CAPÍTULO 6

MODO ASSISTIDO
E
GUIÕES DE INTERACÇÃO

ÍNDICE

6.1.- O Modo Assistido Sintáctico-Semântico (MASS). 164
6.1.1. - Introdução. 164
6.1.2.- Características. 164
6.1.3.- Aspectos de Implementação. 171
6.2.- Guiões de Interacção. 174
6.2.1. - Introdução. 174
6.2.2.- Propriedades. 176
6.2.3.- Sintaxe e Semântica Estática. 178
Declarações Estáticas. 180
Comportamento. 182
6.2.4. - Semântica Operacional: Redes de Petri. 187
Redes Etiquetadas-Guardadas-Interruptíveis. 188
6.2.5.- Semântica dos Guiões. 191
Expressão Atómica. 192
Sequência. 193
Paralelismo Síncrono. 194
Paralelismo Assíncrono. 195
Alternativa. 195
Repetição. 196
Interrupção. 197
Eventos Assíncronos. 198
6.2.6. - Exemplos. 198
6.2.7. - Edição de GI e geração das Redes de Petri. 216
6.3.- Exemplo de Aplicação do MASS. 216
6.4.- Sumário. 219

6.1.- O Modo Assistido Sintáctico-Semântico (MASS).

6.1.1. - Introdução.

Qualquer método que se pretenda desenvolver para a concepção sistemática de sistemas interactivos tem por primeiro passo obrigatório a adopção de um mo-delo de interacção. Este modelo do comportamento interactivo deve ser sufi-cientemente abstracto para que possa representar as características de entrada, de saída e de comunicação dos sistemas visados, de forma afastada em relação aos detalhes necessários para produzir uma efectiva implementação. O modelo deverá definir um conjunto de propriedades, ou requisitos, que irão constituir uma referência para quem concebe e para quem constrói a IU.

Por um lado, o modelo deverá fornecer uma base de referência para que cer-tas características de comportamento dos sistemas desenvolvidos possam ser confrontadas com a própria perspectiva do utilizador sobre o funcionamento do sistema. Por outro lado, características gerais dos utilizadores podem também ser consideradas na construção do modelo. Apenas desta forma, modelos de in-teracção podem ter utilidade como modelos que suportam a concepção de siste-mas interactivos com elevado grau genérico de usabilidade.

O modelo de interacção que no âmbito do trabalho desta tese foi desenvolvi-do visando tais objectivos, designa-se por *Modo Assistido Sintáctico-Semântico* (MASS). Apresenta-se a seguir a relação entre este modelo e o modelo dos arqué-tipos, as extensões que a este introduz, como adopta princípios que são oriun-dos de uma análise de características comuns dos utilizadores, e como pode ser implementado sobre interfaces linguísticas já existentes, promovendo a sua re-cuperação tecnológica.

Para além de apresentar o MASS, este capítulo introduz também o formalis-mo dos *Guiões de Interacção* (GI), a notação desenvolvida para a especificação de controladores do diálogo seguindo o modelo MASS. Os Guiões de Interacção pos-suem um conjunto de construções que permitem a especificação dos diálogos a alto nível e de forma modular. A partir destas especificações a IU é depois gera-da automaticamente.

As construções existentes nos GI suportam todas as propriedades de inter-acção definidas no modelo MASS, tendo ainda poder expressivo para, tal como se mostra na secção 6.2, especificarem diversos outros tipos de IU.

6.1.2.- Características.

O *Modo Assistido Sintáctico-Semântico* (MASS) é um modelo conceptual da IU de tipo ***RC*** ou ***DC***, ou seja, que se destina quer a quem concebe quer a quem tem que implementar a IU. O modelo é interiorizável e exteriorizável, i.é., deve existir no interior dos destinatários, mas pode ser igualmente comunicado

usando uma notação. É estrutural pois é construído segundo um dado princípio orientador, e é genérico pois é aplicável a várias classes de objectos ou problemas.

Comecemos por relacionar o MASS com o modelo dos arquétipos do qual, em grande medida, é uma abstracção, já que sintetiza algumas das propriedades de interacção apresentadas por estes. No modelo dos arquétipos, a ideia base con-siste em implementar um modelo de interacção por edição estrutural sobre ter-mos incompletos que representam o processo correcto de construção de certos objectos, representados como termos de uma álgebra. Como igualmente se sali-entou, o modelo dos arquétipos acomoda quer uma perspectiva linguística quer a sua correspondente algébrica do processo de construção interactiva de objec-tos, com base na sua estrutura sintáctica. A interacção dirigida pela estrutura sintáctica dos objectos a construir tem, no mínimo, a vantagem de permitir eli-minar qualquer possibilidade de ocorrência de erros sintácticos na representa-ção interna destes objectos. Esta evidente vantagem, bem como as facilidades de manipulação e a generalidade da representação dos arquétipos, induziu a sua aplicação no contexto da interacção entre utilizadores e aplicações fundamen-talmente baseada na utilização de uma *linguagem de comandos*.

Pensando-se agora que os objectos a construir passam a ser *comandos inter-activos,* representáveis quer por estruturas sintácticas quer por termos algébri-cos, a estes é, portanto, igualmente aplicável o modelo de edição, construção ou manufactura dos arquétipos. Em última análise, exceptuando as interfaces por manipulação directa, que se situam num outro paradigma, todas as IU podem ser abrangidas por este modelo, dado possuirem, mais ou menos claramente, uma *linguagem de comandos*, ou seja, uma linguagem para activação da funcio-nalidade da camada computacional.

Adicionalmente, qualquer que seja a linguagem de comandos, e qualquer que seja a complexidade semântica intrínseca a cada comando, é observável que uma linguagem interactiva deste tipo é, em geral, uma linguagem bastante simples e com uma grande regularidade estrutural, dado ser construída à volta da noção concreta de *comando*. Embora no contexto deste modelo a representa-ção de um comando seja mais complexa que a simples consideração da sua es-trutura sintáctica, ainda assim, esta orientação favorece uma perspectiva modu-lar de concepção e de desenvolvimento, de grande utilidade.

Do ponto de vista sintáctico, cada comando a construir e a passar à aplica-ção é uma estrutura parametrizada simples, constituída por um símbolo termi-nal, um *token* que é o seu identificador, e uma sequência de argumentos com tipo que irão ser instanciados interactivamente pelo utilizador. Qualquer que venha a ser a sua representação, um *comando* pode, assim, ser sempre visto co-mo sendo um *arquétipo simples*.

Duas perspectivas de utilização do modelo MASS podem ser aqui considera-das para apresentação das suas características, nomeadamente, a sua aplica-ção em sistemas a serem desenvolvidos e a sua aplicação a sistemas já existen-tes. Dado que a aplicação do modelo no desenvolvimento da IU de sistemas a serem construídos é tratado nos capítulos seguintes, integrando-o até num mé-todo sistemático, apresentaremos aqui as suas propriedades no contexto da sua utilização para a recuperação tecnológica de sistemas existentes, nos quais a funcionalidade da aplicação é invocada recorrendo a uma linguagem formal, explícita para o utilizador (por exemplo a formulação de uma "interrogação"[1] a uma base de dados usando uma "linguagem de interrogação"[2]).

O modelo MASS visa, antes de mais, definir as propriedades a que a lingua-gem de interacção a construir deve obedecer, por forma a que se possam atingir os objectivos de usabilidade e de orientação para implementação atrás referidos. Enunciemos e analisemos de seguida, uma por uma, cada uma destas proprie-dades.

? ***Assistência Sintáctica****: o modelo de interacção deverá garantir que a linguagem de interacção a usar a nível da IU esconderá do utilizador todos os detalhes da sintaxe concreta da linguagem de comunicação com a aplicação; por outro lado, recorrendo ao mecanismo de edição estrutural dos arquétipos, deverá garantir que qualquer frase sintetizada é sintacticamente correcta.*

Este requisito é justificável, do ponto de vista da usabilidade dos sistemas, se tivermos em consideração que qualquer utilizador de um sistema interactivo pensa de modo informal, fala em língua natural e é aqui obrigado a exprimir as suas intenções numa linguagem formal de interacção. Em resultado, é notória uma grande distanciação entre a liberdade expressiva que o utilizador usufrui noutros contextos, e a rigidez e inflexibilidade da sintaxe fixa da linguagem de comandos aceite pela aplicação. Assim, a distância articulatória é elevada, o que implica baixa usabilidade e grande probabilidade de erro sintáctico. A *assistên-cia sintáctica* sugerida no MASS, e implementável seguindo um esquema de edi-ção orientada pela estrutura semelhante à anteriormente formalizada nos arqué-tipos, favorece uma diminuição da distância de articulação por garantir a au-sência de erros sintácticos e seu subsequente tratamento. A navegação estrutu-ral está muito ligada, como se viu atrás, à permanente escolha de uma dentre várias alternativas. Em resultado, o estilo de interacção conhecido que melhor se adequa a este tipo de navegação estrutural é o de *interacção por menus*, de-vendo desde já salientar-se que interacção por

[1] *query*.

[2] *query language.*

menus não é um fim mas apenas o melhor meio para tal disponível na tecnologia de interacção.

Esta utilização de menus tem igualmente a ver com uma outra característica do comportamento de qualquer utilizador: o *não-determinismo*. De facto, ao de-sejado determinismo de um sistema interactivo opõe-se o *não-determinismo* do utilizador, que, conforme caracteriza Dix em [Dix 90], e havia já sido igualmen-te apontado em [Martins e Oliveira 90], assume pelo menos duas facetas:

- *incerteza procedimental*, ou seja, completo desconhecimento do sistema sobre qual vai ser a acção ou escolha seguinte do utilizador;
- *incerteza nos dados*, ou seja, a incerteza do utilizador sobre quais os valo-res sobre os quais pode actuar correctamente, ou dentre os quais pode realizar uma escolha válida num dado contexto.

A *incerteza nos dados* é uma fonte de não-determinismo de origem semânti-ca, ou seja, relacionada com o estado momentâneo da aplicação, e será aborda-da no ponto seguinte.

A *incerteza procedimental* é uma fonte de não-determinismo que no modelo MASS é tratada usando um esquema de orientação sobre o conjunto válido de escolhas possíveis, sempre que o controlo do diálogo se encontra do lado do uti-lizador, em particular tendo em consideração *condições de contexto*.

Seja d o alfabeto de entrada, e seja d_f o subconjunto de símbolos de d que, num dado contexto, podem ser aceites. Assim, a expressão matemática que corresponde à escolha e utilização de um símbolo do alfabeto pode escrever-se como,

$$\text{let } \mathbf{x} \in d \text{ in } \mathbf{f(x)}$$

em que $\mathbf{x} \in d$ representa uma escolha não-determinista sobre o alfabeto d. Porém, o problema pode surgir a dois níveis. Em primeiro lugar, se, apesar da especificação, o utilizador introduz um símbolo $x \in d$ tal implica a invocação de uma rotina de erro, seguindo a abordagem tradicional em compilação. Porém, a interacção segue muito mais um modelo de interpretação, ou seja, em que a in-teracção é realizada seguindo um modelo "demand-driven", pelo que o modelo de interpretação pode permitir evitar correcção de erros ao garantir que as entradas do utilizador são válidas.

Contudo, o facto de uma entrada do utilizador ser correcta, ou seja, estar contida em d, não é suficiente, dado que, num determinado contexto de inter-acção, e apesar do símbolo pertencer ao alfabeto, é também necessário que tal símbolo seja, em tal contexto, aceitável ou reconhecível. Assim, a especificação completa deverá ser a que indica que, para um dado contexto de interacção, se deve ter,

$$\text{let } \mathbf{x} \in d_f \text{ in } \mathbf{f(x)}$$

Tal significa que os diferentes menus apresentados ao utilizador devem ser baseados em informação contextual, por forma a que as opções seleccionáveis sejam apenas as que, nesse contexto, são aceitáveis.

Note-se igualmente que, colocada esta camada interactiva entre o utilizador e a linguagem formal, as frases desta passam a ser construídas de modo "lazy" [Henderson 84], i.é., símbolo a símbolo, sendo cada um destes símbolos asso-ciado a acções interactivas do utilizador, podendo de imediato serem realizadas acções pela IU.

Assim, símbolo a símbolo, valor a valor, frases correctas do ponto de vista sintáctico são construídas. No modelo "batch", ou "eager", por oposição a "lazy", os comandos são introduzidos textualmente pelo utilizador e, apenas quando completos, passados ao possível "parser", que só então analisará a sua validade.

? ***Assistência Semântica****: o modelo de interacção deverá garantir que os valores que podem num dado contexto ser atribuídos aos argumentos das operações, dependem do contexto actual, ou seja, do estado da camada computacional. Prendendo-se esta propriedade com a anteriormente designada "incerteza nos dados", então, sempre que o utilizador tem que seleccionar um de vários valores possíveis, é vantajoso que apenas a gama de valores aceitáveis lhe seja proporcionada, bem como mecanismos de selecção;*

A assistência semântica sugerida pelo modelo, significa que sempre que um valor deve ser sintetizado o mesmo deverá ser semanticamente correcto, não sendo portanto passível de gerar situações de erro. A *assistência sintáctica* em conjunto com a *assistência semântica,* permitem a construção de IU exibindo um comportamento com uma característica muito importante, que se traduz por uma perspectiva "profiláctica" (e não "terapêutica") da interacção. Podem assim, à partida, evitar-se erros sintácticos ou frases sem significado semântico e, em consequência, libertar a camada computacional do ónus do tratamento e recu-peração de erros. Adicionalmente, mensagens de erro e acções do utilizador pa-ra o seu tratamento, naturalmente elementos perturbadores da efectiva interac-ção, são minimizadas, com evidente vantagem.

? ***Valores por Omissão***[3]*: o modelo de interacção poderá garantir que certas variáveis, cujos valores devem ser lidos, possam ser aceites sem terem sido instanciadas recorrendo à utilização de valores por omissão. Estes valores tanto podem ser especificados previamente, como podem ser pré-definidos em função do tipo de dados.*

[3] *default values.*

A utilização de valores por omissão para a instanciação de variáveis permite que termos incompletos possam ser aceites sem erro, sendo o termo completo construído usando tais valores e apresentado ao utilizador para confirmação.

? ***Flexibilidade na Ordem de Leitura de Argumentos****: a ordem de aceitação dos argumentos de um comando deverá obedecer a um critério lógico, ou seja, não deve ser imposta ao utilizador uma ordem rígida apenas por conveniência de implementação. Por exemplo, se os dois argumentos de um dado comando não têm entre si qualquer relação lógica, então a ordem da sua leitura é, naturalmente, livre. Noutras situações, uma ordem fixa deverá ser contemplada.*

Finalmente, não se deve esquecer que a prática assistida pode tornar-se uma escolha do utilizador que, acaba por aprender, por experiência, a linguagem subjacente da aplicação, e por achar mais cómodo e rápido escrever direc-tamente nesta, ainda que com menor segurança. Tem-se assim a seguinte pro-priedade:

? ***Comutação de Modo****: o modelo de interacção deverá permitir que, nas implementações em que tal faz sentido, a assistência sintáctico-semântica possa ou não ser usada pelo utilizador, através de um mecanismo que faci-lite a comutação entre modo assistido e modo "batch". A comutação deve, idealmente, poder ser feita a meio da síntese de um comando.*

Esta comutação deverá poder ser feita em determinados pontos da própria construção de um comando, sendo particularmente útil a comutação de modo não-assistido para modo assistido. Deste modo, sempre que o utilizador esquece a sintaxe concreta a utilizar, ou não sabe como introduzir os valores a associar a um argumento estruturado (por exemplo um registo), tem a possibilidade de invocar o modo assistido. Este esquema de comutação coloca alguns problemas de implementação já que, obviamente, passam a existir duas representações internas que devem ser mantidas consistentes para que se possa comutar entre elas.

? ***Memória****: o modelo de interacção, ao basear-se num paradigma de edição, deverá privilegiar a edição sobre o último comando sintetizado, exibindo portanto capacidade de memorização.*

Esta propriedade vai de encontro à necessidade de apoiar o utilizador na realização de actividades de carácter repetitivo, procurando que este possa

reu-tilizar comandos anteriormente introduzidos, a partir dos quais, por edição, pode construir o comando pretendido. O grau de memorização (ou *história*) é uma decisão de implementação.

São estas as seis características fundamentais do modelo MASS, quer este se aplique a sistemas já existentes ou a novos sistemas. O modelo visa claramente

a diminuição da distância articulatória na interacção, ao interpor entre o utili-zador e a linguagem da aplicação uma linguagem de interacção simples, de fá-cil articulação, e baseada num paradigma que, em termos cognitivos, é facil-mente assimilável, que é o paradigma da construção por edição passo-a-passo. Do ponto de vista da distância observacional, e apenas no que diz respeito à fa-se transaccional da interacção, ou seja, até que algum comando seja sintetizado e passado à aplicação, o modelo é observacionalmente simples dado basear-se em navegação estrutural. Esta é implementável usando um estilo de interacção muito simples e comum, uma hierarquia de menus sensíveis ao contexto.

A adopção de modelos de interacção baseados em assistência sintáctica não é uma ideia nova, podendo ser encontrados trabalhos semelhantes na área das linguagens de comandos [Ukelson e Rodeh 86], das bases de dados[4] [Cha 91], dos ambientes interactivos [Dewan e Solomon 87] e, naturalmente, dos editores estruturados [Minör 90]. No entanto, a maioria destes trabalhos ou não trata ou trata de forma incompleta os aspectos relacionados com a semântica estática, associada à validação de dados, e com a assistência semântica.

Assim, o MASS, com todas as características enunciadas, apresenta-se como um modelo de interacção por edição estrutural que não se limita aos aspectos de assistência sintáctica, importantes mas insuficientes tanto mais que sintaxe válida é uma propriedade também dependente do contexto, ou seja, da semânti-ca. A assistência semântica é fundamental no modelo já que só a sua considera-ção, por muito onerosa que venha a ser a sua efectiva implementação, possibili-ta a criação de IU que, satisfazendo aos requisitos do modelo, exibem um com-portamento diferente do usual, pois, entre outras características, permitem uma abordagem "profiláctica" à ocorrência de erros.

Note-se finalmente que o MASS possui algumas características que têm a ver com a consideração de um modelo simples e genérico do utilizador. No entanto, enquanto que para modelos mais complexos se adopta uma representação explí-cita (cf. o sistema a apresentar no capítulo 9), propriedades mais genéricas po-dem ser introduzidas de forma explícita em modelos mais simples. No MASS, ca-racterísticas de comportamento genéricas

[4] sem esquecer o hoje "histórico" Modo Assistido do DBASE III.

dos utilizadores são consideradas mas representadas de forma implícita, i.é., contidas nas propriedades do modelo.

A natural tendência para comportamento erróneo do utilizador é tratada recorrendo a assistência sintáctica e semântica. A imprevisibilidade do compor-tamento do utilizador é limitada ao não-determinismo controlado de ter que ser realizada uma escolha dentre várias possíveis e válidas. O conhecimento vago sobre a linguagem de interacção é também contemplado, não só pela já referida assistência sintáctica, como também pela aceitação de comandos incompletos quanto à definição dos seus argumentos, incompletude resolvida quer pela con-sideração de valores por omissão quer pela invocação de interacção para a sua instanciação completa.

6.1.3.- Aspectos de Implementação.

Consideremos em primeiro lugar como a implementação do MASS pode ser feita sobre uma aplicação tradicional, baseada em interacção usando uma linguagem de comandos, seja esta do tipo "batch" puro, baseada num interpretador de li-nhas de comandos[5], ou mesmo baseada em menus e "forms". Em qualquer dos casos, e para que o exemplo tenha sentido, admite-se que a interacção existente com a aplicação não possui as características de assistência do MASS.

Assim, e qualquer que aquela seja, admite-se que é recuperável, ou seja, que se pode facilmente eliminar tal código, substituindo-o pelo código do MASS. Em tais condições, o código do MASS torna-se um nó que é colocado entre o uti-lizador e o, eventual, "parser" da linguagem de comunicação com a aplicação. Tendo esta linguagem uma gramática bem definida, é então possível construir, com base na mesma, uma nova linguagem de interacção com o utilizador que, baseando-se em tecnologia de interacção mais actual, possibilite a construção de frases da linguagem de interacção com a aplicação, no mínimo, sintactica-mente correctas. A assistência semântica depende, neste caso, da existência prévia, ou da possibilidade de extensão da camada computacional nesse senti-do, de funções que permitam indagar o estado da aplicação sempre que ao nível da IU se necessitar de informação semântica ou contextual, que é fundamental para a efectiva implementação de tal tipo de assistência.

Admitindo como verificáveis todas estas condições, então o nodo correspon-dente à introdução do MASS para a aplicação, surge como um componente inter-médio que passa a aceitar, em modo "lazy" e assistido, as entradas do utiliza-dor, e as faz corresponder a uma frase aceitável pelo "parser", e que a este será comunicada.

---

[5] *command-line interpreter*.

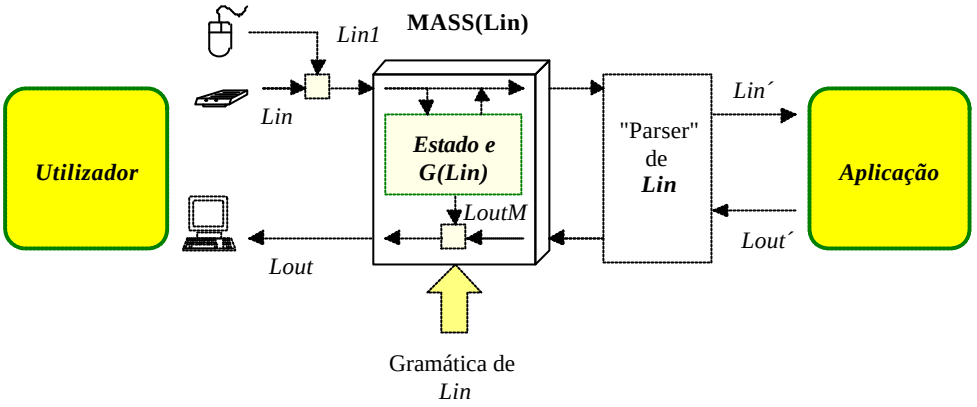

Fig. 6.1 - Modo Assistido: Recuperação Tecnológica.

Esta perspectiva é ilustrada na figura 6.1, onde as entradas do utilizador, realizadas em modo "lazy" sobre um conjunto contextualmente válido de opções, são intersectadas pelo módulo MASS, que possui uma representação interna da linguagem de interacção efectiva, permitindo assim que um dado comando cor-rectamente sintetizado, seja passado ao "parser" da aplicação para execução.

Considere-se que na figura 6.1 L_{in} representa a linguagem interactiva com base em comandos previamente existente como veículo de comunicação com a aplicação. Formalmente, esta é a linguagem L_G[6]. Seja L_{in1} a linguagem de meta-comandos associada ao modo assistido, linguagem simples, basicamente com-posta pelos símbolos que correspondem às acções de navegação estrutural, por exemplo { ? , ? , ?, ?, ?, ?, ?, ... }, e aos valores introduzidos para certas variá-veis.

Esta linguagem possui características diferentes das de uma convencional linguagem de comandos ou de programação, dado que quer o significado indivi-dual de cada um dos seus símbolos terminais, quer a validade sintáctica e o si-gnificado de qualquer estruturação destes em frases, é dependente do contexto em que tais símbolos são recebidos. Por exemplo, a frase "? ? ? 10 35" pode, em certos contextos ser construída enquanto que noutros não. Por outro lado, e em função do contexto em que foi introduzida, assim lhe será atribuído um signifi-cado, não possuindo portanto um significado único. Por outro lado, a linguagem é bastante redundante, o que beneficia o utilizador que passa a possuir diversas formas de exprimir as acções desejadas. Por exemplo, e admitindo que um me-nu circular com três opções é apresentado ao utilizador com a primeira opção seleccionável à partida, são frases semanticamente equivalentes, num exemplo finito, as seguintes, que correspondem a várias formas de seleccionar a primeira opção do menu: "?", "? ? ?", "? ? ? ?", "? ? ? ? ?".

[6] leia-se a linguagem gerada pela gramática G.

Deste modo, torna-se bastante complexo, e sem nenhum sentido particular, associar uma gramática à linguagem L_{in1} dado esta ser, fundamentalmente, um conjunto de *tokens* sem estruturação relevante e cuja interpretação individual é dependente do contexto em que tal símbolo surge, contexto este possivelmente modificado símbolo a símbolo.

No entanto, e dado que se pretende manter a linguagem original, para que o utilizador possa comutar entre esta e o MASS a qualquer momento, então a lin-guagem total de entrada do sistema, seja ela designada por L_{input}, vai passar a ser a reunião da linguagem L_{in}, que é a linguagem L_G gerada pela gramática G, com a linguagem L_{in1}. Representaremos esta linguagem de entrada como,

$$L_{input} = L_{in} \; ? \; L_{in1}$$

onde o símbolo ?visa indicar que a linguagem de entrada passa a ser uma se-quência de uma ou mais frases da linguagem L_{in} seguidas de uma ou mais fra-ses da linguagem L_{in1}, ou uma sequência de uma ou mais frases da linguagem L_{in1} seguidas de uma ou mais frases da linguagem L_{in}. Trata-se portanto de um entrelaçamento[7] de duas linguagens, com pontos bem definidos de comutação de uma para outra.

A expressão indica claramente a existência de dois *modos* de interacção. Sendo *modos* contextos particulares de interpretação, torna-se agora perceptível que ainda que a linguagem de entrada entrelace frases duma e doutra lingua-gem, os pontos de comutação definidos correspondem também à comutação de interpretadores das frases.

Ainda que muitas críticas possam ser feitas à existência de *modos* em siste-mas interactivos enquanto elementos potencialmente perturbadores dos proces-sos cognitivos e comportamentais do utilizador, modos em geral conotados com a noção de contexto e de linguagem a empregar em tal contexto, mesmo nas IU baseadas em manipulação directa[8], supostamente sem modos[9], modos podem ser reconhecidos em cada uma das classes de objectos visualizados.

No caso do MASS, dois *modos* completamente distintos poderão ser usados, sob controlo do próprio utilizador, o que se traduz na possibilidade de os meca-nismos de diálogo poderem ser convenientemente ajustados pelo utilizador ao seu próprio conhecimento da interacção com o sistema (cf. utilizador experiente e menos experiente).

Por outro lado, a linguagem de saída L_{output} vai passar a ser também a reunião da linguagem de saída já existente L_{out} e da linguagem de saída

---

[7] *interleaving.*

[8] onde cada objecto interactivo representa um mini-contexto de diálogo dado apenas aceitar certas acções do utilizador.

[9] *modeless.*

produ-zida pelo MASS. De facto, dado que após a síntese de um comando este é passa-do à aplicação, esta deverá responder ao comando da mesma forma como até à introdução do MASS respondia. Porém, a introdução do MASS vai fazer com que objectos relativos ao *retorno semântico,* a ser produzido em resultado dos meta-comandos accionados pelo utilizador, devam igualmente surgir na saída (L_{outM}). Estes objectos podem ser objectos de apresentação, mensagens de erro, ou ou-tras mensagens geradas a nível do controlador do MASS, ou até a "string" corres-pondente ao "unparsing" da estrutura interna que representa o estado actual da instanciação do comando que está a ser sintetizado. Assim, a linguagem de saí-da considerando o modo MASS pode também ser representada como,

$$L_{output} = L_{out} \; ? \; L_{outM}$$

Um dos primeiros exemplos de recuperação tecnológica usando o MASS é reportado em [Creissac e Martins 92] onde o MASS foi aplicado à linguagem fun-cional de tipo Lisp desenvolvida na Universidade do Minho e designada *xmetoo* [Pinto 89], permitindo que todos os programas (protótipos) desenvolvidos na lin-guagem passassem a possuir uma IU assistida em vez da usual interface orien-tada à linha de comando.

A especificação formal das características do MASS num contexto onde este é utilizado como modelo de interacção para novas aplicações foi realizada no capí-tulo anterior, tendo por base a utilização de arquétipos como representações dos objectos a construir. A especificação formal do MASS no contexto da recuperação tecnológica de aplicações já existentes, tendo por base uma dada linguagem de interacção descrita sob a forma de uma gramática, foi já realizada, podendo ser encontrada em [Pinto 87], tendo sido, a título de exemplo, desenvolvido um pro-tótipo da aplicação do MASS sobre MS-DOS, criando-se um MS-DOS assistido[10].

Como se referiu no ínicio desta secção, o MASS é um modelo conceptual da interacção que propõe um conjunto de princípios a serem seguidos na concep-ção e implementação de IU. As características e propriedades do modelo MASS podem ser satisfeitas mesmo que o desenvolvimento seja realizado de forma ma-nual. Porém, a construção de "ferramentas" de apoio à geração de IU em modo MASS é de grande importância para a facilidade de aplicação do modelo.

O formalismo que se apresenta na secção seguinte, os *Guiões de Interacção,* permitem não só a especificação de controladores do diálogo satisfazendo aos princípios do MASS mas também a sua geração automática, desta forma facili-tando a construção de protótipos da IU.

[10] curiosamente, à data não existiam utilitários do tipo do DOS-SHELL.

6.2.- Guiões de Interacção.

6.2.1. - Introdução.

O formalismo dos *Guiões de Interacção (GI)*, apresentado numa versão inicial em [Martins et al. 90], tem por objectivo descrever, a um alto nível de abstracção, o controlo do fluxo da interacção que o controlador do diálogo deve realizar, para fazer corresponder aos eventos recebidos do utilizador as alterações de estados da IU e da aplicação que lhes são associadas.

Um GI vai descrever, de uma forma não procedimental, todos os passos que devem acontecer no diálogo, seja a aceitação de eventos do utilizador, seja a alteração de estado da IU, seja a comunicação no sentido IU-utilizador, seja até alguma informação sobre erros, ao nível apenas da interacção, até que possa ser construída, sintetizada, "manufacturada" uma *primitiva computacional*, i.é., um comando da aplicação com todos os seus argumentos, tendo sido garantido pe- lo diálogo prévio que o mesmo vai corresponder a uma invocação correcta da correspondente rotina da aplicação, correcção sintáctica e semântica.

Ainda que tal não seja impeditivo de outro tipo de utilizações, como aliás se procurará exemplificar adiante, deve desde já salientar-se que a concepção do formalismo dos GI foi realizada tendo por objectivo principal a especificação de controladores de diálogo satisfazendo o modelo MASS, o modelo de interacção proposto. Assumindo, de momento, esta perspectiva como primordial, cada GI é uma abstracção de controlo e de dados que é posicionada entre os diálogos externo e interno para realizar a sua devida sincronização, seguindo um modelo de interacção baseado na edição estrutural de comandos e dados da aplicação.

Cada GI pode, assim, ser visto como um mini-MASS, dado ter por missão es-pecificar uma interface assistida específica para cada um dos comandos da apli-cação que podem ser sintetizados. Este é o grau de granularidade que, numa primeira fase, é usado para especificar o diálogo.

Porém, a síntese de um comando implica a síntese dos seus argumentos, que, do ponto de vista das interacções mais elementares necessárias, pode ser um processo complexo, não só porque estes podem ter estruturas complexas, mas também porque os seus valores devem ser válidos. Enquanto que em certos casos a possibilidade de erro tem que ser considerada, dado que a introdução do valor foi realizada de forma livre, tal validação é desnecessária se, usando as informações sobre o estado da camada aplicativa (*o contexto semântico*), a leitu-ra for feita corresponder a uma escolha correcta entre várias escolhas correctas possíveis.

Ainda relativamente aos dados, pretende-se que a notação possua suficiente eficácia expressiva para, de uma forma simples, poder especificar diferentes mo-dos de sintetizar tais argumentos, por exemplo,

sequencialmente, segundo uma ordem arbitrária e até de modo paralelo, o que é relevante tendo em atenção que, por vezes, a ordem destas entradas deve obedecer a certas restrições, o que significa que a sua ordem de leitura não é livre.

Por outro lado, para certos tipos de dados mais comuns, poder-se-ão desen-volver esquemas de interacção pré-definidos e parametrizados, que possibilitem considerar, a menos do necessário parâmetro que será um invariante de dados, que tal interacção possa ser considerada atómica (como se fosse um *evento*)[11].

Ao colocarem-se os GI entre os dois diálogos, é assumido um controlo do ti-po *balanceado*, i.é., controlo realizado numa componente independente. Esta perspectiva deve ser também a perspectiva do especificador do diálogo ao usar os GI. Antes de mais ele está a especificar como o controlador medeia e sincro-niza os dois diálogos, ainda que com algum grau de abstracção.

Finalmente, em termos da relação prática entre as decisões de grau de mo-dularidade dos GI e da sua utilização, a simplicidade oferecida é grande, dado que é estabelecida uma clara relação de um para um entre os GI e as operações a invocar na camada computacional. A especificação da estruturação correcta destas invocações seguindo uma ordem temporal, é realizada numa segunda fase, onde *combinadores* são usados sobre os GI já especificados. Esta estratégia "bottom-up" é talvez, metodologicamente, a mais natural, ainda que uma abor-dagem "top-down" possa também ser usada.

Para além desta adequação dos GI para especificar diálogos seguindo o mo-delo MASS, como se demostrará posteriormente através de um conjunto repre-sentativo de exemplos, pretende-se que os GI possuam um poder expressivo ele-vado, uma grande clareza e modularidade, que facilitem a sua utilização na es-pecificação de diálogos com maior tipo de exigências. Excepção é feita aos diá-logos por manipulação directa, nos quais a característica mais relevante a ser especificada em relação com os eventos é o retorno semântico ao nível da apre-sentação[12]. Tal como os GI, a maioria das notações para a especificação de diá-logos não possui mecanismos para a especificação de apresentações, dado estas terem que ser realizadas a muito baixo nível.

6.2.2.- Propriedades.

Antes de compararmos a notação dos *Guiões de Interacção*, em termos da suas capacidades expressivas e características, com algumas das notações que no

---

[11] Como se verá, esta possibilidade foi mesmo experimentada no sistema GAMA que se apresenta no capí-tulo 8.

[12] *presentation feedback*.

ca-pítulo 4 foram descritas para a especificação de diálogos, comecemos então por apresentar as suas propriedades características.

No desenvolvimento da notação dos *Guiões de Interacção* foram diversas as preocupações de a prover de construções com capacidade expressiva ajustada à especificação de IU actuais, sem no entanto esquecer os requisitos de origem, ou seja, a especificação natural de interfaces satisfazendo o MASS. Note-se, a título de exemplo, que ainda que o modelo MASS não requisite IU multi-caminho, os GI possuem capacidade expressiva para as especificar e prototipar.

Vejamos então as principais propriedades que se alegam para a notação, e que, mais adiante, procuraremos demonstrar com vários exemplos:

- *Composicionalidade*, ou seja, a possibilidade de, usando um conjunto de combinadores, se poderem construir especificações complexas a partir de GI mais atómicos;

- *Reutilização local*, ou seja, a possibilidade de guiões mais elementares poderem ser *importados* para a descrição de GI mais complexos, bem como também *incluídos*, partilhando neste caso o espaço de variáveis do inclusivo;

- *Concorrência dos sub-diálogos*, expressável quer como diálogos síncronos quer como assíncronos;
- *Especificação de comportamento externo* sob a forma de especificação dos traços aceitáveis, ou seja, das sequências de eventos aceitáveis pelo GI;

- *Especificação declarativa do comportamento interno* sob a forma de triplos *evento-condição-acção*, representando as transições de estado internas ao controlador do diálogo;

- *Tratamento de eventos assíncronos*, que são os eventos que resultam de acções do utilizador de carácter não-determinista (cf. “interrupts” na IU), e que implicam uma resposta imediata do controlador, qualquer que se-ja o estado de interacção em que se encontre. Exemplos destes eventos são os eventos CANCEL, OK, RESET e APPLY, cuja semântica se apresenta posteriormente;

- *Interacção para síntese de dados*, que são os argumentos das operações que devem ser invocadas na camada computacional, considerando que qualquer restrição na ordem da síntese destes argumentos deve ser con-siderada, e ainda que, para alguns tipos de dados, objectos de

interac- ção[13] específicos podem ser considerados usando a tecnologia de inter-acção interacção disponível (cf. “widgets”, etc.);

? *Tratamento de erros*, ou melhor, especificação de situações de excepção, resultantes da não observação completa do modelo MASS, ou seja, pela ocorrência de erros que apenas podem surgir caso a implementação da IU não use a informação semântica disponível na aplicação, ignorando pois certas directivas de assistência semântica sugeridas pelo MASS;

? *Localização dos efeitos da interacção*, através da declaração explícita dos efeitos de cada evento no estado interno da aplicação, da apresentação, ou do controlador do diálogo;

? *Proximidade da especificação da camada computacional*, através de uma representação dos objectos da aplicação de forma tão abstracta quanto a que resulta da sua especificação formal;

? *Valores por Omissão*, que podem ser definidos em cláusula apropriada para determinados argumentos, e que permitem a sua utilização caso nenhuma interacção para a sua alteração tenha sido realizada, de acordo aliás com o sugerido pelo MASS;

? *Abstracção da apresentação*, propriedade que pode ser tomada apenas como estratégia de não comprometimento com semântica de baixo nível, mas que possibilita que, na prática, se possam ter *apresentações*[14] dife-rentes para o mesmo objecto de interesse;

? *Possibilidade de prototipagem*, ou seja, por mais complexa que possa ser a integração na notação das propriedades anteriores, esta é ainda exe-cutável, podendo construir-se e executar-se protótipos das especifica-ções, o que se considera de valor indispensável[15].

6.2.3.- Sintaxe e Semântica Estática.

Apresenta-se nesta subsecção a sintaxe abstracta dos GI, a semântica estática relativa à especificação da aplicação da qual derivam, e ainda a forma como cada cláusula sintáctica se associa às propriedades descritivas apresentadas na secção anterior, descrições que se escrevem em CAMILA.

---

[13] *interactors*.

[14] *views*.

[15] As Redes de Petri que, como se verá, não só darão semântica operacional aos GI, como serão também automaticamente geradas a partir das especificações dos GI, serão a garantia da executabilidade destes.

Para que esta descrição possa ser completa, em particular relativamente à semântica estática, devemos começar por modelar a estrutura da especificação da *aplicação* e da especificação do *controlador do diálogo*. A especificação da aplicação vai ser modelada tomando em consideração apenas os constituintes que nos interessam para que se possa fazer a validação da especificação do con-trolador. A este nível de interesse e de abstracção, podemos então considerar a especificação da aplicação como sendo o tuplo:

```
AplSpec ::   Vars: IdVar -> IdType    ; variáveis da aplicação e tipos
             Ops : IdOp -> FuncOp     ; nomes e aridades das operações
             Pres : IdOp -> FuncOp    ; nomes das pré-condições
IdType, IdOp = SYM
IdPre, IdVar = SYM
FuncOp ::    Inp : IdType-list Out : IdType
```

onde se distinguem a função finita que faz corresponder a cada identificador de variável da especificação o seu respectivo tipo, a função finita que a cada iden-tificador de operação associa a respectiva funcionalidade e o conjunto dos iden-tificadores das pré-condições.

Por seu lado, uma especificação do controlador do diálogo usando guiões é, em essência, uma correspondência entre os identificadores e as descrições dos guiões de interacção.

```
CDSpec = GIName -> GIDescr    ; nomes e descrições dos guiões
GIName = SYM
GIType = [Decision | Synth | ValSynth];
```

A descrição de um guião, identificada por GIDescr e que se detalhará a se-guir, indicará qual o *tipo do guião*, um de três tipos possíveis, designadamente, um guião do tipo *Decision*, do tipo *Synth*, do tipo *ValSynth* ou mesmo sem tipo particular definido. Ainda que sejam de tipos diferentes, a estrutura sintáctica dos vários tipos de guiões é igual, existindo apenas diferenciação semântica.

Os guiões do tipo *Decision* são utilizados na especificação da interacção de mais alto nível, ou seja, da interacção que corresponde à realização de uma se-lecção dentre várias opções possíveis. As opções possíveis correspondem à acti-vação de outros guiões de qualquer tipo.

Guiões do tipo *Synth* são os guiões cujo resultado final, caso se concluam de forma natural, ou seja sem interrupções, consiste na síntese de um comando da aplicação, com os seus respectivos argumentos, e da respectiva

invocação. Estes são os únicos guiões que podem produzir alterações no estado da aplicação.

Os guiões do tipo *ValSynth* servem para invocar operações de consulta do estado da aplicação, sintetizando *valores de variáveis* necessários para que o contexto de interacção se mantenha actualizado, e retorno semântico possa ser oferecido ao nível da apresentação.

Apresenta-se de seguida (cf. Guião 1) um primeiro exemplo de um GI, que pretende especificar o diálogo necessário para invocar a operação de inserção de uma palavra lida num dicionário, *InsPal(pal, sig)*. Ainda que algumas claúsulas sejam neste exemplo apresentadas em branco, e, como tal, pudessem ser omiti-das, como primeiro exemplo decidiu-se apresentá-las, com o objectivo de mos-trar todas as cláusulas que podem ser utilizadas nos GI.

O guião é apresentado usando a sintaxe concreta definida, com o objectivo de que possa servir de referência ao longo da posterior apresentação. Note-se desde já que a descrição de cada guião é dividida, conforme o exemplo que se apresenta, em duas secções: as *declarações estáticas* e a *descrição do comporta-mento*. Tem-se então:

GIDescr :: Decls: GIDecls Behav: GIBehav

As declarações estáticas introduzem todos os identificadores que irão ser usados no guião, enquanto que nas declarações de comportamento se introdu-zem as cláusulas que descrevem todo o comportamento interactivo sob controlo do guião.

Guião 1 (Exemplo):

DefGI *GInsPal*
Declarations
TYPE *Synth*
SYMBOL *{ InsPal, InserePal }*
ARGS *pal: Pal ; sig: Sig*
VAR-UI
VAR-APL
VAR-CTRL
SUBGI
EXTERNAL

Behaviour
CONTEXT *not(FullDic( ))*
INIT *pal = ""; sig = ""*

```
EVSEQ input(pal) . input(sig)
TRANS
    input(pal): (pal != "") && not (ExistPal(pal)) => Null
                EXCEP   pal == ""   => out("Palavra Nula !")
                        pal != ""   => out("Já existe !")
    Ok:
    Cancel:
EXEC InsPal(pal, sig)
EndGI
```

Descrevem-se de seguida cada um destes blocos e as respectivas cláusulas, apresentando os invariantes correspondentes à sua correcção.

Declarações Estáticas.

As declarações estáticas constam de várias cláusulas, a maioria das quais são opcionais, podendo até ser omitidas em certos tipos de guiões.

```
GIDecls :: [ValT: ValType]              ; tipo de valor devolvido caso o
           Symbol : SYM-set             ; guião seja de tipo ValSynth.
           Type : GIType
           Args: IdVar -> IdType            ; declarações
           VAR-UI: IdVar -> IdType
           VAR-CTRL : IdVar -> IdType
           VAR-APL ::  Ldecl: IdVar -> IdType   ; declarações
                       Atribs: IdVar -> IdVar   ; atribuições
           Extern : GIName-set
           SubGi : GIName-set
```

A cláusula TYPE indica o tipo do guião. A cláusula SYMBOL apresenta um conjunto de símbolos, ou *tokens*, que constituem a classe de equivalência de identificadores que podem ser reconhecidos como associados ao nome interno da operação da aplicação. A cláusula ARGS serve para declarar as variáveis, e seus tipos, que são variáveis locais ao guião, ou seja, cujos valores podem ser acedidos, para escrita ou leitura, localmente ao guião. A correcção semântica da cláusula depende da obediência ao invariante seguinte,

```
FUNC inv-ARGS(gid: GIDescr, aplSpec: AplSpec) : Bool
; a declaração dos tipos dos argumentos do guião está correcta
RETURNS
        let (  types = ran(Args(Decls(gid))) )
        in
           types ?  ran(Vars(aplSpec));
```

que especifica que todos os identificadores de tipos usados nas declarações das variáveis na cláusula ARGS devem pertencer ao conjunto dos identificadores de tipo da especificação da camada computacional.

Outras declarações de variáveis podem ser utilizadas nos guiões. A cláusula VAR-UI é utilizada para definir variáveis cujos conteúdos são necessários para a sincronização com a apresentação. Correspondem a variáveis da apresentação a que o GI deve aceder para actualização dos seus valores. A cláusula VAR-APL declara o conjunto de variáveis da aplicação acessíveis, apenas para consulta, e respectivos tipos. Pode declarar também um conjunto de atribuições entre variáveis do controlador e variáveis da aplicação. A correcção semântica desta cláusula passa por garantir que todos os identificadores de variáveis usados tenham sido declarados, e que, adicionalmente, para cada um deles o seu tipo esteja correcto. O invariante correspondente é especificado como:

```
FUNC inv-VAR-APL(gid: GIDescr, aplSpec: AplSpec) : Bool
; a declaração das variáveis da aplicação acessíveis está correcta
RETURNS
        let ( FFDecl = Ldecl(VAR-APL(Decls(gid))),
              FFAt = Atribs(VAR-APL(Decls(gid))),
              FFVarSpec = Vars(aplSpec),
              AplVarIds = dom(FFDecl),
              ParesAt = { (idvctrl, FFAt[idvctrl]) | idvctrl ? dom(FFAt)
},
              IdsCtrl = dom(FFAt), IdsApl = ran(FFAt) )
        in
          (1)   AplVarIds ? dom(FFVarSpec) ?
          (2)   all( idv ? AplVarIds : FFVarSpec[idv] ==
FFDecl[idv]) ?
          (3)   IdsCtrl ? AplVarIds ? IdsApl ? dom(FFVarSpec) ?
          (4)   all( (idvctrl, idvapl) ? ParesAt :
                    FFVarSpec[idvctrl] == FFVarSpec[idvapl] );
```

O invariante especifica que uma cláusula VAR-APL estará correcta se todos os identificadores de variáveis da aplicação forem válidos (1), i.é., tiverem sido definidos na especificação; se para cada um deles o tipo que lhe foi associado na cláusula é correcto relativamente ao especificado (2), e, caso atribuições entre variáveis do controlador e variáveis da aplicação sejam declaradas, então, os identificadores de ambas devem existir (3) devendo ser compatíveis os seus tipos (4) declarados.

Finalmente, na cláusula VAR-CTRL declaram-se variáveis locais ao controla-dor. Não tendo sido definido um sistema de tipos particular para os guiões, os tipos usados nas declarações destas variáveis locais são os tipos pré-definidos para a linguagem de especificação. Não se impõem restrições

particulares aos nomes das variáveis, devendo no entanto tais identificadores ser disjuntos dos usados noutras cláusulas por questões de legibilidade.

```
FUNC inv-VAR-CTRL(gid: GIDescr) : Bool
; a declaração dos nomes das variáveis da cláusula VAR-CTRL está correcta,
; ou seja, os seus identificadores são disjuntos dos usados noutras cláusulas
RETURNS
        let ( CtrlVarIds = dom(VAR-CTRL(Decls(gid))),
              UIVarIds = dom(VAR-UI(Decls(gid))),
              AplVarIds = dom(Ldecl(VAR-APL(Decls(gid)))) )
        in
           CtrlVarIds ? (UIVarIds ? AplVarIds) == { };
```

As cláusulas SUBGI e EXTERNAL correspondem à utilização dos mecanismos de *inclusão* e *importação* de guiões. Declaram-se na cláusula SUBGI todos os identificadores de guiões que são sub-guiões do guião que se descreve, devendo as descrições daqueles estar contidas no guião principal. Estes sub-guiões têm, por herança ou regras de visibilidade, acesso ao espaço de variáveis locais do guião onde se incluem. Na cláusula EXTERNAL declaram-se todos os guiões que podem ser invocados para uma melhor estruturação do guião principal, mas que não vão partilhar o seu espaço de variáveis. É portanto um mecanismo de importação que possibilita reutilização de guiões, razão pela qual a interferência tem que ser controlada sob pena de se ter grande complexidade semântica, o que se consegue evitar.

Comportamento.

A especificação do comportamento de cada guião de interacção é feita através de cinco cláusulas principais, conforme se define a seguir.

```
GIBehav :: Init : IdVar -> ExpValue          ; inicialização de variáveis
           Contxt : [BoolExp]                ; contexto
           EvSeq : ExprComp                  ; sequência de eventos
           Trans : TrDescr                   ; transições de estado
           Exec : [ExecDescr]                ; invocação da operação
```

A cláusula INIT é utilizada para atribuir valores iniciais aos argumentos ou variáveis locais aos guiões. Deve obedecer ao invariante seguinte, que especifica que os identificadores declarados na cláusula INIT devem estar

contidos na reu-nião dos identificadores de argumentos e dos identificadores das variáveis do controlador.

```
FUNC inv-INIT(gid: GIDescr) : Bool
; a declaração dos nomes das variáveis a inicializar está correcta, ou seja,
; ou são argumentos ou são variáveis do controlador

RETURNS
        let ( ids = dom(Init(Behav(gid))),
              args = dom(Args(Decls(gid))),
              ctrls = dom(VAR-CTRL(Args(Decls(gid)))),
              locais = args ? ctrls )
        in
           ids ? locais;
```

A cláusula CONTEXT é um predicado que especifica a condição de contexto, relacionada com o estado actual da IU e, principalmente, da aplicação, ou seja, é uma regra lógica, de valor variável no tempo, que determina se num dado mo-mento o guião pode ou não ser activado. No exemplo apresentado, apenas se o resultado da expressão *not(FullDic())*, ou seja, se o resultado da negação lógica do resultado da invocação da função da aplicação *FullDic()*, que testa se o referi-do dicionário está vazio, for verdadeiro, o guião pode ser activado. A inclusão da cláusula de contexto nos guiões de uma forma explícita, não oferece qualquer dificuldade definicional, dado que esta resulta directamente da análise da espe-cificação da camada computacional. Em geral, trata-se da simples transcrição da pré-condição, ou parte desta, da operação a sintetizar pelo guião.

É no entanto necessário garantir que a correcção desta cláusula se verifica, o que passa por se usar a linguagem correcta para a sua expressão, por invocar funções existentes e, para cada uma destas, utilizar como argumentos valores ou variáveis de tipos correctos relativamente à sua funcionalidade definida.

A completa correcção da especificação do guião relativamente à especifica-ção da camada computacional, resulta pois da conjunção das correcções parci-ais de cada uma das cláusulas, tal como definidas pelos correspondentes inva-riantes. O invariante geral aplicável a cada guião poderá então escrever-se co-mo:

```
FUNC inv-APLGID(gid: GIDescr, aplSpec: AplSpec) : Bool
; o guião está correcto
RETURNS
        inv-ARGS(gid, aplSpec) ? inv-VARS-APL(gid, aplSpec) ?
        inv-INIT(gid) ? inv-CONTEXT(gid, aplSpec) ?
        inv-EVSEQ(gid, aplSpec)
```

A cláusula EVSEQ especifica quais os eventos externos ao guião necessários à sua transição interna de estado, até que possa atingir o seu estado final de um modo temporalmente estruturado. Trata-se de uma cláusula de especifica-ção de comportamento. Esta descrição de comportamento faz-se, de forma con-creta, compondo eventos usando combinadores para sequenciação, para concor-rência síncrona, para concorrência assíncrona, para repetição e para a expres-são de alternativa. Nesta cláusula podem usar-se quer identificadores de even-tos quer identificadores de guiões. Sendo os eventos atómicos, vejamos os casos mais interessantes em que as expressões utilizadas envolvem combinadores de identificadores de guiões, e o seu respectivo significado.

exp1 . exp2 ; sequência dos comportamentos determinados pela
; *exp1* seguida dos comportamentos dados por *exp2*.

exp1 + exp2 ; alternativa entre os comportamentos especificados
; pela *exp1* e pela *exp2*.

exp* ; repetição, 0 ou mais vezes, da expressão de compor-
; tamento especificada por *exp*;

exp1 || exp2 ; paralelismo síncrono entre o comportamento defini-
; do pela *exp1* e o definido pela *exp2*, decorrendo as
; actividades da *exp1* e da *exp2* em paralelo mas com
; garantia de sincronização final.

exp1 | exp2 ; paralelismo assíncrono entre os comportamentos
; definidos pela *exp1* e pela *exp2*, terminando esta
; expressão quando um deles terminar.

exp1 [exp2] ; o comportamento especificado pela *exp1* pode, em
; qualquer ponto da sua execução, ser interrompido
; pela *exp2*; se tal acontecer, *exp2* realiza o seu com-
; portamento até ao fim, só então podendo de novo
; realizar-se o resto do comportamento de *exp1*.

Analisaremos de seguida algumas cláusulas EVSEQ indicando o respectivo significado. A cláusula de comportamento incluída no exemplo apresentado,

EVSEQ *input(pal)* **.** *input(sig)*

estrutura os eventos *input(pal)* e *input(sig)*, que são interacções para leitura de duas "strings", leituras consideradas elementares e daí serem consideradas eventos, segundo uma ordem sequencial. A cláusula de comportamento seguin-te,

EVSEQ *GInit* **.** *(GInsPal* **+** *GRemPal* **+** *GViewSign)****** **.** *End*

especifica um comportamento mais complexo, sendo neste caso usados os nomes dos guiões em representação dos eventos que os activam. Trata-se de um tipo de expressão de comportamento em geral associada a guiões do tipo *Deci-sion*, que se usam para guiar a interacção ao mais alto nível, e onde se realizam escolhas sobre as operações a executar, correspondendo a escolhas sobre os guiões nesse instante activáveis em função do contexto de interacção.

A expressão especifica um comportamento correspondente a *traços* que são obrigatoriamente iniciados pelo evento *GInit*, seguidos por zero ou mais repeti-ções dos eventos apresentados na alternativa, terminando com o evento *End*. São traços possíveis resultantes da expressão de comportamento apresentada,

< GInit, End> , < GInit, GInsPal, End > , < GInit, GRemPal, GInsPal, End >,
< GInit, GInsPal, GRemPal, GViewSig, ... , End >, etc.

Note-se que o traço *< GInit, GRemPal, GInsPal, End >*, ainda que seja a este nível de especificação um traço possível, irá provavelmente nunca acontecer, pois a camada semântica irá impôr a restrição de que sempre que um dicionário esteja vazio, a operação de remoção de uma palavra (guião *GRemPal*) não deve-rá poder ser activada.

De seguida, surge a cláusula TRANS onde se descrevem as transiçõe s inter-nas de estado do guião, e que correspondem à reacção especificada à ocorrência dos eventos. Esta especificação das transições internas de estado é realizada de forma declarativa através de um conjunto de regras, cada uma delas especifica-da como sendo, basicamente, um triplo, *evento-condição-acção*, opcionalmente com uma regra de tratamento de excepções.

```
TrDescr =   Ev : IdEvent -> InfTrans      ; identificador de evento
InfTrans :: Cond : [BoolExp]              ; condição
            Act : [ActionCode]            ; acções a realizar
            Exc : [ExcCode-list]          ; acções de erro ou outras
```

Em termos de sintaxe concreta, como pode ser observado no exemplo, as re- gras de transição consistem em estruturas da forma,

evento: condição ? acções [EXCEP acções]

onde se especifica que a ocorrência de um dado evento desencadeia a acção ou acções especificadas, caso a condição indicada seja verdadeira. Opcionalmente, caso a condição seja falsa, acções, mais de informação de erro do que de trata-mento destes, ou mesmo outras, podem ser executadas. A acção nula é repre-sentada pelo símbolo *Null*.

A cláusula EXEC é utilizada nos guiões que fazem a ligação à aplicação. Esta ligação pode ser feita quer para invocar funcionalidade que altera o estado da aplicação (caso dos guiões *Synth*) quer para invocar funcionalidade de consulta do estado da aplicação para transferência de dados (cf. os guiões *ValSynth*), correspondendo à necessidade de se possuir conhecimento de certos valores do estado da aplicação que devem ser transmitidos à camada de apresentação (o *retorno semântico*), ou que são importantes na percepção do actual contexto de interacção, que pode influenciar a transição de estados de qualquer guião.

Para além dos eventos declarados na cláusula EVSEQ, e que correspondem aos eventos "de interesse" do guião[16], eventos de carácter assíncrono, ou seja, que não fazem parte da estruturação temporal de eventos apresentada na cláu-sula, podem ser reconhecidos por alguns guiões. Os eventos assíncronos que são considerados na descrição de guiões são os seguintes:

CANCEL : *evento que permite cancelar o diálogo especificado pelo guião, devolvendo o controlo da interacção ao estado de interacção ante-rior à invocação do guião cancelado.*

OK : *quando este evento é declarado na cláusula TRANS, a expressão descrita em EXEC não é automaticamente avaliada após o fim da sequência de eventos em EVSEQ, mas apenas quando o evento OK for accionado. Esta declaração é equivalente a ter-se escrito o evento assíncrono OK em sequência, mas no final, com os eventos descritos em EVSEQ.*

RESET : *evento que permite reinicializar o diálogo especificado no guião. Este evento está permanentemente disponível.*

APPLY : *evento que é semanticamente equivalente ao evento OK, ainda que implique a reinicialização do guião, permitindo assim a sua ime-diata reactivação.*

6.2.4. - Semântica Operacional: Redes de Petri.

[16] Numa abordagem baseada em processos estes eventos seriam designados como o *alfabeto* do processo.

Para além da sintaxe abstracta e da semântica estática dos guiões, é também indispensável realizar a especificação do fluxo da sua execução, em particular quando combinados através dos combinadores apresentados. Como se referiu no capítulo 3, existem vários formalismos e notações capazes de especificarem comportamentos concorrentes, em particular, Petri Nets, CCS e CSP. Por outro lado, e conforme se sumariou no capítulo 4, várias são as notações criadas para a especificação de diálogos multi-caminho e, até, concorrentes puros. Muitas destas notações possuem uma semântica que é delegada na semântica definida para as notações em que se baseiam, sendo estas, quase invariavelmente CSP e CCS.

Seria igualmente atraente para o autor sediar a semântica das expressões de comportamento dos guiões num dos formalismos baseados em álgebras de processos, fosse ele CSP ou CCS. No entanto, e ainda que a nível da semântica de alto nível tal pudesse ser feito, uma característica importante dos guiões te-ria de, em tal caso, ser mascarada. Os *guiões de interacção não são processos,* mas tão só descritores de mini-controladores de diálogo, nestes especificados a um nível mais alto do que nos usuais processadores de eventos, também estáti-cos, mas não composicionais.

Pretendendo-se descrever a semântica operacional dos guiões, das cláusulas de expressão de comportamento destes e da sua composição, com preocupações até de que o formalismo empregue, para além de rigoroso, possa também servir de base à execução de tais especificações, então, num contexto como o actual em que não se trata de especificar processos, as Redes de Petri parecem ser o formalismo adequado dadas as suas reconhecidas capacidades para especificar fluxos de controlo ou até de informação.

Uma das desvantagens principais apontadas às Redes de Petri, a falta de modularidade nos modelos mais básicos, é ultrapassável considerando a sua estruturação em subredes. Por outro lado, a sua aplicação na especificação dos guiões é facilitada, dado que a abordagem usada é intrinsecamente modular.

Este grau de modularidade faz com que não seja necessário utilizar Redes de Petri muito complexas, quer em estrutura quer quanto ao tipo de informação a representar. Adoptaram-se assim as Labelled Petri Nets (LPNs)[Peterson 77], que são Redes de Petri com marcações[17] que, para além de especificarem o fluxo de controlo, tal como se pretende, permitem também um tratamento linguístico, pois são em geral associadas a aceitadores ou reconhecedores de linguagens. No entanto, para a sua utilização efectiva na definição operacional dos guiões, foi necessário adicionar-lhes mais alguma capacidade descritiva.

[17] *Marked Petri Nets.*

Em primeiro lugar foi acrescentada a possibilidade de se poder associar a uma dada transição uma *condição*. Mesmo que a transição possa "disparar" por estarem satisfeitas todas as respectivas regras de marcação, apenas se esta *condição* for verdadeira a transição poderá, efectivamente, "disparar". Estas *condições* permitem assim a introdução do tão importante *contexto dinâmico,* alterável no tempo, e que pode condicionar o fluxo de controlo pré-determinado.

Em segundo lugar, foi considerada a possibilidade de se interromper a exe-cução de uma rede para que uma outra rede possa ser executada por completo, só então sendo o controlo devolvido à rede inicial.

Apresenta-se a seguir a definição formal das Redes de Petri que vão ser aqui usadas que, porque se distinguem das Redes de Petri Etiquetadas usuais, quer por poderem ter transições condicionadas, quer por poderem ser interrompidas, iremos designar por *Redes de Petri Etiquetadas-Guardadas-Interruptíveis*[18].

Redes Etiquetadas-Guardadas-Interruptíveis.

Formalmente, uma Rede de Petri R, é um tuplo,

$$R = < P, T, F >$$

onde P representa um conjunto de estados, T é um conjunto de transições tal que P ? T = ? e F ? (P x T) ? (T x P) é a relação de fluxo, que inclui pares em (P x T) que representam as relações entre lugares e transições, os lugares e arcos de entrada, e pares em (T x P) que representam as relações entre transições e lugares, os lugares e arcos de saída.

Uma Rede de Petri Marcada é uma Rede de Petri à qual se associam funções de marcação, sendo representada pelo tuplo,

$$R_{PM} = < R, M_S, M_F, M_A >$$

onde R é uma Rede de Petri, e M_S, M_F e M_A são marcações, i.é., conjuntos de lugares que são subconjuntos de P. M_S representa a marcação inicial da rede. M_F representa a marcação final da rede, ou seja, a que se espera atingir, sendo M_A a marcação actual da rede[19].

---

[18] Entre os termos interrompível (de, pode ser interrompido ou suspenso) e interruptível (de, pode passar a interrupto ou suspenso) derivados do verbo *interromper*, ambos inexistentes no dicionário de português do autor, ao contrário das derivações semelhantes do verbo *corromper*, tais como corrompível e corruptível, ambas assinaladas, o autor, confrontado com a necessidade de traduzir o termo inglês *interruptible*, decidiu-se pela tradução mais lógica e próxima na língua portuguesa, ou seja *interruptível* (cf. corruptível).

[19] Se a cada lugar da rede se pretenderem associar zero ou mais *tokens*, as marcações devem passar a ser funções de tipo s ? ? , associando a cada lugar da rede o número de *tokens* nele contidos. No caso presente, e dado que um lugar ou está marcado ou não, uma marcação pode ser apenas um conjunto.

Considerando uma particular marcação M, diz-se que um lugar *p* ? P está *marcado*, logo activo, se *p* ? M, senão o lugar diz-se *não marcado*, ou inactivo. Diz-se que uma transição *t* ? T se encontra habilitada, i.é., pode disparar, se e só se, sendo,

Pin = { p ? P | (*p* , *t*) ? F } ; *lugares de entrada da transição*

e

Pout = { p ? P| (*t* , *p*) ? F } ; *lugares de saída da transição*

se verificar que

(? *p* ? *Pin* . *p* ? M) ? (? *p* ? *Pout* . *p* ? M)

ou seja, se todos os lugares de entrada da transição se encontrarem marcados, i.é., activos, e todos os lugares de saída não estiverem marcados, ou seja, estive-rem inactivos.

O disparo de uma transição *t* é designado um *passo elementar* da evolução da rede, e do que ela modela, representando-se em geral por M[*t* > M´, representando M a marcação anterior ao disparo de *t*, e em que *t* está habilitada, e M´ a marcação após o seu disparo. Em resultado do disparo da transição, uma nova marcação da rede passa a existir, M´, calculável a partir de M segundo a expressão

M´ = M - *Pin* ? *Pout*

Se existe uma transição *t* segundo a qual M[*t* > M´, então diz-se que M´ é uma marcação atingível[20] a partir da marcação M. Conforme se referiu no capí-tulo 3, se, segundo uma dada marcação, várias forem as transições habilitadas, a conflitualidade é resolvida de forma não-determinista.

A *sequência de disparos* de uma R_{PM} consiste na sequência dos disparos das suas transições, *w*, a que corresponde uma sequência de marcações que levam a rede de uma marcação M_1 a uma marcação M_n, o que se denota por

$M_1[w > M_n$

A execução de uma R_{PM} é definida por um algoritmo muito simples, que se resume a três passos principais:

1) A marcação da R_{PM} é inicializada com a marcação M_S;

2) Uma sequência de disparos de transições habilitadas, *w*, tem lugar;

3) A execução termina quando uma das seguintes condições se verificarem:

[20] *reachable*.

? Nenhuma transição estiver habilitada (situação de *deadlock*).

? A marcação final M_F tiver sido atingida.

Uma Rede de Petri Etiquetada[21] é uma R_{PM} particular que é definida como sendo o triplo,

$$R_{PE} = < R_{PM}, ?, ? >$$

em que ? representa um alfabeto finito de símbolos associados às transições e que pode incluir o símbolo ? (ou *null*), representativo da "string" vazia, e ? é a função que etiqueta cada transição com um desses símbolos ? : T ? ?. Deste modo, uma *sequência de disparos* de uma R_{PM}, corresponderá à evolução das marcações conforme $M_1[w > M_n$, segundo uma sequência $w = t_1, t_2, \ldots, t_n$ a que se pode agora associar a "string" correspondente a $?(w) = ?(t_1)\ ?(t_2) \ldots ?(t_n)$ que pertencente a ?*.

A linguagem associada a uma R_{PE}, denotada por $L(R_{PE})$ pode ser definida como

$$L(R_{PE}) = \{ ?(w) \mid M_S [w > M_F \}$$

ou seja, a linguagem, ou sequência de "strings", que pode ser aceite ou reconhe-cida por uma R_{PE} considerando todas as possíveis transições que levam a rede da marcação inicial à marcação final.

As *Redes de Petri Etiquetadas-Interruptíveis*, R_{PEI}, são R_{PE} que têm a propriedade de poderem ser interrompidas na sua execução por uma outra rede. A regra do "disparo" de transições da rede interruptível R_{PEI} passa portanto a ser diferente. Sendo A e B duas R_{PEI} em que A pode ser interrompida por B, o que é denotado por A[B], então, a regra para o disparo de uma qualquer transição *t* de A passa a ser a seguinte:

> Uma transição *t* da rede A pode disparar segundo as regras de uma R_{PE} mas apenas se a marcação actual de B for a inicial ou a final.

As *Redes Etiquetadas-Guardadas-Interruptíveis*, R_{PEGI}, aqui sugeridas para a definição operacional dos GI, são uma subclasse das R_{PEI}, dado imporem sobre estas algumas restrições fundamentais:

1) A marcação inicial de uma R_{PEGI}, obedece à regra de que $\#M_S = 1$, ou seja, apenas um lugar está activo;

2) A marcação final de uma R_{PEGI}, obedece à regra de que $\#M_F = 1$, ou seja, apenas um lugar é considerado final;

---

[21] *Labelled Petri Net.*

3) A cada transição pode ser associada uma *condição* que deve ser verdadeira para que esta, ainda que habilitada, possa ser disparada. A definição da rede é aumentada com a inclusão de uma função de transi-ções para expressões booleanas, C: T ? BoolExp.

4) A cada transição podem ser associadas acções, a serem executadas caso a transição dispare, mas que não alteram nem a estrutura nem a marca-ção da rede.

Na maioria dos exemplos que serão considerados e, em particular, ao serem usadas redes para descrever a semântica dos guiões, não se terá a preocupação de definir nenhuma função de "labelling" ? : w ? ? particular, assumindo-se, o que é aliás comum na utilização destas redes, que os próprios eventos associa-dos às transições constituem o alfabeto fundamental ?. A razão para esta consi-deração prende-se com o facto de que, embora se pretenda reter a possibilidade de realizar uma posterior análise linguística, é exactamente sobre os traços dos eventos que tal poderá vir a ser feito. Assim sendo, as "strings" que nos interes-sam aqui poder considerar coincidem com os próprios identificadores dos even-tos.

Apresenta-se de seguida, combinador a combinador, a semântica operacio-nal dos GI, definindo para cada construção a Rede de Petri correspondente ao comportamento especificado, sendo certo que esta será a rede posteriormente gerada a partir da especificação do guião.

6.2.5.- Semântica dos Guiões.

Comecemos por apresentar a estrutura de uma rede que representa o fluxo da execução de um guião genérico.

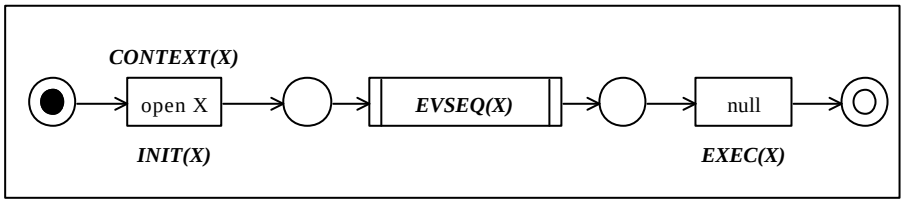

Fig. 6.2 - Rede para um Guião Genérico.

Em favor da coerência da apresentação das diversas redes que definem a se-mântica das construções dos guiões, seguir-se-ão as seguintes regras simples:

a) *Serão sempre apresentadas as marcações inicial e final da rede. A primei-ra é representada por lugares com círculo a preto indicando a existência de um*

token. A segunda é representada por lugares desenhados usando duplos círculos. Excepções serão objectivamente indicadas.

b) *As transições serão representadas por rectângulos, inscrevendo-se no interior destes o evento que provoca o seu disparo, na sua zona superior a condição para tal disparo e na sua zona inferior as acções a executar aquando do disparo. Rectângulos com margens representam abstracções de subredes. Transições simples indicarão o evento associado.*

A Fig. 6.2 representa então a execução genérica de um guião, tendo em con-sideração as suas cláusulas de comportamento. Como se pode interpretar da figura, para qualquer guião, a partir de uma marcação inicial simples, a rede só é de facto passada para um estado de actividade após o disparo da sua primeira transição, que é, no entanto, condicionada pelo valor lógico da condição que se associa à cláusula *CONTEXT*. Se esta cláusula for inexistente a transição não é guardada, sendo a regra de disparo a usual, ou seja, neste caso logo que ocorra o evento *open*. Analisando ainda esta primeira transição, verifica-se a existência de uma acção a ser executada. Trata-se de uma acção genérica, que se designou por *INIT*, e que corresponde à execução de todas as acções especificadas na cláu-sula *INIT* do guião.

Disparada, dentro destas condicionantes, a transição inicial, o guião passa a estar activo, sendo o seu comportamento especificado por uma rede cuja cons-trução é inteiramente dependente da expressão contida na cláusula *EVSEQ*. A semântica particular de cada combinador que pode ser utilizado em *EVSEQ* é de seguida apresentada, restando quanto a este exemplo afirmar que, terminado de forma normal, i.é. sem que eventos assíncronos tenham surgido, o comporta- mento especificado em *EVSEQ*, através de uma transição automática, conforme se indica referindo o evento *null*, atinge-se a marcação final, eventualmente após a execução da acção especificada na cláusula *EXEC*, que pode existir ou não.

Vejamos de seguida a semântica de cada um dos combinadores das expres-sões de comportamento, tendo sempre em atenção que estas expressões podem combinar quer eventos quer guiões.

Expressão Atómica.

A cláusula *EVSEQ* mais simples que pode ser declarada num guião, para além da cláusula nula, é a que inclui um único identificador de guião ou um evento.

São exemplos destes tipos de expressões de comportamento as cláusulas que se apresentam a seguir.

EVSEQ *input(pal)*

EVSEQ *DoConsPal(sig)*

Consideremos então a rede apresentada na figura 6.3 que especifica a ex-pressão de comportamento do tipo EVSEQ *GI(v)*, onde *GI* representa o identifica-dor do guião e *v* um argumento simbólico. Nesta rede designar-se-á por X o gui-ão que contém a cláusula.

Tal como no exemplo genérico apresentado na figura 6.2, a primeira transi-ção, disparável neste caso pela ocorrência do evento *start*, só pode disparar de facto se a cláusula de contexto do guião invocado for verdadeira. Disparada esta transição, o *token* transita para o lugar L2, habilitando de imediato três transi-ções. Uma delas, correspondente à ocorrência do evento assíncrono *cancel*, não é condicionada, podendo disparar de imediato logo que tal evento ocorra.

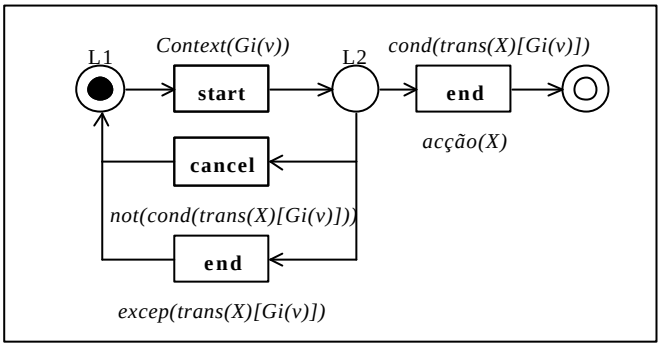

Fig. 6.3 - Rede para a expressão EVSEQ GI(v).

Este evento *cancel* não necessita de ser declarado para ser considerado em qualquer expressão de comportamento. A ocorrência do evento *end* irá terminar a execução do guião GI(v). No entanto, duas possibilidades existem consoante as condições associadas às duas transições habilitadas e relacionadas ainda com o evento. No primeiro caso, a condição que na cláusula TRANS se associa à ocorrência do evento é verificada e, caso seja verdadeira, a acção descrita na cláusula executada, atingindo-se a marcação final. No segundo caso, situação que é complementar da anterior, a condição é falsa, e as acções indicadas na sub-cláusula de excepção EXCEP são executadas. Não sendo atingido o estado final, o diálogo que foi realizado segundo o guião GI(v) é anulado.

Sequência.

A expressão de sequênciação, **exp1 . exp2**, é representável pela rede genérica da figura 6.4, e que consiste numa ligação entre as redes representadas por *EXP1* e *EXP2*, ligação que se estabelece fazendo coincidir a marcação final da primeira com a marcação inicial da segunda.

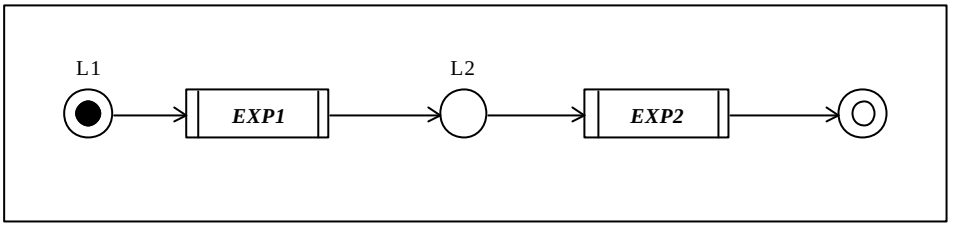

Fig. 6.4 - Rede para a Sequenciação.

A combinação sequencial não é, naturalmente, comutativa, sendo porém associativa, ou seja,

$$(exp1 \,.\, exp2) \,.\, exp3 = exp1 \,.\, (exp2 \,.\, exp3)$$

considerando uma análise de comportamento que exclui os comportamentos resultantes de situações de erro ou da ocorrência de eventos assíncronos.

Paralelismo Síncrono.

O paralelismo síncrono é denotado pela expressão **exp1 || exp2**, significando que os comportamentos expressos pelas duas expressões podem evoluir parale-lamente, ainda que devam sincronizar antes que o comportamento global espe-cificado possa terminar.

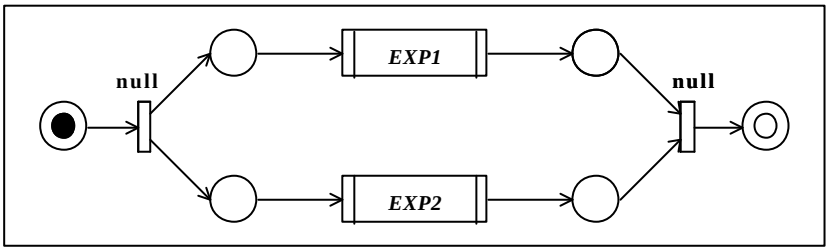

Fig. 6.5 - Rede para Paralelismo Síncrono.

podendo observar-se que, a partir da marcação inicial, por uma transição de tipo nulo, os dois lugares sucessores são activados podendo, a partir de então, a expressão de comportamento *EXP1* e a expressão *EXP2* evoluir de forma inde-pendente. Idealmente, o alfabeto de eventos de *EXP1* e de *EXP2* deverá ser dis-junto, por forma a facilitar o seu reconhecimento e atribuição, considerando-se que a sua ocorrência não é tratada de forma efectivamente paralela, mas antes considerando uma fila de eventos onde estes são colocados de forma entrelaça-da[22].

[22] seguindo uma estratégia de *interleaving*.

No entanto, se se considerarem os eventos assíncronos, tais como *cancel*, a identificação do destino da sua ocorrência, ou seja, se se aplicam a *EXP1* ou a *EXP2*, podem apenas ser considerados a nível da efectiva implementação.

Paralelismo Assíncrono.

O paralelismo assíncrono, que pode ser expresso pela expressão de comportamento **exp1 | exp2**, traduz-se numa rede que demonstra que as duas expres-sões de comportamento podem decorrer em paralelo, ou seja, em independên-cia, sendo certo no entanto que tal comportamento é dado por terminado logo que uma das expressões atinja o seu estado final. Para que tal possa ser expres-so por uma rede, basta que se considere que qualquer que seja a última transi-ção das expressões de comportamento que atinja a marcação final da rede, é esta que se considera efectivamente concluída. A rede correspondente tem a se-guinte configuração:

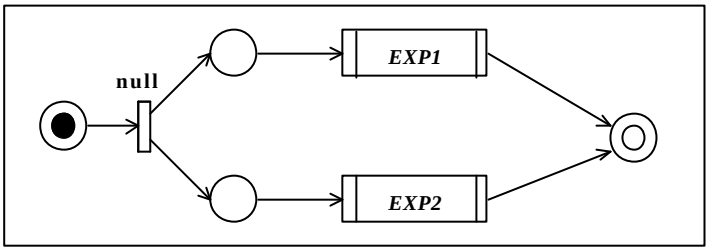

Fig. 6.6 - Rede para o Paralelismo Assíncrono.

Após uma transição inicial automática, eventos são recebidos e transições são realizadas nas subredes *EXP1* ou *EXP2*, conforme os mecanismos referidos para o paralelismo síncrono, sendo aqui o estado final atingido pela subrede que primeiro completar a sua sequência de eventos, assim terminando o comporta-mento paralelo dado pela expressão.

Alternativa.

A expressão de alternativa **exp1 + exp2**, corresponde à possibilidade de se exe-cutar o comportamento descrito por *exp1* ou o descrito por *exp2*. A partir de tal selecção apenas o alfabeto de eventos definido no comportamento da expressão escolhida é aceite e, possivelmente, disponibilizado ao utilizador. Este compor-tamento global é descrito pela rede da figura 6.7

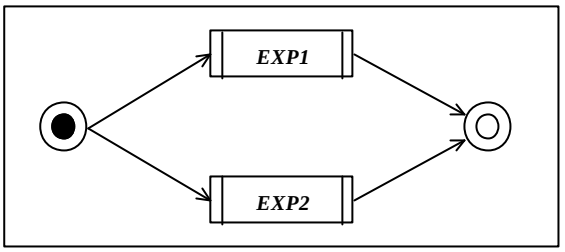

Fig. 6. 7 - Rede para Alternativa.

segundo a qual o primeiro evento que surgir activará a expressão de comporta-mento identificada por *EXP1* ou *EXP2*. A expressão seleccionada evoluirá até atingir o seu estado final.

Repetição.

A expressão de repetição **exp1*** corresponde à possibilidade de se executar o comportamento descrito por *exp1* zero ou mais vezes. Temos, neste caso, uma expressão de comportamento que só por si não termina, sendo a sua terminação determinada ou pela ocorrência de eventos assíncronos ou pela terminação que está associada à expressão de comportamento com que esta expressão de repe-tição se combina. Esta é uma situação particular que tem que ser objecto de uma análise especial, tendo em consideração a expressão geral em que esta sub-expressão é introduzida. A rede correspondente à repetição apresenta-se na figura 6.8 numa marcação não inicial, por forma a que seja claro o facto de que a transição correspondente aos eventos CANCEL e OK está sempre habilitada, e que corresponde à única transição de acesso à marcação final.

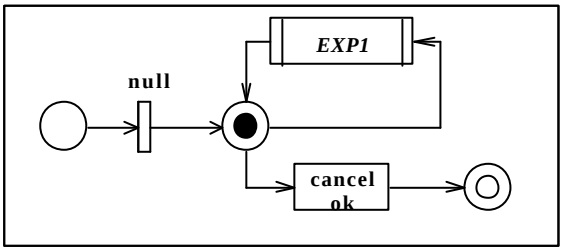

Fig. 6.8 - Rede para a Repetição.

Na expressão **exp1* . exp2**, duas formas de terminação da repetição sobre a *exp1* podem ser consideradas. Ou a que resulta da ocorrência de um evento as-síncrono CANCEL ou OK antes que qualquer evento aceite por *exp2* tenha ocorri-do, ou exactamente a ocorrência de um evento reconhecido por *exp2*.

Em expressões como **exp1* || exp2** ou **exp1* | exp2**, o fim do comporta-mento da expressão composta é completamente determinado, a menos da

ocor-rência de um qualquer evento assíncrono, pelo fim do comportamento da *exp2*. Poderia parecer, numa primeira leitura, que então não faria sentido considerar tal combinação, porém, como existe actividade associada à recepção de eventos, as duas expressões acabam por garantir que são aceites eventos quer de uma quer de outra subexpressão, aos quais se associam actividades, terminando no entanto o comportamento apenas quando *exp2* terminar.

Por outro lado a expressão composta **exp1* + exp2**, sendo uma opção, tem por significado a escolha entre um comportamento que, ou segue *exp1* e apenas pode ser terminado por um evento assíncrono, ou segue *exp2*.

Interrupção.

A expressão de interrupção **exp1[exp2]**, especifica que o comportamento indica-do pela *exp1* pode, a qualquer momento, ser interrompido pela activação do comportamento especificado na *exp2*. A partir do momento em que o comporta-mento especificado por *exp1* é interrompido pela *exp2*, então, apenas eventos relacionados com transições especificadas em *exp2* serão aceites. Quando *exp2* chegar ao fim, qualquer que seja tal trajecto, a *exp1* volta a definir o controlo, exactamente no ponto em que este foi interrompido.

Este comportamento traduz-se, simbolicamente, pela rede da figura 6.9.

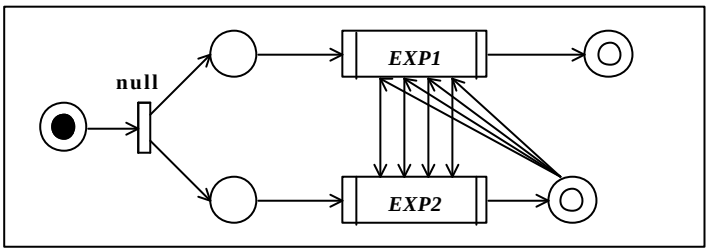

Fig. 6.9 - Rede com Interrupções.

Como se pode verificar pela semântica informal dada para esta expressão de comportamento, qualquer de seja *exp1*, i.é., qualquer que seja a rede que repre-senta a expressão, seria necessário, para que se pudesse expressar com detalhe o comportamento *exp1[exp2]*, que a qualquer lugar da rede que representa *exp1* fosse colocado em paralelo o lugar inicial da rede que representa *exp2*, de modo a que o evento correspondente à primeira transição de *exp2* pudesse a qualquer momento ser aceite. Por outro lado, logo que aceite este evento, dado que a rede *exp1* não poderá evoluir enquanto a rede *exp2* não atingir a sua marcação final, será necessário simular a remoção do *token* da rede *exp1*, o que pode ser feito aceitando apenas eventos em *exp2* até que *exp2* termine. Finalmente, quando a *exp2* terminar, ou seja a rede respectiva atingir a marcação final, o *token* que foi, simuladamente, retirado

de *exp1* deve ser, simuladamente, considerado de novo colocado no respectivo lugar, podendo prosseguir o comportamento expres-so por *exp1* de forma normal, ainda que interruptível, até terminar.

Toda esta semântica não pode ser expressa pela notação gráfica associada às Redes de Petri, pelo que, ainda que esta seja a semântica operacional para a construção, apenas uma Rede de Petri genérica, ou seja, procurando represen-tar todas as possíveis *exp1* e *exp2*, e suas possíveis interligações, pode ser apre-sentada.

Eventos Assíncronos.

Apresentam-se de seguida as extensões a introduzir na rede genérica de execu-ção de um guião, pelo facto de se assumirem como reconhecíveis os eventos OK e CANCEL.

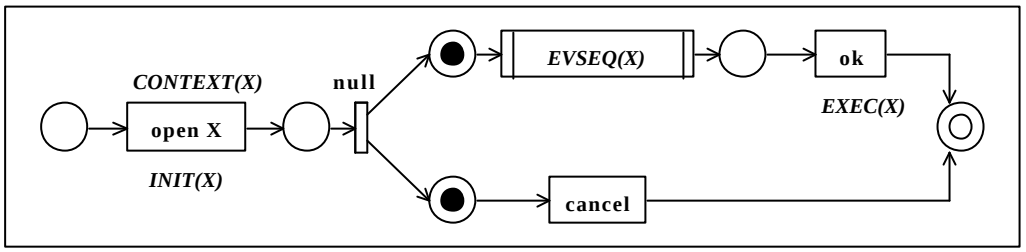

Fig. 6.10 - Rede com OK e CANCEL.

Apresenta-se na figura 6.10 a rede genérica na qual se consideram eventos OK e CANCEL. A rede é apresentada não numa marcação inicial, mas sim consi-derando-se que o evento *open* já ocorreu, o que habilita as transições correspon-dentes quer ao evento *cancel* quer a EVSEQ(X). Habilitada a transição correspon-dente ao evento *cancel*, e qualquer que seja o estado do subdiálogo especificado por EVSEQ(X), logo que ocorra um evento *cancel* o guião atinge a sua marcação final.

A inclusão do evento OK, dado que corresponde à confirmação do diálogo que foi realizado segundo a expressão em EVSEQ(X), traduz-se pela introdução de um lugar adicional após a transição correspondente a EVSEQ(X), onde é de-positado o *token* logo que esta termina, e pela transição correspondente ao even-to *ok*. A menos da ocorrência do evento *cancel* a rede só atinge a sua marcação final pela ocorrência deste evento.

Os eventos RESET e APPLY são, respectivamente, equivalentes a CANCEL e OK apenas com a diferença de que o guião é automaticamente reinicializado.

6.2.6. - Exemplos.

Apresentam-se nesta subsecção alguns exemplos de especificações de diálogos de várias características usando GI, procurando ilustrar o seu poder expressivo em exemplos concretos. Começaremos por pequenos exemplos que visam criar alguma familiarização com o formalismo dos guiões e com a sua utilização, de seguida apresentando exemplos mais completos. Onde tal se justificar, o exem-plo especificado usando GI será igualmente especificado noutras notações por forma a poder estabelecer-se, no contexto do exemplo, alguma comparação das capacidades expressivas e da simplicidade notacional.

A maioria dos exemplos apresentados são clássicos em literatura de IHC, o que significa que não se procuraram exemplos específicos para os quais os GI se mostrassem particularmente vocacionados. Os GI são mesmo postos em compa-ração com dois dos formalismos considerados dos mais expressivos dentro de duas classes de formalismos, designadamente os GTNs dentro dos ATNs, e SPI dentro dos formalismos baseados em processos, devendo salientar-se o facto relevante de que GTNs não são executáveis, enquanto que SPI e GI o são, ou seja, permitem gerar protótipos dos controladores especificados.

Exemplo 6.1 (Entradas por Ordem arbitrária) :

Especificar um diálogo onde é lido um comando e um argumento, por uma qualquer ordem, podendo quer o comando quer o argumento ser corrigidos um número infinito de vezes. O diálogo só deve terminar com a entrada do coman-do de confirmação, Ok, numa situação em que comando e argumento foram já introduzidos. O comando introduzido deve ser válido.

```
DefGI GIEx1
Declarations
        TYPE Synth
        ARGS com: Str ; arg: Str
Behaviour
        INIT com = ""; arg = ""
        EVSEQ input(com)* || input(arg)*
        TRANS
                input(com): ExistCom(com) => Null
                                        EXCEP   out("Comando Errado !")
                Ok:
        EXEC Perform(com, arg)
EndGI
```

O guião apresentado, apesar de ser do tipo *Synth*, não realiza de facto a sín- tese de um comando a invocar mas antes aceita, segundo as regras enunciadas, um comando e um seu argumento, e, caso o comando seja

válido, passa-o para execução a um interpretador. A cláusula EVSEQ especifica o paralelismo síncro-no entre sequências de entradas de valores para *com* e de entradas de valores para *arg*. A declaração do evento *Ok* garante que, logo que *com* e *arg* possuam valores lidos, a ocorrência do evento termina o diálogo. A condição de excepção associada à ocorrência do evento de leitura do comando garante que, caso este não seja válido, o evento é desprezado e, em independência da subexpressão de leitura do argumento, uma nova leitura de um comando deve ser realizada.

Note-se, conforme a semântica do operador de repetição (ver Fig. 6.8), que só usando a construção repetitiva se pode garantir a possibilidade de serem reintroduzidos valores para as variáveis.

Finalmente, e propositadamente, ambas as variáveis foram definidas como sendo do tipo "string", pelo que poderia colocar-se a questão de saber como dis-tinguir entre os eventos "string" associados ao comando e ao argumento. Uma solução possível poderia ser a especificação de um conjunto de dispositivos de entrada cujos identificadores poderiam servir de prefixos aos eventos, passando--se a identificar os eventos por triplos identificador de dispositivo, evento e va-lor, o que muito baixaria o grau de abstracção da especificação. Admite-se pois que o reconhecimento da origem, e também do destino, dos eventos, possa ser feito por outros componentes da arquitectura de interacção, elevando-se desta forma o nível de abstracção na descrição do diálogo. Conforme Cockton salienta em [Cockton 90], estes tipos de decisões mostram que a delegação de um certo grau de responsabilidade noutros componentes da arquitectura da IU pode simplificar o controlo ao nível da especificação do diálogo, ainda que demasiada distribuição do mesmo possa ser prejudicial para a sua clareza e análise.

Antes de se apresentar este mesmo exemplo especificado no formalismo dos GTNs de Cockton, deve referir-se que GTNs consistem de um sistema de regras, em que cada regra apresenta a estrutura,

$$starts : aSentence \Rightarrow Procedure_1; \ldots ; Procedure_n;$$
$$\rightarrow endpoint$$

em que *starts* se refere a nodos da rede de transição de estados, designando por exemplo *all* todos os nodos da rede, sendo outros designados usando opera-dores da teoria de conjuntos, sendo *aSentence* uma simples condição, avaliada como verdadeira ou falsa, e que sendo verdadeira implica a execução dos proce-dimentos indicados, passando-se a execução para o nodo indicado por *endpoint* que, em muitos casos, assume o valor *same*, ou seja, refere os mesmos nodos referidos em *starts*.

A especificação do exemplo anterior ficaria, usando GTNs, da forma:

```
all : arg_event => set_arg;
-> same

all : cmd_event => set_cmd;
-> same

all : ok_event => set_ok;
-> same

all : got_arg & got_cmd & got_ok
=> call_linkage ; unset_arg_and_cmd;
-> same

all : ok_event_but_incomplete
=> not_all_there_message; dequeue_ok
-> same

all : ok_event_and_complete => get_events; pass_to_linkage;
-> same
```

É de notar que não é realizada qualquer definição da semântica de frases tais como *set_ok*, *got_arg*, *ok_event_and_complete*, etc., não é feita qualquer con-sideração sobre tipos, não é claramente indicada qual a ligação à camada com-putacional e qual a ligação ao estado interno do diálogo. A especificação não contempla a possibilidade de se reintroduzirem os valores, quer para o comando quer para o argumento, e também não trata qualquer tipo de situação de erro.

Em resultado, fica-se com a sensação que o nível de abstracção é de tal mo-do elevado que tudo fica por especificar e várias diferentes interpretações podem ser realizadas, o que induz ambiguidade indesejável.

Considere-se agora o exemplo especificado usando SPI [Alexander 87], que se baseia na linguagem *eventCSP*, um subconjunto da linguagem de processos CSP. O diálogo é especificado como sendo um conjunto de processos sequen-ciais, cujo comportamento se especifica por eventos, representando cada evento uma interacção ou uma actividade do diálogo. A descrição do comportamento dos processos é realizada usando as seguintes construções simples:

(e ? P) ; *prefixo* : aceita o evento *e* e comporta-se como o processo **P**

(P [] Q) ; *escolha* : comporta-se como **P** ou como **Q**

P ; Q ; *sequência* : realiza o comportamento de **P** e de seguida o de **Q**

P || Q ; *paralelismo* : **P** e **Q** paralelos sincronizam em eventos comuns.

Especifiquemos então de seguida o Exemplo 6.1 em SPI. Vamos considerar em primeiro lugar dois processos, *le_cmd* e *le_arg*, e depois criar um processo de controlo que os irá sincronizar.

```
le_cmd = ( cmd_off ? input1 ?
           ( cancel ? cmd_off ? le_cmd
           [] ok ? ( cmd_on? ? skip
                   [] cmd_off? ? erro_com1 ? le_cmd )
           [] cmd ? ( valido? ? cmd_on ? le_cmd
                   [] invalido? ? erro_com2 ? le_cmd )
         ))

le_arg = ( arg_off ? input2 ?
           ( cancel ? arg_off ? le_arg
           [] ok ? ( ha_arg? ? skip
                   [] nao_ha_arg? ? erro_arg ? le_arg )
           [] arg ? arg_on ? le_arg
           ))

le_com_arg = ( le_com || le_arg synchronized on { skip } ) ; execute
```

Como se pode verificar pela especificação, sendo associada a cada evento uma actividade elementar, a profusão de eventos resultante desta granularidade é grande. Por outro lado, a dissociação entre a especificação em *eventCSP*, que apenas especifica a sequência da ocorrênciada dos eventos, da especificação das variáveis que representam o estado interno do diálogo, realizado numa outra estrutura e segundo uma outra linguagem, faz com que eventos artificiais devam ser introduzidos, quer para cumprirem o papel de "flags", quer para controlo do estado interactivo. Na especificação apresentada, é este o papel dos eventos *arg_on*, *arg_off*, *cmd_on* e *cmd_off*. Outros eventos introduzidos na especificação correspondem apenas a pontos onde é necessário fazer uma consulta ao estado da aplicação. É este o caso dos eventos identificados por *valido?* e *invalido?* que testam junto da aplicação a existência do comando que foi introduzido. Outros ainda, relacionam-se com a ocorrência de erros. É ainda de referir, quanto a este aspecto da sobreutilização dos eventos, que estes têm origens e significados diferentes e pouco claros, e que a única forma de realizar alguma distinção sobre a sua

semântica é através de um judiciosa escolha dos seus nomes. Esta tarefa é ainda mais ingrata se tivermos em atenção que, quando processos são colocados em paralelismo, sincronizam, por omissão, em eventos com o mesmo identificador. Devido a este facto, e no sentido de evitar erros, o autor decidiu-se por explicitar na expressão de paralelismo o conjunto dos eventos onde os dois processos devem, de facto, sincronizar, no exemplo o evento *skip* que correspon-de a fim de actividade.

O processo de sincronização, *le_cmd_arg*, termina o seu comportamento invocando o interpretador para a execução da respectiva operação da aplicação.

Note-se ainda a dificuldade do formalismo exprimir, de forma simples, a ocorrência de eventos assíncronos, tais como *cancel* e *ok*, que acabam por ter que ser explicitamente tratados, como todos os outros, segundo um esquema de "event-handling", bem evidente nos exemplos pelo recurso a uma estrutura con-dicional para a sua aceitação e tratamento.

A simplicidade, clareza e capacidade expressiva dos GI, relativamente a estes dois formalismos, alegadamente simples e expressivos, são, tomando este primeiro exemplo como referência, claramente superiores.

Exemplo 6.2 (Ordem arbitrária dos argumentos) :

Especificar um diálogo para desenhar uma caixa num sistema de CAD, onde o utilizador, após seleccionar num menu o comando respectivo, introduz, por uma ordem arbitrária, os argumentos do comando, que são dois pontos que representam as coordenadas dos cantos superior esquerdo e inferior direito da caixa a desenhar, sendo de seguida invocada a aplicação. O diálogo pode ser anulado a qualquer momento.

```
DefGI GIEx2
Declarations
        TYPE Synth
        ARGS p1: Point ; p2: Point
Behaviour
        EVSEQ input(p1) || input(p2)
        TRANS
                Cancel:
        EXEC DrawBox(p1, p2)
EndGI
```

O GI é, neste caso, muito simples, dado que necessita apenas de especificar que a obtenção de um ponto é colocada em paralelismo síncrono com a leitura do outro, deste modo sem referência a qual pode ser sintetizado primeiro, sendo no entanto certo que as duas leituras

sincronizam quando ambas tiverem sido realizadas, podendo então a operação da aplicação ser invocada. Caso o evento *cancel* seja accionado, conforme de pode verificar pela semântica apresentada na figura 6.10, o diálogo termina.

Este exemplo não é especificado usando GTNs ou SPI, dado ser, de facto, de alguma forma semelhante ao exemplo anterior.

Exemplo 6.3 :

Especificar um diálogo para "Login" num sistema de computação segundo o qual o utilizador introduz o seu "username" primeiro, e, de seguida, a sua "password", devendo o diálogo ser repetido caso a "password" não esteja correcta.

DefGI *GIEx3*
Declarations
TYPE *Synth*
ARGS *user: Str ; passw: Str*
Behaviour
INIT *user = ""; passw = ""*
EVSEQ *input(user)* **.** *input(passw)*
TRANS
input(passw): PasswOk(user, passw) => Null
EXCEP out("Password Errada !")
Cancel:
EndGI

Conforme se pode verificar pela rede apresentada na Fig. 6.3, quando uma condição de excepção é accionada o diálogo especificado no guião é reiniciali-zado, voltando-se à marcação inicial da rede. Daí que nada mais necessite de ser especificado no exemplo em questão.

Em SPI, esta mesma especificação teria a seguinte estrutura, conforme a apresentação feita em [Alexander 87],

Logon = (prompt_for_user ?
(user_ok ? Pwd
[] not_user_ok ? errmsg1 ? Logon
))

Pwd = (prompt_for_pwd ?
(pwd_ok ? skip

```
                [] not_pwd_ok ?  errmsg2 ?  Logon
            ))

    CmdLev =  ( prompt ?
                    ( logon_cmd ?  skip
                    [] other ?  errmsg3 ?  CmdLev
                ))
```

A análise e comparação desta especificação com a especificação usando GI vem reforçar os comentários produzidos aquando do exemplo 6.1. É ainda importante reafirmar-se que não se apresentam nos exemplos SPI as especifica-ções em eventISL que fazem a ligação à aplicação e ao estado da IU, o que im-plicaria adicional complexidade, sendo estes aspectos já contemplados nos GI.

Exemplo 6.4 :

Especificar o diálogo com um editor de texto, no qual este aceita eventos com origem no "rato" ou no teclado. Os eventos devem ser interpretados no contexto de um editor de texto. O "rato" aceita apenas eventos Down e Up, por esta ordem, enviando ao editor de texto a posição actual do cursor logo que o evento Down surge. O teclado aceita eventos que correspondem à entrada de um caracter, armazena os caracteres entrados num "buffer" de linha, e quando recebe o caracter NewLine envia o conteúdo do "buffer" para o editor que a in-sere na posição corrente do cursor.

Antes de apresentar a especificação deste exemplo usando GI, deverá ter-se em atenção o facto de que este foi apresentado pela primeira vez em [Cardelli e Pike 85] no contexto da linguagem de programação de processos Squeak. Todos os formalismos de especificação de IU baseados em processos usam este exemplo, pelo facto de ele ser ajustado a tal perspectiva. De facto, e embora não sendo complexo, o exemplo envolve três processos com comportamentos muito simples, dois dos quais são dispositivos físicos permanentemente geradores de eventos, e um terceiro processo que, neste caso, representa a aplicação a que se destina a interacção e que vai servir de processo de controlo e sincronização.

Enquanto exemplo interessante na exemplificação do funcionamento de pro-cessos, ele parte do princípio que as aplicações tratam directamente com os eventos recebidos dos dispositivos físicos, o que não acontece actualmente.

Procurando reforçar estas ideias, começaremos por mostrar como o exemplo se especifica em SPI (cf. [Alexander 87]) e ilustra a aplicação de processos.

Em primeiro lugar, descrevamos o processo que vai controlar a actividade do "rato".

Mouse = (DOWN ? get-position ? send-position ? UP ? Mouse)

Este processo começa por aceitar o evento que corresponde ao pressionar do botão do dispositivo, DOWN. De seguida, a posição actual do "rato" é recebida e comunicada. Em seguida o processo aceita o evento UP, que corresponde ao soltar do botão, retomando o comportamento descrito.

```
Kbd =   ( get-char ?
            ( newline ?  send-line ?  Kbd
            [] text-char ?  add-to-line ?  Kbd
        ))
```

O processo que representa o teclado aceita caracteres que adiciona à linha em formação caso não sejam o caracter *NewLine*. A entrada de *Newline* faz com que o processo envie a linha construída para o editor de texto.

Note-se que, dado os processos *Kbd* e *Mouse* não terem eventos em comum, ao serem colocados em paralelo, tal corresponde a livre paralelismo, o que quer dizer que não são impostas quaisquer restrições à ordem das entradas do utili-zador.

Kbd | | Mouse

O processo correspondente ao editor de texto, recebe e guarda as sucessivas posições do "rato" (eventos de sincronização com o processo *Mouse*) e as linhas enviadas pelo teclado (evento de sincronização com o processo *Kbd*), actualizan-do coerentemente o seu estado.

```
Text =  ( send-position ?  save-position ?  Text
        [] send-line ?  write-line ?  Text
        )
```

O comportamento final é descrito pela expressão,

Kbd | | Mouse | | Text

A especificação usando GI não pode, naturalmente, seguir uma lógica de processos e de comunicação entre estes, pois guiões não são processos. Temos pois que seguir uma lógica diferente, assente na ideia de que guiões

especificam uma camada que se coloca entre a aplicação e a camada mais baixa do diálogo, especificando exactamente a camada que vai ser responsável pelo controlo dos eventos recebidos, ou seja, especificando o controlador do diálogo.

Vamos então dividir a especificação em dois guiões principais que controlam os eventos gerados pelo "rato" e pelo teclado, tendo em atenção que estes guiões são responsáveis pela consistência dos estados internos dos envolvidos.

Comecemos por especificar o guião correspondente aos eventos do dispositi-vo "rato".

```
DefGI Rato
Declarations
        TYPE Synth
        ARGS p: Ponto
        VAR-CTRL pos: Ponto
Behaviour
        EVSEQ ( Down . input(p)* . Up )*
        TRANS
              input(p):  => pos := p ; NovaPos(p)
EndGI
```

O guião é declarado como sendo do tipo *Synth* pois, embora não tenha que sintetizar nenhum comando, vai aceder, para alteração, à camada aplicativa. O guião tem uma variável interna e declara uma variável do controlador à qual tem que ter acesso para alteração, constituindo este o mecanismo de passagem de informação para outros guiões, pois guiões não comunicam entre si. O guião *Rato* especifica um comportamento segundo o qual após a aceitação de um evento *Down*, repetidas transmissões da posição do "rato" são aceites, cada uma delas actualizando o valor da variável *pos* do controlador e actualizando ainda o valor de tal posição na aplicação, por invocação de uma função da aplicação pa-ra tal efeito.

A ocorrência do evento *Up*, que termina a sequência infinita anterior (cf. semântica discutida na subsecção 6.2.5), permite a repetição do comportamento iniciado com o evento *Down*. Note-se como com o operador ***** aplicado a uma ex-pressão se especifica comportamento sem terminação típico dos processos.

Vejamos de seguida o guião para o teclado.

```
DefGI Kb
```

```
Declarations
        TYPE Synth
        ARGS c: Char ; linha : Str
        VAR-CTRL pos: Ponto
Behaviour
        INIT linha = ""
        EVSEQ input(c)*
        TRANS
            input(c): (c != "NL") =>  linha := linha + c
                                EXCEP linha := "" ; InsLinha(linha, pos)
EndGI
```

O comportamento expresso consiste da aceitação eterna de caracteres, que enquanto forem diferentes do caracter *NL* são introduzidos numa linha. Porém, logo que o caracter *NL* for introduzido, a variável linha é inicializada e invocada a função da aplicação que insere a linha construída na posição actual do "rato".

Finalmente, o controlo global será especificado por um guião que coloca em paralelismo a aceitação dos eventos do "rato" e do teclado, ou seja, os guiões de-clarados como externos *Kb* e *Mouse*.

```
DefGI Control
Declarations
        TYPE
        EXTERNAL Kb, Mouse
Behaviour
        EVSEQ Kb || Mouse
EndGI
```

O acesso à variável do controlador *pos* é crítico, pois ambos os guiões lhe têm acesso concorrente dado estarem em paralelo. Porém, dado que os eventos são disjuntos e consumidos pelo controlador a partir de uma fila de eventos, tal problema deixa de existir aquando da execução.

Note-se finalmente que, apesar de todas as considerações anteriores, os GI apresentam uma especificação simples e razoavelmente clara, sendo as expres-sões de comportamento semelhantes, ou até mais ricas (cf. processo *Mouse*), às apresentadas na especificação em SPI. A sincronização entre processos existen-te em SPI não tem contrapartida nos GI dado não serem processos, pelo que tal descrição usando GI é implícita. No exemplo, em SPI a sincronização é realiza-da pelo processo *Text* que representa a aplicação. Nos GI, esta sincronização é realizada pelas expressões de comportamento e pelas cláusulas de tratamento dos mesmos.

Exemplo 6.5 : (Um comando interruptível).

Especificar um diálogo para síntese de um comando que faz com que uma linha entre dois pontos seja desenhada pela aplicação. Durante a síntese do comando, o tipo de linha corrente, definido por um simples número, poderá ser alterado.

Vamos considerar neste exemplo três guiões. O primeiro especifica o diálogo necessário para que se possam mudar os atributos da linha a desenhar pela aplicação, o que corresponde a ler um número e comunicar à aplicação tal valor lido como sendo o tipo actual de linha (guião *LineType*).

```
DefGI LineType
Declarations
        TYPE Synth
        ARGS num: Integer
Behaviour
        EVSEQ input(num)
                Cancel:
        EXEC SetLineType(num)
EndGI
```

De seguida vamos apresentar o guião que descreve o comportamento para a síntese do comando (guião *Line)* que, ao ser transmitido à aplicação, vai permitir o desenho da linha especificada.

```
DefGI Line
Declarations
        TYPE Synth
        ARGS p1, p2 : Point
Behaviour
        EVSEQ input(p1) || input(p2)
                Cancel:
        EXEC DrawLine(p1, p2)
EndGI
```

Finalmente, apresenta-se o guião que controla o comportamento global para a construção de uma linha (guião *LineCtrl*).

```
DefGI LineCtrl
Declarations
        TYPE Synth
```

EXTERNAL *Line, LineType*
Behaviour
EVSEQ *Line[LineType]*
Cancel:
EndGI

Esta especificção, usando GTNs conduziria a uma grande profusão de esta-dos e "flags" cuja apresentação seria incomportável. Usando SPI, duas hipóteses poderiam ser consideradas. A primeira hipótese poderia ter por base os combi-nadores de processos do SPI, que não condiziria a nenhuma solução já que ne-nhum dos combinadores SPI contempla este comportamento. A segunda consis-tiria em considerarem-se os dois processos em paralelo com o processo que cor-responde à leitura da linha activo, e outro, para alteração do tipo de linha inac-tivo. Ao surgir o evento que, por sincronização, activa o segundo processo, este accionaria um evento-flag que desactivaria o primeiro. A partir de então, só o segundo processo aceita eventos e, ao terminar, deve accionar um evento que volta a activar o primeiro. Ainda que logicamente claro este fluxo de controlo, ele é bastante complexo de ser especificado em SPI, dado ter que se garantir que o retorno do controlo ao primeiro processo é realizado no ponto em que este foi suspenso.

Exemplo 6.6 : (Uma Rede de Menus).

Especificar um diálogo sequencial baseado em menus, que ao primeiro nível oferece três opções A, B e C, opções que podem estar activas ou inactivas em função das suas condições de activação que designaremos por C(A), C(B) e C(C). A opção Fim deve igualmente estar disponível para encerrar o diálogo.
A opção A corresponde à invocação de uma operação sem parâmetros. A opção B deve dar acesso à síntese de um comando com dois argumentos a serem li-dos em sequência. Finalmente, a opção C deve dar acesso a um submenu com três opções, C1, C2 e C3.

Teremos então a seguinte especificação com GI, apresentada seguindo uma ordem "top-down". O primeiro guião corresponde ao menu principal do sistema, no qual se distinguem as operações seleccionáveis, se as suas condições de con-texto forem verdadeiras.

DefGI *MenuPrinc*
Declarations
TYPE *Decision*
EXTERNAL *A, B, C, Fim*
Behaviour
EVSEQ *(A + B + C)** **+** *Fim*

EndGI

O guião especifica um comportamento segundo o qual quatro opções são co-locadas como disponíveis. As opções *A, B* e *C* são repetidas vezes seleccionáveis, sempre em alternativa à opção de *Fim* que terminará o diálogo.

DefGI *A*
Declarations
TYPE *Synth*
Behaviour
CONTEXT *C(A)*
EXEC *op-A()*
EndGI

DefGI *B*
Declarations
TYPE *Synth*
ARGS *arg1, arg2 : INT*
Behaviour
CONTEXT *C(B)*
EVSEQ *input(arg1)* **.** *input(arg2)*
EXEC *op-B(arg1, arg2)*
EndGI

O guião *B* lê, em sequência, dois argumentos e invoca a respectiva operação da aplicação.

DefGI *C*
Declarations
TYPE *Decision*
EXTERNAL *C1, C2, C3, Exit*
Behaviour
EVSEQ *(C1 + C2 + C3)** **+** *Exit*
EndGI

Finalmente, a opção *C* corresponderia à activação do guião C, do tipo menu, onde as quatro opções apresentadas são oferecidas ao utilizador. Os guiões *Fim* e *Exit* poderiam ter as seguintes descrições:

DefGI *Fim*
Declarations
    TYPE *Synth*
Behaviour
    EXEC *CloseApl()*
EndGI

DefGI *Exit*
EndGI

O guião *Fim* fecha a aplicação e o guião *Exit* é apenas um ponto de controlo, não lhe sendo associado qualquer comportamento particular.

Exemplo 6.7 : (Mini Linguagem para um sistema de CAD).

Especificar o diálogo com um sistema de CAD que aceita três comandos principais, respectivamente para desenho de uma linha, de uma caixa ou rectângulo, e para zoom. Seleccionado o comando a partir de um menu, os seguintes requisitos de interacção devem ser observados:

- *O comando zoom deve poder interromper qualquer dos outros, por forma a que um ponto possa ser colocado com toda a precisão;*
- *O comando zoom aceita os seus argumentos, o factor de "zooming" e um ponto central, por uma ordem qualquer;*
- *O comando de desenho de uma linha tem dois pontos como parâmetros;*
- *A qualquer momento da definição do comando de desenho de uma linha, o utilizador pode mudar o tipo de linha ou a sua cor;*
- *O comando caixa aceita os seus argumentos, canto inferior esquerdo e canto superior direito, por uma ordem qualquer;*
- *Um comando CANCEL deve estar disponível por forma a cancelar o comando actual em qualquer passo da sua síntese.*

Considere-se ainda, para efeitos de especificação que a gramática para esta mini-linguagem de CAD é a seguinte :

line-cmd ::= **Drawline** point point
box-cmd ::= **Drawbox** lower-left upper-right
zoom-cmd ::= **Zoom** number point

colour-cmd ::= **Newcolour** number
point ::= coordx coordy
line-type ::= **Linetype** number
escape-cmd ::= **Esc**

Analisemos por passos a especificação, usando GI, desta linguagem de interacção. O comando *zoom* deve aceitar os seus argumentos por uma ordem li-vre e, depois, invocar a operação da aplicação. Este guião usa o guião *GPoint*, criado para sintetizar o valor de um ponto, dado que vários comandos vão ne-cessitar de tal subdiálogo. Trata-se de um pequeno exemplo de reutilização de guiões previamente definidos, um dos mecanismos de reutilização disponibiliza-dos na linguagem dos GI.

DefGI *Zoom*
Declarations
TYPE *Synth*
ARGS *p1: Point ; n: INT*
EXTERNAL *GPoint: Point*
Behaviour
EVSEQ *GPoint(p1)* **||** *input(n)*
TRANS *Cancel:*
EXEC *Zoom(p1, n)*
EndGI

O guião *GPoint* tem a si associada a declaração do tipo do valor que sintetiza por forma a que não existam dúvidas sobre a compatibilidade da sua utilização nas expressões em que aparece.

DefGI *Colour*
Declarations
TYPE *Synth*
ARGS *c: INT*
Behaviour
EVSEQ *input(c)*
TRANS *Cancel:*
EXEC *NewColour(c)*
EndGI

O guião *Colour* lê um número inteiro e invoca a operação da aplicação que altera o valor da cor actual.

DefGI *LineType*
Declarations
TYPE *Synth*
ARGS *num: Integer*
Behaviour
EVSEQ *input(num)*
TRANS *Cancel:*
EXEC *LineType(num)*
EndGI

O guião *LineType* especifica o diálogo necessário para a alteração do tipo de linha do sistema. Este guião deverá estar disponível no menu principal da apli-cação e disponível no contexto da construção do comando de linha. Assim, o co-mando de linha, descrito pelo guião *Line*, tem que ser considerado em conjunto com todos os que o podem interromper numa mesma expressão de controlo desse comportamento. Para tal foi especificado o guião *LineCtrl*.

DefGI *Line*
Declarations
TYPE *Synth*
ARGS *p1, p2 : Point*
Behaviour
EVSEQ *input(p1) || input(p2)*
TRANS *Cancel:*
EXEC *DrawLine(p1, p2)*
EndGI

A expressão de comportamento do guião *LineCtrl* especifica que o guião *Line* pode ser interrompido, em alternativa, pelos guiões *Zoom*, *Colour* e *LineType*, declarados, naturalmente, como guiões externos a este.

DefGI *LineCtrl*
Declarations
TYPE *Synth*
EXTERNAL *Line, LineType, Colour, Zoom*
Behaviour
EVSEQ *Line[LineType + Colour + Zoom]*
EndGI

O mesmo se fez relativamente ao guião *Box*, criando-se o guião *BoxCtrl* que especifica que o comportamento em *Box* pode ser interrompido pelo guião *Zoom*.

```
DefGI BoxCtrl
Declarations
        TYPE Synth
        EXTERNAL Zoom
Behaviour
        EVSEQ Box[Zoom]
EndGI
```

```
DefGI Box
Declarations
        TYPE Synth
        ARGS p1: Point ; p2: Point
        EXTERNAL GPoint: Point
Behaviour
        EVSEQ GPoint(p1) . GPoint(p2)
        TRANS
                Cancel:
        EXEC DrawBox(p1, p2)
EndGI
```

```
DefGI GPoint: Point
Declarations
        TYPE ValSynth
        ARGS x: INT ; y: INT
Behaviour
        EVSEQ input(x) || input(y)
        TRANS
                end:  => GPoint := mkPoint(x, y)
EndGI
```

Resta agora apresentar-se o menu principal do sistema, que pode ser repre-sentado pelo guião:

DefGI *MenuCAD*
Declarations
TYPE *Decision*
EXTERNAL *BoxCtrl, LineCtrl, Colour, LineType, Zoom, Exit*
Behaviour
EVSEQ *(BoxCtrl + LineCtrl + Colour + LineType + Zoom)** **+** *Exit*
EndGI

ainda que pudessemos também considerar a possibilidade de colocar os coman-dos principais, representados pelos guiões *BoxCtrl*, *LineCtrl* e *Zoom*, em paralelo com as alternativas simples de *Colour* e *LineType*, conforme a expressão

$$((BoxCtrl \mid\mid LineCtrl \mid\mid Zoom) + (Colour + LineType))^*$$

6.2.7. - Edição de GI e geração das Redes de Petri.

Definida a estrutura sintáctica dos GI, tornou-se óbvia a necessidade de criar uma ferramenta de suporte à criação, em modo interactivo, dos GI, com capa-cidade de realizar a imediata validação destes. Por outro lado, se à criação com validação se juntar a geração automática das Redes de Petri relativas aos guiões criados, então a ferramenta possuirá a funcionalidade ideal.

Usando o Synthesizer Generator (SGen) [Reps e Teitelbaum 89] foi possível desenvolver esta ferramenta, de facto um editor estruturado para a construção interactiva dos GI, com garantia da sua correcção sintáctica [Rocha 93].

Para além da ferramenta assegurar, usando edição estrutural, a correcção sintáctica dos guiões, um mecanismo de *view*, intrínseco ao SGen, permite que, a partir da sintaxe dos guiões, as respectivas Redes de Petri sejam geradas de modo automático, tendo bastado, para tal, fornecer ao SGen a estrutura sintác-tica da linguagem que as representa.

Esta ferramenta é apresentada com mais detalhe aquando da apresentação do sistema GAMA-X, o que será feito no capítulo 8.

6.3- Exemplo de Aplicação do MASS.

Um dos mais interessantes e exigentes exemplos de sistemas interactivos onde o MASS constituiu o modelo de referência para a construção da IU, foi o projecto SOUR[23] no qual o objectivo era a construção de um sistema de CASE[24] baseado num repositório de objectos, sistema no qual mecanismos de comparação de ob-jectos, classificação automática e de "interrogações com grau de incerteza"[25] deveriam ser implementados, devendo a facilidade de utilização do sistema ser um ponto a ter em atenção. Dos utilitários mais exigentes em termos de "input" são de referir o Classificador de Objectos (*o Conceptualizer*) e o Sistema de Inter-rogação Inteligente com Incerteza (o *Intelligent fuzzy-Query System*). O *Concep-tualizer* deveria dar todo o apoio sintáctico e semântico ao utilizador permitindo--lhe inserir de forma garantidamente correcta toda a muita informação de clas-sificação de um objecto na hierarquia de objectos do sistema, apresentando-lhe a estrutura do objecto a introduzir e, para cada atributo dependente do contex-to, os valores passíveis de lhe poderem ser atribuídos, assim se procurando ga-rantir a correcção semântica destes [Systena e SSS 94].

A proximidade de alguns destes requisitos de interacção com as próprias ca-racterísticas do MASS levou a que algumas directivas do mesmo fossem aplica-das no projecto.

A assistência semântica para as operações de inserção de objectos no repo-sitório é evidenciada pela figura seguinte que representa o ecrã correspondente à operação de "conceptualização", sendo de realçar a escolha da classe do objec-to a inserir, realizada a partir de uma classe garantidamente existente dado que deve ser seleccionada a partir da hierarquia de classes no momento existente no repositório. A apresentação da hierarquia de classes da forma como é realizada no ecrã apresentado na figura 6. 11 é uma decisão de projecto da IU, ainda que corresponda à estratégia de assistência semântica advogada.

---

[23] de SOftware Use and Reuse, projecto EUREKA EU-379, envolvendo o INESC Braga e três empresas do grupo Olivetti.

[24] na realidade um sistema de CASE parametrizado, ou seja, um sistema de META-CASE.

[25] *fuzzy queries*.

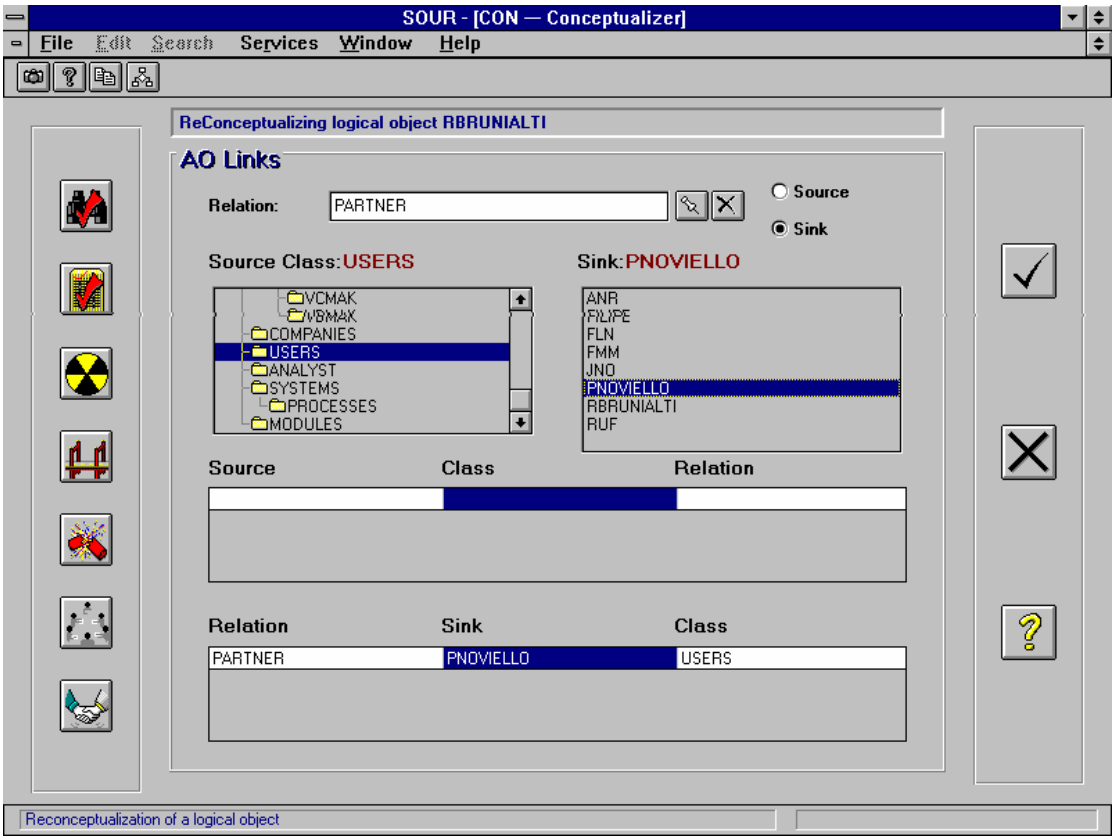

Fig. 6.11 - Conceptualização Assistida.

Note-se ainda que seleccionada uma classe, os identificadores das respecti-vas instâncias são apresentados ao utilizador para que este possa estabelecer li-gações entre o objecto a ser classificado e qualquer destes (um *link*). Deste mo-do a ligação será sempre realizada entre objectos de facto existentes no sistema. A figura 6.12 é um outro exemplo de assistência semântica, neste caso no con-texto do Sistema IQS. Neste sistema de "query" duas preocupações foram resol-vidas recorrendo a algumas das ideias nesta tese expostas. Em primeiro lugar, as diversas formas sintácticas para construção de frases de "query" foram anali-sadas, e de seguida sete *arquétipos* foram considerados como representativos das várias formas de interrogação do repositório. As "queries" passaram, assim, a ser "templates" a instanciar pelo utilizador de forma interactiva, recorrendo a menus e escolhas sobre listas de dados válidos.

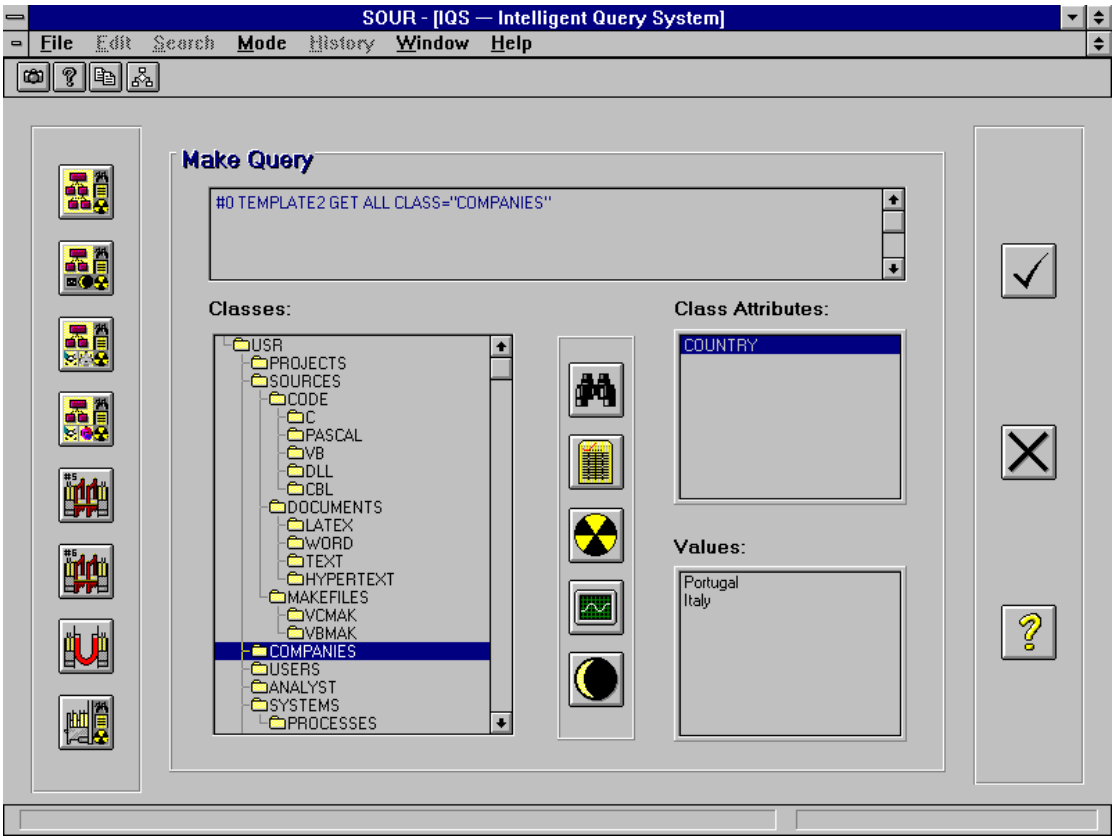

Fig. 6.12 - Exemplo de "Query" Assistido.

A figura 6.12 representa uma "janela" do IQS onde se realiza a síntese de uma "query" com a estrutura correspondente a um dos "templates" definidos. De notar a apresentação dos valores semanticamente seleccionáveis como valores correctos para atribuição a dado atributo. Este tipo de assistência semântica vai permitir que interrogações ao repositório semanticamente incongruentes não possam nunca ser realizadas.

Note-se até que, neste contexto em que a linguagem de interacção é uma lin-guagem de "query" sobre um repositório, o esforço de proporcionar contexto se-mântico adequado à formulação de expressões de interrogação semanticamente correctas, pode ser considerado, no mínimo, equivalente ao esforço usual de op-timização realizado sobre as estruturas sintácticas da linguagem de "query", que não garante em geral sucesso no "query" dado apenas se preocupar em optimi-zar os percursos lógicos ideais da interrogação e não a obtenção de resultados.

6.4.- Sumário.

Apresentaram-se neste capítulo duas das principais contribuições desta tese, o modelo MASS e o formalismo dos Guiões de Interacção. Como se procurou

mos-trar através da apresentação de exemplos da sua aplicação, são até contribui-ções suficientemente genéricas para que possam ser consideradas fora do en-quadramento sistémico e metodológico que é proposto como contexto da tese.

Em primeiro lugar sugeriu-se um modelo de interacção, o *Modo Assistido Sintáctico-Semântico*, o MASS, que junta ao paradigma da interacção por edição estrutural dos *arquétipos*, uma base linguística que permite a sua utilização num grande número de aplicações interactivas. Por outro lado, e tendo em aten-ção algumas características comuns dos utilizadores, foram incorporadas no modelo determinadas propriedades que procuram satisfazer tais características, e que devem ser vistas pelos implementadores como requisitos a satisfazer pela IU. Mostrou-se ainda como o MASS pode ser usado na recuperação tecnológica de IU, caso a comunicação com a aplicação assuma uma base linguística.

A implementação do MASS em novas aplicações pode ser realizada usando o paradigma dos arquétipos apresentado no capítulo 5, em particular no contexto de aplicações que assumem uma clara vocação para a construção de objectos com uma estrutura de representação rica (cf. objectos gráficos). Para aplicações interactivas de tipo mais geral e desenvolvidas de raiz, como se advoga no con-texto da tese, a implementação do MASS é apoiada pelo formalismo dos Guiões de Interacção.

Os GI, formalismo para a especificação de controladores de diálogo, foram igualmente apresentados, tendo sido definida uma semântica operacional, com base em Redes de Petri particulares, que garante que protótipos dos controlado-res assim especificados podem ser executados, o que se considera crucial.

O poder expressivo dos GI foi comparado, em exemplos concretos, com outros formalismos, mesmo com formalismos baseados em processos, podendo a sua capacidade expressiva ser considerada, no mínimo, equivalente, indepen-dentemente de comparações de facilidade de utilização, que não foram realiza-das. No entanto, e contrariamente à maioria das notações de especificação com que foram comparados, ou que foram anteriormente referenciadas, a utilização dos GI é facilitada pela existência de ferramentas de apoio, e por possibilitarem a geração de protótipos dos controladores do diálogo e da própria IU.

Exemplos de obediência ao MASS em contextos de grande exigência quanto a facilidades interactivas foram igualmente apresentados, em particular no âmbi-to do desenvolvimento do sistema de CASE designado por SOUR, e das respecti-vas "ferramentas" de Classificação e Interrogação sobre um repositório de objec-tos.