um programa USPL para a linguagem SPL consiste em calcular o UFTS que lhe corresponde.

Este mecanismo já foi explicado na secção 2.4, tendo sido apresentado o exemplo do UFTS para o programa da figura 2.2.

Um forma muito conveniente de visualizar e manipular um UFTS passa pela utilização de um tipo específico de $\mathcal{P}$-diagrama. Dado um programa $\mathcal{P}$, um $\mathcal{P}$-diagrama é um grafo $\mathcal{D}$ construído da seguinte forma[Val00]:

- Os nodos são os elementos de um conjunto finito $\mathcal{N}_\mathcal{D}$ de asserções $\phi, \varphi, \ldots$ (definidas sobre as variáveis flexíveis de $\mathcal{P}$) que verificam:

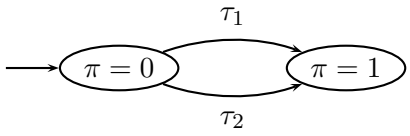
$$\models \Phi \supset \bigvee_{\phi \in \mathcal{N}_\mathcal{D}} \phi$$

  Os nodos que formam o menor subconjunto de $\mathcal{N}_\mathcal{D}$ que ainda verifica esta condição dizem-se iniciais.

- Os ramos são etiquetados pelas relações de transição do programa $\mathcal{P}$ e o seu conjunto $\mathcal{R}_\mathcal{D}$ verifica a seguinte condição ($\phi \xrightarrow{\tau}$ representa o conjunto de nodos que são destino de ramos etiquetados com τ e que tenham origem em ϕ):

$$\forall \tau \in \mathcal{R}_\mathcal{D}, \phi \in \mathcal{N}_\mathcal{D} \cdot \models \tau \wedge \phi \supset \bigvee_{\varphi \in \phi \xrightarrow{\tau}} \varphi$$

O tipo de $\mathcal{P}$-diagrama que iremos usar procura reflectir a estrutura de controlo do programa que lhe deu origem e, como tal, será designado com $\mathcal{P}$-diagrama de controlo. Formalmente, esta classe de $\mathcal{P}$-diagramas caracteriza-se pelo facto de possuírem tantos nodos quantos os possíveis valores para a variável de controlo π de $\mathcal{P}$ e das asserções serem todas do tipo $\pi = n$, percorrendo n todos os valores possíveis para π. Por exemplo, para o programa da figura 2.2, obtemos o seguinte $\mathcal{P}$-diagrama de controlo[3]:

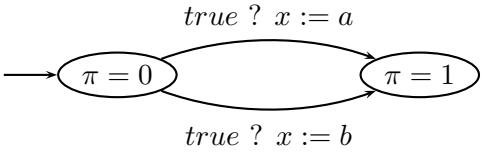


Como se pode reparar, um $\mathcal{P}$-diagrama de controlo representa claramente todas transformações sofridas pela variável de controlo π ao longo da execução do programa. Este facto permite, sem perda de expressividade e com um ganho de clareza, que em vez de se decorar os ramos do $\mathcal{P}$-diagrama com as relações de transição, apenas se apresente a parte dessas relações que dizem respeito à condição de activação originada directamente pelas condições das instruções[4] e, no caso das transições originadas por instruções de atribuição, qual a variável

[3]Nos $\mathcal{P}$-diagramas apresentados nesta tese os nodos iniciais são sempre apontados por uma seta sem origem e nunca são desenhadas as transições nulas por uma questão de clareza (estas corresponderiam a um lacete em cada nodo).

[4]Se atentarmos ao processo de determinar um UFTS apresentado na secção 2.4.2, esta condição corresponde precisamente ao primeiro termo de cada uma das relações de transição apresentadas.

modificada e o seu novo valor. Estas duas fórmulas serão separadas por um ponto de interrogação para reforçar o significado da condição. Para facilitar a percepção dos UFTSs serão utilizados, sempre que possível, estes diagramas para os representar, sendo, por exemplo, o $\mathcal{P}$-diagrama anterior apresentado como:



Embora um UFTS represente o sistema de transição que modela um programa, a notação lógica das relações de transição não é conveniente para a fase de optimização que se segue. Nesta fase é necessário que este sistema de transição esteja representado por extensão e não por compreensão, tal como é descrito pelas relações de transição. Neste sistema de transição explícito, os ramos são etiquetados pela relação de transição que lhe deu origem, indo estas representar o papel de acções possíveis de executar em cada estado, das quais o agente deve escolher as que lhe permitem maximizar a função de utilidade.

Formalmente, o que pretendemos obter é um sistema de transição $\langle \Sigma, I, \delta \rangle$ com os seguintes componentes:

- Σ é o conjunto dos estados.
- $I \subseteq \Sigma$ é o conjunto dos estados iniciais e determina-se como:

$$\{s \parallel\!\models \Theta \mid s \in \Sigma\}$$

- $\delta : \Sigma \to \mathcal{T} \to \Sigma \to \mathbb{B}$ representa os seus ramos, cujas etiquetas são retiradas do conjunto das transições $\mathcal{T}$. Um ramo entre s e s' etiquetado por $\tau \neq \tau_I$ existe neste sistema sse τ ocorrer entre esses dois estados, ou seja:

$$\delta\ s\ \tau\ s' \equiv \langle s, s' \rangle [\rho_\tau]$$

  Por razões que serão apresentadas na secção seguinte, a transição nula só é considerada num estado deste sistema de transição quando não existir outra transição que possa ocorrer nesse estado, ou seja:

$$\delta\ s\ \tau_I\ s \equiv \forall \tau \neq \tau_I \cdot \neg\, (\exists s' \cdot \langle s, s' \rangle [\rho_\tau])$$

Um facto relativamente óbvio, mas bastante importante, relacionado com estes sistemas de transição é que qualquer programa USPL dá origem a um sistema de transição determinístico, ou seja, que verifica a seguinte propriedade:

$$\forall s, s', s'', \tau \cdot \delta\ s\ \tau\ s' \wedge \delta\ s\ \tau\ s'' \supset s' = s''$$

Dado que o sistema de transição é sempre determinístico, por vezes iremos usar uma representação alternativa para δ, cujo tipo é $(\Sigma \times \mathcal{T}) \hookrightarrow \Sigma$ e que se define como:

$$\delta(s, \tau) = \begin{cases} s' & \text{se } \delta\ s\ \tau\ s' \\ \bot & \text{se } \forall s' \in \Sigma \cdot \neg \delta\ s\ \tau\ s' \end{cases}$$

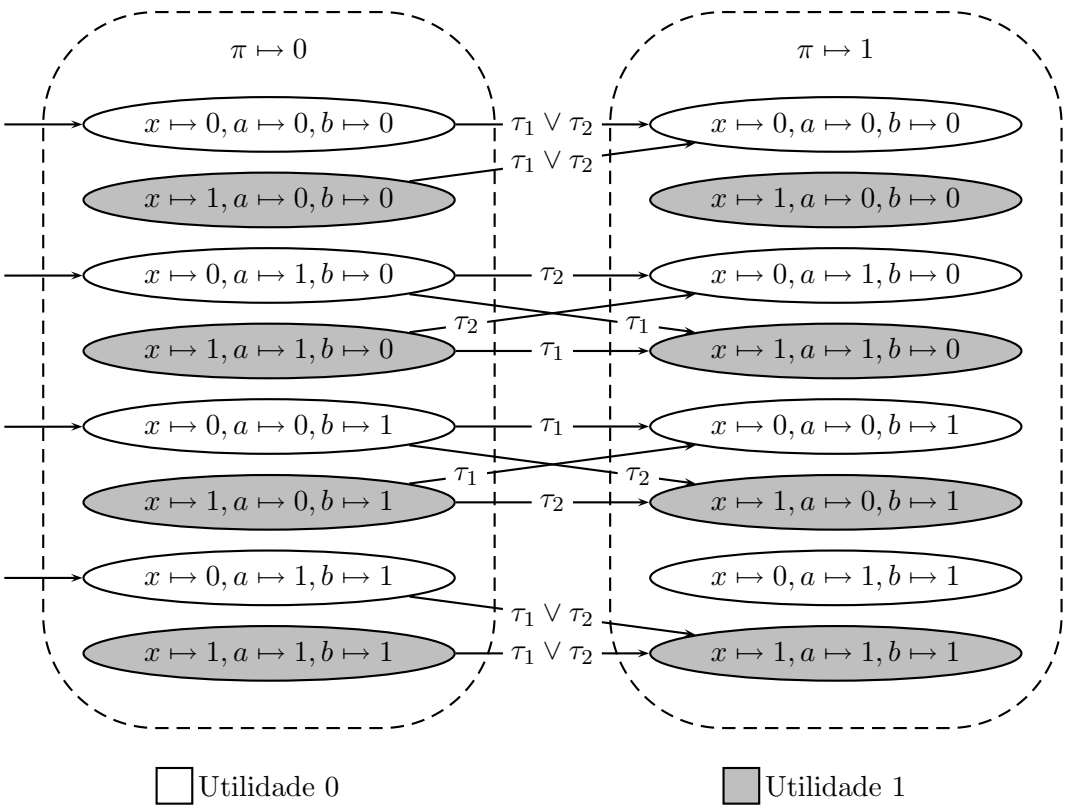


Figura 4.1: Sistema de transição do exemplo da figura 2.2.

Continuando com o exemplo da figura 2.2, obtém-se a partir do respectivo UFTS o sistema de transição apresentado na figura 4.1[5].

4.2 Cálculo da estratégia óptima

Depois de determinado o sistema de transição é necessário determinar qual a melhor estratégia de actuação de acordo com o critério de decisão adoptado. Tal como foi referido na secção 3.1.3, estas estratégias deverão ser não determinísticas para que seja possível obter o maior número de computações em $\rho\ u\ \Omega$, e sem memória para que as computações resultantes sejam computações do programa original.

Definição 4.1 *Dado o sistema de transição determinístico correspondente a um UFTS, uma estratégia válida π é definida formalmente como uma função com o tipo $\Sigma \to 2^{\mathcal{T}}$, que indica para cada estado quais as transições que podem ocorrer nesse estado, e que verifica as seguintes condições*[6]*:*

[5]Para simplificar, quando entre dois estados mais do que uma transição é possível, apenas é desenhado um ramo etiquetado com uma disjunção das transições. Os estados também foram preenchidos de acordo com a respectiva utilidade para melhor se entender o processo posterior de optimização.

[6]Normalmente, as estratégias serão representadas pela letra grega π. Embora esta letra

- *Correcção: as acções escolhidas pela estratégia num estado deverão ser possíveis de executar nesse estado.*

$$\forall s \in \Sigma, \tau \in \pi(s) \cdot \delta(s, \tau) \neq \perp$$

- *Pro-actividade: a estratégia tem que escolher pelo menos uma das transições disponíveis num estado.*

$$\forall s \in \Sigma \cdot \pi(s) \neq \emptyset$$

Se a primeira condição para que uma estratégia seja válida é relativamente óbvia, o mesmo não se passa com a segunda. A razão pela qual a transição nula teve um tratamento especial na definição do sistema de transição explícito e consequentemente na definição de estratégia válida tem origem na conjunção de dois factos:

- A obrigatoriedade de τ_I pertencer ao conjunto das transições $\mathcal{T}$ de qualquer FTS.
- A existência de um critério de justiça, que abrange todas as transições originadas pelas instruções do SPL consideradas nesta tese.

Estes factos surgiram em consequência da necessidade de modelar correctamente a concorrência entre processos e da definição da semântica da lógica temporal, usada para especificar propriedades sobre programas SPL (ver anexo A). Aparentemente, estas razões não se aplicariam directamente neste trabalho, e, como tal, poderíamos relaxar estas obrigatoriedade e evitar estes problemas. No entanto, como se verá nas secções seguintes, acabámos por utilizar a lógica temporal para especificar algumas propriedades sobre os programas e, num futuro próximo, pretendemos extender estas técnicas a programas concorrentes[7], não sendo possível deixar de as considerar.

A primeira consequência destes factos é que o FTS resultante da optimização tem sempre que incluir, independentemente da estratégia óptima calculada, a transição nula. A segunda consequência é a obrigatoriedade, expressa na definição 4.1, de uma estratégia válida ser pro-activa e de apenas poder escolher a transição nula quando nenhuma transição diligente puder ocorrer. A necessidade de pro-actividade deve-se ao facto de uma estratégia poder implicitamente escolher apenas a transição nula num estado s se $\pi(s) = \emptyset$. Tecnicamente, uma computação onde, a partir de um certo ponto, se permanece indefinidamente num estado onde é possível executar alguma instrução diligente não é válida porque se viola a condição de justiça. Esta obriga a que, caso uma transição esteja continuamente activa num estado, ela tenha de ser executada nalgum ponto.

seja também utilizada para denotar a variável de controlo, decidimos também utiliza-la nesta situação, pois é frequentemente usada com esse significado na literatura consultada. Normalmente deverá ser fácil deduzir qual o seu significado a partir do contexto onde estiver inserida.

[7] Aliás, estas são as razões fundamentais para adoptar o SPL como linguagem de trabalho, tal como foi visto no capítulo 2.

Caso uma estratégia escolhesse apenas a transição nula para executar num determinado estado onde pudesse ocorrer uma transição diligente, iria existir uma computação onde, a partir de um certo ponto se permaneceria indefinidamente nesse estado. No entanto, esta computação não é válida no programa original e, como tal, não pode existir no programa resultante.

Dada uma estratégia válida é possível restringir o UFTS original e obter um FTS usando o seguinte procedimento. Seja $val : \Sigma \rightarrow expbool(V)$ a função que determina, para cada estado, a expressão booleana que caracteriza a valoração das variáveis nesse estado, definida formalmente como:

$$val = \lambda s. \bigwedge_{x \in V} x = s[x]$$

Sendo π uma estratégia válida, então a nova condição de activação de uma transição τ pode ser definida como:

$$\bigvee_{s \in \Sigma, \tau \in \pi(s)} val(s) \tag{4.1}$$

Dado um UFTS $\langle V, \Theta, \mathcal{T}, u, \rho \rangle$ e uma estratégia válida π, o FTS que resulta de restringir esse UFTS por π é definido pelo tuplo $\langle V', \Theta', \mathcal{T}', \mathcal{J}', \mathcal{C}' \rangle$ onde:

$$\begin{aligned}
V' &\equiv V \\
\Theta' &\equiv \Theta \\
\mathcal{T}' &\equiv \left\{ \rho_\tau \wedge \bigvee_{s \in \Sigma, \tau \in \pi(s)} val(s) \,|\, \tau \in \mathcal{T} - \{\tau_I\} \right\} \cup \{\tau_I\} \\
\mathcal{J}' &\equiv \mathcal{T}' - \{\tau_I\} \\
\mathcal{C}' &\equiv \emptyset
\end{aligned}$$

Dado este procedimento, podemos definir precisamente o objectivo do processo de optimização como a tentativa de obter uma estratégia óptima π^*, tal que o conjunto das computações do FTS que resulta de restringir o UFTS original por essa estratégia seja precisamente igual a $\rho\ u\ \Omega$.

Obviamente, no caso geral tal não vai ser possível. Suponha-se o seguinte programa com um critério de decisão diferente dos especificados no capítulo 2:

```
local x:[0..2] where x=0

P(x,??) :: [
  [x:=1; x:=0] or [x:=2; x:=0];
  [x:=1; x:=0] or [x:=2; x:=0]
]
```

Como as estratégias não possuem memória, não é possível identificar qualquer subconjunto de computações deste programa. Por exemplo, não é possível com estratégias sem memória escolher apenas as computações onde `x` toma sucessivamente os valores $1, 0, 1, 0$ e $2, 0, 2, 0$. Se incluirmos na linguagem um

critério de decisão com este comportamento não será possível calcular uma estratégia óptima. Uma questão que fica em aberto é se, considerando apenas os critérios de decisão definidos no capítulo 2, é ou não possível calcular uma estratégia óptima. Extrapolando os resultados obtidos na área dos problemas de decisão de Markov (ver secção 3.4.1), isso deverá pelo menos ser possível para o critério `discounted`.

Para finalizar a questão das estratégias, falta apenas referir um pormenor relacionado com o processo de restrição de um UFTS por uma estratégia. Na definição apresentada anteriormente assumiu-se sem grandes explicações que, exceptuando a transição nula, todas as transições continuariam a ser justas. No entanto, seria possível que algumas delas passassem a ser compassivas, introduzindo dessa forma uma espécie de memória na execução de uma estratégia. Esta memória permitiria controlar o não-determinismo das estratégias, forçando a execução de certas transições. Em princípio, com este mecanismo, seria possível alargar o conjunto de critérios de decisão para os quais é possível determinar estratégias óptimas, sem se alterar a noção de computação válida. O único problema desta abordagem (e também o motivo pela qual não foi adoptada) é que a reconstrução de um programa SPL a partir do FTS sintetizado deixaria de ser possível no caso geral. Para tal, seria necessário que qualquer transição compassiva tivesse uma instrução SPL que lhe desse origem, o que não se verifica com a sintaxe actual da linguagem. Como se verá na secção 4.3, mantendo a abordagem actual essa reconstrução pode sempre ser realizada.

4.2.1 Optimização usando o algoritmo *value iteration*

A forma como é feita a optimização varia de acordo com o critério de decisão adoptado. Como foi referido na secção 3.5, as técnicas que melhor se adaptam à optimização no caso do USPL são as utilizadas nos MDPs, principalmente porque fornecem mecanismos de optimização para critérios de decisão baseados em horizontes infinitos, sem que, no entanto, seja necessário expressar as preferências sobre execuções.

No caso dos três critérios adoptados nesta tese, apenas dois são baseados num horizonte infinito, nomeadamente, os critérios `discounted` e `look-at-end`. Obviamente, o critério `look-ahead` toma as decisões tendo por base um horizonte finito. Embora na secção 3.4.1 apenas tenha sido apresentado o algoritmo *value iteration*, que permite calcular estratégias quando se usa como critério a recompensa total descontada (ao qual corresponde o nosso critério `discounted`), esse algoritmo acabou por ser utilizado, com ligeiras alterações, para determinar as estratégias óptimas com qualquer dos três critérios de decisão.

Nesta secção, vamos começar por apresentar algumas definições que são comuns aos algoritmos de optimização dos três critérios, sendo depois apresentados os detalhes específicos de cada um.

O objectivo do algoritmo *value iteration* é determinar a função de valoração $V : \Sigma \to \mathbb{R}$ que representa a utilidade de um estado caso se adopte uma estratégia óptima a partir desse estado. Na versão tradicional do algoritmo, apresentada na figura 3.1, esta função é calculada por aproximações sucessivas até que se atinja um erro máximo inferior ao parâmetro ϵ. Para que este algo-

ritmo possa ser aplicado na optimização dos sistemas de transição é necessário definir como se obtém um MDP, definido por um tuplo $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, a partir de um UFTS, definido por um tuplo $\langle V, \Theta, \mathcal{T}, u, \rho \rangle$, ou do respectivo sistema de transição explícito, definido por um tuplo $\langle \Sigma, I, \delta \rangle$:

- S, o conjunto de estados do MDP, é equivalente ao conjunto Σ de todas as possíveis valorações das variáveis em V.

- A, o conjunto de acções, é equivalente ao conjunto $\mathcal{T}$ das transições. É de relembrar que cada transição τ é originada por uma instrução do programa, representando estas as acções possíveis de executar pelo agente modelado.

- $R : \Sigma \to \mathcal{T} \to \mathbb{R}$, que representa a função de recompensa. Nos MDPs as preferências não são representadas por utilidades sobre os estados, mas sim por funções de recompensa que determinam o ganho imediato obtido por se realizar cada acção. Esta função será definida como:

$$R(s,\tau) = \begin{cases} u(\delta(s,\tau)) - u(s) & \text{se } \delta(s,\tau) \neq \bot \\ -\infty & \text{se } \delta(s,\tau) = \bot \end{cases}$$

  Dado que o algoritmo *value iteration* assume que R é uma função total, é necessário definir alguma recompensa mesmo quando uma transição não esteja disponível. Naturalmente, esta recompensa deverá ser suficientemente negativa para evitar que essas transições sejam escolhidas. A definição destas recompensas como $-\infty$ resolve o problema e não implica a necessidade de se compararem infinitos para se determinar a estratégia óptima, pois a função R verifica sempre a seguinte propriedade:

$$\forall s \in \mathcal{S}, \exists \tau \in \mathcal{A} \cdot R(s,\tau) > -\infty$$

- $T : \Sigma \to \mathcal{T} \to \Sigma \to \{0,1\}$ representa a relação de transição e define-se trivialmente como:

$$T(s,\tau,s') = \begin{cases} 1 & \text{se } (\delta\ s\ \tau\ s') \\ 0 & \text{se } \neg(\delta\ s\ \tau\ s') \end{cases}$$

Depois de determinada a função de valoração óptima V^*, a estratégia óptima é determinada por um processo ligeiramente diferente do usual, pois neste caso pretendemos obter uma estratégia não determinística onde, em cada estado, são seleccionadas todas as acções que permitem maximizar a função de valoração. Formalmente a estratégia óptima determina-se como:

$$\tau \in \pi^*(s) \text{ sse } \tau = \operatorname*{argmax}_{\tau \in \mathcal{A}} \left[R(s,\tau) + \gamma \sum_{s' \in \mathcal{S}} T(s,\tau,s') V^*(s') \right] \tag{4.2}$$

Como foi referido na secção 3.4.1, existem algumas condições para que seja possível determinar a função de valoração V^*. Estas condições verificam-se trivialmente se o conjunto das acções e dos estados for finito. Esta foi a razão porque decidimos restringir, logo na definição da linguagem, os tipos de dados

para $s \in \mathcal{S}$ **do**
    $V_1(s) := 0$
od
$t := 1$
repetir
    $t := t + 1$
    para $s \in \mathcal{S}, a \in \mathcal{A}$ **do**
        $Q_t^a(s) := R(s,a) + \sum_{s' \in \mathcal{S}} T(s,a,s') V_{t-1}(s')$
        $V_t(s) := \max_a Q_t^a(s)$
    od
ate $\forall s \in S \cdot V_t(s) = V_{t-1}(s)$

Figura 4.2: Algoritmo *value iteration* usado com o critério `look-at-end`.

permitidos apenas a gamas de valores. Desta forma a finitude do conjunto de estados é sempre garantida. A longo prazo é desejável relaxar esta restrição, pois nalguns casos, mesmo com tipos aparentemente infinitos é possível, recorrendo a técnicas de abstracção como as referidas na secção 3.4.1, calcular a função de valoração óptima.

Implementação do critério de decisão `discounted`

Como facilmente se constata, este critério corresponde exactamente ao critério de optimização das recompensas totais descontadas dos MDPs. Como tal, foi usado o algoritmo *value iteration* exactamente como foi apresentado na figura 3.1. Para além do factor de desconto, que é definido no próprio programa USPL e que está inerente ao próprio critério de decisão, este algoritmo também necessita do parâmetro ϵ que deverá ser definido quando se proceder à optimização. Como já foi referido na secção 3.4.1, este algoritmo termina sempre a sua execução e V_t não difere de V^* mais do que $(2\epsilon\gamma)/(1-\gamma)$.

Implementação do critério de decisão `look-at-end`

No caso do critério `look-at-end` foi utilizada a variante do algoritmo *value iteration* apresentada na figura 4.2. As únicas diferenças são a não utilização do factor de desconto e a eliminação do erro no critério de paragem. Obviamente, para determinar a estratégia óptima depois de se determinar V^*, é necessário seja utilizado $\gamma = 1$ na equação (4.2).

A justificação para a utilização deste algoritmo deve-se à possível equivalência entre o nosso critério `look-at-end` e o critério do limite da soma das recompensas utilizado nos MDPs. Embora não tenha sido encontrada uma referência onde se apresentasse um algoritmo concreto para determinar estratégias óptimas com esse critério, em [Put90] sugere-se que podem ser utilizadas variantes do *value iteration* para esse fim. Um dos problemas com esta modificação é que se perde a garantia de terminação e de optimabilidade que se herdava

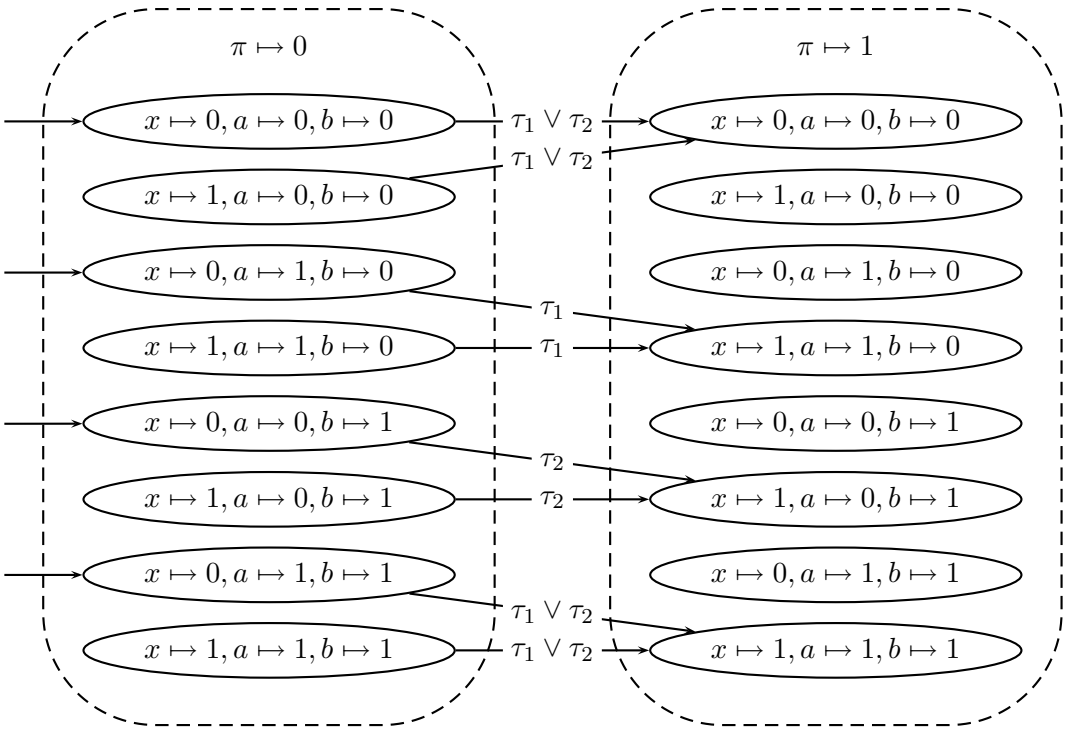


Figura 4.3: Sistema de transição optimizado do exemplo da figura 2.2.

dos MDPs no caso anterior. No entanto, quanto à terminação é mais ou menos fácil de verificar que, caso o programa termine sempre a sua execução[8], este algoritmo irá garantidamente terminar a sua execução: nos estados terminais a valoração é sempre nula e, por propagação, pelo menos a partir da n-ésima iteração a valoração nos estados que distam n passos dos estados terminais também será sempre constante, dado que exceptuando os lacetes nos estados terminais não existem ciclos no sistema de transição.

Se aplicarmos este método para determinar a estratégia óptima do nosso exemplo habitual[9], e depois restringirmos os sistema de transição explícito com essa estratégia, obtemos o sistema de transição apresentado na figura 4.3.

Implementação do critério de decisão `look-ahead`

Para a optimização com este critério foi utilizada a variante do algoritmo *value iteration* apresentada na figura 4.4. Neste algoritmo, em vez de o critério de paragem ser definido pela diferença de dois valores sucessivos de V, existe um número de iterações que é determinado à partida pelo parâmetro N definido no programa USPL. Também é utilizado um ciclo `enquanto` em vez do `repetir até` para permitir que quando $N = 1$ não seja feita nenhuma iteração. Tal

[8]No sentido em que para todas as computações existe um instante a partir do qual apenas a transição nula ocorre.

[9]Como é óbvio neste caso o programa termina sempre a sua execução e, como tal, é possível aplicar este critério.

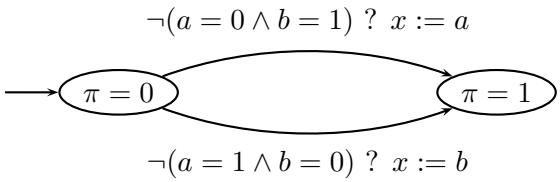
para $s \in \Sigma$ **do**
$V_1(s) := 0$
od
$t := 1$
enquanto $(t < N)$ **do**
$t := t + 1$
para $s \in \Sigma, a \in \mathcal{T}$ **do**
$Q_t^a(s) := R(s,a) + \sum_{s' \in \Sigma} T(s,a,s') V_{t-1}(s')$
$V_t(s) := \max_a Q_t^a(s)$
od
od

Figura 4.4: Algoritmo *value iteration* usado com o critério `look-ahead`.

como no caso do critério `look-at-end`, depois de se determinar V^* é necessário que seja utilizado $\gamma = 1$ na equação (4.2) para se calcular correctamente a estratégia óptima. Obviamente, neste caso nem sequer se colocam os problemas de terminação.

4.3 Cálculo de um programa SPL a partir do sistema de transição optimizado

A primeira coisa a fazer depois de se determinar o sistema de transição explícito e a estratégia óptima, é determinar o novo FTS de acordo com o processo definido na secção 4.2. No caso do exemplo em análise, o novo FTS é conveniente visualizado pelo seguinte $\mathcal{P}$-diagrama de controlo[10]:



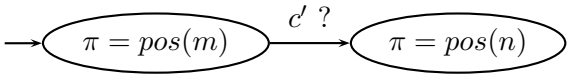
Um dos resultados mais interessantes para esta tese é referido no seguinte teorema, e tem a ver com a possibilidade de se obterem programas SPL correctos para todos os FTS calculados pelo processo de optimização apresentado neste capítulo.

Teorema 4.1 *Qualquer que seja a estratégia utilizada para optimizar um UFTS (mesmo que não seja pro-activa), é sempre possível obter um programa SPL cujo modelo seja o FTS resultante.*

[10] É de notar que, por questões de simplificação, a condição de activação que é apresentada no diagrama não é igual à definida na equação (4.1), mas apenas equivalente.

Prova: O processo de restrição de um UFTS por uma estratégia origina um FTS que tem exactamente as mesmas relações de transição, apenas com a condição de activação mais forte. Sendo assim, esse FTS pode ser representado por um $\mathcal{P}$-diagrama exactamente com a mesma forma do original. Assim, a prova desta propriedade será realizada por indução na estrutura do $\mathcal{P}$-diagrama resultante.

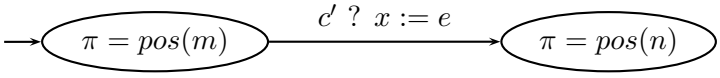
- Se a instrução original era m: `await` c n: obtemos o $\mathcal{P}$-diagrama



  onde $c' \supset c$. A este $\mathcal{P}$-diagrama pode corresponder o programa SPL:

$$m\texttt{: await } c' \ n\texttt{:}$$

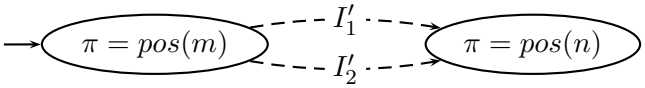
- Se a instrução original era m: `guard` c `do` x `:=` e n: obtemos o $\mathcal{P}$-diagrama



  onde $c' \supset c$. A este $\mathcal{P}$-diagrama pode corresponder o programa SPL:

$$m\texttt{: guard } c' \texttt{ do } x \texttt{ := } e \ n\texttt{:}$$

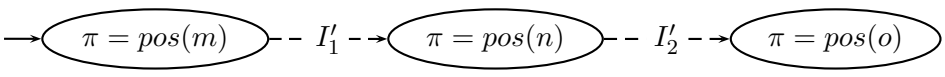
- No caso de uma escolha não determinística m: I_1 `or` I_2 n: obtemos o $\mathcal{P}$-diagrama



  onde as linhas tracejadas pretendem representar os $\mathcal{P}$-diagramas aos quais correspondem os programas SPL I_1' e I_2', que resultam da optimização dos blocos de instruções I_1 e I_2. A este $\mathcal{P}$-diagrama pode corresponder o programa SPL:

$$m\texttt{: } I_1' \texttt{ or } I_2' \ n\texttt{:}$$

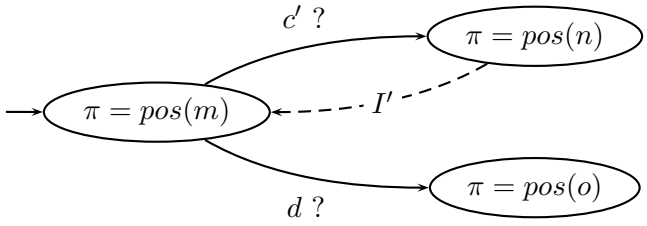
- No caso de uma sequenciação m: I_1 `;` n: I_2 o: obtemos o $\mathcal{P}$-diagrama



  onde as linhas tracejadas pretendem representar os $\mathcal{P}$-diagramas aos quais correspondem os programas SPL I_1' e I_2', que resultam da optimização dos blocos de instruções I_1 e I_2. A este $\mathcal{P}$-diagrama pode corresponder o programa SPL:

$$m\texttt{:}\ I'_1\ \texttt{;}\ n\texttt{:}\ I'_2\ o\texttt{:}$$

- No caso de um ciclo $m\texttt{:}\ \texttt{while}\ c\ \texttt{do}\ n\texttt{:}\ I\ o\texttt{:}$ obtemos o $\mathcal{P}$-diagrama



  onde $c' \supset c$, $d \supset \neg c$, e a linha tracejada pretende representar o $\mathcal{P}$-diagrama ao qual corresponde o programa SPL I', que resulta da optimização do bloco de instruções I. A este $\mathcal{P}$-diagrama pode corresponder o programa SPL:

$$m\texttt{: loop forever do [await}\ c'\ \texttt{;}\ n\texttt{:}\ I'\texttt{] or await}\ d\ o\texttt{:}$$

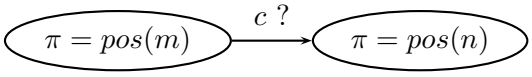
  De referir que numa instrução $l\texttt{:}\ \texttt{loop forever do}\ m\texttt{:}\ \textit{instrução}\ \widehat{m}\texttt{:}\ \widehat{l}\texttt{:}$

$$pos(l) = pos(m) = pos(\widehat{m}) = pos(\widehat{l})$$

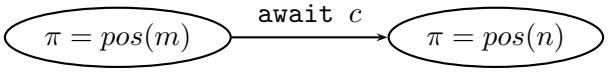
□

A demonstração deste teorema sugere um algoritmo por reescrita de grafos para calcular o programa SPL correspondente a um FTS resultante do processo de tradução. O primeiro passo consiste em obter um grafo cuja estrutura é semelhante ao $\mathcal{P}$-diagrama de controlo resultante, mas onde os ramos são decorados por instruções SPL simples. Tal como foi referido na prova, os ramos podem ser decorados de duas formas:

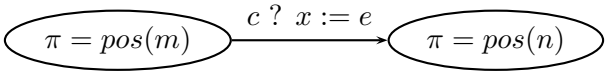
- Quando não há mudança no valor de uma variável diferente de π, temos um ramo da seguinte forma:



  No grafo resultante este ramo será substituído por:



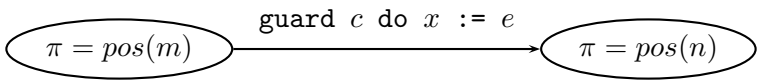
- Caso contrário temos:



```
out x:[0..1] where x=0
in a:[0..1]
in b:[0..1]

P :: [
    guard !(a=0 /\ b=1) do x := a
  or
    guard !(a=1 /\ b=0) do x := b
]
```

Figura 4.5: Programa SPL equivalente ao programa da figura 2.2.

No grafo resultante este ramo será substituído por:



Depois de determinado o grafo inicial, e enquanto tal for possível, aplicam-se as seguintes regras de reescrita:

- Se existirem dois ramos com a mesma origem e o mesmo destino, esses ramos são substituídos por apenas um, também com a mesma origem e o mesmo destino, decorado com uma escolha não determinística das duas instruções que etiquetavam os ramos originais.
- Se existirem dois ramos em que o destino do primeiro é igual à origem do segundo, e não existir mais nenhum ramo com destino ou origem igual ao destino do primeiro, esses ramos são substituídos por apenas um, com a origem do primeiro e o destino do segundo, decorado com uma sequenciação das duas instruções que etiquetavam os ramos originais.
- Se existirem dois ramos em que o primeiro é um lacete e a origem do segundo é igual à origem do primeiro, esses ramos são substituídos por apenas um, com origem e destino iguais ao do segundo, decorado com uma escolha não determinística entre a instrução que etiquetava o segundo e um `loop forever` da instrução que etiqueta o primeiro.

Usando este algoritmo, obtemos para o exemplo usual o programa que é apresentado na figura 4.5, concluindo-se assim o processo de transformação do programa USPL da figura 2.2 num programa SPL equivalente.

4.3.1 Algumas considerações sobre as instruções de ciclo

Na reconstrução dos programas SPL a partir dos FTSs é estranhamente utilizado o ciclo `loop forever` em vez do esperado ciclo `while`. Esta nuance deriva indirectamente da semântica associada a algumas instruções `while`, nomeadamente quando estas estão inseridas no contexto de uma escolha não determinística. Atente-se no programa USPL na figura 4.6 em que esta situação acontece.

```
local x:[0..2]

P(x,look ahead 1) :: [
    while !(x=2) do x := (x+1)
  or
    x := 1
]
```

Figura 4.6: Programa SPL com problemas no ciclo `While`

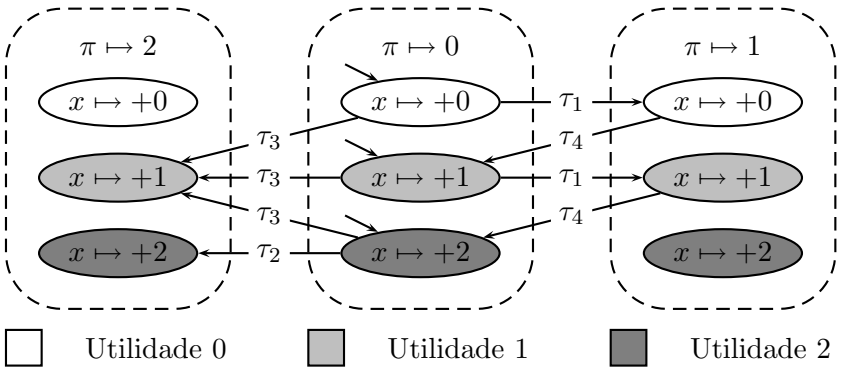


Figura 4.7: Sistema de transição referente do exemplo da figura 4.6

Normalmente, a semântica associada a um programa como este faria supor que temos duas alternativas de execução: ou executar um ciclo em que se incrementa a variável `x` sucessivamente de uma unidade até até esta atingir o valor 2 e depois terminar, ou atribuir o valor 1 à variável `x` e terminar imediatamente. No entanto, se aplicarmos as regras semânticas definidas na secção 2.4.2 obtemos o sistema de transição da figura 4.7.

Observando este sistema de transição verifica-se que o comportamento do programa não é bem o esperado: depois de qualquer iteração da instrução `while` é possível executar a atribuição `x := 1` e terminar o programa sem executar as restantes iterações. Este problema da semântica do SPL está bem documentado, sendo, por exemplo, referido no problema 0.1 de [MP95].

Devido a este problema semântico o resultado de optimizar um sistema de transição pode por vezes dar origem a que não seja possível reconstruir um programa SPL com ciclos `while`. Para tal basta que a melhor estratégia escolha apenas parte das transições geradas pela mesma instrução `while`. Se não fosse possível colocar esta instrução numa escolha não determinística nunca haveria qualquer problema, porque nunca seria possível deixar de escolher todas as suas transições (relembro, mais uma vez, que não é possível escolher só a instrução nula). Por exemplo, depois de calcular a estratégia óptima para este exemplo, obtém-se o sistema de transição da figura 4.8, para o qual não é possível obter um programa SPL onde apenas seja permitido usar ciclos `while`.

Em [MP95] Manna e Pnueli apontam duas possíveis soluções para este problema:

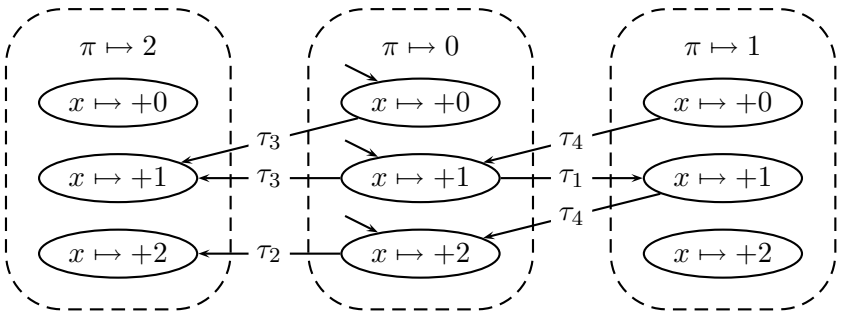


Figura 4.8: Sistema de transição optimizado do exemplo da figura 4.6

```
local x:[0..2]

P :: [
    loop forever do [await (x=1); x:=(x+1)]
  or
    await (x=2)
  or
    guard !(x=2) do x:=1
]
```

Figura 4.9: Programa SPL equivalente ao programa da figura 4.6

- Não permitir a existência de instruções `While` como sub-instruções de escolhas não determinísticas.
- Mudar a semântica das instruções `While`, introduzindo uma etiqueta depois da execução do corpo que não é equivalente à etiqueta antes do teste da condição.

Caso se optasse por qualquer uma destas soluções já não existiria este problema e não seria necessário efectuar a reconstrução dos programas com ciclos `loop forever`. No entanto, não se adoptou nenhuma delas porque a ferramenta STeP também não as adoptou, e, caso o fizéssemos, não seria possível utilizá-la. Se permitirmos a utilização dos ciclos `loop forever`, e usarmos o algoritmo apresentado na secção anterior, obtemos o programa SPL da figura 4.9 que é correctamente modelado por este sistema de transição.

Dado que a instrução `while` apresenta estes problemas semânticos, poderia pensar-se que talvez fosse preferível utilizar no USPL a instrução `loop forever`. No entanto, isso não é verdade porque surgiriam problemas de resolução mais difícil devido à semântica peculiar desta instrução e devido à obrigatoriedade de se verificarem as condições de justiça. Atente-se no programa da figura 4.10, onde, para facilitar a exposição do problema, foi utilizado o tipo `integer`, não previsto no USPL, cujos habitantes são inteiros de precisão ilimitada. Devido à semântica do `loop forever`, neste programa temos, em cada instante, duas hipóteses de evolução: ou incrementar `x` de uma unidade ou terminar. A estratégia óptima neste caso parece trivial, i.é, nunca terminar e incrementar

```
local x:integer

P(x,look-ahead 1) :: [
    loop forever do x := x+1
  or
    await true
]
```

Figura 4.10: Programa USPL com a instrução `loop forever`

sempre `x`, devendo a instrução `await true` ser eliminada do programa SPL resultante. No entanto, as computações desse programa não são sequer execuções do programa USPL: a transição associada à instrução `await true` é justa e, como está continuamente activa, tem de ser executada em todas as computações válidas. As condições de justiça impedem que a estratégia acima referida seja uma estratégia válida. Este problema é bastante subtil e provavelmente com uma resolução difícil caso se continuem a usar estratégias sem memória. Quando se utiliza o `while` este problema não existe porque, para todas as transições excepto a nula (cujo problema já foi resolvido), nunca acontece de o valor de π antes e depois da sua execução ser igual.

Para concluir a análise dos problemas derivados dos ciclos, é de referir que a semântica das instruções `while` e `loop forever` varia ligeiramente nas diversas referências consultadas sobre o SPL. Nalguns casos são definidas duas relações de transição para um `while`, enquanto que noutros esta instrução dá origem a apenas uma relação de transição com duas modalidades. O `loop forever` tanto é identificado com a instrução `while true`, como é uma instrução diferente com a semântica referida acima. Naturalmente, estas diferenças condicionam a forma como a tradução de programas USPL em programas SPL é feita. Como um dos nossos objectivos consistiu em poder utilizar a ferramenta STeP para efectuar algumas provas no refinamento, decidimos utilizar a semântica definida em [BBC+97].

4.4 Refinamento de programas USPL

Em vez de transformar directamente um programa USPL num programa SPL equivalente usando o processo de optimização descrito anteriormente, também é possível tentar efectuar transformações graduais dos programas, num típico processo de refinamento, que leve ao programa SPL equivalente. Nesta secção são apontadas algumas pistas sobre como se poderia fazer esse refinamento, sendo já apresentadas alguma possíveis regras heurísticas de transformação. No entanto, este trabalho é ainda muito preliminar, devendo esta secção ser vista mais como uma indicação de um possível trabalho de investigação futuro.

Como é normal no refinamento, nem todas as regras de transformação podem ser aplicadas incondicionalmente, sendo usual que o programa original tenha de verificar algumas propriedades para que essa aplicação seja possível. Nas regras que serão aqui apresentadas, essas condições são sempre expressas

através de fórmulas temporais definidas sobre programas SPL, sendo possível utilizar a ferramenta STeP [BBC+97] para efectuar a prova formal da verificação dessas condições. Aliás, esta foi uma das razões fundamentais pela qual se optou pelo SPL como linguagem de partida para a definição da nossa linguagem de programação baseada em funções de utilidade.

Nas regras que se seguem, $\mathcal{P}'$ representa o programa SPL que se obtém do programa USPL $\mathcal{P}$ pela simples remoção da função de utilidade e do critério de decisão que parametrizam o processo. O conjunto das computações de $\mathcal{P}'$ é formado por todas as execuções de $\mathcal{P}$ que verificam as condições de justiça. U representa a expressão, definida sobre as variáveis locais do programa, que define a função de utilidade do processo.

O primeiro conjunto de regras permite-nos obter directamente um programa SPL a partir do programa USPL. Uma destas regras deverá ser o passo final de um processo de refinamento.

Regra 4.1 *Sempre que a função de utilidade de um programa USPL é constante, o programa SPL equivalente é idêntico. Formalmente, esta regra aplica-se quando a seguinte asserção é válida:*

$$\mathcal{P}' \models \exists k \cdot \Box(U = k)$$

Ao contrário desta regra fundamental que pode ser aplicada com qualquer critério de decisão, as regras seguintes apenas se aplicam com critérios de decisão específicos.

Regra 4.2 *Se num programa USPL que utilize o critério de decisão* `look-at-end` *todas as execuções terminarem com a mesma utilidade, então o programa SPL equivalente é idêntico. Formalmente, esta regra aplica-se quando a seguinte asserção é válida:*

$$\mathcal{P}' \models \Diamond\exists k \cdot \Box(U = k)$$

As regras agora apresentadas agora são regras para aplicar em passos intermédios do refinamento, ou seja, regras que transformam programas USPL em programas USPL equivalentes. Começamos com algumas regras que fazem transformações apenas na função de utilidade.

Regra 4.3 *A expressão que define a função de utilidade pode sempre ser substituída por outra expressão que, no programa em causa, tenha sempre um valor idêntico. Formalmente, sendo U' uma expressão definida sobre as variáveis locais do programa, se verificar a validade da asserção*

$$\mathcal{P}' \models \Box(U = U')$$

então U pode ser substituída por U' no programa.

Regra 4.4 *Se o critério de decisão do programa for* `look-at-end` *ou* `look-ahead` *então qualquer parte constante da expressão que define a função de utilidade*

pode ser retirada. Formalmente, se tivermos para um dado programa $\mathcal{P}$ cuja função de utilidade é

$$U = \Omega + \Delta$$

podemos reduzi-lo a um programa USPL equivalente com função de utilidade

$$U = \Omega$$

se se demonstrar a validade da seguinte asserção:

$$\mathcal{P}' \models \exists k \cdot \Box(\Delta = k)$$

A regra seguinte faz transformações no corpo do programa e não apenas na função de utilidade. Devido à sua especificidade, só podem ser aplicada com critérios de decisão muito restritos.

Regra 4.5 *Se um programa USPL com critério de decisão* `look-ahead 1` *apresentar a escolha não determinística, onde I_1 e I_2 representar blocos de instruções arbitrários,*

l: `[guard` c_1 `do` x_1 `:=` e_1`;` I_1`] or [guard` c_2 `do` x_2 `:=` e_2`;` I_2`]`

e verificar a seguinte condição

$$\mathcal{P}' \models \Box\,(at\_l \wedge c_1 \wedge c_2 \supset U[x_1 := e_1] > U[x_2 := e_2])$$

então é possível substituir a escolha não determinística anterior pela seguinte:

l: `[guard` c_1 `do` x_1 `:=` e_1`;` I_1`] or [guard` $c_2 \wedge \neg c_1$ `do` x_2 `:=` e_2`;` I_2`]`

Para além de permitir chegar ao programa SPL equivalente sem passar pelo processo de optimização estas regras de refinamento tem ainda algumas vantagens adicionais:

- Abrem a possibilidade de se optimizar programas com espaço de estados infinito, coisa que não pode ser feita com as técnicas de optimização utilizadas nesta tese.
- Permitem transformar programas USPL em programas USPL mais simples, que possam ser mais fáceis de optimizar com as técnicas derivadas dos MDPs.

Como já foi referido, este trabalho é muito preliminar. Nomeadamente é necessário descobrir muito mais regras para transformação do corpo do programa, tal como a última apresentada, e provar pelo menos a correcção destas regras, ou seja, demonstrar que o conjunto de computações do programa resultante de aplicar a regra é sempre um subconjunto das computações originais.

Capítulo 5

Discussão

Para finalizar esta tese vamos apresentar brevemente algumas das conclusões mais relevantes deste trabalho e dedicar algum espaço a discutir possíveis evoluções para trabalho futuro. Esta última secção é particularmente importante e extensa, pois esta investigação está longe de se encontrar finalizada, devendo, nomeadamente, prosseguir a curto prazo com a introdução de primitivas de concorrência no USPL.

5.1 Conclusões

Após a conclusão parcial deste trabalho de investigação, a principal conclusão é a de que é possível conceber uma linguagem de programação simples e explícita para agentes orientados por funções de utilidade. A linguagem USPL surge da extensão de um subconjunto da linguagem imperativa SPL, através da parametrização do programa com uma função de utilidade que caracteriza as preferências do agente a especificar. Esta pequena alteração tem grandes consequências no comportamento esperado do programa, nomeadamente no que toca ao comportamento associado às escolhas não determinísticas.

O processo utilizado para compilar programas escritos nesta linguagem, consistiu em desenvolver um transformador para a linguagem SPL da qual derivou. Embora o SPL seja uma linguagem de especificação para programas reactivos, não possuindo por isso um compilador ou interpretador que permita executar os seus programas, seria relativamente fácil compilar os seus programas pois, considerando apenas o subconjunto de instruções utilizado, é uma linguagem imperativa relativamente simples. Sempre que possível tentou-se especificar formalmente os problemas em análise, nomeadamente, no tocante a equivalência entre um programa USPL e um programa SPL, que constitui o conceito fundamental desta tese.

A aplicabilidade e eficiência da teoria apresentada nesta tese foi demonstrada através do desenvolvimento de uma aplicação que permite compilar programas USPL para SPL, e gerar os ficheiros de provas para o STeP no caso do refinamento. Esta aplicação foi desenvolvida na linguagem Haskell [ABB+98].

Por enquanto, a linguagem desenvolvida só permite a especificação de agentes isolados do seu ambiente, mas mesmo neste estado do seu desenvolvimento

```
local dinheiro:[0..10]
local moedas:[0..10] where moedas=0
local continua:[0..1] where continua=1

P :: [
  while continua=1 do
    [
      [
        guard ((dinheiro=6/\moedas=10)\/dinheiro>=7\/dinheiro=5)
        do dinheiro := (dinheiro-5)
      or
        guard ((dinheiro=6/\moedas<=9)\/(dinheiro=4/\moedas=10)\/dinheiro=3)
        do dinheiro := (dinheiro-3)
      or
        guard ((dinheiro=4/\moedas<=9)\/dinheiro=2)
        do dinheiro := (dinheiro-2)
      ];
      guard !(moedas=10) do moedas := (moedas+1);
      [
        guard (dinheiro=1\/dinheiro=0) do continua := 0
      or
        continua := 1
      ]
    ]
]
```

Figura 5.1: Máquina de destrocar moedas em SPL

já é possível implementar pequenos casos de estudo com algum interesse prático. Por exemplo, usando a aplicação desenvolvida foi possível transformar o programa USPL que especifica uma máquina de destrocar moedas no equivalente programa SPL que é apresentado na figura 5.1[1].

Este exemplo também serve para demonstrar a capacidade de abstracção da linguagem USPL, pois, enquanto que o programa USPL original é bastante inteligível, o programa SPL que o implementa apresenta algumas situações, aparentemente anómalas, que um programador provavelmente não contemplaria ao fazer uma implementação manual da especificação:

- Existe um comportamento pouco óbvio quando já se gastaram 10 moedas. Dado que a máquina vai de certeza bloquear na próxima instrução, então é preferível tentar minimizar rapidamente o dinheiro que falta destrocar. Este problema não existiria se fosse possível fazer atribuições simultâneas, sendo as variáveis dinheiro e moedas modificadas simultaneamente. Na prática, como nunca se gastam 10 moedas para destrocar qualquer quantia

[1]Foi utilizado como critério de paragem do algoritmo *value iteration* um erro máximo de 0.1.

permitida, esta situação nunca se vai verificar.

- Quando se chega a valores inferiores ou iguais a 1$00 é indiferente continuar ou não, pois para o programa é indiferente terminar o ciclo, e ficar indefinidamente a executar a transição nula, ou ficar bloqueado na escolha não determinística dentro do ciclo.

A maior complexidade, e interesse, deste trabalho deveu-se à necessidade de lidar com programas com possíveis ciclos infinitos, e à consequente necessidade de implementar mecanismos para comparar computações infinitas partindo de uma especificação das preferências através de uma função de utilidade definida sobre os estados. Este problema não tem uma solução universalmente aceite, tendo sido necessário incluir na própria linguagem USPL primitivas para especificar qual o critério de decisão a utilizar. Para os critérios de decisão definidos até ao momento, o processo de transformação de um programa USPL no equivalente programa SPL foi implementado à custa de técnicas previamente desenvolvidas na área dos *Markov Decision Processes*.

O maior contributo desta tese consistiu precisamente na integração, com alguns cuidados de correcção formal, de uma série de conceitos e técnicas de áreas relativamente dispersas na concepção de uma linguagem de programação baseada em funções de utilidade. Para além desta integração, fez-se um estudo preliminar de uma técnica de refinamento gradual para programas USPL. Tirando partido da existência de técnicas e ferramentas para provar propriedades especificadas em lógica temporal sobre programas SPL, definiu-se um conjunto de regras condicionais que permitem transformar simbolicamente programas USPL em programas USPL mais simples, ou directamente em programas SPL.

5.2 Trabalho futuro

No tocante às perspectivas de trabalho futuro, vamos começar por referir algumas das ideias que serão exploradas para melhorar a linguagem USPL na sua versão actual, sendo os problemas derivados da introdução da concorrência analisados na secção 5.2.1.

A um nível mais teórico existe ainda bastante trabalho a desenvolver, nomeadamente com o objectivo de consolidar os fundamentos do processo de optimização implementado. Um dos objectivos a muito curto prazo consiste em demonstrar formalmente que as estratégias determinadas pelo processo de optimização apresentado na secção 4.2.1 são realmente óptimas ou, possivelmente, até completas, de acordo com as definições formais dos critérios decisão apresentadas na secção 2.4.1. Outra tarefa muito importante consiste em demonstrar formalmente a correcção das regras de refinamento apresentadas na secção 4.4. Neste momento, estas regras, embora aparentemente úteis, são apenas empíricas, podendo ser questionada a sua correcção. Julgo que, para a maior parte das regras apresentadas, essa prova deverá ser relativamente simples. Obviamente, também é necessário descobrir mais regras de refinamento para que este método de optimização possa ser usável por si só, sem que seja necessário utilizar o processo de optimização apresentado na secção 4.2.

No tocante aos critérios de decisão, também é muito importante tentar identificar formalmente classes de critérios de decisão para os quais é possível calcular estratégias óptimas. Na secção 4.2 apenas foi demonstrado que no caso geral tal não é garantido. Se estas tarefas não forem possíveis de concretizar com definições genéricas de critérios de decisão, então deverão ser feitas pelo menos para os critérios de decisão já apresentados.

Também é necessário demonstrar, ou refutar, a possibilidade de se traduzir as preferências expressas por uma função de utilidade numa especificação razoável em lógica temporal. Se for possível caracterizar desta forma as computações escolhidas por um critério de decisão, então será também possível utilizar técnicas de síntese semelhantes às apresentadas na secção 3.2 para determinar os programas SPL equivalentes aos programas originais.

Uma das hipóteses interessantes a explorar passa pela utilização de conceito de computação válida diferente, que ignore possíveis variáveis locais do processo (num conceito aproximado da simulação proposta em [KMP94]). A vantagem desta alteração seria a possibilidade de o processo utilizar estratégias com memória sem os inconvenientes referidos na secção 3.1.3.

A um nível mais prático também existem várias tarefas a concretizar e tecnologias para explorar, nomeadamente ao nível da determinação das estratégias óptimas com técnicas derivadas dos MDPs. Para começar é necessário implementar o algoritmo *policy iteration*, pois ao contrário do algoritmo *value iteration* onde, no caso da recompensa total descontada, apenas se garante a obtenção de estratégias ϵ-óptimas, com o *policy iteration* temos a garantia de obtenção de estratégias realmente óptimas [Put90]. Desta forma, deixaria de ser necessário especificar o erro pretendido aquando da optimização. Como no nosso caso as transições são sempre determinísticas também deve ser explorado o algoritmo fortemente polinomial apresentado em [PT87], que é específico para MDPs com esta particularidade.

Para que as técnicas apresentadas nesta tese possam ser aplicadas a programas com uma dimensão real, também é necessário começar a considerar utilizar técnicas de redução do espaço de estados, tais como as apresentadas na secção 3.4.1. A técnica apresentada em [DG97] parece particularmente simples de aplicar, podendo, neste caso, os $\mathcal{P}$-diagramas ser utilizados para codificar as partições. Por exemplo, no caso do programa da figura 2.2 o número de estados do sistema de transição explícito poderia ser reduzido para metade antes de ser aplicado o algoritmo *value iteration*. Outra técnica de optimização que convém explorar é a apresentada em [LD95] num trabalho que apresenta muitas semelhanças com o nosso (ver secção 3.4.2).

Como as técnicas de resolução de MDPs são apenas mais um mecanismo utilizado no processo de optimização, e não um objectivo deste trabalho de investigação, também será considerada a possibilidade de se utilizar pacotes de software para resolução de MDPs desenvolvidos por terceiros.

Finalmente, e já sem relação directa com os MDPs, é necessário implementar mais critérios de decisão, como, por exemplo, o limite da média das recompensas, e tentar utilizar no protótipo desenvolvido alguma ferramenta para simplificar expressões inteiras e booleanas, por forma a que os programas SPL resultantes sejam mais inteligíveis. Nalguns dos exemplos apresentados, essa

optimização foi feita manualmente para facilitar a compreensão dos programas.

5.2.1 A introdução da concorrência

A primeira grande questão em relação à introdução da concorrência no USPL diz respeito à forma sintáctica como irá ser feita. Sem perder de vista os objectivos desta linguagem, julgo que a melhor opção consistirá em adoptar um paralelismo apenas ao nível do processo com comunicação por memória partilhada. Cada processo, com a respectiva função de utilidade, representará um agente do ambiente cuja funcionamento interno é estritamente sequencial. A opção pela comunicação apenas via memória partilhada é a que implica menos alterações na definição actual da linguagem, pois não é necessário acrescentar instruções explícitas para comunicação. Também não obriga à existência de transições compassivas nos sistemas de transição, dado que estas apenas derivam de primitivas de comunicação síncrona por mensagens. Este modelo de comunicação também é frequentemente adoptado em trabalhos de síntese de sistemas reactivos, como por exemplo, em [WTD91]. Com estas opções as alterações ao USPL são mínimas, passando a ser possível coexistirem vários processos em vez de apenas um, e permitindo a existência de variáveis locais aos processos.

Fixados os aspectos sintácticos, surge-nos a segunda grande questão: qual o objectivo concreto do USPL concorrente? No caso actual, o objectivo consistia em especificar um agente orientado por uma função de utilidade por forma a obter a respectiva implementação numa linguagem tradicional. No caso concorrente, será que se pretende traduzir todos agentes para a respectiva implementação ou apenas um deles? Neste momento, julgo que a segunda opção é a que tem mais interesse prático. O objectivo continua a ser obter uma implementação em SPL de um agente orientado por uma função de utilidade, tendo agora em atenção que esse agente coexiste numa sociedade competitiva com outras entidades que são também modeladas por agentes deste tipo. Este enquadramento é muito interessante porque, na maior parte das situações reais, o programador não conhece concretamente a forma como as entidades externas estão implementadas, mas pode ser fácil perceber apenas quais são os seus objectivos e quais as opções de actuação que têm ao dispor. É mais fácil tentar modelar uma entidade que não controlamos por um agente orientado por uma função de utilidade do que tentar adivinhar qual o programa concreto que a implementa.

Fazendo o paralelismo com o problema da síntese de programas, no trabalho actual apenas tratamos da síntese de um sistema fechado, cujo comportamento depende unicamente do seu estado interno, enquanto que com a concorrência o processo a optimizar é um sistema aberto, que interage com o seu ambiente e cujo comportamento depende desta interacção [KMTV00]. Como se viu na secção 3.2, existem muitos trabalhos na área de síntese relacionados com este problema. Recentemente começam também a surgir trabalhos cujo objectivo é sintetizar implementações para todos os agentes de um sistema distribuído [KV01], mas sabe-se, através de alguns resultados da teoria dos jogos, que o problema de saber se existe ou não uma implementação para todos os processos

```
local x:[-1..1] where x=0

P (-abs(x),look-at-end) :: [
  x := x+1 or skip;
]

||

Q (0,look-at-end) :: [
  x := x-1 or skip;
]
```

Figura 5.2: Exemplo de programa USPL concorrente

é no caso geral indecidível [PR90]. Este resultado, e a falta de referências neste tipo de síntese também contribuíram para a adopção desta estratégia.

Definidos, muito genericamente, os objectivos e a forma sintáctica do USPL concorrente, passamos agora a uma análise muito breve dos novos problemas que surgirão devido a este enriquecimento e tentamos apontar algumas pistas para a sua resolução.

Na versão actual da linguagem o grande problema com que nos deparamos foi a necessidade de comparar sequências infinitas de utilidades. Obviamente, esse problema vai continuar no USPL concorrente, mas irá surgir um novo problema que é o da incerteza aquando da tomada de decisões. No modelo actual não existe qualquer dos tipos de incerteza apontados na secção 3.1.1. Todas as acções são determinísticas e, obviamente, em cada momento é possível determinar com certeza qual o estado actual da execução. No USPL concorrente as acções continuarão a ser determinísticas mas, devido à existência de vários processos, surgirá a incerteza quanto ao estado global da computação.

Para ver em que medida esta incerteza afecta o processo de decisão considere-se o possível exemplo de programa USPL concorrente apresentado na figura 5.2, onde o processo que se pretende optimizar é o `P`. Note-se que `Q` representa uma terceira entidade da qual se conhece concretamente o comportamento esperado, pois a sua função de utilidade é constante. Neste programa, `P` não consegue saber se `Q` já executou ou não a sua escolha, pois não consegue distinguir as situações em que `Q` decide executar a instrução `skip` e aquelas em que este ainda não decidiu. Obviamente, isto é verdade porque `P` não consegue aceder ao valor da variável de controlo de `Q`. Esta incerteza não consegue ser resolvida em tempo de execução e, como tal, o processo de decisão tem de coexistir com ela.

Neste momento, prevê-se que a optimização no caso concorrente se vá inspirar fortemente em duas áreas: nos problemas de decisão de Markov parcialmente observáveis (POMDPs) e na teoria dos jogos. A primeira técnica será útil nos casos em que os agentes do ambiente envolvente podem ser considerados irracionais (i.é, a sua função de utilidade é constante). Nestes casos temos um problema praticamente igual ao actual, com a única excepção que o agente não

sabe com exactidão qual o estado global em cada instante. Esta é precisamente a diferença que caracteriza os POMDPs em relação aos MDPs, sendo uma evolução natural e imediata do trabalho actual.

Se os restantes agentes tiverem funções de utilidade não constantes, o processo de optimização deve ter em atenção que esses agentes vão agir racionalmente e vão tentar prever a estratégia óptima do agente a optimizar ao calcular a sua melhor estratégia. Naturalmente, estas tentativas de previsão levam a uma regressão infinita no processo de modelação dos oponentes, sendo necessário usar técnicas de optimização que lidem com esta situação. A teoria dos jogos lida precisamente com este tipo de problemas de decisão em sociedades de seres racionais. O seu princípio básico consiste em modelar essa interacção como um tradicional jogo em que os jogadores devem executar acções alternadamente[2].

Muito sucintamente, um jogo na forma normal (o tipo de jogo mais simples) consiste num tuplo $\langle N, (A_i), (\succsim_i)\rangle$ onde[3] [OR94]:

- N é um conjunto finito de jogadores;
- para cada jogador $i \in N$, A_i é o conjunto não vazio de acções disponíveis para o jogador i;
- para cada jogador $i \in N$, $\succsim_i$ é a sua relação de preferência[4] definida sobre $A = \times_{j \in N} A_j$.

O facto de a relação de preferência de cada jogador i estar definida sobre A, e não sobre A_i, é a característica que distingue um jogo na forma normal de um problema de decisão. Num problema de decisão uma entidade apenas se preocupa com as consequências das suas acções, enquanto que num jogo podem ser relevantes para um jogador as acções escolhidas pelos restantes. Existem vários mecanismos que permitem, dado um jogo, determinar as acções que cada jogador irá escolher. Estes mecanismos denominam-se conceitos de solução e os vectores de acções por eles determinados denominam-se pontos de equilíbrio. O conceito de solução mais utilizado na teoria dos jogos é o equilíbrio de Nash [Nas50]. Dado um jogo na forma normal $\langle N, (A_i), (\succsim_i)\rangle$, um equilíbrio de Nash é um vector $a^* \in A$ de acções que, para todo o jogador $i \in N$, verifica a seguinte propriedade[5]:

$$\forall a_i \in A_i \cdot (a^*_{-i}, a^*_i) \succsim_i (a^*_{-i}, a_i)$$

Essencialmente, este conceito diz-nos que um conjunto de estratégias apenas é aceitável quando para cada um dos intervenientes, se forem fixadas as restantes estratégias, não for possível obter um melhor resultado. Na perspectiva do desenvolvimento de agentes informáticos que actuam numa rede aberta,

[2]Note-se que esta alternância não nos deve trazer problemas, pois é conhecido que a convenção de que cada jogador deve actuar à vez pode ser substituída pela noção de acções simultâneas mais convencional na modelação da concorrência [AMP95].

[3]Quando nos referirmos a um tuplo $(x_i)_{i \in N}$, constituído por um valor para cada elemento de N, será omitido sempre que a clareza não seja afectada o qualificador "$i \in N$".

[4]Neste contexto, uma relação de preferência é uma relação binária completa, reflexiva e transitiva.

[5]Dado um vector $a = (a_j)_{j \in N}$, e um índice $i \in N$, a_{-i} representa o vector $(a_j)_{j \in N \setminus \{i\}}$ e (a_{-i}, a_i) representa todo o vector a.

a importância do equilíbrio de Nash também pode ser vista da seguinte forma [KP97]: a possibilidade de um processo revelar a sua estratégia sem que os outros tenham alguma vantagem permite implementar programas sem preocupações de garantir a inviolabilidade do código.

Dentro da área da teoria dos jogos, as técnicas que potencialmente poderão ter mais interesse para o nosso trabalho dizem respeito aos jogos repetidos com informação incompleta, com possibilidade de repetição infinita [AM95]. Neste tipo de jogo, os jogadores repetem (indefinidamente) o mesmo jogo, normalmente referido como jogo constituinte. Em cada interacção os jogadores escolhem as suas acções simultaneamente e conhecem todas as acções previamente escolhidas por todos os jogadores. A falta de informação é modelada através de *information sets* que, essencialmente são conjuntos de estados globais que o jogador não consegue distinguir, sendo estes o domínio das estratégias calculadas. No entanto, para que esta teoria se aplique ao USPL concorrente há que perceber de que forma um programa com ciclos arbitrários pode ser correctamente modelado por uma repetição de um jogo.

Para finalizar esta análise sucinta do potencial da teoria dos jogos falta apenas referir uma velha questão que tem causado muitos conflitos com alguns investigadores da teoria da decisão. Normalmente estes argumentam que a teoria dos jogos tem apenas um carácter preditivo, não podendo ser usada para prescrever uma estratégia óptima para um jogador. A justificação para este facto passa quase sempre pelo mesmo argumento: a teoria dos jogos assume que todos os jogadores são racionalmente perfeitos e na vida real isso não acontece. No entanto, no caso de entidades artificiais essa questão não se coloca podendo esta teoria ser aplicada com sucesso. Aprecie-se a seguinte frase de Rosenschein e Zlotkin sobre este assunto [RZ94]:

> *"Embora a teoria dos jogos contribua para a compreensão de muitas áreas de investigação, existe uma semelhança muito forte entre esta teoria e os sistemas de computação heterogéneos. Os computadores, sendo entidades pré-programadas, tornam concreta a noção de estratégia que tem um papel tão fundamental na teoria dos jogos (...) Este jogador idealizado é um modelo imperfeito para o comportamento humano, mas muito apropriado para computadores.*

Um exemplo de sistema que incorpora técnicas da teoria dos jogos e cujos objectivos são muito semelhantes ao nosso foi desenvolvido por Koller e Pfeffer [KP97], e poderá ser um bom ponto de partida para a investigação futura. Este sistema, denominado Gala, parte de uma descrição de um jogo com informação incompleta, analisa-a usando técnicas muito eficientes e produz estratégias racionais para os diferentes jogadores, compatíveis com as noções de equilíbrio da teoria dos jogos. Embora tenha objectivos semelhantes aos pretendidos, este sistema não pode facilmente ser utilizado para optimizar programas USPL: primeiro, a linguagem de descrição nele utilizada é muito orientada para os jogos enquanto que o USPL é mais parecida com uma linguagem de programação normal, sendo a conversão entre elas aparentemente complicada; segundo, o nosso sistema pretende optimizar um programa e devolver a estratégia racional descrita sob a forma de um programa convencional que possa ser directamente

executado, enquanto que no sistema Gala são calculadas estratégias probabilísticas que não se adequam correctamente a este objectivo; finalmente, a maior limitação é que, embora a linguagem de descrição permita ciclos, o sistema só funciona correctamente se estes terminarem sempre, ou seja, não permite os ciclos infinitos que constituem o caso mais interessante do USPL.

Apêndice A

Lógica temporal

A linguagem SPL foi desenvolvida por Manna e Pnueli como parte de um método para provar que programas reactivos satisfazem as suas especificações [MP95]. Neste método, estas especificações são formuladas em lógica temporal linear. No capítulo 4 é utilizada lógica temporal várias vezes para expressar formalmente propriedades sobre programas SPL, sendo, por isso, a interpretação das fórmulas aqui apresentada feita sobre FTSs.

A lógica temporal linear extende a lógica de 1ª ordem através da introdução de um conjunto de operadores temporais que permitem expressar de forma natural e sucinta propriedades de precedência, invariância ou ocorrência sistemática de eventos ao longo do tempo. Os operadores (e seu significado informal) que iremos utilizar são os seguintes:

- $\Box p$. Sempre no futuro p: a fórmula p é válida no instante presente e em todos os instantes futuros.
- $\Diamond p$. Algures no futuro p: a fórmula p é válida no instante presente ou em alguns dos instantes futuros.
- $p \mathcal{U} q$. p até que q: a fórmula p é válida do instante presente e em todos os estados futuros até que q seja válida.
- $p \mathcal{W} q$. p à espera de q: a fórmula p é válida do instante presente e em todos os estados futuros até que eventualmente q seja válida.
- $\bigcirc p$. A seguir p: a fórmula p é válida no próximo instante.
- $p \Rightarrow q$. q é consequência de p: em qualquer instante, sempre que p ocorre, q também ocorre.

Estes operadores não são independentes entre si. Entre outras, podemos definir as seguintes equivalências:

$$\begin{aligned} \Box p &\equiv p \mathcal{W} Falso \\ \Diamond p &\equiv \neg \Box (\neg p) \\ p \Rightarrow q &\equiv \Box (p \supset q) \\ p \mathcal{U} q &\equiv (p \mathcal{W} q) \wedge (\Diamond q) \end{aligned}$$

As fórmulas da lógica temporal são interpretadas sobre uma computação σ e sobre uma posição específica k dessa computação. Se uma fórmula p for válida na posição k de σ escreve-se:

$$(\sigma, k) \models p$$

Esta validade determina-se de acordo com as seguintes regras:

- Para uma fórmula de estado p[1]:

$$(\sigma, k) \models p \quad \equiv \quad s_k \Vdash p$$

- Para as conectivas de 1ª ordem, a semântica define-se como usual nessa lógica. Por exemplo:

$$\begin{aligned} (\sigma, k) \models \neg p &\equiv (\sigma, k) \not\models p \\ (\sigma, k) \models p \vee q &\equiv (\sigma, k) \models p \text{ ou } (\sigma, k) \models q \end{aligned}$$

- Para os operadores temporais temos:

$$\begin{aligned} (\sigma, k) \models \Box p &\equiv \forall j \geq k \cdot (\sigma, j) \models p \\ (\sigma, k) \models \Diamond p &\equiv \exists j \geq k \cdot (\sigma, j) \models p \\ (\sigma, k) \models p \mathcal{U} q &\equiv \exists j \geq k \cdot (\sigma, j) \models q \wedge (\forall k \leq i < j \cdot (\sigma, i) \models p) \\ (\sigma, k) \models p \mathcal{W} q &\equiv (\sigma, k) \models p \mathcal{U} q \ \vee \ (\sigma, k) \models \Box p \\ (\sigma, k) \models \bigcirc p &\equiv (\sigma, k+1) \models p \\ (\sigma, k) \models p \Rightarrow q &\equiv (\sigma, k) \models \Box(p \supset q) \end{aligned}$$

Uma fórmula temporal p é válida numa computação σ sse for válida no seu estado inicial:

$$\sigma \models p \quad \text{sse} \quad (\sigma, 0) \models p$$

Uma fórmula temporal p é válida para um programa $\mathcal{P}$ sse for válida para toda a $\mathcal{P}$-computação σ, representando-se:

$$\mathcal{P} \models p$$

Os autores apresentam também um conjunto de regras que permitem provar a validade de fórmulas temporais em programas. Por exemplo, considerando o FTS de um programa $\mathcal{P}$ representado pelo tuplo $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, para provar a validade de uma fórmula $\Box p$, onde p é uma fórmula de estado pode ser utilizada a regra *INV* que é apresentada na figura A.1, onde φ representa uma fórmula de estado auxiliar. As premissas desta regra implicam, respectivamente, que φ é válida nos estados iniciais da computação, que implica a asserção p cuja invariância se quer demonstrar, e que todas as transições preservam a sua validade. Naturalmente, se todas estas premissas se verificarem tanto φ como p são invariantes do programa.

[1] Uma fórmula de estado não contem qualquer conectiva temporal.

$$\frac{\begin{array}{ll}\text{I1.} & \Theta \supset \varphi \\ \text{I2.} & \varphi \supset p \\ \text{I3.} & \forall \tau \in \mathcal{T} \cdot \rho_\tau \wedge \varphi \supset \varphi'\end{array}}{\mathcal{P} \models \Box p}$$

Figura A.1: Regra INV

Apêndice B

Exemplo de compilação de um programa USPL

No capítulo 4 apenas foi apresentado, para demonstrar o processo de transformação de um programa USPL para um programa SPL equivalente, um exemplo demasiado simples, que nem sequer incluia ciclos. Para colmatar esta falha, é apresentado neste anexo um exemplo ligeiramente mais complicado, mas onde já existe um ciclo com possibilidade de execução infinita. O exemplo apresentado na figura B.1 corresponde a um programa que procura tornar o valor da variável x o mais próximo possível de 0, com o objectivo de terminar o mais depressa possível o ciclo que o constitui. Esse objectivo é representado pela função de utilidade $-|x|$, sendo adoptado neste caso o critério de decisão `look-ahead 1`. De notar que neste caso não seria possível utilizar o critério de decisão `look-at-end`, pois existem execuções onde x é alternadamente incrementado e decrementado.

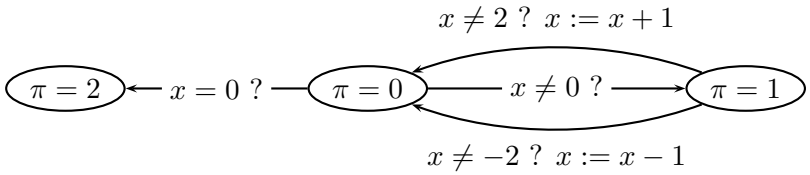
```
local x:[-2..2]

P(0-abs(x), look-ahead 1) :: [
  while !(x=0) do
    [
      guard !(x=2) do x := (x+1)
    or
      guard !(x=-2) do x := (x-1)
    ]
]
```

Figura B.1: Exemplo de um programa USPL com um ciclo.

O UFTS correspondente a este programa USPL é o seguinte:

$$
\begin{aligned}
V &= \{x, \pi\} \\
\Theta &= (\pi = 0) \\
\mathcal{T} &= \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_I\} \\
\rho_{\tau_1} &= (\pi = 0) \wedge (x \neq 0) \wedge (\pi' = 1) \\
\rho_{\tau_2} &= (\pi = 0) \wedge (x = 0) \wedge (\pi' = 2) \\
\rho_{\tau_3} &= (\pi = 1) \wedge (x \neq 2) \wedge (\pi' = 0) \wedge (x' = x + 1) \\
\rho_{\tau_4} &= (\pi = 1) \wedge (x \neq -2) \wedge (\pi' = 0) \wedge (x' = x - 1) \\
u &= \lambda s. - |s(x)| \\
\rho &= \lambda\ u\ \Omega\ .\ \uparrow_{\curvearrowright_u^1}\ \Omega
\end{aligned}
$$

Este UFTS é convenientemente visualizado pelo $\mathcal{P}$-diagrama de controlo seguinte:



Depois de expandir o UFTS para um sistema de transição explícito, de acordo com a secção 4.1, obtém-se o sistema de transição da figura B.2.

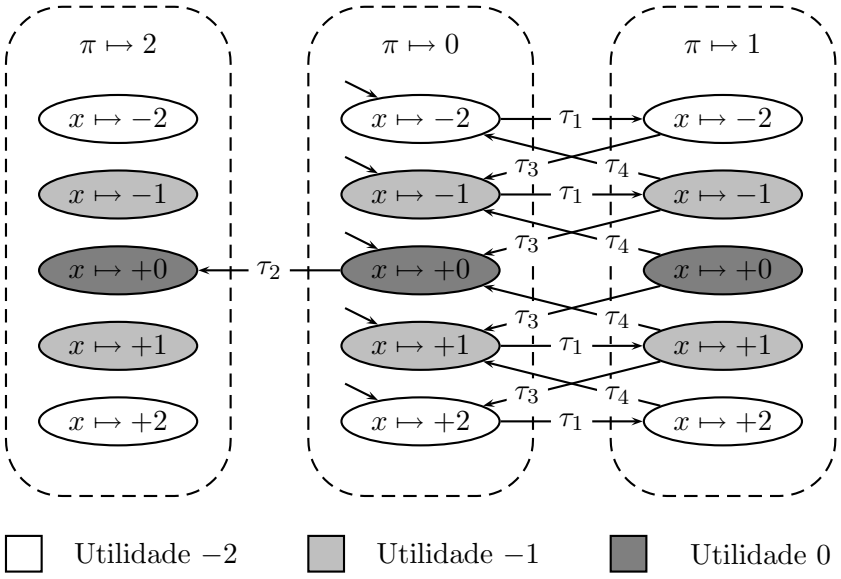


Figura B.2: Sistema de transição do exemplo da figura B.1.

Depois de determinada a estratégia óptima para este sistema de transição através do algoritmo *value iteration* apresentado na secção 4.2.1, o sistema de transição que se obtém do anterior depois de eliminados os ramos que não são seleccionados por essa estratégia é o apresentado na figura B.3.

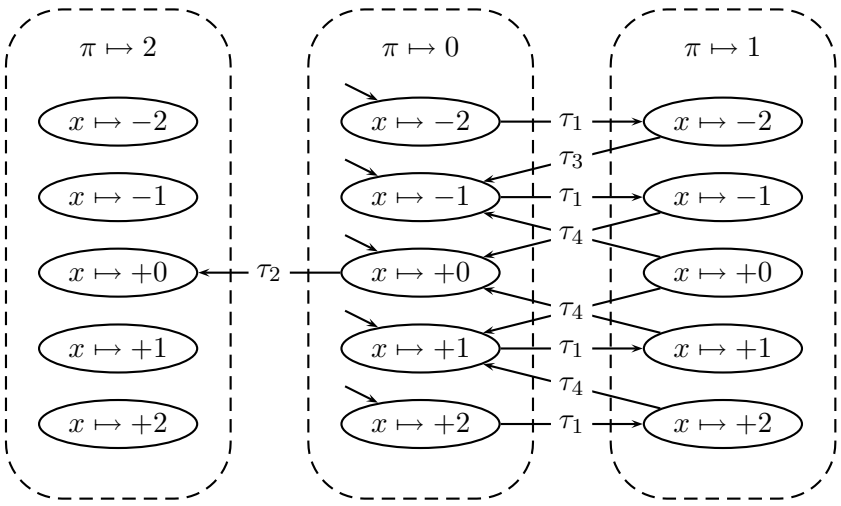
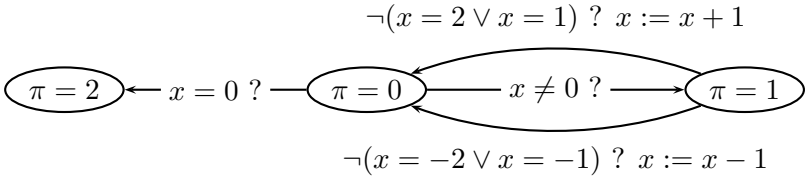


Figura B.3: Sistema de transição optimizado do exemplo da figura B.1.

Note-se que, tal como seria de esperar, quando $x = -1$ deixou de ser possível decrementar, e, quando $x = 1$ deixou de ser possível incrementar, garantindo-se, assim, uma convergência ideal para $x = 0$. A este sistema de transição corresponde um FTS que pode ser representado pelo seguinte $\mathcal{P}$-diagrama de controlo:



Finalmente, o programa SPL que corresponde a este FTS, e que é equivalente ao programa USPL original, é apresentado na figura B.4. Note-se que, neste caso, a instrução `loop forever` pode ser convertida para um `while`, pois o ciclo não estava inserido no contexto de uma escolha não determinística. Uma situação interessante com este exemplo é que, caso se usa-se um critério `look-ahead` n com $n >= 5$, o programa SPL resultante era exactamente igual ao original. A razão para este facto deve-se à possibilidade de, mesmo que se decremente quando $x = -1$ ou se incremente quando $x = 1$, em 5 passos existir a possibilidade de chegar ao valor 0.

```
local x:[-2..2]

P :: [
  while !(x=0) do
    [
      guard !(x=2 \/ x=1) do x := (x+1)
    or
      guard !(x=-2 \/ x=-1) do x := (x-1)
    ]
]
```

Figura B.4: Programa SPL equivalente ao exemplo da figura B.1.

Bibliografia

[ABB+98] L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, P. Wadler, S. Jones (editor), and J. Hughes (editor). Haskell 98: A non-strict, purely functional language. Technical report, 1998.

[AM95] Robert Aumann and Michael Maschler. *Repeated Games With Incomplete Information*. The MIT Press, 1995.

[AMP95] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid System II*, volume 999 of *LNCS*. Springer-Verlag, 1995.

[BBC+97] Nokolaj Bjorner, Anca Browne, Eddie Chang, Michael Colón, Bernd Finkbeiner, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomás Uribe. *STeP: The Stanford Temporal Prover Educational Release (User's Manual)*. Computer Science Department, Stanford University, October 1997.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BG96] Fahiem Bacchus and Adam Grove. Utility independence in a qualitative decision theory. In *Principles of Knowledge Representations and Reasoning (proocedings of KR'96)*, pages 542–552, 1996.

[BT96] Ronen Brafman and Moshe Tennenholtz. On the fondations of qualitative decision theory. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, 1996.

[BT97] Ronen Brafman and Moshe Tennenholtz. Modeling agents as qualitative decision makers. *Artificial Intelligence*, 94(1–2):217–268, 1997.

[BT00] Ronen Brafman and Moshe Tennenholtz. An axiomatic treatment of three qualitative decision criteria. *Journal of the ACM*, 47(3):452–482, 2000.

[Den87] Daniel Dennett. *The intentional stance*. The MIT Press, 1987.

[DG97] Thomas Dean and Robert Givan. Model minimization in markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 106–111, 1997.

[DKKN95] Thomas Dean, Leslie Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constrains in stochastic domains. *Artificial Intelligence*, 76(1–2):35–74, 1995.

[ES88] Allen Emerson and Jai Srinivasan. Branching time temporal logic. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 123–172. Springer-Verlag, 1988.

[Fis94] Michael Fisher. A survey of concurrent metatem - the language and its applications. In D. Gabbay and H. Ohlbach, editors, *Temporal Logics - Proceedings of First International Conference*, volume 827 of *LNAI*, pages 480–505. Springer-Verlag, 1994.

[Fis95] Michael Fisher. Towards a semantics for concurrent metatem. In M. Fisher and R. Owens, editors, *Executable Modal and Temporal Logics*, volume 897 of *LNAI*. Springer-Verlag, 1995.

[Fre93] Simon French. *Decision Theory: An Introduction to the Mathematics of Rationality*. Ellis Horwood, 1993.

[HMK+98] Milos Hauskrecht, Nicolas Meuleau, Leslie Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth International Conference on Uncertainty in Artificial Intelligence*, 1998.

[How60] Ronald Howard. *Dynamic Programming and Markov Processes*. The MIT Press, 1960.

[KLC95] Leslie Kaelbling, Michael Littman, and Anthony Cassandra. Planning and acting in partially observable stochastic domains. Technical Report CS-96-08, Brown University, Providence, RI, 1995.

[KMP94] Yonit Kesten, Zohar Manna, and Amir Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *LNCS*, pages 273–346. Springer-Verlag, 1994.

[KMTV00] Orna Kupferman, P. Madhusudan, P.S. Thiagarajan, and Moshe Y. Vardi. Open systems in reactive environments: Control and synthesis. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory*, volume 1877 of *LNCS*, pages 92–107. Springer-Verlag, 2000.

[KP97] Daphne Koller and Avi Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94(1–2):167–215, 1997.

[KV01] Orna Kupferman and Moshe Vardi. Syntesizing distributed systems. In *Proceedings of LICS'01 (IEEE Symposium on Logic in Computer Science)*, 2001. To appear.

[LD95] Shieu-Hong Lin and Thomas Dean. Generating optimal policies for high-level plans with conditional branches and loops. In *Proceedings of the Third European Workshop on Planning*, pages 205–218, 1995.

[LDK95] Michael Littman, Thomas Dean, and Leslie Kaelbling. On the complexity of solving markov decision processes. In *Proceedings of the Eleventh International Conference on Uncertainty in Artificial Intelligence*, 1995.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, 1995.

[MT98] P. Madhusudan and P.S. Thiagarajan. Controllers for discrete event systems via morphisms. In D. Sangiorgi and R. de Simone, editors, *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 18–33. Springer-Verlag, 1998.

[Nas50] John Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36:48–49, 1950.

[OR94] Martin Osborne and Ariel Rubinstein. *A Course in Game Theory*. The MIT Press, 1994.

[Pap94] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st FOCS*, pages 746–757, 1990.

[PT87] Christos Papadimitriou and J. Tsitsiklis. The complexity of markov chain decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[Put90] Martin Puterman. *Stochastic Models*, volume 2 of *Handbooks in Operations Research and Management Science*, chapter 8, pages 331–434. North-Holland, 1990.

[Rao96] Anand Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Agents Breaking Away (Proceedings of MAAMAW'96)*, volume 1038 of *LNCS*, pages 42–55. Springer-Verlag, 1996.

[RG95a] Anand Rao and Michael Georgeff. Bdi agents: From theory to practice. In V. Lesser, editor, *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319. The MIT Press, 1995.

[RG95b] Anand Rao and Michael Georgeff. Formal models and decision procedures for multi-agent systems. Technical Report 61, Australian Artificial Intelligence Institute, 1995.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[RZ94] Jeffrey S. Rosenschein and Gilad Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*. The MIT Press, 1994.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In E.W. Mayr and C. Puech, editors, *Proceedings of STACS'95*, volume 900 of *LNCS*, pages 1–13. Springer-Verlag, 1995.

[TR94] Jonathan Tash and Stuart Russell. Control strategies for stochastic planner. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[Val00] José Valença. Elementos teóricos da programação 3: Notas teóricas. Departamento de Informática, Universidade do Minho, 2000.

[Var01] Moshe Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 1–22. Springer-Verlag, 2001.

[VW94] Moshe Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Wer96] Eric Werner. Logical foundations of distributed artificial intelligence. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 57–117. John Wiley & Sons, Inc., 1996.

[WJ95] Michael J. Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[Woo95a] Michael Wooldridge. This is myworld: The logic of an agent-oriented dai testbed. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents: Theories, Architectures and Languages*, volume 890 of *LNAI*, pages 160–178. Springer-Verlag, 1995.

[Woo95b] Michael Wooldridge. Time, knowledge, and choice. In M. Wooldridge, J. Müller, and M. Tambe, editors, *Intelligent Agents II (Proceedings of ATAL'95)*, volume 1037 of *LNAI*, pages 79–96. Springer-Verlag, 1995.

[WTD91] Howard Wong-Toi and David Dill. Synthesizing processes and schedulers from temporal specifications. In *Proceedings of the 1990 Computer-Aided Verification Workshop*, volume 531 of *LNCS*. Springer-Verlag, 1991.