Inteligência Artificial

Apontamentos para as aulas

Luís Miguel Botelho

Departamento de Ciências e Tecnologias da Informação
Instituto Superior de Ciências do Trabalho e da Empresa

Setembro de 2018

Sistemas de Regras de Produção

Índice

1 EXEMPLO DE SISTEMA DE REGRAS DE PRODUÇÃO (DESCRIÇÃO CONCETUAL) 5

2 RESOLUÇÃO DE CONFLITOS7

3 IMPLEMENTAÇÃO DAS REGRAS DO HERÓI E DO VILÃO 9

4 IMPLEMENTAÇÃO EM PROLOG DE UMA FERRAMENTA PARA CRIAÇÃO DE SISTEMAS DE PRODUÇÃO14

Sistemas de Regras de Produção

As regras SE…ENTÃO são as estruturas mais usadas para representar conhecimento em sistemas baseados em conhecimento. Existem dois tipos de regras: as regras condição-conclusão e as regras condição-ação (mais vulgarmente chamadas regras de produção). Em ambos os casos, as regras podem ser descritas através de estruturas com o formato IF *LHS* THEN *RHS*, em que LHS significa o lado esquerdo da regra ("*left hand side*") e RHS significa o lado direito da regra ("*right hand side*").

O raciocínio usado com este método de representação de conhecimento pode ser encadeado para trás (no caso das regras condição-conclusão) ou encadeado para a frente (tanto nas regras condição-conclusão como nas regras condição-ação).

O presente texto incide essencialmente sobre os Sistemas de Regras de Produção, ou seja, sistemas baseados em conhecimento procedimental, cuja base de conhecimento contém regras *SE condição ENTÃO ação*. Os sistemas de produção são usados para controlar sistemas, sejam eles de *hardware* (e.g., *robots*) ou de *software* (e.g., agentes inteligentes).

Sendo sistemas baseados em conhecimento, os sistemas de produção têm também uma base de conhecimento, um motor de inferência e uma interface. As regras do sistema e as ações que elas controlam são armazenadas na base de conhecimento. Existem também factos que representam o componente dinâmico do estado do mundo ou que representam informação estática que nunca é alterada. O programa que se encarrega de aplicar as regras aos factos é o motor de inferência. Finalmente, a interface é aquilo que permite a comunicação entre o sistema de produção e o exterior. Em sistemas de natureza profissional, essa interface poderá ser uma interface gráfica sofisticada. Nas aulas, a interface será sempre rudimentar. Em geral, será um programa que se manda executar e que apresenta os resultados da execução.

O funcionamento de um sistema de produção é o resultado daquilo que o motor de inferência faz às regras e factos existentes. O motor de inferência dos sistemas de produção são ciclos (virtualmente infinitos, embora alguns sejam feitos para terminar em certas circunstâncias). Em cada volta do ciclo, o motor de inferência verifica as condições de todas as regas e seleciona aquelas cuja condição estiver satisfeita. Ao conjunto de regras com a condição satisfeita chama-se conjunto de conflito porque é como se fossem várias regras a competir entre si para serem usadas. Seguidamente, o motor de inferência escolhe uma dessas regas e executa as suas ações. A escolha de uma das regras com a condição satisfeita dependem de critérios que serão explicados na secção 2. Depois, o ciclo repete-se: selecionar regras com a condição satisfeita, escolher uma delas, executar e assim por diante.

Dois aspetos deste funcionamento merecem alguma clarificação: *(i)* a razão pela qual, em cada volta do ciclo, não são executadas as ações de todas as regras com a condição satisfeita; e *(ii)* as circunstâncias em que o ciclo termina.

As regras que podem ser usadas são aquelas que tiverem a condição satisfeita pelos factos. Se, numa dada altura, duas regras tiverem a condição satisfeita, o sistema escolhe uma delas e executa a sua ação. Por definição, as ações alteram o estado do mundo. Por exemplo, depois da a ação dar um passo ser realizada o personagem que deu o passo deixou de estar onde estava e passou a estar noutro sítio – o estado alterou-se. Noutro exemplo, após um programa ter feito *download* de um ficheiro, o estado do mundo altera-se. Uma cópia do ficheiro passou a estar no disco desse programa.

Por alterarem o estado do mundo, depois de uma ação ser executada, as condições das regras que estavam satisfeitas podem deixar de estar, e as condições não satisfeitas podem passar a estar. Esta é a razão pela qual não se executam as ações de todas as regras com a condição satisfeita. Assim que as ações de uma delas são executadas, as outras regras poderão deixar de ser aplicáveis. A única coisa sensata a fazer é repetir o ciclo e identificar de novo as regras cuja condição passa a estar satisfeita.

Num caso geral, o estado do mundo é alterado por diversos intervenientes, por exemplo, pela simples passagem do tempo. Quando vários atores podem agir sobre o mundo, aquilo que não é verdade numa altura pode passar a ser verdade noutra altura. Como um sistema de regras de produção é um sistema que executa determinadas ações quando determinadas condições são verdadeiras, mesmo que num certo instante nenhuma dessas condições sejam verdadeiras, num instante futuro, algumas delas poderão ficar satisfeitas. Ou seja, num caso geral, o ciclo de um motor de inferência não pode terminar; pode parar apenas quando o programa morrer.

No entanto existem tipos de aplicações em que as únicas alterações de interesse que ocorrem no mundo são operadas pelo próprio sistema de regras de produção (ou apenas pelos dispositivos de hardware ou de software controlados por ele). Nesses mundos mais simplificados, o mundo só é alterado apenas pela execução de ações da responsabilidade do sistema de regras de produção. Quando nenhuma regra tiver a condição satisfeita, não será executada nenhuma ação. Como mais nada pode alterar o mundo, o seu estado deixará de se alterar. Isso implica que as condições das regras (que só dependem do estado do mundo) nunca mais serão verdadeiras. Ou seja, não vale a pena o motor de inferência continuar a verificar se as regras têm a condição satisfeita. Portanto, o ciclo do motor de inferência pode parar quando nenhuma regra tiver a condição satisfeita.

1 Exemplo de Sistema de Regras de Produção (descrição concetual)

As regras de produção especificam a execução de ações quando a sua condição se encontra satisfeita. As regras de produção são encadeadas para a frente.

No exemplo usado para apresentar este assunto, temos um mundo quadriculado com um vilão e um herói, como se mostra na Figura 1.

.

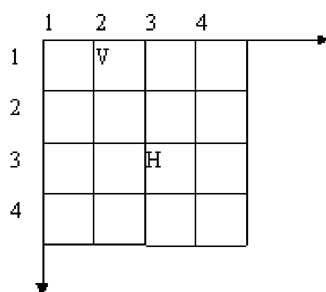

Figura 1 - O mundo do vilão

O vilão ocupa uma quadrícula fixa no mundo (1,2), e o herói vai movimentar-se de acordo com a aplicação de um conjunto de regras de produção apresentadas na Figura 2. O objetivo é controlar o herói desde a sua posição atual (3,3) até à mesma posição do vilão. Quando o herói tiver atingido o vilão, deverá liquidá-lo. As regras são escritas do ponto de vista do herói.

Regra 1 SE a posição atual for a mesma que a posição do vilão
ENTÃO Liquidar o vilão

Regra 2 SE o número da coluna da posição atual for superior ao número da coluna da posição do vilão
ENTÃO Dar um passo para a esquerda

Regra 3 SE o número da coluna da posição atual for inferior ao número da coluna da posição do vilão
ENTÃO Dar um passo para a direita

Regra 4 SE o número da linha da posição atual for inferior ao número da linha da posição do vilão
ENTÃO Dar um passo para baixo

Regra 5 SE o número da linha da posição atual for superior ao número da linha da posição do vilão
ENTÃO Dar um passo para cima

Figura 2 - Regras de produção

As ações possíveis no mundo do vilão são “liquidar o vilão” e “dar um passo” para baixo, para cima, para a direita e para a esquerda. Além das regras, um sistema baseado em regras de produção tem que ter um mecanismo de aplicação das regras e a programação das ações.

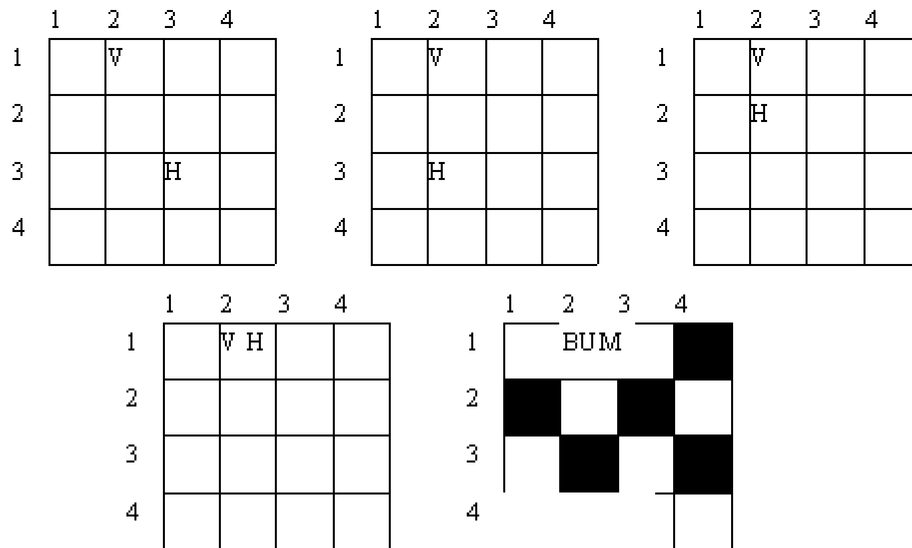

Figura 3 - Sequências de ações do herói

Na Figura 3 mostra-se a sequência de ações efetuadas pelo herói, de acordo com as regras de produção representadas na Figura 2. No início, as posições do vilão (1,2) e do herói (3,3) são introduzidas. O sistema determina as regras com a condição satisfeita. Tanto a regra 2 como a regra 5 têm a condição satisfeita. Em princípio, qualquer delas poderia ser usada. O sistema escolhe a regra 2 (não há razão especial para preferir a regra 2 em relação à regra 5). De acordo com a regra 2, o herói dá um passo para a esquerda, alterando a representação interna do estado atual (a posição do herói passa a ser (3,2)). A segunda configuração exibida na Figura 3 representa o resultado desta ação.

Devido à alteração na posição do herói, o sistema volta a verificar se alguma regra tem a condição satisfeita. Apenas a regra 5 tem a condição satisfeita pelo que o herói dá um passo para cima ficando na posição (2,2).

Devido à nova alteração da posição do herói, o sistema volta a verificar as suas regras. De novo, a regra 5 é a única com a condição satisfeita. Como resultado desta ação, o herói e o vilão ficam ambos na posição (1,2) (quarto quadro da Figura 3).

A alteração da posição do herói conduz o sistema a verificar as condições das suas regras. Desta vez, a regra 1 é a única cuja condição está satisfeita. De acordo com a regra 1, o herói mata o vilão.

2 Resolução de conflitos

Quando é usado raciocínio encadeado para a frente, pode acontecer que as condições de mais que uma regra fiquem satisfeitas. Nesta eventualidade o sistema tem que escolher a que vai usar porque não poderá usar mais do que uma. De facto, ao executar a ação da primeira regra com a condição satisfeita, o estado do mundo altera-se e, consequentemente é possível que as regras que tinham a condição satisfeita deixem de a ter.

Ao conjunto de regras com a condição satisfeita chama-se conjunto de conflito ("conflict set"). Aos critérios usados pelo sistema para selecionar apenas uma regra do conjunto de conflito chama-se política ou estratégia de resolução de conflitos ("conflict resolution strategy"). Os vários sistemas usam diferentes políticas de resolução de conflitos constituídas por vários critérios, entre os quais os seguintes:

1. Nenhuma regra pode ser usada duas vezes com os mesmos dados
2. Escolhe-se a regra que depende dos dados mais recentes
3. Escolhe-se a regra mais específica (i.e., a regra cuja condição é constituída pelo maior número de subcondições)
4. Escolhe-se a regra mais importante (neste caso, as regras têm que ser associadas a um fator de importância ou prioridade).
5. Escolhe-se a regra cuja ação seja a mais importante

O primeiro critério é bastante útil porque evita que a mesma regra possa continuar a ser usada devido aos mesmos factos. Ainda assim, há muitos sistemas que não implementam esse critério, nas suas políticas de resolução de conflitos. Nesse caso cabe ao programador criar mecanismos que invalidem a utilização repetida da regra.

A escolha da regra que depende de factos mais recentes (critério 2) é muito útil em sistemas que devem reagir rapidamente às alterações do mundo em que existem. De facto, se o sistema continuar a reagir a informação mais antiga, negligenciando alterações recentes, pode acontecer que o seu comportamento se torne desatualizado. No entanto, haverá aplicações em que seria preferível reagir a alterações mais antigas.

De acordo com o critério 3, quando duas ou mais regras têm a condição satisfeita, escolhe-se a regra mais específica. De facto, se o programador criou regras mais específicas do que outras é porque tinha em mente a sua utilização, sempre que as circunstâncias o permitissem.

Os critérios 4 e 5 permitiriam ao programador atribuir graus de importância (ou de prioridade) às regras ou mesmo às ações. Desta forma, se duas regras tiverem a condição satisfeita, seria natural que o sistema escolhesse aquela com maior importância ou aquela cujas ações fossem mais importantes. Infelizmente, este critério tem sido muito criticado porque a importância de regras e de ações pode variar, de acordo com as circunstâncias.

Pode defender-se que as estratégias de resolução de conflitos consubstanciam uma forma de usar conhecimento que não está explicitamente representado no sistema. Por exemplo, se duas regras tiverem a condição satisfeita, há conhecimento que escolhe uma ou a outra. Mas esse conhecimento não está representado explicitamente no sistema. É conhecimento implícito. E a utilização de conhecimento implícito, em sistemas baseados em conhecimento, pode ser mal considerada porque o comportamento do sistema não depende apenas do conhecimento representado explicitamente. É como se houvesse a lei (escrita explicitamente) e depois houvesse uma lei invisível sobre a forma como a lei explícita pode ser aplicada.

Esta razão e também por ser muito mais simples levou-nos a optar por não usar nenhuma estratégia de resolução de conflitos: sempre que duas regras podem ser usadas, o sistema escolhe uma. O engenheiro do conhecimento não deve saber como é que o sistema escolhe. Deve preparar as regras do seu sistema para situações deste tipo.

A decisão de não recorrer a estratégias de resolução de conflitos tem porém consequências. As regras têm de estar equipadas com condições e com ações cujo único propósito é controlar a sua própria aplicabilidade. Por exemplo, se uma regra não dever ser aplicada repetidamente com os mesmos dados, ela deve ser desenhada de tal maneira que, depois de ser aplicada a primeira vez com determinados dados,

não voltará a ser usada com os mesmos dados. Em geral, é necessário que a condição da regra dependa de um dado facto de controlo e que a ação da regra altere ou remova esse facto de controlo para que a regra deixe de poder ser usada de novo. Consequentemente, as regras de produção, em sistemas sem estratégias de resolução de conflitos, têm conhecimento sobre o domínio do problema e conhecimento de controlo da própria regra.

3 Implementação das regras do herói e do vilão

Na secção anterior, analisaram-se as regras de produção necessárias para controlar o herói, mas apenas do ponto de vista conceptual; sem os detalhes da implementação.

O propósito desta secção é a implementação das regras, recorrendo à ferramenta computacional *PSys*, a qual facilita a definição de sistemas de regras de produção.

O *PSys* é essencialmente um motor de inferência, escrito em Prolog, que processa regras de produção escritas com uma determinada sintaxe. Para executar um sistema de regras de produção, recorrendo ao *PSys*, basta executar o procedimento Prolog `psys/0`, cuja implementação se encontra no ficheiro `PSys.pl`.

Sendo um procedimento em Prolog, o `psys/0` pode ser integrado facilmente em qualquer outro programa em Prolog, o que permite estender as suas capacidades com naturalidade. Além disso, tendo acesso ao código Prolog, também não é difícil alterar a definição do `psys/0`.

Mas talvez a vantagem mais significativa de termos o *PSys* implementado em Prolog, é que podemos usar, quer na condição das regras, quer na ação, qualquer recurso da linguagem de implementação e qualquer recurso criado em Prolog pelo utilizador do *PSys*.

O *PSys* está disponível na secção Sistemas do sítio web da cadeira (http://iscte.pt/~luis/aulas/ia). Tem de se fazer o *download* do PSys.zip, e guardar o PSys.pl numa diretoria à escolha.

Como o *PSys* foi implementado em Prolog, a forma mais direta e fácil de o usar é recorrer a um interpretador de Prolog, por exemplo o SWI Prolog.

Para usar o *PSys* são necessários os seguintes 4 passos gerais:

1. Usando o SWI-Prolog, criar um novo ficheiro .pl para o *sistema de produção*
2. Importar o *PSys* para que se possam especificar e usar regras de produção. Para isso, escrever uma linha como a seguinte, no início do ficheiro, especificando o *pathname* completo do PSys
3. Depois de importar o *PSys*, podem escrever-se os seguintes componentes do sistema:
   a. As regras de produção do sistema
   b. As ações a usar nas regras. Também pode acontecer que não seja necessário definir ações específicas, se for considerado que as ações da própria linguagem Prolog são adequadas.
   c. Predicados adicionais que tenham eventualmente de ser usados
   d. Programa de interface que chama o *PSys*. Nos casos mais simples, não é necessário definir a interface; bastará chamar o procedimento `psys/0`.
4. Usar o sistema definido, chamando o programa de interface

Assumiremos que a criação de um novo ficheiro no interpretador Prolog não carece de explicação. Passaremos de imediato para a importação do PSys.

No início do ficheiro em que pretendemos escrever as regras do Sistema, é necessário importer o ficheiro PSys.pl, o que pode ser feito usando a diretiva *ensure_loaded/1*:

```
:- ensure_loaded('C:/Systems/Psys/PSys.pl').
```

O operador `:-` no início da linha é fundamental. Não estamos a definer um predicado em Prolog, estamos a dizer ao interpretador para executar uma diretiva (carregar um ficheiro, neste caso). Também é importante reter que as barras de separação de diretorias no *pathname* devem ser as que são usadas no Linux (e não as que são usadas no Windows), mesmo que estejamos a usar um PC com Windows.

Finalmente, o pathname especificado é totalmente inventado. Em cada computador, o *PSys* estará guardado numa diretoria diferente. É necessário dar o pathname complete do PSys.pl, no computador onde o estivermos a usar.

A parte mais substancial da utilização do *PSys* é a definição de regras e das outras definições potencialmente necessárias. Começamos pelas regras. Para cada regra definida concetualmente na secção 1, escreveremos a versão computacional, recorrendo ao *PSys* e à sua sintaxe.

Todas as regras apresentadas na secção 1 referem o herói, o vilão e as suas posições. As posições são pares de coordenadas (x, y) num referencial com a origem no canto superior esquerdo, em que x se mede no eixo das abcissas, o qual cresce para o lado direito, e o y diz respeito ao eixo das ordenadas, o qual cresce para baixo, como na Figura 1 e na Figura 3.

O nome do herói será especificado por um facto do predicado *hero/1*, por exemplo `hero(batman)`. Este facto poderá ser criado pelo programa de interface ou de qualquer outra forma válida no Prolog. O nome do vilão será especificado num facto do predicado *villain/1*, por exemplo `villain(joker)`, o qual poderá ser criado, no início, pelo programa de interface.

As posições dos dois personagens serão mantidas por dois factos do predicado pos/2, por exemplo `pos(batman, (3, 2))` e `pos(joker, (5,5))`. As posições iniciais dos personagens poderão ser criadas e armazenadas inicialmente pelo programa de interface.

Para podermos escrever as regras temos de saber os nomes dos procedimentos que implementam as ações referidas nas regras. Temos uma ação em que o herói mata o vilão, e temos quarto ações em que o herói dá um passo para a esquerda, para a direita, para cima e para baixo. Para já, suporemos que estas ações estão implementadas pelos procedimentos prolog *kill/2*, *step_left/1*, *step_right/1*, *step_up/1* e *step_down/1*, por exemplo `kill(batman, joker)` e `step_left(batman)`.

Agora podemos escrever cada uma das 5 regras analisadas na secção 1. Começamos pela primeira regra especificada na secção 1:

SE a posição atual for a mesma que a posição do vilão
ENTÃO Liquidar o vilão.

```
if(hero(H) and pos(H, Pos) and villain(V) and pos(V, Pos))
then kill(H, V).
```

As regras PSys são estruturas if/then com letra minúscula. Depois do if, vem a condição da regra, a qual pode envolver conjunções (and), disjunções (or), e negações (\+). Cada regra tem de ser terminada por um ponto final.

Na condição da regra anterior, hero(H) serve para instanciar a variável H com o nome do herói, recorrendo ao facto do predicado *hero/1* que mantém o nome do herói. Passa-se o mesmo com o vilão e a variável V. As duas condições pos(H, Pos) e pos(V, Pos) instanciarão a variável Pos com a posição do herói e do vilão, o que é possível apenas se as duas posições forem a mesma (porque foi usada a mesma variável para ambas). Portanto a condição desta regra, escrita em PSys, significa *"Se o herói e o vilão estiverem na mesma posição"*.

Nesta condição, o herói mata o vilão: kill(H, V).

Seguidamente, surgem as regras que controlam os passos do herói.

SE o número da coluna da posição atual for superior ao número da coluna da posição do vilão
ENTÃO dar um passo para a esquerda

```
if(hero(H) and pos(H, (Xh, Yh)) and
    villain(V) and pos(V, (Xv, Yv)) and
    Xh > Xv)
then step_left(H).
```

Na tecnologia baseada em Prolog, como é o caso do *PSys*, é muito fácil e natural especificar e processar estruturas de dados compostas. Por exemplo, as estruturas (Xh, Yh) e (Xv, Yv) representam dois pares de coordenadas. Como nesta regra foi necessário usar uma das coordenadas de cada posição, foi conveniente especificar as posições por estruturas (X, Y). Na regra anterior, em que não havia necessidade de separar o número da coluna (X) do número da linha (Y), usamos a variável Pos, a qual quando instanciada tomará o valor de uma estrutura (X, Y).

A segunda regra pode ser escrita com uma ligeira alteração. Como a regra apenas necessita saber o número da coluna da posição de cada personagem (Xh > Xv), o número da linha não é relevante. Quando uma variável não tem interesse, ela pode ser substituída pelo *underline* (_). Desta forma informa-se quem lê que aquela grandeza não é usada:

```
if (hero(H) and pos(H, (Xh, _)) and
    villain(V) and pos(V, (Xv, _)) and
    Xh > Xv)
then step_left(H).
```

O facto de surgirem dois *underlines* na mesma regra não implica que as grandezas substituídas tenham o mesmo valor. Significa apenas que têm um valor qualquer que não é usado.

A representação das outras três regras em PSys é muito semelhante:

SE o número da coluna da posição atual for inferior ao número da coluna da posição do vilão
ENTÃO dar um passo para a direita

```
if (hero(H) and pos(H, (Xh, _)) and
    villain(V) and pos(V, (Xv, _)) and
    Xh < Xv)
then step_right(H).
```

SE o número da linha da posição atual for inferior ao número da linha da posição do vilão
ENTÃO dar um passo para baixo

```
if (hero(H) and pos(H, (_, Yh)) and
    villain(V) and pos(V, (_, Yv)) and
    Yh < Yv)
then step_down(H).
```

Desta vez, na regra acima, são as colunas que não são usadas.

SE o número da linha da posição atual for superior ao número da linha da posição do vilão
ENTÃO Dar um passo para cima

```
if (hero(H) and pos(H, (_, Yh)) and
    villain(V) and pos(V, (_, Yv)) and
    Yh > Yv)
then step_up(H).
```

Para que as regras acabadas de escrever possam ser usadas, é necessário definir as ações que elas comandam, isto é, as ações `kill(H, V)`, `step_left(H)`, `step_right(H)`, `step_down(H)` e `step_up(H)`.

As ações das regras terão de comandar os sistemas para que as regras foram feitas. Se o herói e o vilão fossem *robots*, as ações teriam de comandar os seus controladores, enviando-lhes os comandos que eles reconheceriam, através dos *device drivers* disponibilizados pelos *robots*. Neste caso, temos personagens de *software*. Se esses personagens de *software* fossem programas independentes do sistema de regras de produção, teriam uma API (*Application Programming Interface*) através da qual poderiam ser comandadas. Nesse caso, as ações a definir no *PSys*, teriam de usar os métodos disponibilizados na API (no caso de se tratar de uma API orientada por objetos).

No mundo do herói e do vilão que estamos a considerar, as ações dos personagens são totalmente implementadas no próprio *PSys*. A ação em que um personagem mata outro apenas tem de criar uma alteração, no estado do sistema, que possa ser interpretada por ele e pelos utilizadores como se da morte da vítima se tratasse. No nosso caso, será escrita uma mensagem no ecrã do computador e serão removidos os factos temporários usados, fazendo com que o sistema de regras termine a sua execução porque deixará de haver regras cuja condição esteja satisfeita.

```
kill(X, Y):-
   retract(hero(_)), retract(villain(_)),
   retractall(pos(_, _)),
   writelist([X, ' kills ', Y]), nl.
```

Os passos para a direita, para a esquerda, para baixo e para cima escreverão uma mensagem no ecrã do computador, removerão os factos correspondentes às posições dos personagens em causa, calcularão as

novas posições desses personagens e criarão os factos com as novas posições para que o sistema saiba que as personagens que se moveram ocupam novas posições.

```
step_left(A):-
  retract(pos(A, (X, Y))),
  X1 is X - 1,
  assert(pos(A, (X1, Y))),
  writelist([A, ' moves left.']), nl.

step_right(A):-
  retract(pos(A, (X, Y))),
  X1 is X + 1,
  assert(pos(A, (X1, Y))),
  writelist([A, ' moves right.']), nl.

step_down(A):-
  retract(pos(A, (X, Y))),
  Y1 is Y + 1,
  assert(pos(A, (X, Y1))),
  writelist([A, ' moves down.']), nl.

step_up(A):-
  retract(pos(A, (X, Y))),
  Y1 is Y - 1,
  assert(pos(A, (X, Y1))),
  writelist([A, ' moves up.']), nl.
```

Temos agora que implementar o programa de interface para que o utilizador possa usar o sistema de produção, no mundo do herói e do vilão.

Basta um programa que cria os factos *hero/1*, *villain/1*, e *pos/2*, relativos aos nomes dos dois personagens e às suas posições iniciais. Podemos chamar hero_and_villain ao programa de interface. O programa, deve receber quatro argumentos: o nome do herói e a sua posição inicial, e o nome do vilão e a sua posição inicial. Depois de criar os factos iniciais o programa de interface – *hero_and_villain/4* – mandará executar o sistema de regras de produção:

```
villain_and_hero(Hero, PosH, Villain, PosV):-
   assert(hero(Hero)),
   assert(villain(Villain)),
   assert(pos(Hero, PosH)),
   assert(pos(Villain, PosV)),
   psys.
```

O procedimento `psys` aplica o motor de inferência às regras e ações definidas e aos factos iniciais, o que resulta na sucessiva aplicação de regras até que nenhuma delas possa ser aplicada. Nessa altura, a execução do `psys` termina.

Finalmente, o último passo consiste em executar o programa *hero_and_villain/4*, no interpretador Prolog (por exemplo, no SWI-Prolog). No exemplo que se segue, o herói será o Batman, inicialmente posicionado na posição (3, 2), e o vilão será o Joker, inicialmente posicionado em (5, 5) (Figura 4).

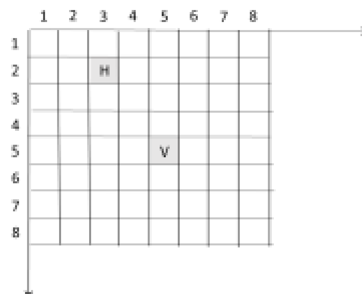

Figura 4 – Posições iniciais do Batman e do Joker

Nesta configuração, o Batman dará três passos para baixo e dois para a direita. A ordem pela qual o Batman dará os seus passos em direção ao Joker não é determinada. Depois de alcançar o Joker, o Batman liquidá-lo-á.

```
?- hero_and_villain(batman, (3, 2), joker, (5, 5)).
batman moves right.
batman moves right.
batman moves down.
batman moves down.
batman moves down.
batman killed joker
true.

?-
```

O que vemos na interação é que o Batman começa por dar dois passos para a direita e só depois se move para baixo. Se, em vez deste mundo representado de forma rudimentar, tivéssemos um mundo e personagens 3D, em vez das mensagens que vemos surgir no ecrã (para informar o utilizador daquilo que se está a passar), veríamos os movimentos do Batman e a sua imagem a três dimensões. Mas para isso, teria de haver um programa de animação 3D. Nessa realidade, as ações definidas teriam de dar ordens ao simulador para que a animação surgisse aos nossos olhos.

4 Implementação em Prolog de uma Ferramenta para criação de Sistemas de Produção

[A implementação não faz parte da matéria atual de Inteligência Artificial]

A presente secção centra-se na implementação de um sistema de representação de conhecimento e de raciocínio com regras de produção.

O motor de inferência é essencialmente um ciclo. Em cada volta do ciclo, são verificadas as condições de todas as regras da base de conhecimento. As regras com a condição satisfeita são agrupadas numa estrutura chamada conjunto de conflito ("conflict set"). As ações das regras do conjunto de conflito não podem ser todas executadas porque a execução das ações de uma das regras poderá invalidar as condições de outras regras do conjunto de conflito. Consequentemente escolhe-se apenas uma das regras do conjunto de conflito e executa-se a sua ação. Todo o processo se repete até que o conjunto de conflito é o conjunto vazio.

```
psys :-
   repeat,
      conflict_set(Set),
      process_conflict_set(Set), !.

process_conflict_set([]):- !.
process_conflict_set(Set):-
   select_rule(Set, rule(_,Action)),
   perform_action(Action),
   !,
   fail.

conflict_set(Set):-
   findall(rule(Cond, Act), (rule(Cond, Act), satisfied(Cond)), Set).

% simple minded conflict resolution strategy
select_rule([R|_], R).

% simple, but maybe sufficient, condition evaluator
satisfied(Cond):- call(Cond).

perform_action(A):-
   trace_level(0),
   !,
   call(A).
perform_action (A):-
   action_notice(A, Msg),
   write(Msg), nl,
   call(A).
```

Figura 5 – Ciclo principal do raciocínio num sistema de produção

O predicado *conflict_set/1* cria o conjunto de regras da base de conhecimentos cuja condição está satisfeita (i.e., o conjunto de conflito), e o predicado *process_conflict_set/1* processa o conjunto de conflito produzido. Se este for o conjunto vazio, o processamento termina. Caso contrário, é selecionada uma regra do conjunto de conflito e a sua ação é executada resultando numa possível alteração da memória de trabalho.

A seleção de uma regra do conjunto de conflito, da responsabilidade do predicado *select_rule/2*, é arbitrária no sentido em que a decisão da sua escolha não deve estar descrita na documentação do sistema. Poderia ser escolhida uma regra à sorte, poderia ser escolhida a primeira ou a última das regras do conjunto de conflito. A escolha é da responsabilidade do sistema mas o Engenheiro do Conhecimento, o utilizador desta ferramenta, não pode contar com um determinado critério de escolha.

O predicado *satisfied/1* serve para determinar se uma condição está satisfeita. Se o formato das condições for igual ao usado na linguagem Prolog e se os factos forem factos em Prolog, *satisfied/1* é o mesmo que *call/1*. Se o formato das condições ou a forma como os factos são armazenados for diferente do usado no Prolog, *satisfied/1* terá de ser alterado.

O predicado *perform_action/1* depende igualmente do formato com que as sequências de ações são especificadas. Se esse formato for o mesmo do formato de uma sequência de objetivos Prolog, *perform_action/1* será o mesmo que *call/1*. Na definição da Figura 5, *perform_action/1* tem uma função adicional. Se o nível de notificação pretendido (*trace-level*) for 0, a execução das ações é muda. Se o nível de notificação pretendido for 1, a execução das ações é acompanhada da impressão de mensagens. O nível de notificação está definido no predicado *trace_level/1*. As mensagens estão definidas na tabela *action_notice/2*.

Além de se poder ligar e desligar a exibição de mensagens antes da execução das ações, este mecanismo de notificação tem a vantagem de permitir definir novas ações com base em ações já existentes sem herdar obrigatoriamente a impressão de mensagens de notificação. Se pretendermos herdar a notificação, em vez de se usar diretamente uma ação, usa-se o procedimento *perform_action/1*.

Além do ciclo principal, um sistema de regras de produção tem que ter as definições das ações primitivas que podem ser usadas na parte direita das regras e que podem ser usadas também na definição das ações do domínio da aplicação. Como a implementação é em Prolog, todas as ações do Prolog podem ser usadas nas regras de um sistema (e.g., write, read, assert, retract).

As mensagens a serem exibidas antes da execução de cada ação devem ser definidas pelo predicado *action_notice/2* com se exemplifica na Figura 6.

```
action_notice(conclude(P), Msg):-
  term_to_atom(P, AP),
   atomic_list_concat([AP, ' was concluded.'], Msg).

action_notice(remove(P), Msg):-
  term_to_atom(P, AP),
   atomic_list_concat([AP, ' is no longer true.'], Msg).

action_notice(create_goal(P), Msg):-
  term_to_atom(goal(P), AP),
   atomic_list_concat([AP, ' was created.'], Msg).

action_notice(remove_goal(P), Msg):-
  term_to_atom(goal(P), AP),
   atomic_list_concat([AP, ' was removed.'], Msg).

action_notice(retract, '').
```

Figura 6 – Mensagens a imprimir para cada ação

term_to_atom/2 já existe no Prolog. Converte qualquer termo (simples ou composto) num átomo para que possa ser concatenado com outros átomos num único átomo. *atomic_list_concat/2* também já existe no Prolog. Produz um único átomo que resulta da concatenação dos elementos atómicos de uma lista. *action_notice/2* recebe uma ação e produz um átomo com a mensagem a imprimir.

Para utilizar o mecanismo de inferência analisado, primeiro cria-se a base de conhecimento, incluindo as ações do domínio e as mensagens a serem imprimidas antes da execução de cada ação. Depois usa-se o *assert/1* do Prolog para inicializar a memória de trabalho. Finalmente, chama-se o predicado *psys/0* para efetuar a inferência.