Sistemas baseados em Agentes
Textos de apoio

António Silva

2013-01-30

Conteúdo

Parte I
Agentes Inteligentes segundo o Chimera

No ambiente de desenvolvimento de Agentes Inteligentes Chimera, uma extensão do LPA Win-Prolog, um agente é funcionalmente composto por:

- **Código**, que define a sua função e comportamento, construído em LPA-Prolog, conferindo-lhe a inteligência necessária,
- **Comunicações**, para poder comunicar com outros agentes,
- **Eventos** aos quais é capaz de reagir, usando uma *Event Queue* assíncrona e *Handlers* apropriados.

1 Comunicações

No momento em que um agente é criado, arranca automaticamente um servidor de comunicações que funciona em background, monitorizando a sua actividade. Se outro agente tentar efectuar uma ligação, o servidor detectará esse pedido e servi-lo-á de forma transparente, sem necessitar de qualquer programação adicional ou de interacção com o utilizador.

Quando, por outro lado, se tornar necessário ligar a outro agente, a única informação requerida para efectuar essa operação é o **Domain Name** (ou o endereço IP) em que o agente destino se encontra a correr bem como a indicação de qual a **Porta** a que se encontra ligado.

Uma vez a ligação estabelecida, tudo o que é preciso para enviar informação entre agentes é construir um termo Prolog e chamar um predicado (*agent_post*) que envie esse termo ao agente destinatário.

2 Eventos

Num programa controlado por eventos, a sua execução é interrompida automaticamente sempre que um evento ocorre. Esse evento deverá ser tratado adequadamente, após o que o programa recomeça onde tinha sido interrompido.

O tratamento das comunicações no Chimera é também feita recorrendo a eventos. Quando uma mensagem contendo um termo chega a um agente, ocorre um evento. Seja o que for que o agente esteja a realizar nesse momento, o controle é passado para um *"event handler"* específico, definido pelo utilizador e que especifica o que fazer nessa eventualidade.

Efectivamente, em Chimera, enviar um termo a outro agente é o mesmo que agendar um evento nesse outro agente. Esse processo é assim designado por *"event posting"* traduzindo a ideia de que um evento é enviado ao agente remoto.

É possível encapsular qualquer estrutura de dados (números, variáveis, listas,...) nesses termos Prolog utilizados para envio de mensagens entre agentes. O termo recebido no destino será idêntico ao enviado, com a única excepção de uma variável contida no termo enviado ser duplicada à chegada, não mantendo assim qualquer relação com a enviada.

Os eventos podem ser definidos pelo utilizador, com nome e estrutura de dados totalmente especificada por ele, ou serem definidos pelo próprio sistema. Os eventos gerados automaticamente pelo sistema ocorrem sempre que ligações sejam estabelecidas ou interrompidas e sempre que mensagens sejam enviadas ou recebidas. A sua sintaxe está descrita na Secção "Os eventos de sistema". Os eventos definidos pelo utilizador são gerados mediante a invocação de predicados *agent_post/3*. Os dois tipos de eventos podem ser distingidos pela sua estrutura de dados, uma conjunção no caso dos eventos de sistema e um tuplo nos eventos de utilizador.

O Chimera oferece ainda uma possibilidade adicional interessante: é possível agendar um evento no próprio agente.

3 Construção de um agente Chimera

Numa óptica mais prática que a utilizada no princípio do texto, um agente Chimera é visto como compreendendo dois componentes:

1. Um **Socket** suportando as comunicações TCP/IP
2. Um **Handler** constituído por um predicado Prolog com 3 parâmetros e que será chamado automaticamente sempre que um dado evento ocorra. Para cada tipo de evento deverá existir um handler específico, caso se pretenda garantir uma resposta própria a esse evento.

Para criar um agente é necessário especificar 3 elementos: o seu nome, o nome (*functor*) dos predicados Prolog que permitem fazer o tratamento dos seus eventos de sistema (*handlers*) e, opcionalmente, a porta TCP/IP a que vai estar ligado.

O predicado utilizado para o efeito é o *agent_create/3*:

agent_create(Agent_name, Handler, Port)

A porta pode ser especificada pelo programador mas, caso não o seja, ser-lhe-á atribuído automaticamente um valor pelo sistema. Se for especificada uma Porta que esteja já a ser usada pelo sistema, ocorrerá um erro.

Por último, para poder utilizar o Chimera e as suas facilidades é necessário assegurar o carregamento da biblioteca respectiva usando o predicado:

ensure_loaded(system(chimera))

4 Comunicações

As comunicações entre dois agentes exigem o estabelecimento prévio de uma conexão mediante a chamada do predicado *agent_create/4* (não confundir com o predicado *agent_create/3* que é utilizado para criar o próprio agente):

agent_create(Agent_name, Link, IP_address, Port)

Neste predicado, **Agent_name** é o agente em que o predicado é executado, ou seja, o agente iniciador do pedido de ligação, **Link** é um número inteiro atribuído pelo sistema e destina-se a identificar essa ligação específica (o agente pode ter várias ligações estabelecidas simultaneamente), o **IP_address** e **Port** constituem o endereço do agente remoto com quem se pretende estabelecer a ligação.

Após o estabelecimento de uma conexão, o envio de uma mensagem resume-se a provocar a geração de um evento no agente destino usando para o efeito o predicado *agent_post/3*:

agent_post(Agent_name, Link, Term)

A invocação em **Agent_name** de um predicado com esta sintaxe implica o agendamento de um evento com uma estrutura de dados contendo **Term** no agente remoto ligado através da conexão identificada por **Link**. O conteúdo de **Term** constituirá a mensagem enviada ao agente remoto.

De notar que um agente pode agendar um evento em si próprio. Nesse caso, o segundo parâmetro do predicado **agent_post/3** deverá ser uma lista vazia ([ ]).

Uma ligação pode ser fechada mediante o predicado **agent_close/2**. O predicado **agent_close/1** é utilizado para fechar o agente propriamente dito, bem como qualquer ligação que ele tenha estabelecido.

5 O tratamento dos eventos

Pode-se dizer que num ambiente de programação como o Chimera, uma aplicação é basicamente composta por *"user events"* e os seus *"handlers"*.

O tratamento dos eventos é efectuado através de predicados designados por *"event handlers"* recebendo três parâmetros: o nome do agente que o agenda, o identificador da conexão que o liga ao agente remoto onde o evento será gerado e o termo contendo a estrutura de dados associada ao evento.

Os Event Handlers são constituídos por predicados Prolog criados pelo utilizador com a seguinte estrutura:

handler_name(Agent_name, Link, Event)

Event descreve o evento ao qual o handler diz respeito, e **Link** refere-se a um número identificando a ligação através da qual a comunicação entre agentes foi estabelecida.

De notar que o **handler_name** deverá ser o mesmo que foi declarado no 2º parâmetro do predicado *agent_create/3* que criou o agente.

6 Os eventos de sistema

Os eventos de sistema são gerados de forma automática pelo sistema sempre que alguns acontecimentos especiais ocorram. Todos eles têm associada uma estrutura de dados sob a forma de conjunções.

A lista de eventos de sistema inclui os seguintes:

(close, Port)

É gerado quando o agente é terminado mediante a invocação do predicado *agent_ close/ 1.*

(close, Host, Port)

É gerado quando uma ligação entre agentes é terminada mediante a invocação do predicado *agent_ close/2.*

(create, Port)

É gerado quando um agente é criado mediante a invocação do predicado *agent_ create/ 1.*

(create, Host, Port)

É gerado quando uma ligação entre agentes é requerida mediante a invocação do predicado *agent_ create/2.* Não traduz a efectiva concretização da ligação, apenas o pedido de estabelecimento, independentemente do seu sucesso ou insucesso. É, pois, aconselhável assegurar-se, mediante um mecanismo de *handshake*, que a ligação se concretizou.

(error, Code, What)

Traduz a ocorrência de um problema numa dada ligação entre dois agentes. O parâmetro **What** virá instanciado com um átomo Prolog que indica qual o problema específico ocorrido.

(open, Host, Port)

É gerado quando um pedido de ligação é aceite no agente remoto.

(read, Host, Port, Term)

É gerado quando uma mensagem é recebida, ou seja, quando um evento pedido por um agente origem é gerado no agente destino.

(write, Host, port, Term)

É gerado quando uma mensagem é enviada.

Mais informação sobre estes eventos pode ser encontrada no Apêndice B do texto contido no ficheiro CHI_REF.pdf.

7 Exemplo de demonstração

Nas páginas 35 a 46 do documento "Agents for Win-Prolog - Programming Guide and Technical Reference" (ficheiro pdf contido no sub-directório Docs do directório de instalação do Win-Prolog) é descrita a aplicação "Brains" elaborada com o intuito de servir de demonstração das técnicas descritas atrás [1].

[1] Está no sub-directório Examples do directório de instalação do Win-Prolog

Parte II
Notas sobre a componente visual do WINProlog

Estas notas procuram resumir e sistematizar os conceitos e técnicas de criação de interfaces descritos nos capítulos 3 a 6 do documento 'LPA WinProlog Win32 Programming Guide' (ficheiro pdf win_w32).

Em WinProlog todos os objectos gráficos são considerados janelas.

8 Criação de Janelas

As janelas são criadas usando o predicado *wcreate/8*, com a seguinte sintaxe:

wcreate(Nome, Tipo, Titulo, X, Y, L, W, Estilo)

Por exemplo:

wcreate(win1, text, `Janela`**, 200, 125, 400, 200, 0)**

O 1º parâmetro será o identificador da janela. O 3º parâmetro diz respeito ao texto a afixar no título. O 2º especifica o tipo de janela, neste caso de texto. Os 4º e 5º argumentos dão a posição da janela e os 6º e 7º as suas dimensões.

O último parâmetro (**Estilo**) não tem efeito neste caso, assumindo assim o valor zero, já as *text windows* têm um estilo fixo, não parametrizável.

A partir do momento da criação da janela mediante a invocação de *wcreate/8*, todas as operações sobre esta janela serão referidas a ela, através do seu identificador.

Exemplo:

wsize(win1, Left, Top, Width, Depth)

Este predicado altera ou verifica a posição e tamanho duma janela.

Uma janela ou controlo (em si também uma janela) dentro de outra janela são identificados por meio de uma conjunção da forma **(Nome, Id)**, por exemplo:

wsize((win1,2), Left, Top, Width, Depth)

Uma janela deve ser fechada usando o predicado *wclose(Nome)*.

9 Diálogos standard

Exemplo:

msgbox(Titulo, Mensagem, Estilo, Codigo)

O **Estilo** é um valor numérico que permite especificar as características visuais e de comportamento do diálogo. Mais à frente, na Secção 'Estilos das janelas', este assunto será tratado com mais detalhe. O **Código** é o valor numérico retornado pelo diálogo, consoante o botão que tenha sido premido pelo utilizador.

Existem outros diálogos standard disponíveis como *aboutbox*, *open*, *save* e *print*.

10 User windows

Estas janelas são criadas com o predicado

wcreate(Nome, user, Titulo, X, Y, L, W, 0)

Para colocar controlos (um botão, por exemplo) na janela:

wcreate((Nome, Id), button, Texto, X,Y,W,D,Estilo)

O parâmetro **Estilo** como veremos à frente permite definir características e comportamentos do controlo.

11 Dialog windows

As diferenças entre janelas dos tipos *dialog* e *user* podem ser resumidas da seguinte maneira:

- A *dialog* é independente do ambiente do WinProlog, existe no top level desktop, enquanto a *user* se encontra integrada na janela
- a posição da janela é absoluta e não relativa à janela principal
- o parâmetro de estilo tem significado, ou seja o seu estilo pode ser especificado
- permite a tab navigation

Estes diálogos são criados mediante

wcreate(Nome, dialog, Titulo, X, Y, W, D, Estilo)

Um exemplo de valor para o Estilo será 16'80c80000 (invisível). Mudando o primeiro dígito 8 para 9 tornará a janela visível. Note-se que estes são os estados iniciais definidos na criação da janela mas o estado visível/invisível da janela pode ser mudado programaticamente em qualquer altura mediante o predicado wshow/2:

wshow(Nome, P)

Os valores do parâmetro **P** determinam os seguintes comportamentos:

P	**resultado**
0	invisível
1	visível
2	minimizado
3	maximizado

Assim como com uma *user window*, é também possível povoar um diálogo com controlos, usando o comando *wcreate/8*.

12 Control windows

Em WinProlog estão disponíveis os controlos que seria de esperar encontrar. As *Control classes* disponíveis incluem *button*, *edit*, *static*, *listbox* e *combobox*, entre outras. Abaixo serão referidas as principais, juntamente com predicados específicos para a sua manipulação.

Os objectos gráficos (controlos) são criados mediante o predicado *wcreate/8*.

wcreate((Nome, Id), Tipo, 'Texto', X,Y,W,D,Estilo)

O parâmetro **Estilo** pode conter um valor como **16'50010000 (hex)** que, entre outras características, permite especificar que a janela é *child/visible* (**5**) e qual o tipo de botão (o último **0**). No caso de um botão, poderíamos ter assim:

wcreate((Nome, Id), button, 'Start', X,Y,W,D,16'50010000)

12.1 Button

Os dois predicados com mais interesse na manipulação dos botões de comando são o *wenable/2* e o *wbtnsel/2*:

wenable((Name, Id),P)

Este predicado, que é aliás comum a outros controlos, permite inibir ou autorizar a interacção do utilizador da interface com o controlo. O valor do parâmetro **P** (0 ou 1) decide se o controlo está inibido ou não.

wbtnsel(((Name, Id),P)

Este predicado permite seleccionar ou desseleccionar programaticamente o botão de comando, dependendo de qual o valor (1 ou 0) do parâmetro **P**.

12.2 Edit

A classe *Edit* corresponde às *TextBox* de outras linguagens visuais. Pode ser controlada usando o predicado *wtext/2*:

wtext((Name, Id),Text)

Se o parâmetro **Text** estiver instanciado com um valor, o conteúdo do controlo passará a ser esse valor. Em caso contrário, a variável **Text** será instanciada com o conteúdo do controlo.

Note-se que este predicado pode ser aplicado a outras classes de controlos, nomeadamente à classe *button*, em que permite modificar programaticamente o texto do botão e à classe *combobox*.

12.3 Listbox

Os predicados utilizados para manipular esta classe de controlos são os seguintes:

wlstadd((Name, Id), Pos, Text, Item)

Este é o predicado que permite povoar a *listbox*, adicionando-lhe linhas com o conteúdo **Text**. **Pos** especifica qual a posição em que a nova linha será inserida. A numeração das linhas nas *listbox* começa no valor 0. O valor –1 corresponde à posição de defeito, que será a próxima linha livre ou, no caso de se ter imposto a ordenação automática da *listbox*, a linha adequada para a ordenação ser mantida. A ordenação automática pode ser especificada através do estilo usado na criação do controlo. O parâmetro **Item** é opcional e permite associar um valor numérico ao conteúdo da linha.

Uma linha pode posteriormente ser eliminada mediante o predicado *wlbxdel/2*.

Para controlar a selecção de linhas numa listbox são usados os predicados *wlstsel/2* e *wlstsel/3*. O primeiro permite seleccionar ou desseleccionar uma linha de uma *listbox* em que apenas uma linha seleccionada seja permitida [2]:

wlstsel((Name, Id), Pos)

O seu comportamento é o seguinte: se **Pos** for uma variável não instanciada, será devolvida a posição da linha actualmente seleccionada, ou o valor –1 caso nenhuma o esteja. Se **Pos** estiver instanciada com um valor numérico, a linha que lhe corresponde será seleccionada. Se esse valor numérico for –1, qualquer linha que esteja seleccionada no momento, deixará de o estar. O segundo predicado pode ser utilizado de forma mais genérica:

wlstsel((Name, Id), N, Status)

Este predicado permite especificar ou conhecer o estado de selecção (**Status**) de qualquer linha da listbox, dado o seu identificador numérico (**N**). Assim, é possível seleccionar uma linha específica (**N**) da *listbox* ou, ao contrário, desseleccioná-la, dependendo esse comportamento, de estar ou não instanciada a variável **Status**.

wlstfind((Name, Id), Início, Text, Pos)

Este predicado permite efectuar pesquisa de informação dentro da lisbox. O parâmetro **Início** especifica qual a posição de início da busca (mais uma vez –1 é o valor de defeito, ou seja, neste caso a primeira linha, ou então a linha de ordem **N**). **Text** deverá conter o texto a pesquisar e, caos este seja encontrado, a sua posição (nº de linha) será devolvida através da variável **Pos**.

Com o predicado *wlstget/3* é possível obter o conteúdo de uma determinada linha de uma *listbox*:

wlstget((Name, Id), Pos, String, Item)

em que **String** irá receber o contúdo da linha que se encontra na posição **Pos**.

12.4 Combobox

Esta classe de controlos funciona de forma semelhante à da *Listbox*, com a óbvia adição de um campo *Edit*. Para controlar esse campo, é usado o predicado *wtext/2* já referido atrás.

[2] Note-se que este é um comportamento que, caso pretendido, deve ser especificado mediante um estilo apropriado (ver Secção 'Estilos de janelas').

12.5 Static

Esta classe de objectos corresponde ao que noutras linguagens visuais se designa por *label*. Destina-se a apresentar informação ao utilizador da interface. Pode assumir a forma de um *label* de texto ou um ícone.

Exemplos:

label - **wcreate((win1,1), static, 'Texto', X, Y, W, D, 16'50800001)**

ícone - **wcreate((win1,1), static, 'iconfilename', X, Y, W, D, 16'50000003)**

No segundo caso, em vez de apresentar um texto, o controlo apresenta um ícone gráfico, contido num ficheiro passado como 3º parâmetro.

12.6 ScrollBar

A classe *Scrollbar* reúne dois tipos diferentes de barras de deslocamento: indepedentes e associadas a outras janelas. Os objectos desta classe são criados por predicados do tipo:

wrange((Name, Id), Tipo, L1, L2)

Os valores que **Tipo** pode assumir são:

- 0 - autónoma
- 1 - horizontal, integrada numa janela
- 2 - vertical, integrada numa janela

L1 e **L2** referem-se aos limites da gama de valores controlados pela barra de deslocamento.

wthumb((Name, Id), Tipo, T)

Este predicado permite fixar ou obter a posição **T** do cursor na barra de selocamento. O parâmetro **Tipo** pode assumir os mesmos valores usados em *wrange*, e com os mesmos efeitos.

12.7 Grafix

Esta classe de objectos serve de contentor para a apresentação de objectos gráficos.

wcreate((Name, Id), grafix, '', X, Y, W, D , 16'50800000)

Uma vez criado, é possível enviar-lhe primitivas gráficas encapsuladas entre os delimitadores*gfx_ begin* e *gfx_ end* que definem um contexto específico:

```
gfx_begin((Name, Id))
 gfx(ellipse(X,Y,E1,E2)),
gfx_end((Name, Id)).
```

Entre esses dois delimitadores é possível incluir qualquer número de primitivas gráficas. A lista de primitivas gráficas disponíveis (50) está disponível no Apêndice E do ficheiro WIN_REF.

Uma vez desenhados, esses objectos gráficos não são actualizados automaticamente. Assim sendo, é necessário prever handlers adequados que, caso necessário, procedam ao seu redesenho. Na secção 'Windows handlers' este tema é revisitado.

12.8 Menus

Os menus são criados usando o predicado *wmcreate/1*:

wmcreate(Menu)

em que **Menu** é um átomo Prolog.

A partir desse momento, items podem ser adicionados ao menu, mediante o predicado *wmnuadd/4*:

wmnuadd(Menu, Position,'Label', Item)

Este predicado adiciona um **Item** ao **Menu** na posição **Position** com o texto **Label**. Position pode ser –1, caso em o novo item será inserido a seguir aos items pré-existentes ou ser um inteiro positivo indicando o ponto de inserção (as posições em menus começam em zero).

Podem existir três tipos de items num menu: numéricos, separadores e sub-menus. Os primeiros são os mais comuns e correspondem a um valor entre 1000 e 61439 para o parâmetro **Item**. Os separadores correspondem ao valor 0, enquanto para obter um sub-menu **Item** deverá conter um átomo Prolog identificando esse sub-menu.

Existem vários outros predicados para manipular menus. O predicado *wmnudel/2* serve para eliminar elementos do menu, enquanto *wmnuget/4* permite obter os dados relativos a um dado elemento do menu.

O parâmetro **Menu** pode ser um átomo correspondendo a um menu criado pelo predicado *wmcreate* mas pode ainda apresentar os valores 0 e 1. No 1º caso o novo item será acrescentado à barra de menus do próprio WinProlog e no 2º caso ao menu de sistema do WinProlog.

Outra forma, mais útil, de inserir um menu é utilizar o predicado *popupmenu/4* que permite situar um menu num ponto escolhido de uma dada janela e controlar a selecção de um seu elemento.

Mais informação sobre menus e o seu tratamento podem ser encontrados no documento WIN_REF.pdf.

13 Windows Handlers

Para permitir que um dado objecto reaja a um evento específico é necessário instalar *window handlers* que permitam lidar com esse evento. A sua sintaxe é a seguinte:

window_handler(Window, Handler)

O predicado define um functor específico (**Handler**) para os *handlers* associados à janela **Window**. Corresponde a uma declaração dos *handlers* associados àquela janela. A partir do momento em que o functor dos *handlers* está declarado, é possível instalar um *handler* para cada par de objecto/evento, usando predicados do tipo:

handler((Name, Id), Message_type, Data, Result)

Message_type refere-se ao tipo de evento que está a ser tratado, **Data** é um átomo Prolog ou conjunção eventualmente associados a certo tipo de eventos (pouco usado) e **Result** permite, caso necessário, retornar informação usando um átomo Prolog. A cada tipo de evento corresponde uma **Message_type** própria (*msg_button* para o click de um rato, *msg_change* para qualquer alteração numa editbox, por ex.).

A seguir alguns exemplos:

```
windows_handler(win1, win_handler).
win_handler((win1,1), msg_button,_,Text) :- wtext((win1,10), Text).
win_handler((win1,10), msg_change,_,_) :- beep(500, 50).
```

Conforme foi referido relativamente às janelas *grafix* os objectos gráficos nelas desenhados não são persistentes, tendo que ser redesenhados caso por exemplo uma outra janela cubra ainda que temporariamente a janela *grafix*. Há pois necessidade de instalar um tratador desse evento específico, que monitorize a mensagem *msgpaint*.

No exemplo seguinte, *line/5* desenha uma linha num determinado contexto (**Window**) e armazena a acção realizada num facto *grafix*. O handler *grafix_handler* reage a mensagens **msg_paint** repetindo todos os comandos gráficos que tenham sido armazenados em factos *grafix*.

```
line( Window, X0, Y0, X1, Y1 ) :-
    gfx_begin( Window ),
        gfx( polyline(X0,Y0,X1,Y1) ),
    gfx_end( Window ),
    assert( grafix(polyline(X0,Y0,X1,Y1)) ).

grafix_handler( Window, msg_paint, _, _ ) :-
    gfx_begin( Window ),
    forall( grafix( Grafix ), gfx( Grafix )
    ),
    gfx_end( Window ).
```

14 Dialog windows (Mode / Modeless)

A chamada dos diálogos pode ser feita de forma modal ou não modal. A distinção entre estas duas formars pde descrever-se da seguinte maneira:

As janelas de diálogo Modal:

1. representam pedidos de entrada de dados
2. retornam valores através do 4º parâmetro do seu *window handler*
3. devolvem o controlo ao programa que os invocou apenas quando um botão é premido (retornando um valor)

As janelas de diálogo Não Modal:

1. não requerem a introdução de qualquer informação
2. não retornam valores
3. quando são chamados retornam de imediato o controlo ao programa que os invocou, continuando activo e atentos a qualquer evento para o qual possuam um handler apropriado.

Uma invocação Modal efectua-se através da seguinte instrução:
show_dialog(Window)
Uma invocação Não Modal obtem-se desta forma:
call_dialog(Window, Resultado)

15 Fecho de Dialog windows

Os Diálogos não são destruídos após as chamadas a *showdialog/1* ou *calldialog/2*, ficando apenas invisíveis, prontos a ser re-usados. A fim de evitar desperdício de memória, quando os diálogos deixem de ser necessários devem ser fechados mediante *wclose/1*.

16 Estilos das janelas

Cada janela possui um estilo próprio especificado por meio do campo respectivo no predicado *wcreate*. A notação mais compacta para representar o valor desse campo é a hexadecimal que em LPA Prolog usa o seguinte formato:

16'XXXXXXXX

Exemplo:

16'40000000 + 16'10000000 + 16'00040000 = 16'50040000

O valor 16'50040000, resultante da soma dos valores representando os três estilos individuais WS_CHILD, WS_VISIBLE e WS_THICKFRAME, traduz o estilo combinado WS_CHILD + WS_VISIBLE + WS_THICKFRAME, descrevendo uma janela que é 'filha' (dependendo de uma janela 'mãe'), que é visível e possui um bordo grosso.

Para todos os controlos, poderão ser escolhidos os respectivos estilos individuais, que são combinados da mesma forma. Cada classe de objectos possui um conjunto específico de estilos individuais.

Em alternativa a esta forma de especificar estilos, podem ser utilizadas listas Prolog contendo os descriptores dos estilos. O exemplo apresentado acima seria especificado nesta notação da seguinte forma:

[WS_CHILD, WS_VISIBLE, WS_THICKFRAME]

Neste caso, porém, haverá que utilizar o predicado *wccreate/8*. O predicado *wcreate/8* apenas pode ser utilizado com a notação de estilo hexadecimal.

Se bem que as duas notações sejam absolutamente equivalentes, é aconselhável o uso desta última, de forma a aumentar a inteligibilidade do código, facilitando a sua depuração e manutenção.

Parte III
Notas sobre o uso de Timers em WIN Prolog

A realização de temporizadores em LPA-Prolog (versão 4.70) assenta na utilização de dois predicados essenciais, *timer_ create/2* e *timer_ set/2*.

O primeiro permite criar o *timer*, mas sem o disparar:

timer_create(Nome, Predicado)

- **Nome** - Nome do timer
- **Predicado** - predicado que será executado no fim da temporização

Este predicado cria um timer com o nome **Nome** e declara que os seus eventos serão tratados pelo predicado **Predicado/2**.

17 Criação e disparo do temporizador

Considere-se o seguinte predicado:

foo_handler(Name, Status) :- (write(Name-Status), nl) ~> user.

A chamada do seguinte predicado irá criar um timer que usará aquele "handler":

```
?- timer_create( foo, foo_handler ).
<enter>
yes
```

O segundo predicado é usado para disparar o timer:

timer_set(Nome, Intervalo)

- **Nome** - Nome do timer
- **Intervalo** - intervalo de tempo (em ms) dentro do qual o handler será chamado.

Arma o timer com o nome **Nome**. Se **Intervalo** for um inteiro, o timer disparará dentro de **Intervalo** milisegundos. Se for uma conjunção de um par de inteiros, o primeiro inteiro será interpretado como o intervalo em milisegundos e o segundo como o momento do disparo do timer.

```
?- timer\_set( foo, 1000 ).
<enter>
yes
```

O "handler" foo_handler será chamado 1000 ms após a invocação do predicado *timer_ set*. Após esse período de tempo, o "handler" será executado , afixando uma informação do tipo:

foo - (1000,12345)

em que o segundo inteiro corresponde ao momento em que a temporização terminou.

18 Exemplo de aplicação

```
:-dynamic stop_timer/1.

stop_timer(0).

foo_handler( Name, Status ) :-
    ( write( Name-Status), nl ) ~> user,
    stop_timer(X),
    (
    X = 1 ;
    (
    Status = (Intervalo,_),
    timer_set(Name, Intervalo) )
    ).

go :-
    timer_create( foo, foo_handler ), assert(stop_timer(0)),
    timer_set(foo, 3000).

 stop :-
    retract(stop_timer(_)),
    assert(stop_timer(1)).
```

O predicado *foo_handler* modificado, invoca de novo *timer_set* usando o mesmo intervalo, obtendo-se assim, uma temporização cíclica. A sua invocação está, no entanto, dependente do valor de um parâmetro armazenado num facto *stop_timer/1*, pelo que, o ciclo de temporizações sucessivas pode ser interrompido mediante a chamada do predicado *stop*.

Mais informações sobre esta matéria, especificamente sobre os predicados referidos, podem ser recolhidas no ficheiro pdf WIN_REF existente no sub-directório Examples do WIN-PROLOG.

Parte IV
Notas sobre aplicações WINProlog auto-executáveis

Nota prévia:

Qualquer referência a Prolog neste texto diz respeito ao LPA Win-Prolog, especificamente na sua versão 4.320 (ainda que o processo seja basicamente o mesmo para as outras versões recentes).

É possível tornar um ficheiro Prolog auto-executável, isto é, poder corrê-lo dispensando o ambiente de desenvolvimento do LAP Prolog. Para tal, deve ser seguido o procedimento geral descrito e exemplificado no ficheiro pdf "Win_Usr" (páginas 116 a 128), contido no directório "Docs", e que será resumido de seguida. No entanto, para poder tornar auto-executáveis ficheiros Prolog que utilizem o Agent Toolkit em geral ou a biblioteca TCP em particular, devem ser tidos alguns cuidados específicos, sem os quais o método geral recomendado não funciona.

19 Método Geral

Um aplicação Prolog é composta por dois componentes, um executável contendo o sistema Prolog e outro do tipo "overlay" contendo a aplicação compilada que se pretende executar. O primeiro obtem-se copiando e renomeando o ficheiro PRO386W.EXE contido no directório principal do Win-Prolog. O segundo é o produto da compilação do ficheiro Prolog em causa. Ambos os ficheiros deverão ficar com o mesmo nome. Para uma aplicação Prolog poder ser auto-executável deverá em primeiro lugar conter dois predicados específicos, chamados "hooks" que vão controlar todo o funcionamento do programa. Um será o "main_hook", em que pode ser executado algum processamento prévio, inicializações ou chamada de alguma rotina que invoque um ecran de arranque da aplicação. O segundo será o chamado "abort hook" em que é chamado o programa principal. Veja-se o exemplo abaixo:

```
my_main_hook :-
% <perform any initializations you might want here>,
abort.
my_abort_hook :-
% <run the application itself here>,
halt.
```

Uma vez inseridos os dois predicados "hook" a aplicação pode ser testada ou compilada acedendo ao diálogo "Application" através do menu Run/Application (o ficheiro Prolog deve ter sido previamente compilado). Após o teste efectuado com sucesso, pode ser feito o "save" da aplicação compilada preenchendo a editbox respectiva. O nome que se

der a esse ficheiro "overlay" deverá ser o mesmo a utilizar quando se renomear o ficheiro PRO386W.EXE. No entanto, para que o "overlay" resultante não contenha componentes desnecessários é conveniente fazer a sua compilação numa nova sessão do Win-Prolog, onde se faz a carga do ficheiro Prolog em causa, mediante:

?- ensure_loaded(directorio(ficheiro_prolog)).

20 Método específico para o uso do Agent Toolkit

O método a seguir neste caso deve ter em atenção o momento em que as bibliotecas TCP e do Agent Toolkit são carregadas. Em vez de isso ser feito, como é habitual, mediante a directiva inicial

:- ensure_loaded(examples('agents\protocol\agload')), agent_load_files.

O procedimento correcto será fazê-lo de dentro do "main_hook", conforme o exemplo abaixo

```
agent_main_hook :-
  load_agent_files,
  abort.
% abort hook used when running this program as a stand-alone application
agent_abort_hook :-
  (  go
  -> agent_start
     ;
     halt(1)
  ).
go :-
   agent_reset,
   agente1_initialize( Params, EventHandlers ),
   interface,
   agent_create( Params, Me ),
   agent_initialize,
   agent_declare( handlers, EventHandlers ),
   agent_post_event( start, _ ).
load_agent_files :-
   ensure_loaded( tcp ),
   ensure_loaded( agent ),
   ensure_loaded( 'agdef.pro' ),
   tcp_reset,
   agent_reset.
...
```

Note-se que, no exemplo acima, se parte do princípio que os ficheiros das bibliotecas se encontram no mesmo directório que o próprio "overlay", o que em princípio parece ser a solução mais prática. Em caso contrário, dever-se-á alterar os predicados ensure_loaded contidos em load_agent_files.