Construção de Programas Interactivos por Interface Gráfica

Manuel João Próspero

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática
P-2825 Monte de Caparica

RESUMO

Nesta comunicação é descrita uma técnica de implementação, em Prolog, duma ferramenta útil a qualquer programador de aplicações gráficas interactivas, baseada em redes de transição para a especificação do diálogo. O recurso à edição gráfica elimina a necessidade da programação habitual sob a exclusiva forma de texto. Um ambiente adequado proporciona ainda algumas facilidades adicionais, como sejam a detecção automática de erros e a execução acompanhada visualmente passo a passo.

1. Introdução

É do conhecimento comum a dificuldade com que os programadores, em geral, se debatem quando se trata de construir a interface pessoa-computador (também conhecida por "interface com o utilizador") de uma aplicação interactiva concreta. A experiência mostra que há uma tendência para valorizar a funcionalidade, propriamente dita, da aplicação em prejuízo da qualidade da interface com o utilizador final. Tal situação leva, na prática, a que esses programas venham a ser pouco ou quase nada utilizados. Se tentarmos compreender a razão deste facto, logo veremos que é muito raro um programador ter ao seu dispor ferramentas adequadas à implementação desse tipo (complexo) de interfaces. O que se pretende é, portanto, a possibilidade de se especificar, por linguagem própria, a interface com o utilizador e os programas serem automaticamente gerados a partir daí. O programador deixaria então de se preocupar com muitos dos detalhes da codificação, canalizando toda a sua atenção para os problemas realmente importantes e podendo experimentar diversas soluções alternativas, não sendo penalizado por custos elevados.

Nas secções que se seguem relata-se a técnica de implementação, em Prolog, de uma interface gráfica que permite a um programador construir autómatos não deterministas que especificam o diálogo da aplicação com o utilizador final. O conteúdo desta comunicação teve como base um trabalho proposto para final de curso de engenharia informática na UNL, mais concretamente no âmbito da cadeira de Projecto de Programação. Foi realizado por Ricardo Amador, que só por absoluta indisponibilidade de tempo para trabalhar neste artigo não se

apresenta também como co-autor do mesmo. Em todo o caso, os programas aqui apresentados e a maior parte das figuras foram extraídos do relatório [AMA89].

2. Especificação do Diálogo por Redes de Transição

As técnicas para descrição e controlo do diálogo duma aplicação com o utilizador podem agrupar-se em três grandes classes, considerando [PRÓ88]: as redes de transição, as gramáticas (independentes ou não do contexto) e o controlo por eventos.

O poder descritivo de uma notação mede-se pelo conjunto de interfaces que podem ser por ela descritas. Embora o poder descritivo das redes de transição seja inferior ao de outros mecanismos (como as gramáticas sensíveis ao contexto ou a programação dirigida por eventos), aquelas são uma ferramenta muito útil aos programadores de aplicações interactivas pela sua facilidade de utilização e pelo carácter gráfico aliado à sua representação. Assim, uma rede de transição pode ser visualizada por um ou mais diagramas que representam grafos, em que os nós corresponderão a estados e os arcos, dirigidos, a transições de estados.

Como seria de esperar, a construção automática de programas a partir de redes de transição implica sempre a elaboração de um tradutor para a linguagem usada na especificação. Se tal tradução for feita directamente a partir da representação gráfica dos diagramas, sem qualquer representação textual intermédia da responsabilidade do programador, a programação do diálogo da aplicação virá muito simplificada, permitindo então a esse programador debruçar-se exclusivamente sobre os pormenores que contribuam para a qualidade da interface. Por isso, se bem que a forma textual de uma especificação seja a de mais fácil interpretação a nível de programação, é a forma gráfica que permite uma definição mais natural e intuitiva da especificação do diálogo de uma aplicação e, por conseguinte, merece o esforço de se tentar desenvolver uma técnica perseguindo esse objectivo [JAC85].

Do ponto de vista formal, uma rede de transição é caracterizada por um tuplo

$$M = (Q, \Sigma, P, \delta, \gamma, q_0, f)$$

com o seguinte significado dos símbolos utilizados:

- Q — conjunto finito de estados
- Σ — conjunto finito de símbolos de entrada
- P — conjunto finito de acções
- δ — uma aplicação de $Q \times \Sigma$ em Q
- γ — uma aplicação de Q em P
- q_0 — estado inicial, pertencente a Q
- f — conjunto de estados finais, incluído em Q

Deste ponto de vista, uma rede de transição apenas difere de um autómato pela introdução de P e γ. Por esse motivo, e salvaguardando os casos ambíguos, passaremos a utilizar

preferencialmente a designação "autómato" em vez de "rede de transição". No entanto, há que associar a cada estado de um autómato um predicado dito "de transformação" que implemente a aplicação γ.

Os autómatos que permitem ao programador especificar o diálogo são automaticamente traduzidos (compilados) para um conjunto de cláusulas em Prolog. Os arcos são etiquetados pelo que chamamos "evento". Entendemos este como um conceito lato que engloba qualquer acontecimento susceptível de alterar o estado do sistema, desde a satisfação de uma dada condição, até a uma qualquer interrupção do programa causada pelo utilizador, passando inclusivamente pela chamada a outros autómatos (permitindo então, tanto a recursividade directa, como a indirecta). Os eventos básicos são considerados como quaisquer outros autómatos e, por isso, podem ser programados de acordo com as necessidades da aplicação em causa.

Um evento pode ainda ser uma transição vazia, caso em que ele é designado por `nil`. Isto permite transformar um diagrama de tal forma que de um estado final F se não possa transitar para qualquer outro estado, como se mostra na figura 1. Ao emprego da transição vazia nestas situações corresponde sempre a criação de mais um estado no conjunto Q.

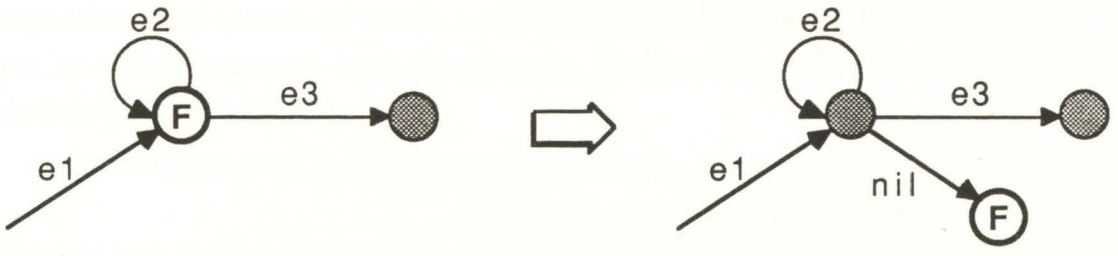

Figura 1 — Utilização do evento `nil`.

3. Interface com a Aplicação propriamente dita

O modelo escolhido para o Sistema de Gestão da Interface com o Utilizador (vulgo UIMS) foi o de Seeheim [ERO87]. Segundo este modelo, a interface comporta-se como uma função que, recebendo os dados do utilizador, os transforma para, dessa forma, serem passados à parte funcional da aplicação (vulgarmente chamada de "aplicação propriamente dita"). O mesmo se passa em sentido contrário, mas em que agora a complexidade da informação, característica da aplicação, vai ser decomposta até ficar em condições de poder ser afixada numa superfície de visualização, por exemplo um ecrã.

Ao predicado anteriormente apresentado como de transformação cabe a responsabilidade de, ao nível de cada estado, transformar adequadamente as estruturas de dados correspondentes. Se tomarmos um dado caminho num determinado autómato, será a composição das transformações associadas aos estados aí encontrados que fará com que, a partir de uma estrutura de dados inicial (dita de entrada, EDE), se obtenha uma estrutura de dados final (dita de saída, EDS). No

caso geral, obviamente, a transformação particular de cada estado terá que ter em conta o evento de que resulta a transição para esse estado. Assim sendo, cada predicado de transformação terá três parâmetros: o evento de chegada ao estado, a estrutura de dados corrente e a estrutura de dados resultante. Caso exista algum estado que não necessite de transformar estruturas de dados, por uma questão de uniformização assumir-se-á a transformação identidade, ou seja, aquela que deixará inalterada a estrutura de dados de entrada. Por motivo idêntico, assume-se que, quando a execução principia, o evento (virtual) que teria conduzido ao estado inicial é o evento `nil`.

Vejamos, em seguida, a maneira de identificar sintacticamente cada evento. Dado que este poderá ser, por sua vez, um subautómato, então, além de um nome (que determinará o que se pode chamar "classe do evento"), haverá que associar também a EDE e a EDS correspondentes. Assim, o termo `menu(Menu,Item)` poderá ser o evento da classe `menu` que representa a escolha `Item` de entre o conjunto especificado em `Menu`. Recorde-se que, na sintaxe do Prolog de Edimburgo (uma norma *de facto* [CLO84]), as variáveis se iniciam por letra maiúscula, o que não acontece com os símbolos de função.

Agora já estamos em condições de explicar a criação de eventos básicos. O exemplo `menu`, dado anteriormente, poderá ser considerado como básico numa certa aplicação. O autómato correspondente terá apenas um estado e o predicado de transformação será responsável pela interacção com o utilizador final da aplicação, apresentando-lhe uma lista de escolhas e devolvendo a opção escolhida:

```
evento_menu(nil,Opções,Opção_escolhida) :-
                                choice(Opções,Opção_escolhida).
```

Repare-se que, como o autómato tem um só estado, necessariamente o evento `nil` teria que figurar no predicado de transformação associado.

4. A execução e o seu controlo

A implementação de um UIMS que foi realizada permite, quer o chamado controlo externo, quer o interno [PFA85]. A diferença entre ambos é que, neste segundo tipo, o controlo da execução é de exclusiva responsabilidade da aplicação.

O nosso interpretador de autómatos é descrito pelo predicado `dialogue/3` (forma "símbolo de função / aridade"), que aceita os três parâmetros que descrevem o autómato a ser executado: o nome (ou classe), a EDE e a EDS. Visto que qualquer outro predicado pode chamar `dialogue`, é nesse sentido que o controlo pode ser considerado como interno.

A título de exemplo, atente-se no diagrama da figura 2. Corresponde este ao autómato principal duma aplicação destinada a manipular uma base de dados. No entanto, o mesmo objectivo é conseguido pelo código que se segue (controlo interno), onde o interpretador de autómatos será chamado sempre que o utilizador final escolha uma opção existente (que não `'SAIR'`):

```
principal(_, _) :-
    repeat,
        dialogue(menu, mPrincipal, Opção),
        processa_opção(Opção),
    Opção == 'SAIR', !.

processa_opção('SAIR').
processa_opção('CRIAR') :- dialogue(criar, _, _).
processa_opção('EDITAR') :- dialogue(editar, _, _).
processa_opção('APAGAR') :- dialogue(apagar, _, _).
```

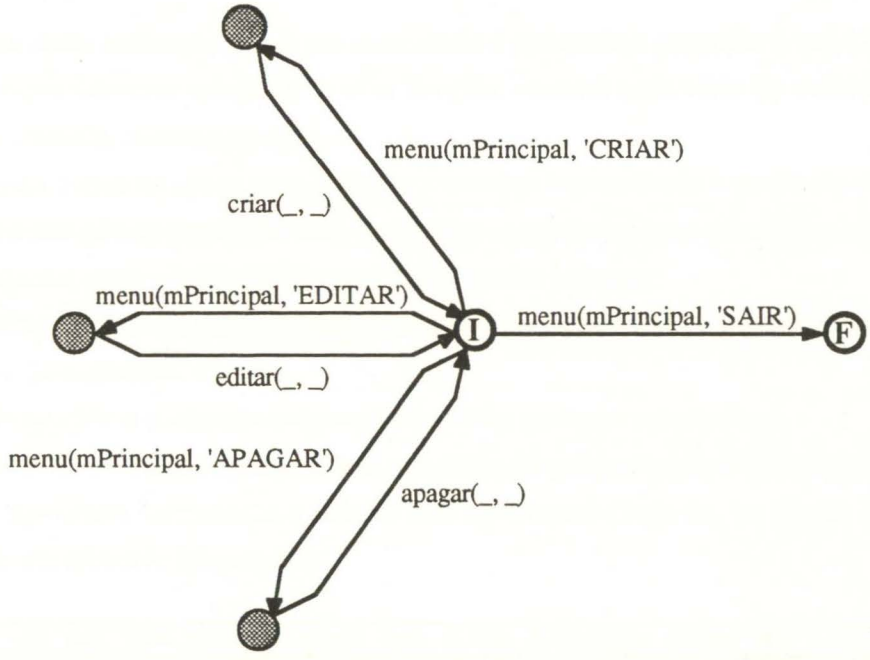

Figura 2 — Autómato principal numa certa aplicação.

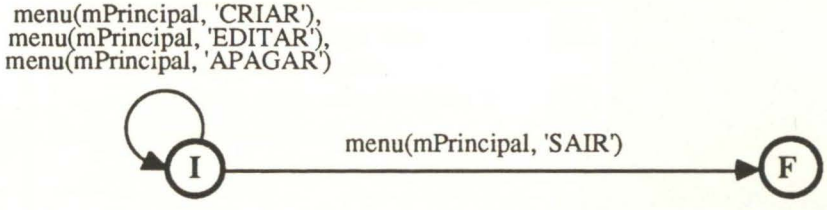

Figura 3 — Autómato com apenas dois estados.

Naturalmente que há muitas outras soluções para este problema. Uma delas será eliminar os estados intermédios da figura 2, englobando as suas transformações no próprio estado inicial. Tal é exemplificado pela figura 3. Repare-se nos três eventos agora associados a um lacete, não havendo qualquer referência aos subautómatos `criar`, `editar` e `apagar`. Estes são activados indirectamente por controlo interno através do predicado de transformação `processa_opção/3` correspondente ao estado inicial:

```
processa_opção(nil, ED, ED) :- !.
processa_opção(ev(menu, mPrincipal, Opção), ED, ED) :-
                                        processa_opção(Opção).
```

A representação do evento `menu` surge aqui numa forma sintáctica diferente, por ter sido utilizado o termo `ev/3`. Esta foi a representação interna escolhida, que tem a vantagem de permitir aos programas uma manipulação mais directa, uma vez que a classe de cada evento passará a figurar também como argumento, mantendo-se inalterado o símbolo funcional `ev`.

5. Compilação

Para que um dado autómato possa ser executado é necessário proceder à sua compilação. O compilador, especialmente criado para essa função, executa uma série de validações sobre o autómato que compila, nomeadamente:

i) identifica todos os arcos sem eventos e estados inacessíveis a partir do estado inicial, removendo-os (do ponto de vista do código compilado) para permitir a continuação da compilação, caso o utilizador/programador assim o deseje,

ii) identifica estados que não conduzem a qualquer estado final e que são potenciais ramos mortos do autómato, e

iii) detecta quando o autómato em causa não tenha qualquer estado final.

Note-se que estes resultados possibilitam um estilo de programação a que poderemos chamar incremental, apoiando fortemente a edição dos autómatos (veja-se, na figura 4, a detecção automática de um arco não etiquetado).

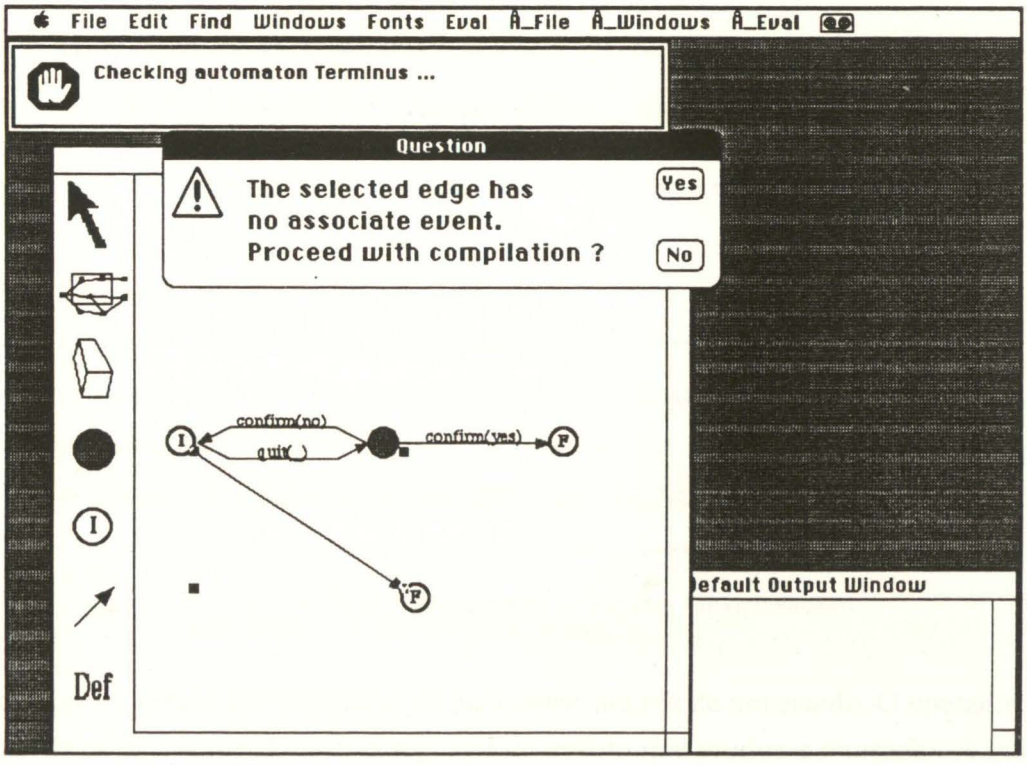

Figura 4 — Estado do ecrã durante uma compilação.

A compilação de um autómato converte-o num conjunto de factos para os seguintes predicados:

`firstState/2` — explicita o estado inicial do autómato

`finalState/2` — explicita os estados finais do autómato

`eventsOut/3` — define as transições de saída para cada estado do autómato

`attribs/3` — enumera os predicados associados a cada estado do autómato.

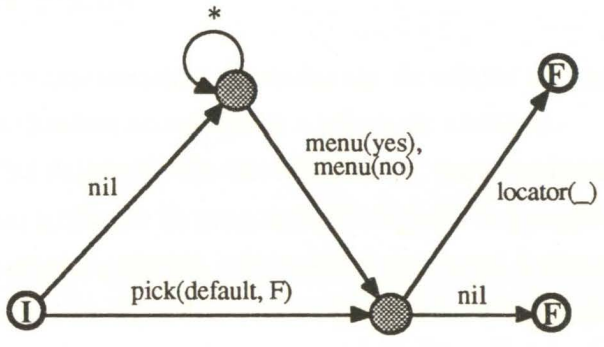

Figura 5 — Um autómato que não acusará erros na compilação.

A representação compilada do autómato da figura 5 seria então a seguinte:

```
                                AUTOMATON: teste

*************************************************************************/

/*****************************   Initial state    *****************************/
firstState(teste, id0).

/*****************************    Final states    *****************************/
/* id9  */
finalState(teste, id9).
/* id7  */
finalState(teste, id7).

/*****************************    Connections     *****************************/
/* id0  */
eventsOut(teste, id0, [id1/ev(pick, default, F), id2/nil]).
/* id1  */
eventsOut(teste, id1, [id9/ev(locator, _), id7/nil]).
/* id2  */
eventsOut(teste, id2, [id2/ *, id1/ev(menu, no), id1/ev(menu, yes)]).

/***************************** States attributes  *****************************/
attribs(teste, id0, [transf=transf_inic]).
attribs(teste, id1, [help=ajuda, show=mostra, transf=transformação,
                undo=desfaz, identif=identifica]).
```

A forma `idn` é gerada automaticamente para nome interno de um estado. O operador "`/`" é usado para separar o nome do estado seguinte do evento que produz a transição. A interface com a aplicação é especificada pelos atributos. Assim, `transf_inic` será o predicado de transformação para o estado inicial.

O autómato da figura 5 apresenta o evento especial "*" no lacete de um dos estados. Isto significa que é aí aceite qualquer evento diferente de `menu(yes)` e de `menu(no)`.

Em muitos casos, é desejável uma maior simplificação no texto identificador de cada evento. Desde que os predicados de transformação sejam devidamente programados, é possível uma sintaxe mais simples, em que se apresente apenas a classe do evento e a sua EDS. Foi o que aconteceu com `locator`, ainda na figura 5.

6. Utilização do editor gráfico

Algumas das principais características do ambiente de edição são apresentadas seguida e resumidamente. Para mais detalhes aconselha-se a leitura de [AMA89].

A gestão da interface foi desenvolvida em MacProlog, para equipamento Macintosh e no sentido de uma extensão ao ambiente de programação daquela linguagem. A edição gráfica dos autómatos é realizada em janelas próprias, que incluem uma zona, à esquerda, para a escolha de utensílios (tal como se mostra na figura 6). A área de desenho é separada da zona de utensílios por uma linha de *split*, relacionada com a opção *viewer*. Esta opção, que se encontra seleccionada na figura 6, permite mover o rectângulo de visualização sobre toda a área de desenho.

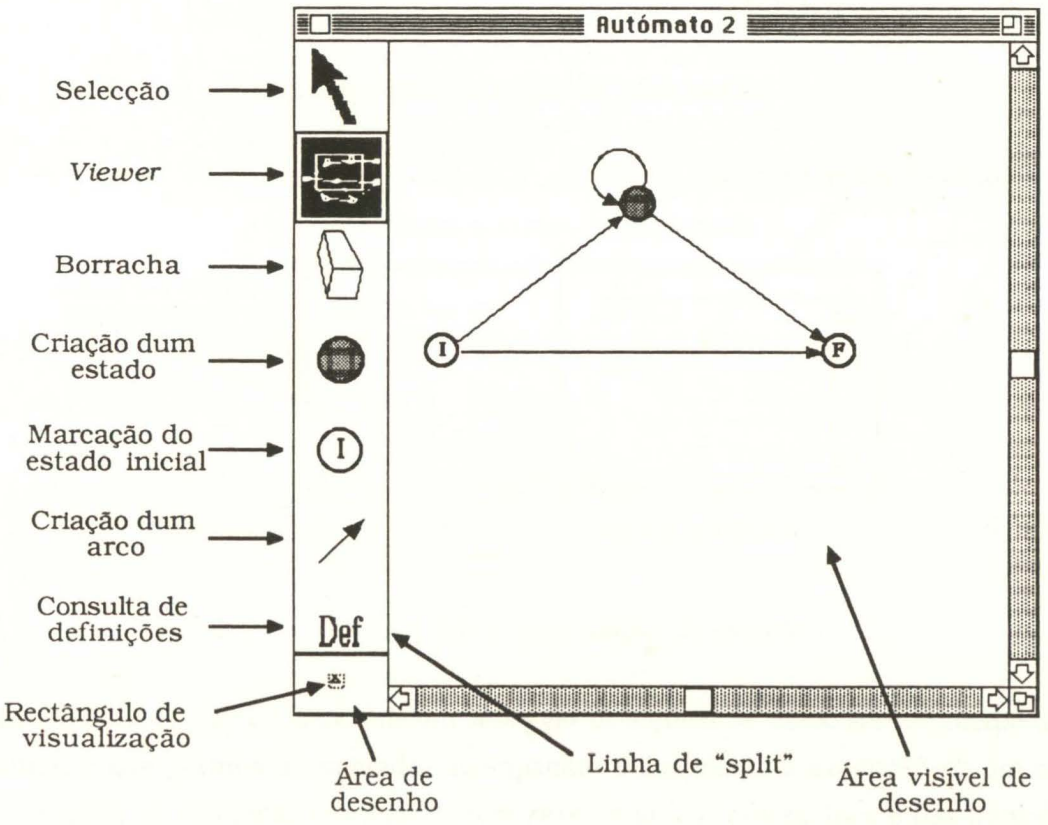

Figura 6 — Janela de edição de autómatos.

Opções como as das operações com ficheiros e dos pedidos de ajuda, entre muitas outras, são accionadas por selecção em menus *pull-down*.

A criação de um estado resulta da escolha de um local ainda não ocupado e dentro da área de desenho. Qualquer estado do qual se não possa transitar é, por definição, um estado final. Assim, após a criação de um novo estado, ele é automaticamente representado como final, deixando de o ser logo que se defina uma transição para qualquer outro estado existente.

Para criar um arco, o utilizador indica apenas o nó de partida e o de chegada. A edição de cada evento associado faz-se depois, de maneira cómoda, editando texto numa janela de diálogo que aparece quando o arco for seleccionado. A figura 7 mostra-nos essa janela, onde aparece a classe, a EDE (**Identifier**) e a EDS (**Value**) do evento. No entanto, se um dado arco possuir mais do que um evento associado, o diálogo de edição é iniciado, de forma automática, pela janela de que é exemplo a figura 8. A identificação dos eventos criados aparece, na forma de texto, sobre os arcos com que aqueles ficaram relacionados.

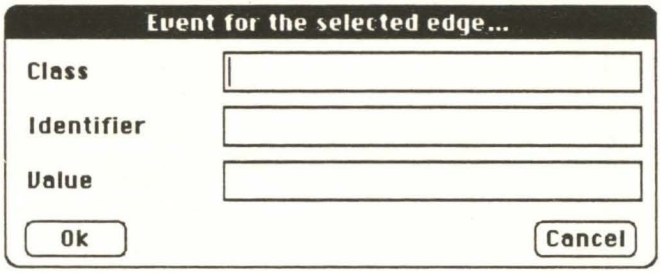

Figura 7 — Janela de edição de cada evento.

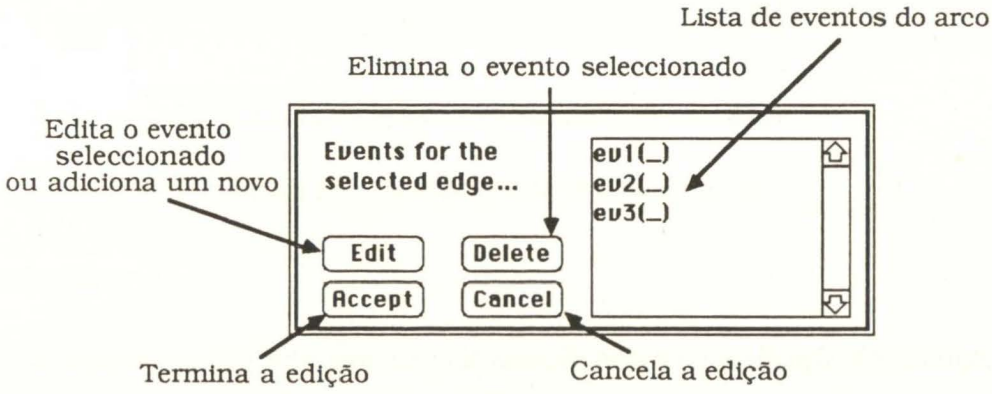

Figura 8 — Diálogo prévio para edição de eventos.

O ambiente de execução foi dotado de um *debugger* de autómatos, associado ao interpretador dos mesmos, e que permite ao operador acompanhar visualmente a execução de um dado autómato através da marcação automática (em *reverse video*) dos estados e das transições percorridas. De notar que estas operações foram concebidas e implementadas coerentemente com o mecanismo de retrocesso da linguagem, desfazendo-se a marcação à medida que se

retrocede no grafo. Neste modo de funcionamento, quando termina a execução de um autómato encontrar-se-á marcado todo um percurso que nos conduz do estado inicial a um estado final. Num modo ainda mais fino, denominado *trace*, a execução do autómato é suspensa em cada estado, logo após a transformação associada ao estado (em que é mostrada a estrutura de dados resultante, podendo o utilizador/programador escolher entre continuar, forçar o falhanço da transformação, ou mesmo fazer abortar a execução) e antes de ser gerado o novo evento de saída (sendo o utilizador/programador interrogado sobre se o evento de saída deve ou não ser gerado, tal como se vê na figura 9).

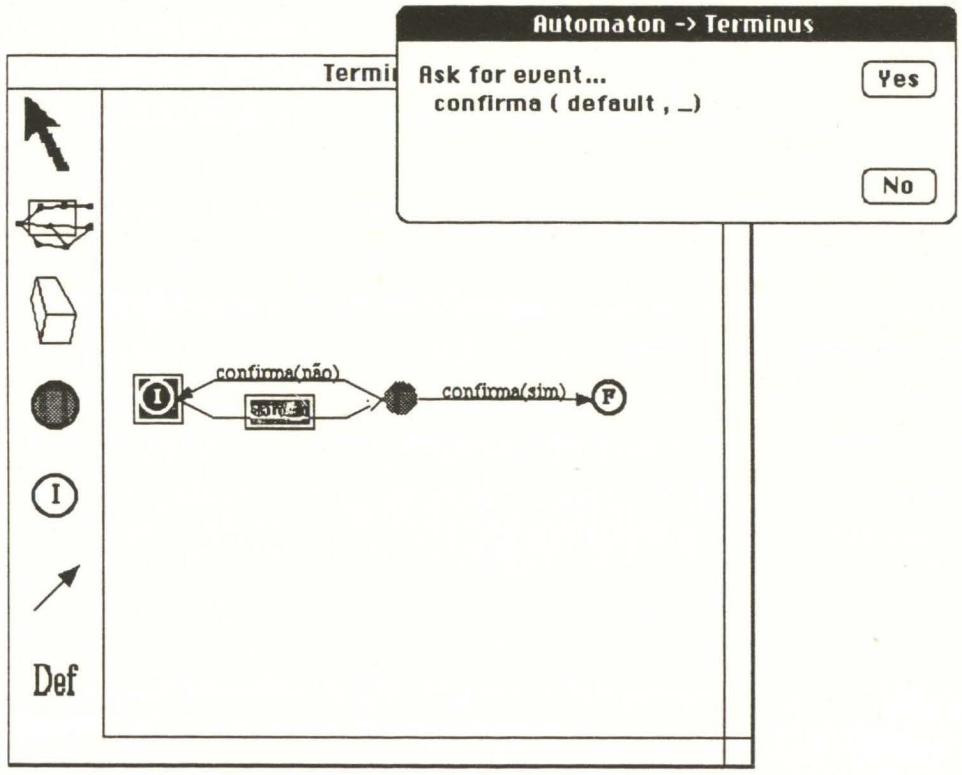

Figura 9 — Funcionamento em modo *trace*.

7. Conclusões

Por muito potente que seja uma determinada notação para a especificação do diálogo, a forma textual em que a maioria delas é apresentada ao programador é, por si só, um obstáculo importante à sua aceitação por parte daquele. Ora, tendo as redes de transição de estado uma representação gráfica natural, o esforço de aprendizagem da sua utilização através dum programa interactivo é relativamente diminuto. No nosso trabalho, a interface gráfica é, portanto, a ferramenta indispensável colocada ao dispor do próprio programador.

Por outro lado, o poder da unificação em lógica, a forma modular de qualquer conjunto de cláusulas e o retrocesso automático (*backtracking*), característicos da linguagem Prolog, permitiram construir um sistema de gestão de interfaces suficientemente poderoso e complexo com uma abordagem a muito alto nível: a edição gráfica.

O código gerado de forma automática é altamente recursivo, o que, ao fim e ao cabo, permitirá o retrocesso durante a execução. No entanto, este facto acarreta um elevado consumo de memória que, num grande número de casos, não se justifica. Uma beneficiação possível para a implementação será, pois, a detecção desses casos e a consequente passagem a algoritmos iterativos.

Como exercício de aplicação, gerou-se um programa gráfico interactivo destinado a ser executado noutro tipo de equipamento e ambiente de programação. Mais concretamente, numa VAXstation II possuindo um interpretador de C-Prolog, tendo GKS (nível 2b) como suporte da programação gráfica. Foram então definidos alguns autómatos de base no sentido de simularem os dispositivos lógicos de entrada gráfica do GKS (*choice*, *locator*, *pick*, etc.) em modo *request*. Para além disso, o programa gerado mais não era do que um editor gráfico de autómatos, semelhante ao que se desenvolveu em MacProlog para o Macintosh.

Como resultado deste exercício podemos concluir que a técnica usada é de aplicação geral e que a nossa própria implementação, embora se encontre escrita em MacProlog sobre um Macintosh II, pode ser utilizada para produzir interfaces para aplicações interactivas num outro ambiente. É claro que em certas configurações de equipamento e ambientes de programação se pode contar com facilidades mais convidativas que noutros. Tal situação deriva do facto da componente de apresentação do UIMS [PFA85], isto é, aquela que tem a ver com a utilização dos dispositivos físicos de entrada/saída, não ter sido alvo da especificação da interface com o utilizador.

8. Referências

[AMA89] Amador, R.
Projecto de Programação — UIMS, Relatório interno — UNL (1989)

[CLO84] Clocksin, W.F.; Mellish, C.S.
Programming in Prolog, Springer-Verlag, 2nd. edition (1984)

[ERO87] Ero, J.; van Liere, R.
User Interface Management Systems, *Eurographics'87 Tutorial*, Amsterdam (1987)

[JAC85] Jacob, R.J.K.
A State Transition Diagram Language for Visual Programming, *Computer*, 18(8) pp 51-59 (August 1985)

[PFA85] Pfaff, G.E. (editor)
User Interface Management Systems, Springer-Verlag (1985)

[PRÓ88] Próspero, M.J.
Estilo declarativo na Programação Gráfica Interactiva: análise e avaliação sobre sistemas em Prolog, Dissertação de Doutoramento — UNL (1988)