GAMA-X

Geração Semi-Automática de Interfaces Sensíveis ao Contexto

José Francisco Creissac Freitas de Campos

Departamento de Informática
Universidade do Minho

UM mes/jfc

© Universidade do Minho 25 de Setembro de 1993

Aos meus Pais.
Às minhas Irmãs.
Enfim, a todos lá em Casa.

"A experiência é Madre de todas as cousas"
Duarte Pacheco Pereira - in *"Esmeraldo de Situ Orbis"*

Ao Engº Mário Martins devo o ter-me iniciado na área da Interacção Humano-Computador, bem como a oportunidade de realizar este trabalho e todo o apoio e orientação que me prestou.

O meu agradecimento a todos no Grupo de Fundamentos da Computação, em especial ao Engº José João - sempre disponível e entusiasta - pelos esclarecimentos e sugestões e pela cedência das suas funções de I/O para XMetoo; ao Zé Carlos, pelo companheirismo e incentivo nos momentos mais difíceis.

O meu obrigado, também, a todos os colegas de Mestrado, em especial aos que passaram Agosto em Gualtar.

Índice

1 Introdução . . . 1
  1.1 Objectivos . . . 1
  1.2 Estrutura da Tese . . . 2
2 Sistemas Interactivos . . . 3
3 Especificação Formal de Interfaces . . . 6
  3.1 Modelos Cognitivos . . . 6
  3.2 Modelos Estruturais . . . 9
  3.3 Especificação Formal de Diálogos . . . 14
  3.4 Fechando a Ponte . . . 18
  3.5 Geração Automática de Sistemas Interactivos . . . 19
4 A Linguagem CAMILA . . . 21
5 Guiões de Interacção . . . 23
  5.1 Tarefas a Especificar . . . 23
  5.2 Leitura dos Argumentos/Visualização do Resultado . . . 23
  5.3 Escolha da Operação . . . 29
  5.4 Especificação do Comportamento . . . 30
    5.4.1 *Petri Nets* . . . 30
    5.4.2 Guiões de Interacção . . . 31
    5.4.3 A Expressão Mais Simples . . . 32
    5.4.4 Sequenciação . . . 32
    5.4.5 Concorrência Síncrona . . . 33
    5.4.6 Concorrência Assíncrona . . . 33
    5.4.7 Alternativa . . . 34
    5.4.8 Repetição . . . 34
    5.4.9 Alguns Casos Particulares . . . 35
    5.4.10 Comandos . . . 36
  5.5 Guiões de Interacção e a Linguagem CAMILA . . . 37
  5.6 Guiões de Interacção e Tipos de Dados . . . 38
  5.7 Guiões de Interacção e Outros Formalismos . . . 38
    5.7.1 Ligação à Componente Semântica . . . 38
    5.7.2 Expressividade . . . 39
    5.7.3 Ligação à Apresentação . . . 39
6 Descritores da Apresentação . . . 40
7 O Sistema GAMA-X . . . 43
  7.1 Arquitectura do Módulo de Interacção com o Utilizador . . . 43
  7.2 Especificação do Controlador de Diálogo . . . 44
  7.3 Especificação do Modelo da Apresentação . . . 45
  7.4 Especificação do Modelo da Aplicação . . . 45
  7.5 Arquitectura do GAMA-X . . . 46
8 Implementação do MIU . . . 48
  8.1 Perspectiva Global . . . 48
  8.2 O Controlador de Diálogo . . . 49
    8.2.1 Protocolos de Comunicação . . . 49
    8.2.2 Animador de *Petri Nets* . . . 52
    8.2.3 O Interpretador de Instruções . . . 53

8.2.4 A Inclusão dos Tipos 53
8.2.5 Estrutura Geral do Controlador 55
8.3 O Modelo da Apresentação 56
8.4 O Modelo da Aplicação 59
8.5 Utilização e Exemplos 60
8.5.1 Modelo da Apresentação 61
8.5.2 Controlador de Diálogo 61
8.5.3 Modelo da Aplicação 61
8.5.4 Utilização 61
8.6 Versão X11 do *Frontend* 62
9 Conclusões 65
Referências 67
A Especificação CAMILA de um Dicionário 70
B Guiões de Interacção para o Dicionário 71
C Descritores de Apresentação para o Dicionário 73
D Especificação do Modelo da Aplicação para o Dicionário 76
E Especificação NYAGSL do Controlador de Diálogo para o Dicionário . . . 77
F Especificação NYAGSL do Modelo da Apresentação para o Dicionário . . 80
G Especificação NYAGSL do Modelo da Aplicação para o Dicionário 83
H Definições Comuns 84
I Comunicações 85
J Mensagens 87
K Controlador de Diálogo 89

Lista de Figuras

1 Um Modelo GOMS 8
2 Gramáticas Formais 8
3 Dialogue Cell 10
4 Modelo RED-PIE 11
5 Modelo RED-PIE com efeitos observáveis 11
6 Modelo de Seeheim 12
7 Modelo MVC 13
8 Arquitectura UMA 13
9 Exemplo de Gramáticas Multiparty 15
10 Diagramas de Transições de Estado USE 16
11 Exemplo de uma Especificação CAMILA 22
12 Rede de um Guião 31
13 Rede para gi(var) 32
14 Rede para o Operador de Sequenciação 33
15 Rede para o Operador de Concorrência Síncrona 33
16 Rede para o Operador de Concorrência Assíncrona 34
17 Rede para o Operador de Alternativa 34
18 Rede para o Operador de Repetição 35
19 Rede para o Operador de Repetição (versão 2) 35
20 Rede para os operadores de Soma e Concorrência 36
21 Rede para os Comandos OK e CANCEL 36
22 Arquitectura do MIU 43
23 Arquitectura do GAMA-X 47
24 Composição do Controlador de Diálogo 56
25 Composição do Modelo da Apresentação 59
26 Composição do Modelo da Aplicação 60
27 Menu Inicial 62
28 Inserção 63
29 Menu Inicial depois da Inserção 63
30 *View* "Portugues" 64
31 *View* "Ingles" 64

1 Introdução

Nas duas primeiras gerações de computadores, a preocupação com a sua facilidade de utilização era um aspecto secundário. Por um lado, as tecnologias de interacção eram primitivas (cartões e fitas perfuradas), o que levava a que fossem utilizadas apenas por técnicos especializados, por outro, o objectivo principal era aumentar a capacidade de cálculo das máquinas.

O interesse pelo estudo da Interacção Humano-Computador surge com a terceira geração de computadores [MO90]. O aparecimento dos sistemas de *time-sharing* e a divulgação das *video display units* (VDU) tornou os computadores mais acessíveis e passíveis de utilizar de forma interactiva. O utilizador passa a ter acesso directo à máquina, deixam de existir "intermediários" e pode estabelecer-se um diálogo entre o utilizador e o *software* que ele pretende utilizar. À combinação computador e *software* capaz de manter um diálogo com o utilizador chamamos **Sistema Interactivo**. O aparecimento do computador pessoal, ao vulgarizar a utilização dos computadores em ambientes não profissionais, incrementou significativamente o interesse por estes sistemas.

Esta divulgação do computador como instrumento de trabalho nas mais diversas áreas, e a constante exigência de sistemas cada vez mais poderosos por parte dos utilizadores (ou talvez pelas leis da concorrência), aliada ao rápido crescimento da capacidade das máquinas, tem levado a um aumento crescente da complexidade dos sistemas *software*. Este processo atingiu um ponto tal que é hoje aceite estar a utilidade de um computador mais dependente da facilidade com que pode ser utilizado do que da sua capacidade intrínseca de cálculo [HH89, LJBS78]. Uma boa Interface Homem-Máquina torna-se, assim, essencial.

1.1 Objectivos

Apesar da grande quantidade de tecnologia disponível, a construção de uma interface é ainda um processo difícil e demorado, sendo, frequentemente, necessário recorrer a protótipos para experimentação e validação das decisões tomadas ao longo do processo de desenvolvimento.

A causa fundamental deste estado de coisas prende-se com a falta de bases teóricas adequadas que permitam conhecer em profundidade os mecanismos pelos quais se estabelece o diálogo entre o Humano e o Computador. É esse conhecimento que permitirá o surgimento dos modelos formais e das ferramentas que possibilitarão um desenvolvimento mais rápido e seguro das interfaces. Torna-se então evidente a necessidade de aplicar métodos formais ao desenho e implementação da camada interactiva dos sistemas, tal como se faz já para a sua camada computacional.

As abordagens actualmente propostas pecam, de um modo geral, pela demasiada separação entre a aplicação e a interface. Embora um sistema interactivo deva ser separado em camada computacional e camada de diálogo, uma boa interface não pode estar "isolada" da aplicação, devendo apresentar suficiente *feedback* semântico para que o utilizador consiga interpretar em cada instante o que se está a passar.

No trabalho aqui descrito é apresentado um método para especificação de interfaces orientadas ao comando (**Guiões de Interacção**) em que é posta especial atenção na ligação semântica entre a interface e a aplicação (cf. Modo Assistido [MO90]).

A especificação de diálogos assíncronos, especialmente os concorrentes, é outro aspecto difícil de tratar pelas abordagens tradicionais (Gramáticas, Diagramas de Transição de

Estado). É proposta a utilização de uma linguagem de descrição de traços possíveis de eventos como modelo de especificação de diálogo dos Guiões, sendo utilizadas *Petri Nets* como modelo de implementação.

Com o conhecimento do funcionamento de um sistema surge a tendência para a sua automatização. Neste contexto é apresentada a arquitectura de um Sistema de Gestão de Interfaces com o Utilizador para a linguagem CAMILA - uma linguagem modular de especificação por modelos -, o GAMA-X. A implementação da componente de *runtime* do sistema (Módulo de Interacção com o Utilizador) permite demonstrar as potencialidades do método proposto.

1.2 Estrutura da Tese

No Capítulo 2 são introduzidas algumas noções fundamentais na área dos Sistemas Interactivos. De seguida, Capítulo 3, apresentam-se as várias sub-áreas de estudo dentro da IHC, bem como exemplos de trabalhos nelas efectuados.

A linguagem Camila é apresentada no Capítulo 4.

No Capítulo 5 são descritos os Guiões de Interacção - o formalismo proposto para a especificação da componente de diálogo das Interfaces - e no 6 os Descritores de Apresentação utilizados para definir o aspecto gráfico da mesma Interface. A arquitectura do sistema GAMA-X é descrita no Capítulo 7. A implementação da componente de *runtime* do sistema é descrita no Capítulo 8, sendo apresentados alguns exemplos de utilização do sistema descrito.

Finalmente, no Capítulo 9, são apresentadas as conclusões do trabalho, bem como pistas para a sua continuação.

2 Sistemas Interactivos

O desenvolvimento de uma Interface é um processo particularmente complexo. Para além dos aspectos metodológicos e tecnológicos presentes no desenvolvimento de qualquer sistema *software*, é necessário ter também em consideração aspectos da área de factores humanos: ergonomia, consistência, facilidade de aprendizagem, etc.

A área de Interacção Humano-Computador (IHC) surge como resposta a esta constatação. Os primeiros trabalhos na área foram efectuados no âmbito da Computação Gráfica e estavam relacionados com aspectos de factores humanos [HH89]. Centenas de regras foram compiladas, embora apenas uma pequena parte tenha sido objecto de validação experimental. Estas regras visavam fundamentalmente definir quais as características de uma boa Interface; não definiam, no entanto, como se deveria desenvolver uma interface de modo a que lhes obedecesse, ou como provar que uma dada interface as verificava.

Com a crescente complexidade dos Sistemas Interactivos, tornou-se evidente serem necessárias, para construir interfaces que obedecessem a essas regras e princípios, ferramentas e metodologias que auxiliassem e guiassem o seu desenvolvimento. Tratava-se de passar de uma fase em que a interface era programada de uma forma mais ou menos *ad hoc*, para outra em que ela podia ser descrita e gerada utilizando metodologias e ferramentas apropriadas. Um marco importante nesta transição foi o estabelecimento do Princípio da Separação (ou da Independência de Diálogo), que propõe que uma aplicação interactiva esteja dividida em duas camadas: a camada computacional, correspondente à aplicação propriamente dita, e a camada de diálogo, responsável pela comunicação entre o utilizador e a camada computacional. O princípio diz ainda serem as duas camadas independentes, tanto a nível de desenho como de implementação.

Foi a partir deste ponto que a IHC se pôde afirmar como área de estudo autónoma, pois o desenho e desenvolvimento da camada interactiva passaram a estar separados do desenho e desenvolvimento da camada computacional.

Para além de permitir o estudo, de modo independente, de metodologias e técnicas óptimas para cada uma das componentes, a adopção deste princípio tem ainda a vantagem de permitir que alterações feitas numa das camadas (principalmente a nível de implementação) possam ter poucas implicações (ou mesmo nenhuma) na outra. A sua implementação é, no entanto, difícil de conseguir e a necessidade de comunicação entre as camadas provoca alguma degradação do desempenho.

Uma primeira forma rudimentar do Princípio da Separação pode ser reconhecida no aparecimento do conceito de *device independence*, que isolou o programador das características particulares dos diferentes aparelhos de interacção.

Numa segunda fase surgiram os geradores de aplicações, que permitiam descrever em linguagens de alto nível os écrans e as leituras da aplicação, e os *Toolkits*, colecções de rotinas de tratamento de écran e leitura/escrita. Exemplos de *Toolkits* conhecidos são o Smalltalk-MVC [Dig87], SunView [Sun90], Xtoolkit [QO90a], Macintosh Toolbox [Ste87], DesQView/X [Rad90] ou o MsWindows [Ste87]. Um tipo particularmente importante de *Toolkit* são os *Window Systems*, vocacionados para fornecerem serviços de gestão de *Workstations*. Dividem o écran físico em écrans virtuais (chamados janelas) possibilitando o desenrolar de várias sessões de trabalho em simultâneo no mesmo posto de trabalho, estando a cada uma delas associada uma janela principal.

Podemos identificar dois tipos básicos de *Window Systems*. Os *Kernel Based*, que são construídos como extensões ao sistema operativo - pelo que ficam limitados à máquina em

que estão a ser executados (por exemplo SunView e MsWindows) -, e os Cliente-Servidor, nos quais o *Window System* passa a ser mais um processo na máquina, funcionando como servidor de pedidos gráficos e de leitura/escrita. Deste modo, é possível ter um processo remoto a fazer pedidos ao *Window System* através da rede e, assim, utilizar os dispositivos de entrada e saída de dados da máquina local. Exemplos deste tipo são o X11 [QO90b] e o NeWS [Ros87].

Com estas ferramentas, era já possível separar o desenho e a implementação da componente de diálogo do desenho e implementação da camada computacional de uma aplicação interactiva. Esta separação dependeria, no entanto, de uma vontade deliberada da equipa de desenvolvimento do sistema para a obter, já que as ferramentas, por si só, não a garantem. Elas limitam-se a fornecer serviços, não impondo políticas. Nesta fase, embora fosse possível implementar Independência de Diálogo, ela não era, ainda, verdadeiramente suportada pelas ferramentas disponíveis.

Foi com o aparecimento dos Sistemas de Gestão da Interface com o Utilizador (SGIU) que o Princípio da Separação passou a ser realmente suportado. Um SGIU é um conjunto de ferramentas para o desenho e a execução da componente de diálogo dos sistemas interactivos [CC90, GR90] que permite desenvolver a Interface de um modo independente do desenvolvimento da componente computacional. Idealmente deverá proporcionar um ambiente interactivo para o desenho, prototipagem, avaliação, implementação ou execução e manutenção de Interfaces com o Utilizador [BCMR90, HH89]. É importante não confundir um SGIU com um *Toolkit*, pois este último fornece facilidades para o desenvolvimento da Interface, mas não um ambiente integrado e uma metodologia para esse desenvolvimento.

A separação entre componente de diálogo e componente computacional, levanta o problema de decidir quem tem o controlo sobre o funcionamento global do sistema. Quando é a componente interactiva a controlar o funcionamento do sistema, a componente computacional é vista como um conjunto de rotinas que a primeira invoca quando necessário, estamos perante o controlo *Dialogue Dominant*. Na situação inversa, em que o controlo está na componente computacional, dizemos que este é *Computacional Dominant*. Entre estes dois extremos podemos ter um controlo misto (as duas componentes invocam-se mutuamente) ou balanceado (existe uma terceira componente que o detém).

Um outro aspecto importante a ter em conta quando se pensa em Interacção Humano-Computador, tem a ver com o tipo de diálogo que pretendemos. Podemos identificar dois tipos básicos de diálogo [Mar92]: diálogo sequencial e diálogo assíncrono. Na categoria de diálogos sequenciais então incluídas interacções do tipo pedido-resposta, linguagens de comandos, navegação por hierarquias de menus e *data entry*. A principal característica dos diálogos sequenciais é a de que, em cada instante, o sistema apresenta apenas uma tarefa ao utilizador. Em diálogos assíncronos são apresentadas ao utilizador, em cada instante, diversas tarefas que ele pode realizar. Incluídas neste tipo de diálogo encontram-se normalmente as interfaces por manipulação directa. Nestas, o utilizador realiza operações manipulando representações visuais dos objectos (os *icons*): se, por exemplo, quiser remover um ficheiro, coloca-o (o seu *icon*) sobre o cesto do lixo (o *icon* da operação de remoção). Associado a este tipo de interacção está o conceito de diálogo *multi-thread*, numa referência à possibilidade que é dada ao utilizador de, em cada momento, prosseguir o diálogo segundo vários caminhos. Se mais que uma tarefa puder ser executada ao mesmo tempo, então o diálogo diz-se concorrente. Embora os dois tipos de interacção sejam utilizados, a tendência é cada vez mais para os diálogos assíncronos, principalmente por manipulação directa, dada a sua aparente maior facilidade de aprendizagem e utilização.

Com o objectivo de conseguir interfaces cada vez mais amigáveis, tem sido levadas a cabo tentativas, especialmente no âmbito da Inteligência Artificial, para incorporar nas interfaces Modelos do Utilizador [MF93]. Deste modo o sistema interactivo tenta analisar o comportamento do utilizador e, por reacção, adaptar-se, em cada momento, às suas capacidades e conhecimento. Tais trabalhos não têm conseguido, no entanto, tanto sucesso como o esperado, pelo que abordagens alternativas têm sido procuradas.

Em [MO90] é argumentado que a complexidade inerente ao ser humano implicará sempre uma incompletude desses modelos face à realidade e em consequência tais interfaces serão sempre, em maior ou menor grau, inadequadas e ineficazes. Para a resolução deste problema é proposta a utilização de modelos implícitos do utilizador. Se conseguirmos identificar as características humanas que podem influenciar o desempenho de um sistema interactivo e concebê-lo de modo a que actue de modo preventivo em relação às mesmas, teremos o problema resolvido. A ideia é então, não reagir *a posteriori* às acções do utilizador, mas agir de um modo preventivo, guiando a interacção de modo a que este não cometa erros e utilize o sistema de uma forma correcta. A este modo de interacção chamou-se Modo Assistido.

Para a definição de Modo Assistido são consideradas como essenciais as seguintes características do utilizador:

- **Incorrecção**. O sistema não deverá permitir ao utilizador executar acções inválidas. Para tal deverá determinar, *a priori*, quais as acções válidas no contexto actual e validar, tão cedo quanto possível, os valores introduzidos.

- **Incompletude**. O sistema só deverá permitir ao utilizador que envie um comando para execução quando todos os seus argumentos estiverem instanciados.

O Modo Assistido caracteriza-se, portanto, por associar à assistência sintáctica a assistência semântica.

Se para um utilizador novato uma interface por menus é, normalmente, mais intuitiva e fácil de utilizar, já um utilizador experiente consegue, usualmente, maior produtividade pela utilização directa dos comandos. Uma interface em Modo Assistido deverá, então, prever dois tipos de interacção:

- Modo Comando - interacção por meio de linha de comandos;

- Modo Menu - interacção por menus.

Sempre que em Modo Comando for detectado algum erro, deverá ser efectuada a mudança automática para Modo Assistido, facilitando, assim, a correção do mesmo.

3 Especificação Formal de Interfaces

Na Engenharia de *Software*, a necessidade de correcção dos sistemas desenvolvidos e de clareza e tratabilidade das especificações produzidas, levou ao aparecimento de Métodos Formais de Especificação. De igual modo, na área de IHC têm sido feitos estudos para aplicar métodos formais ao desenvolvimento de Interfaces (quer derivados dos anteriores, quer desenvolvidos especificamente para este caso).

Os métodos formais aplicados na área da Engenharia de *Software* vêem-no como uma construção matemática e procuram descrever o comportamento dos sistemas de forma rigorosa e abstracta, libertando assim a especificação de pormenores de implementação concreta. Em IHC, para além do *Software* é necessário considerar também o utilizador, com as suas características de imprevisibilidade, imprecisão, incorrecção, etc. Deste modo, a investigação na área tem sido conduzida basicamente segundo uma de duas perspectivas [Abo92]:

- Numa, é dado ênfase ao ponto de vista do utilizador. Tendo como principal preocupação os aspectos psicológicos, cognitivos e sociológicos da interacção, procuram desenvolver-se teorias que modelem o seu comportamento. À classe de modelos produzidos nesta área de estudo chamaremos Modelos Cognitivos.

- Na outra, os investigadores focam mais o ponto de vista do sistema, preocupando-se com os aspectos relacionados com a Engenharia de *Software*, considerando muitas vezes que os aspectos ergonómicos só por experimentação poderão ser analisados. Temos então Modelos Estruturais e Modelos de Especificação da Interface (principalmente Especificação do Diálogo).

Independentemente do seu tipo, um bom modelo deverá possuir, idealmente, as seguintes características [Jac83, Was85]:

- A especificação deverá ser de fácil leitura.
- Deverá ser preciso e formal, não deixando dúvidas quanto ao comportamento do sistema.
- Deverá facilitar a verificação da consistência.
- Deverá ser suficientemente poderoso, de modo a permitir especificar sistemas minimamente complexos.
- Deverá especificar apenas o que o sistema faz e não como o faz.
- Deverá permitir a construção de um protótipo a partir da especificação da interface.
- A sua estrutura deverá estar relacionada com o modelo mental que o utilizador tem do sistema.

3.1 Modelos Cognitivos

O sucesso de uma aplicação interactiva depende, em última análise, da facilidade com que o utilizador consegue aprendê-la e utilizá-la. Quando alguém utiliza um computador, pretende, através dele, realizar uma, ou mais, tarefas: levantar dinheiro numa máquina ATM, ou escrever uma dissertação de mestrado, por exemplo. A facilidade com que

utilizará o sistema, dependerá da relação que existir entre as acções que ele mentalmente considera necessárias para realizar essa tarefa e as interacções com o sistema necessárias para o conseguir. Por outro lado, o utilizador pode não saber exactamente qual a tarefa que pretende realizar (ou como realizá-la) e, nesse caso, o seu desempenho será grandemente afectado pela qualidade da informação que lhe for proporcionada pela interface.

Os Modelos Cognitivos são utilizados, exactamente, para análise e descrição das tarefas que o utilizador tem para realizar e preocupam-se com o conhecimento que o utilizador tem sobre uma determinada tarefa e o modo de a executar. A noção de tarefa varia conforme o nível de abstracção que o modelo descreve, podendo ir de algo tão abstracto como "*abrir um ficheiro*", até algo tão concreto como os toques nas teclas da máquina ATM.

Dois exemplos de Modelos Cognitivos são a Teoria GOMS [FN85] e as Gramáticas Formais propostas por Reisner [Rei81]. Tanto um como outro se baseiam em notações do tipo BNF (Backus Naur Form).

A teoria GOMS divide o comportamento cognitivo humano em quatro componentes:

- **Objectivos**, correspondem às tarefas que o utilizador deve realizar. Objectivos complexos podem ser divididos em subtarefas (as quais podem, elas próprias, ser Objectivos).

- **Operadores**, que são as acções atómicas a nível físico, cognitivo ou perceptual (dependendo do grão de análise) que constituem as acções do utilizador.

- **Métodos**, os "procedimentos aprendidos" para atingir Objectivos, são constituídos por sequências de outros Objectivos e Operadores e dependem do estado da memória do utilizador e da tarefa em questão.

- **Regras de Selecção**, que permitem escolher qual o Método a utilizar para atingir um determinado Objectivo e formam a estrutura básica de controlo da Teoria.

Estes quatro componentes são utilizados para descrever a interacção Homem-Máquina e, experimentalmente, fazer previsões de tempo e de sequências de comandos utilizadas, de modo a prever o desempenho e o comportamento humanos. Na fig. 1, é apresentado um extracto de um Modelo GOMS para a tarefa de posicionar o cursor numa linha de texto [FN85]. Ao Objectivo em questão estão associados três métodos: **use Line-feed Method**, **use Quote-String Method** e **use Scroll-Method**. Cada um deles por sua vez é composto por outros subobjectivos. O utilizador terá que, iterativamente, escolher o método a utilizar até que o cursor esteja na linha pretendida. Em [FN85] é mostrado que a notação utilizada para a descrição da interacção é equivalente a uma Gramática Livre do Contexto.

Reisner utiliza Gramáticas Formais Livres de Contexto para descrever interfaces e comparar analiticamente desenhos alternativos. A metalinguagem proposta por Reisner é composta pelos metasímbolos "::=" (é composto por), "|" (ou), "+" (concatenação) e "-" (em simultâneo). Na fig. 2 são apresentadas algumas produções da definição de uma aplicação de desenho interactiva (ROBOART1) apresentada em [Rei81]. Os terminais são apresentados em maiúsculas e os não terminais em minúsculas. A tónica é posta nas acções que o utilizador terá de recordar durante a execução de uma tarefa para, deste modo, analisar a facilidade com que ele consegue utilizar a interface. Neste exemplo podemos verificar que, para fazer uma figura colorida, o utilizador pode optar por fazer

```
Goal: Locate Line
.     Choose Command                        ... (if task not at
.     [select Goal: use Line-feed Method          current line)
.             .     Goal: specify Command
.             Goal: use Quote-String Method
.             .     Goal: specify Command
.             .     Goal: specify Argument
.             Goal: use Scroll-Method
.             .     Goal: specify Command
.             .     Goal: specify Argument] ... (repeat until
                                                 at task line)
```

Figura 1: Um Modelo GOMS

```
colored shape ::= color + shape | shape + color
color ::= new color | old color | starting default color
new color ::= CURSOR IN RED | CURSOR IN BLUE | ...
old color ::= NULL
starting default color ::= NULL
```

Figura 2: Gramáticas Formais

primeiro a figura e seleccionar a cor depois, ou ao contrário. Para seleccionar uma nova cor ele terá que colocar o cursor sobre ela (as cores são apresentadas no écran), ou, caso pretenda utilizar a cor anterior ou a cor inicial por defeito, não fazer nada (símbolo "**NULL**").

A facilidade de utilização é definida em termos do tamanho do conjunto de símbolos terminais (cada não terminal corresponde a uma acção do utilizador pelo que, quantos mais não terminais, mais acções o utilizador terá de recordar), do comprimento das sequências de terminais necessárias para realizar as tarefas (quantas acções são necessárias para realizar uma dada tarefa) e da consistência estrutural sintáctica (acções similares devem ter estrutura sintáctica semelhante). No fundo está a definir-se a facilidade de utilização em termos da estrutura sintáctica da interface, deixando de lado os conceitos semânticos envolvidos. Esta definição é discutível na medida em que uma tarefa sintacticamente simples pode ser semanticamente complexa e vice-versa. Por exemplo, nos teclados portugueses dos computadores pessoais, a tarefa semanticamente simples de escrever o caracter "{", complica-se sintacticamente com a necessidade de premir três

teclas em simultâneo (sendo que uma delas só pode ser premida depois de as outras duas já o terem sido). Numa fase posterior foram ainda incluídas no modelo previsões sobre o comportamento humano, nomeadamente em termos de tempos de resposta do utilizador.

É pertinente questionarmo-nos sobre a validade da utilização de Gramáticas Livres de Contexto para a descrição das Interfaces. Como tais gramáticas apenas admitem um não terminal no lado esquerdo de cada produção, tal utilização equivale a assumirmos que o processo cognitivo é sequencial e monotónico (já que apenas uma tarefa pode ser realizada em cada instante, o que deixa de fora os diálogos assíncronos) e também que a capacidade humana de atingir um determinado objectivo é independente do contexto actual do utilizador e da sua história. Tais pressupostos dificilmente serão verdadeiros e é a própria Reisner a concluir que, embora sendo um bom ponto de partida, as Gramáticas Formais não são a resposta definitiva, sendo necessário encontrar novos formalismos.

Comparando as características dos dois modelos podemos ver que, sendo os dois baseados em Gramáticas Livres do Contexto, o seu poder descritivo é necessariamente o mesmo. O seu poder analítico é no entanto diferente. Enquanto o modelo GOMS possibilita uma análise experimental, quantitativa, da interacção, abordando o problema da selecção de métodos para a realização de uma dada tarefa, as gramáticas propostas por Reisner estão mais vocacionadas para uma análise qualitativa das interfaces, de forma analítica, mas sem abordar o problema de selecção do método a utilizar [FN85].

Embora a perspectiva de factores humanos seja importante e deva ser, tanto quanto possível, englobada no processo de desenho e desenvolvimento da interface, neste trabalho dá-se particular ênfase aos aspectos de Engenharia de *Software* do desenho das interfaces. O problema é abordado mais ao nível de como construir interfaces e, embora exista a preocupação de incluir mecanismos e características que possibilitem o desenho e implementação de interfaces de qualidade, o problema da determinação do que é uma tal interface não será abordado.

3.2 Modelos Estruturais

Dentro deste tipo de modelos podemos identificar duas classes distintas: modelos que descrevem a estrutura da Interacção Humano-Computador e modelos que descrevem a estrutura do sistema que implementa a Interface. A estes últimos chamaremos **Abstracções Arquitecturais**. Quer num caso, quer no outro, este tipo de modelos é muito importante, pois são eles que vão servir de referência ao processo de desenho e implementação da Interface. Os modelos actuais estão, geralmente, mais orientados para o diálogo sequencial. Dada a natureza pouco estruturada dos diálogos assíncronos (o utilizador pode fazer tudo, ou quase tudo, sempre, ou quase sempre!), o desenvolvimento de Modelos Estruturais desse tipo de diálogo é mais difícil e encontra-se mais atrasado.

Relativamente à primeira classe de modelos, eles preocupam-se, como foi dito, com a descrição do processo de comunicação entre o utilizador e o sistema, descrevendo de forma genérica a estrutura e o fluxo de controlo da Interacção Humano-Computador. Uma possibilidade, inicialmente apresentada por Foley [HH89], é vermos essa interacção de um ponto de vista linguístico. Foley dividiu a interface em quatro níveis:

- **Conceptual**, corresponde aos objectivos e funções do sistema;
- **Semântico**, que engloba as operações de leitura e apresentação de informação;
- **Sintáctico**, as sequências de *tokens* que invocam acções semânticas;

- **Léxico**, o qual define a estrutura dos *tokens*.

Outros autores subdividem ainda estes níveis em subníveis mais detalhados.

Hix e Hartson [HH89] propuseram um modelo linguístico chamado *Dialogue Transaction Model*, baseado em relações simples entre Linguagens Formais e Máquinas de Transições de Estados e que é o Modelo Estrutural do Sistema *Dialogue Management System* por eles proposto. Resumidamente, neste modelo estão definidos objectos linguísticos, e objectos construcionais que os processam. Os objectos linguísticos são:

- **Lexema**, a menor unidade possível de *input*;
- **Token**, a menor sequência de Lexemas que tem significado para a aplicação;
- **Frase**, uma sequência de Tokens.

Os Lexemas são processados por objectos construcionais chamados **Acções**, que fazem corresponder a cada acção do utilizador o correspondente Lexema; os Tokens são processados por **Interacções**, que fazem corresponder a uma sequência de Lexemas o correspondente Token; as Frases são processadas por **Transacções**, que agrupam Tokens para as formar. Cada objecto construcional é composto por objectos constituintes: de leitura, de escrita e de computação. A estrutura do diálogo é então representada, por combinação dos objectos descritos, em diagramas semelhantes a *Flow Charts*.

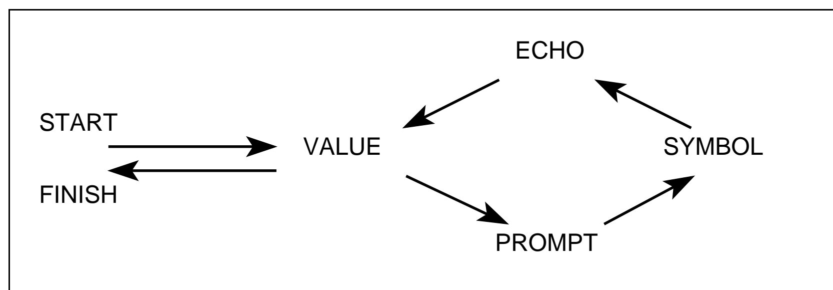


Figura 3: Dialogue Cell

Têm também sido propostos Modelos Estruturais não linguísticos. As *Dialogue Cells*, propostas por Borufka [BP81], são constituídas por quatro elementos (ver fig. 3):

- **Prompt**, que prepara o sistema para leitura e indica o tipo de informação a ser lida;
- **Symbol**, que é o valor lido do utilizador;
- **Echo**, que é a interpretação do simbolo lido feita pelo sistema;
- **Value**, que estabelece a correspondência entre o símbolo e dados utilizáveis pela aplicação.

Uma célula de diálogo descreve diálogo sequencial. É-lhe fornecido um valor inicial e, depois de executados os quatro elementos pela ordem indicada pelas setas, devolve o resultado da avaliação do símbolo lido. Organizando várias células hierarquicamente podemos descrever uma interacção mais complexa.

O modelo de eventos tem também sido utilizado como base para modelos estruturais. Benbasat e Wand [BW84], por exemplo, propuseram um modelo baseados em Eventos de Interacção compostos por um *prompt* (análogo ao Prompt das Células de Diálogo), o *input* (corresponde ao Symbol do modelo anterior) e uma acção de processamento e controlo de fluxo para determinar o próximo evento. Estão também previstas validações de dados, *help* e um mecanismo de *Escape* que permite ao utilizador ignorar o pedido de leitura actual.

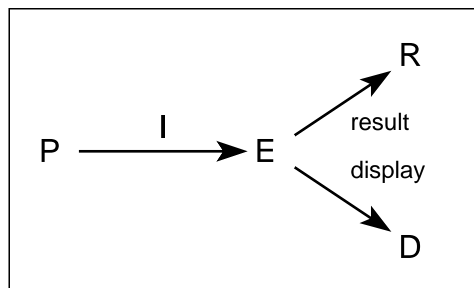


Figura 4: Modelo RED-PIE

Um outro modelo não linguístico é o RED-PIES [DR85]. Segundo este modelo, um sistema interactivo é caracterizado (do ponto de vista do utilizador) pelos comandos que aceita e pelos resultados que produz em função destes. Uma sequência válida de comandos define um programa, ao conjunto de todos os programas chamamos **P**. **E** é o conjunto de todos os efeitos possíveis e a relação entre a sequência de comandos já aceite pelo sistema e o efeito produzido é uma função **I**. Para permitir distinguir efeitos respeitantes ao produto final de efeitos que fazem parte do processo interactivo, **E** é mapeado nos espaços **R** (resultado) e **D** (display) por **result** e **display** respectivamente, dando origem ao esquema da fig. 4.

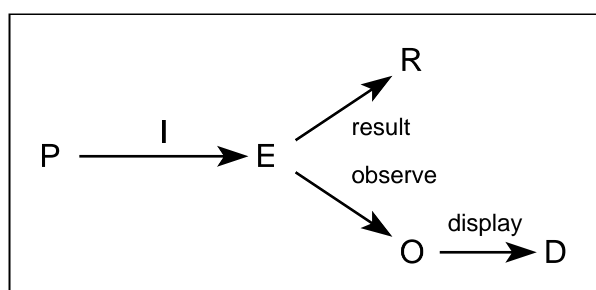


Figura 5: Modelo RED-PIE com efeitos observáveis

Como, por vezes, o volume de informação que o sistema produz é demasiado elevado para ser todo apresentado de uma única vez, torna-se necessário apresentar apenas parte da informação e fornecer comandos que possibilitem a navegação na totalidade da mesma. Por exemplo, ao fazermos um *preview* de um dado ficheiro, é-nos apresentada apenas uma página de cada vez e podemos usar comandos para mudar para a página anterior/seguinte. Estes comandos não alteram verdadeiramente o estado da aplicação, mas apenas o da

interface.

Para permitir a representação de tais comandos torna-se necessário introduzir o espaço de efeitos observáveis (**O**). O esquema passa, agora, a ser o da fig. 5. Agora, **E** é mapeado no espaço **O** (efeitos observáveis), por **observe**, e este, por sua vez, é mapeado em **D** por **display**, o qual, no fundo, selecciona qual dos efeitos deve ser visualizado.

Com base neste modelo são definidas formalmente propriedades dos sistemas interactivos, tais como: reinicialização, completude ou consistência. Mostra-se ainda que é possível estabelecer uma classificação para os comandos.

Note-se que este não é propriamente um modelo da Interface mas do sistema como um todo, já que inclui também a componente computacional (função **result**). Ao não modelar o sistema ao nível do comando/evento do utilizador, mas do programa, o modelo aparenta estar mais vocacionado para sistemas do tipo *batch*.

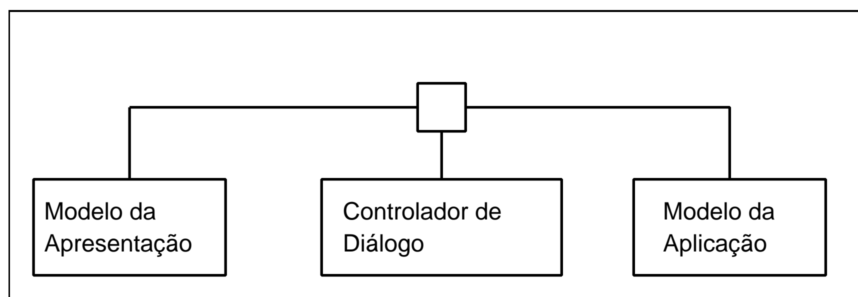


Figura 6: Modelo de Seeheim

No campo das Abstracções Arquitecturais, o Modelo de Seeheim [Gre86] é talvez o modelo mais divulgado. Na sua primeira versão (1987), este modelo divide a Interface em três componentes (ver fig. 6):

- **Apresentação** - lida com a representação física da interface podendo ser visto como o nível léxico;
- **Controlador de Diálogo** - supervisiona as comunicações entre o utilizador e a aplicação podendo ser visto como o nível sintáctico;
- **Modelo da Aplicação** - normalmente presente de forma implícita [MO90], define a interface entre a componente interactiva e o resto do sistema e pode ser visto como o nível semântico da interface.

A separação dos três níveis (léxico, sintáctico e semântico) causa alguns problemas, em particular para diálogos por manipulação directa, pois, sendo necessário completar cada frase antes de ela poder ser avaliada, existe muito pouco *feedback* semântico por parte da aplicação para a apresentação. Consideremos, por exemplo, uma operação de inserção numa base de dados, mesmo que o código indicado seja inválido, só depois de lida toda a restante informação poderá a aplicação detectar o erro e informar o utilizador; entretanto, este esteve a trabalhar inutilmente. Torna-se, assim, evidente que é necessário aproximar a semântica do nível léxico. Tal é conseguido numa segunda versão do modelo, pela integração da informação semântica no Controlador de Diálogo.

Outro Modelo Estrutural conhecido é o Modelo MVC do Smalltalk [Dig87]. Trata-se de um modelo multi-agente que envolve três agentes (ver fig. 7). O agente Model

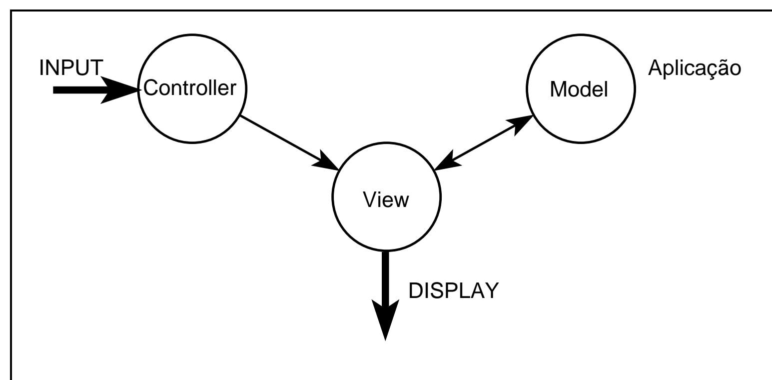


Figura 7: Modelo MVC

corresponde à Aplicação, a View (ou Pane) é a representação da aplicação no écran, o Controller (ou Dispatcher) interpreta o *input* e faz a sua distribuição.

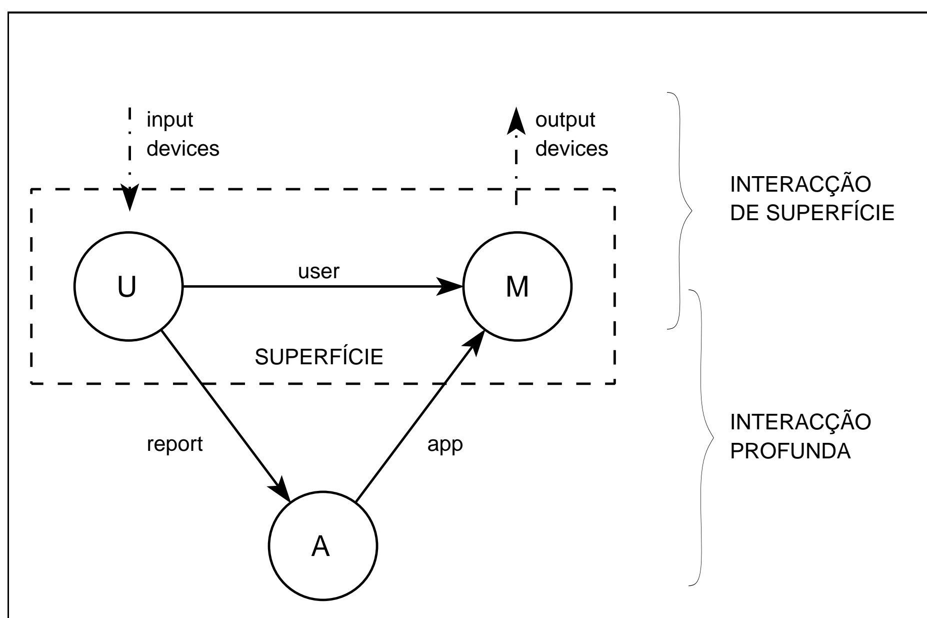


Figura 8: Arquitectura UMA

Em [Too91] é proposta a arquitectura UMA como referência para a Manipulação Directa. Inicialmente é defendida a separação da interacção em Interacção de Superfície, aquela que tem a ver apenas com o estado da interface, e Interacção Profunda, aquela que tem a ver com o estado da aplicação (cf. modelo RED-PIE com efeitos observáveis). A arquitectura proposta (ver fig. 8) é composta por três agentes:

> "um meio M que interpreta operações como mudanças no *display*, uma aplicação A que recebe mensagens *(input, object)* e gera operações no meio e um agente utilizador U que interpreta *input* no contexto do *display* actual e ou envia

> mensagens de *input* para a aplicação, ou implementa interacção de Superfície enviando, espontaneamente, operações para o meio." [Too91]

O agente utilizador em conjunto com o agente meio formam um Meio Activo (*Active Medium*) ou Superfície, o qual permite o desenrolar de diálogo de forma totalmente transparente em relação à aplicação.

Embora com objectivos diferentes, as duas classes de modelos apresentadas (modelos para descrição da interacção e modelos para a descrição da arquitectura) não são mutuamente exclusivas, podemos até dizer que se completam. Os modelos estruturais da interacção guiam a definição e desenvolvimento da linguagem de interacção, permitindo abstermo-nos de pormenores de implementação dessa mesma linguagem (podemos ter duas interfaces, uma por Manipulação directa e outra com uma Linguagem de Comandos, em que a linguagem é a mesma, apenas o modo disponibilizado ao utilizador para gerar as frases é diferente) e prestar atenção apenas aos aspectos estruturais da mesma. As Abstracções Arquitecturais guiam a implementação da componente interactiva que por sua vez deverá implementar um diálogo estruturalmente de acordo com o definido com os métodos anteriores.

3.3 Especificação Formal de Diálogos

Estando definida a estrutura do diálogo é então necessário especificar exactamente que diálogo pretendemos. As primeiras abordagens ao problemas basearam-se em Gramáticas ou em Diagramas de Transição de Estados (DTE's).

As primeiras abordagens gramaticais à especificação de diálogo utilizaram BNF. Em [LJBS78] é apresentada a aplicação de uma variante do BNF, o TBNF, à especificação da interface de uma consola de controlo de um Sistema de Telecontrolo de uma Rede de Energia. TBNF é basicamente um enriquecimento do BNF com operadores de ocorrência opcional e de repetição, bem como com acções colocadas nas meta-expressões. Utilizando os operadores de repetição, a sequência:

> f;f;f;...

poderia ser especificada por

> seqf ::= DSEQ 'f' ';'

em vez de

> seqf ::= 'f' seqf1 | NULL
> seqf1 ::= ';' 'f' seqf1 | NULL

Com estes enriquecimentos consegue-se, por um lado, simplificar a especificação de interacções complexas e, por outro, fazer a ligação à componente computacional. Não é possível, todavia, tornar a Interface sensível ao contexto semântico da camada computacional.

Um dos problemas com este tipo de abordagem prende-se com o facto de no diálogo Humano-Computador, os dois intervenientes não utilizarem a mesma linguagem! A linguagem em que o utilizador "fala" com o sistema e a linguagem em que este lhe responde são diferentes. Em TBNF, por exemplo, este problema é resolvido através das acções, a gramática descreve a linguagem do utilizador e a linguagem que o computador utiliza está implícita nas acções descritas. Deste modo, no entanto, apenas estamos a especificar

formalmente uma das linguagens. Para resolver este problema Shneiderman propôs as *Multiparty Grammars* [Shn82]. Nestas, a cada não terminal da gramática é associada uma etiqueta. Não terminais que descrevam interacção produzida pelo utilizador são etiquetados com um **H**, não terminais relativos ao computador com um **C**. Deste modo é possível, numa única gramática, especificar tanto a linguagem do utilizador como do computador, bem como a relação entre elas. A ligação semântica, no entanto, continua a ser apenas ao nível da invocação das operações.

```
<COMMANDS> ::= <H: OPEN><C: OPEN-ACKNOWLEDGE>|
               <H: CLOSE><C: CLOSE-ACKNOWLEDGE>|
               <H: *><C: OPEN-CLOSE-DIAGNOSTIC>

<H: OPEN> ::= OPEN<H: FILENAME>|
              O<H: FILENAME>

<C: OPEN-ACKNOWLEDGE> ::= [<H: FILENAME>] IS NOW OPEN
```

Figura 9: Exemplo de Gramáticas Multiparty

Na fig. 9 é mostrado um extracto de uma linguagem de comandos muito simplificada apresentada em [Shn82]. Na notação utilizada, os não terminais são escritos entre "<" e ">". O não terminal <H: *> reconhece qualquer *string* e é útil para a detecção de erros. Um não terminal não etiquetado, corresponde a um subdiálogo entre o Humano e o Computador. Os parêntesis rectos em volta de um não terminal indicam que o seu valor é usado para *output*, neste caso, o nome do ficheiro indicado pelo Humano, no comando OPEN, é utilizado pelo Computador na sua mensagem de confirmação.

Jacob [Jac83] apresenta exemplos de aplicação, tanto de BNF como de DTE's, à especificação da interface de um Sistema Militar de Mensagens. Também aqui são adicionadas acções ao BNF, neste caso é feita a diferenciação entre **acções**, que têm a ver com a aplicação, e **respostas**, que têm a ver com a interface. Na notação apresentada é possível ainda introduzir condições nas produções, condições essas que têm de se verificar para que a produção seja reconhecida. Deste modo, é possível introduzir validação semântica ao nível da especificação do diálogo.

A inclusão de acções na gramática levanta, no entanto, um problema. A altura em que elas serão executadas depende de tipo de algoritmo de *parsing* que for utilizado [Gre86]: se o *parsing* utilizado for *top-down*, então as acções são executadas quando as produções a que estão associadas forem usadas, se o *parsing* for *bottom-up*, as acções só são executadas quando as produções forem reconhecidas. Esta dualidade pode tornar confusa a utilização de diferentes notações e torna-as dependentes da implementação dos sistemas que as irão interpretar.

No fundo o problema está na não existência do conceito explícito de estado nas gramáticas. Uma interface é muitas vezes vista como uma máquina de estados, com as interacções a provocarem mudanças nesse estado. É natural, então, que se procure utilizar uma notação em que o conceito esteja presente. Uma escolha óbvia serão os Diagramas

de Transição de Estado, neste caso não se especifica directamente a linguagem (melhor, as linguagens) de interacção, mas quais os estados pelos quais essa interacção passa e as transições entre eles.

No caso geral, a cada transição do DTE está associada uma acção do utilizador. É o conjunto das transições, então, bem como a sequência em que elas podem ocorrer, que define a linguagem do utilizador. Também neste caso são adicionadas acções ao diagrama para deste modo representar as respostas do sistema e introduzir semântica na especificação. As acções podem ser associadas ao estados ou às transições (ou a ambos), mas em qualquer dos casos o instante em que elas serão executadas é bem definido: quando o estado é atingido ou quando a transição é efectuada, respectivamente.

Um dos problemas com os DTE's é o tamanho e complexidade que facilmente atingem quando se tenta especificar um diálogo minimamente realista. Uma das soluções encontradas foi a sua divisão em subdiagramas, os quais possibilitam dividir a especificação do diálogo em unidades lógicas mais pequenas. A associação de um subdiagrama a uma transição indica que ela se deverá efectuar depois de o subdiagrama ser percorrido. Aos diagramas que se podem invocar recursivamente é dado o nome de DTE's recursivos. Estes possuem naturalmente maior poder expressivo que os DTE's normais. Mais uma extensão ainda foi a inclusão no modelo de um estado e de funções que o manipulam, funções essas que estão associadas às transições e que retornam um valor booleano; se o valor retornado for "verdade" então a transição pode dar-se, se for "falso" a transição não se efectua. A este tipo de diagrama chamou-se Diagrama de Transições de Estado Aumentado. Com este tipo de diagrama é agora possível especificar diálogos sensíveis ao contexto, permitindo ou inibindo determinadas transições em função da sua validade semântica. Os DTE's utilizados em [Jac83] pertencem a esta classe de diagramas.

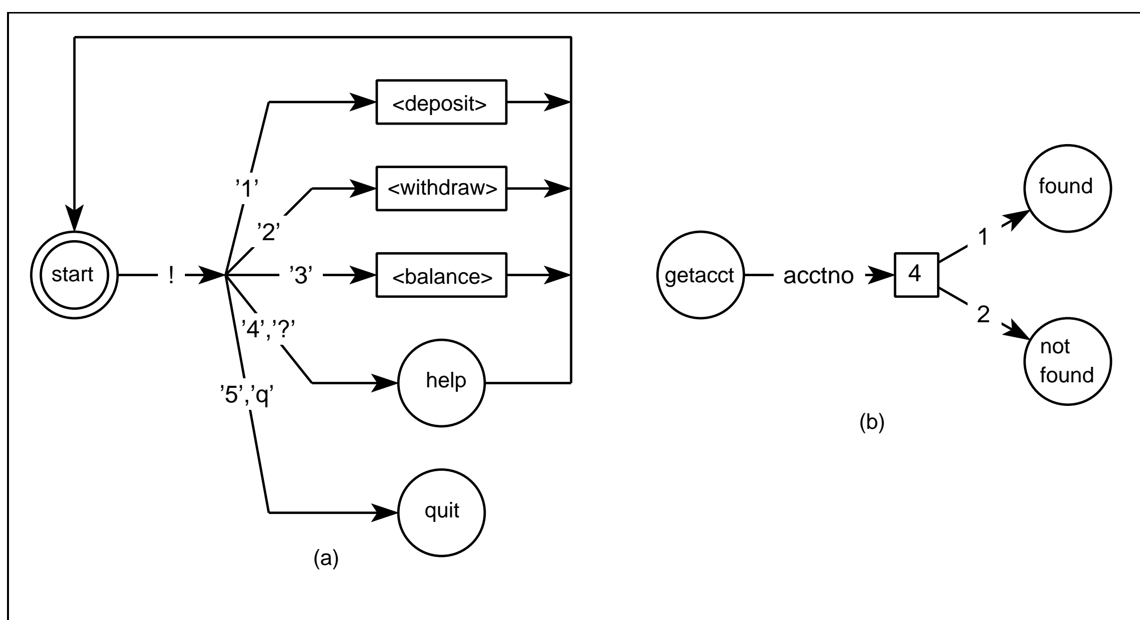


Figura 10: Diagramas de Transições de Estado USE

Um outro exemplo de utilização de Diagramas de Transição de Estados Aumentados são os diagramas USE [Was85]. Na fig. 10 são apresentados dois exemplos, retirados do

artigo referido, de diagramas USE. Nestes diagramas os estados são utilizados para *output*, estando definida uma linguagem para descrição do mesmo, as transições são disparadas pelo *input* do utilizador ou pelo reconhecimento de subdiagramas. As transições podem ser decoradas com acções ou com subdiagramas. No caso de acções, elas são executadas quando se dá a transição; se estiver associado um subdiagrama, o diálogo passa a ser controlado por este. Em qualquer dos casos, o valor retornado, quer pelas acções quer pelos subdiagramas[1], é utilizado para decidir o destino final da transição. Transições que não estejam etiquetadas são utilizadas, como *default*, caso nenhuma outra seja válida, transições etiquetadas com "+" são automaticamente efectuadas e a etiqueta "**!**", associada a um caracter, significa que a transição deve ser efectuada mal seja premida a tecla correspondente a esse caracter (por defeito um *input* é terminado com *return*). Está, no entanto, a misturam-se um pouco a definição sintática da linguagem com pormenores de nível léxico!

Na fig. 10(a) é especificado um menu, premindo as teclas "1" a "3" o diálogo passa a ser controlado pelos diagramas **deposit** a **balance**, a opção "4" ou "?" provoca a transição para o estado **help** para o qual estará definida a apresentação de uma mensagem, "5" ou "q" terminam a interacção. Em (b) é apresentado um exemplo em que a transição depende do valor retornado pela operação, é lido um número de conta, a operação associada a 4 é executada e caso o seu resultado seja um é feita a transição para **found**, caso seja dois é feita a transição para **notfound**.

Nestes exemplos são apenas mostrados os diagramas; a especificação completa do diálogo incluiria ainda a definição do *output* de cada estado e as operações.

O principal problema com as gramáticas e os DTE's é a especificação de diálogos assíncronos, em especial se forem concorrentes. De modo semelhante ao que foi dito para aos Modelos Cognitivos (cf. 3.1), a utilização de gramáticas livres de contexto pressupõe um diálogo sequencial. Os DTE's, por seu lado, permitem descrever as sucessivas mudanças de estado em função de um dado traço de diálogo, mas apresentam dificuldades quando passamos a ter vários traços de diálogo em simultâneo.

Têm sido feitas tentativas para adaptar estes formalismos à descrição deste tipo de diálogos. Em [MVM90] é apresentada uma adaptação de DTE's para a especificação de diálogo concorrente. Aos estados, aqui chamados UIP's (*User Interaction Point*), são associados propriedades que irão influenciar o seu comportamento:

- *Closed-by-next* - o UIP é removido do écran quando um dos seus descendentes for activado;
- *Deactivated-by-next* - quando um descendente é activado, as opções do UIP actual são inibidas, mas ele permanece no écran;
- *Several-descendents* - o UIP pode ter vários descendentes activos em simultâneo;
- *Autoinhibited-options* - quando uma opção é seleccionada, fica inibida até o descendente correspondente ser fechado ou o UIP actual ser re-inicializado.

Em consequência destas propriedades é possível termos mais do que um UIP activo em cada instante. Deste modo, os UIP's deixam de ser estados de um DTE, passando a ser algo mais parecido com condições de uma *Petri Net*.

Outras abordagens têm passado pela procura de formalismos mais poderosos. Em [Gre86] é demonstrado formalmente que o Modelo de Eventos é mais poderoso que as

[1] Neste caso o valor retornado pelas acções não é necessariamente booleano.

Gramáticas Livres de Contexto ou os DTE's. Um exemplo da utilização do Modelo de Eventos é o do sistema IUICE [Stu90]. Neste, a interface é descrita por um conjunto de métodos (entidades responsáveis pelo processamento de eventos) ligados por *event streams*, que implementam a comunicação de eventos entre os métodos. Estão definidos três tipos de métodos:

- Fontes de Eventos, as quais são responsáveis pela aceitação de eventos externos dirigidos ao sistema;
- Métodos Semânticos, que fazem o tratamento dos eventos;
- Métodos Gráficos, cuja principal missão é transformar os eventos em *output* gráfico.

A abordagem por eventos tende a ser, no entanto, pouco abstracta (cf. [Stu90]).

Outras abordagens ao problema utilizam o Paradigma da Programação por Objectos, identificando objectos gráficos no écran e descrevendo de que modo eles estão interligados e comunicam, tanto entre si como com a aplicação e com o utilizador (cf. [Hay90], [Jac86]).

Em [Abo92] é proposta uma metodologia baseada em agentes, sendo o seu comportamento especificado utilizando uma notação baseada em CSP. A definição de um agente é dividida em quatro partes:

- a especificação interna, que descreve quais os estados possíveis para o agente, os estados em que ele pode iniciar actividade e as operações possíveis para alterar os estados;
- a especificação de comunicações, que liga o agente a canais através dos quais as operações são enviadas/recebidas;
- a especificação externa, que descreve os eventos em que o agente pode participar;
- a relação do comportamento derivado, a qual faz a unificação das três anteriores, relacionando os eventos em que um agente pode participar com os estados possíveis que ele pode atingir em função dessa participação.

A utilização de processos comunicantes para a especificação de diálogo é proposta em [HS90].

Em [MAH90] é proposta a primeira versão de um novo formalismo, Guiões de Interacção, para a descrição do diálogo. Cada GI pode ser visto como um objecto, sendo utilizados DTE's para a descrição do diálogo gerado por cada guião.

3.4 Fechando a Ponte

A descrição do diálogo tem apenas a ver, de um modo geral, com a estrutura sintáctica da linguagem. Existem ainda duas outras questões com que temos que nos preocupar para que tenhamos uma descrição completa da camada interactiva: o aspecto visual da interface e o modo como é feita a ligação entre a componente de diálogo e a componente computacional do sistema. Vamos chamar a estes dois elementos da descrição da interface Modelo da Apresentação e Modelo da Aplicação respectivamente (cf. Modelo de Seeheim). Nem sempre eles existem como entidades autónomas, mas de um modo ou outro estão sempre presentes.

Dada a pouca atenção de que têm sido alvo, pode dizer-se que estes dois aspectos são os parentes pobres da especificação de interfaces. Os Modelos da Apresentação

utilizam normalmente linguagens de descrição gráfica (cf. [Was85]), que permitem definir características de cor, posicionamento, etc. dos objectos no écran. Os Modelos da Aplicação estão normalmente implícitos [MO90], ou seja, mais ou menos *hardcoded* na componente de *runtime* dos sistemas. Limitam-se, na maioria dos casos, a ser uma listagem das operações que podem ser invocadas na aplicação.

3.5 Geração Automática de Sistemas Interactivos

Quanto maior é o conhecimento sobre o funcionamento de um determinado sistema, maior é a vontade e a necessidade sentida para a sua automatização. Na área das interfaces já se verifica esse fenómeno. Com o aparecimento de métodos para a especificação, surgem também tentativas para a geração automática.

Em primeiro lugar convém clarificar o que se entende por Geração Automática. Ferramentas como os Geradores de Aplicações, que a partir de uma especificação mais ou menos formal do sistema, geram o código necessário à sua implementação, não são ferramentas de Geração Automática. Eles limitam-se a fazer a tradução do sistema, de uma notação mais abstracta para uma notação mais concreta. Geração Automática consiste exactamente em gerar a própria especificação do sistema!

Obviamente que essa especificação não poderá nascer do nada. Quando construímos uma interface partimos de uma descrição da camada computacional do sistema interactivo que pretendemos implementar. No caso de a pretendermos gerar automaticamente a base de trabalho será também essa descrição, que neste caso deverá ser formal pois deve caracterizar precisamente a camada computacional. É também evidente que será necessário pré-determinar qual o tipo de diálogo que pretendemos ver gerado: se pretendemos interacção por comandos, menus ou manipulação directa, com diálogo síncrono ou concorrente, etc. Idealmente um sistema deverá permitir gerar mais do que um tipo de diálogo

Um segundo ponto que deve ser realçado é que a Geração Automática, excepto em situações muito particulares, não é possível. Tendo que ver, tanto o aspecto gráfico da interface, como o próprio estilo de interacção, com questões tão subjectivas como o gosto pessoal[2], dificilmente um sistema automático conseguirá um resultado satisfatório sob esse ponto de vista. Um aspecto talvez até mais importante prende-se com a própria estrutura sintáctica do diálogo. No caso de menus, por exemplo, de que modo deverão as operações ficar agrupadas? Uma especificação da camada computacional tem pouca informação que permita tomar decisões a esse nível. Um sistema de geração deverá, assim, apresentar uma primeira versão da interface, realizada com base na especificação formal da aplicação, e disponibilizar ferramentas para a sua manipulação e adaptação. Se o próprio sistema final gerado for adaptável, tanto melhor. Não devemos, então, falar de Geração Automática, mas de Geração Semi-Automática.

Um exemplo de um sistema deste tipo é o MIKE [OJ86]. Neste, uma interface por defeito é gerada a partir das assinaturas das operações do sistema, com base no tipo de cada operação. Assim, é criado um menu inicial com as operações que não têm tipo (assume-se que correspondem aos comandos do sistema), e a leitura de cada argumento da função é feita, ou directamente segundo um método pré-definido para o seu tipo, ou seleccionando uma das operações que devolvam um valor do tipo pretendido. Esta primeira versão pode depois ser trabalhada utilizando editores apropriados que o sistema providencia. O MIKE não gera, no entanto, a validação semântica das frases construídas,

[2] Os menus deverão ser azuis ou cor-de-rosa? Prefiro um *options menu* ou uma *radio box*?

nem prevê mecanismos para que essa validação possa mais tarde ser incluída na descrição da interface. Deixa à aplicação a responsabilidade dessa tarefa, fornecendo duas rotinas **CommandEnable** e **CommandDisable** que permitem activar ou desactivar comandos. Fica no entanto de fora a validação semântica dos argumentos das operações. Em parte, tal situação fica, provavelmente, a dever-se ao facto de o ponto de partida do MIKE ser apenas a sintaxe dos comandos do sistema. Para ser possível a geração de validação semântica torna-se então necessário proceder a uma análise não só da sintaxe, mas também da semântica das operações (ie. a sua especificação).

4 A Linguagem CAMILA

CAMILA é uma linguagem de especificação formal modular e concorrente, por modelos, actualmente em desenvolvimento na Universidade do Minho.

Uma especificação CAMILA é uma hierarquia de Módulos, que podem, ou não, executar concorrentemente. Um módulo consiste na definição de um conjunto de Tipo de Dados e de Operações (ie. funções com estado). Os tipos de dados são definidos utilizando os tipos básicos:

- INT - Inteiros;
- STR - *Strings*;
- SYM - o tipo dos Símbolos do XMetoo, pode ser utilizado como Booleano estando definidas as constantes *true* e *false*.

e os seguintes modelos:

- Etiquetação
- Listas
- Conjuntos
- Funções Finitas
- Relações
- Tuplos
- Alternativas

A cada definição de tipo pode estar associado um Invariante, isto é, uma função Booleana que permite restringir os valores do tipo a um subconjunto dos valores tornados possíveis pela sua definição[3]. A definição das operações é feita por duas condições:

- uma pré-condição que expressa quais as relações que devem existir entre os argumentos e entre eles e o estado para que a utilização da operação seja válida;
- uma pós-condição que expressa a relação entre o resultado da operação e/ou o estado depois da sua invocação e os argumentos e estado inicial.

Cada módulo pode receber outros módulos como parâmetros, utilizando as operações e tipos de dados neles definidas para a sua própria definição. Este mecanismo de composição é possível devido à existência do conceito de Interface. Uma Interface pode ser vista como o tipo de um módulo na medida em que contém a declaração dos tipos de dados e operações que um módulo deve definir para que lhe obedeça. Este mecanismo permite também um elevado grau de encapsulamento pois, ao utilizarmos um módulo, temos apenas que nos preocupar com a sua interface, o que permite que a sua implementação/especificação possa ser alterada de um modo completamente transparente.

[3] Um exemplo típico é o tipo Data. A definição

Data = INT×INT×INT

permite valores como

(36, 13, 25)

o qual é claramente inválido. É então necessário associar ao tipo um invariante que expresse as limitações impostas a uma data.

```
interface STACK =
    sorts Elem, Pilha, Bool;
    state Pilha;
   events INIT:->,
          PUSH:Elem->,
          POP:->Elem,
          TOP:->Elem,
          EMPTY:->Bool;
end;

module StackAsList(E:ITEM): STACK =
 sorts Elem = E.Objecto,
       Bool = SYM,
       Pilha = Elem-list;
events INIT      = sigma'=<>,
       PUSH(e)   = sigma'=cons(e,sigma),
       ...;
end;
```

Figura 11: Exemplo de uma Especificação CAMILA

Na figura 11 pode ver-se um extracto de especificação retirado de [Roc91]. Nela, é apresentada a interface de uma *Stack* e um módulo que a respeita. Na interface são declarados os tipos (**sorts**) e as operações (**events**) que o módulo deverá definir, bem como o tipo do estado do módulo. Para cada operação é indicada a sua assinatura (tipos dos seus argumentos e tipo do seu resultado), como todas as operações têm acesso ao estado ele não necessita ser incluído nas assinaturas. O módulo *StackAsList* especifica uma *Stack* genérica e obedece à interface anterior. O estado inicial de cada operação é representado por **sigma**, o estado final por **sigma'**. O módulo tem um parâmetro **E** que deverá obedecer à interface ITEM que não é aqui apresentada mas declara apenas a espécie Objecto. Supondo que tínhamos um módulo *Integers* do tipo ITEM que definia Objecto como INT, podemos escrever uma *Stack* de inteiros do seguinte modo:

StackAsList(Integers())

Um sistema especificado em CAMILA consiste, do ponto de vista de quem o deve utilizar, no conjunto de operações e tipos de dados da interface do módulo de mais alto nível da especificação[4]. Cada operação tem uma assinatura que deve ser respeitada e a sua utilização só é válida quando a pré-condição a ela associada se verifica.

É com base nesta informação que partimos para a definição do sistema de Geração de Interfaces em Modo Assistido.

[4] No caso da fig. 11 teríamos as espécies **Elem**, **Pilha** e **Bool** e as operações **INIT**, **PUSH**, **POP**, **TOP** e **EMPTY**.

5 Guiões de Interacção

Neste Capítulo é feita a apresentação do formalismo Guiões de Interacção (GI), desenvolvido para a especificação de Controladores de Diálogo em Modo Assistido, logo com um elevado grau de informação semântica. Esta apresentação é feita tanto ao nível de uma linguagem para a sua descrição como ao nível da sua especificação formal.

Para auxiliar a descrição de GI's utilizar-se-á a especificação CAMILA de um dicionário apresentada no Apêndice A.

Como foi visto (cf. Capítulo 4), a interface de um módulo CAMILA define uma linguagem de comandos através da qual ele pode ser utilizado. Neste caso temos a linguagem formada pelos comandos:

- INIT: $\rightarrow$ /* Constructor do Sistema */
- INSPAL: Pal $\times$ Sig $\rightarrow$ /* Inserção de uma Palavra */
- REMPAL: Pal $\rightarrow$ /* Remoção de uma Palavra */
- CONSPAL: Pal $\rightarrow$ Sig /* Consultar o Significado */
- EMPTYDIC: $\rightarrow$ Bool /* Dicionário Vazio? */
- EXISTPAL: Pal $\rightarrow$ Bool /* Existe a Palavra? */

Estas visão da componente computacional como um conjunto de operações, leva-nos, desde logo, a pensar num tipo de controlo *Dialogue Dominant*.

5.1 Tarefas a Especificar

Desde cedo se verificou que a especificação monolítica de uma interface é inviável. Os Guiões de Interacção resolvem este problema suportando, como veremos, a especificação de subdiálogos que são depois compostos para obter o diálogo global do sistema. Vamos então separar o diálogo em fases e procurar formas de especificar cada uma delas. Consideraremos as seguintes fases de diálogo:

- Selecção da Operação a executar;
- Leitura dos argumentos da operação/visualização do seu resultado.

5.2 Leitura dos Argumentos/Visualização do Resultado

Por uma questão de conveniência na apresentação vamos começar por apresentar os Guiões de Interacção que especificam a segunda fase referida, a leitura dos argumentos e invocação de uma operação. A estes Guiões iremos chamar Guiões de Interacção do tipo SYNTH dado que descrevem a síntese de um comando.

Consideremos a operação INSPAL e vejamos como especificar o diálogo necessário à sua correcta invocação[5]. A primeira decisão a tomar é qual o nome a dar ao GI. Chamemos-lhe GInsPal. Para além do seu nome, é necessário também declarar quais os argumentos a ler (neste caso **pal** e **sig**) e qual a expressão a calcular depois de terminada a leitura (**INSPAL(pal, sig)**).

[5] Nunca é de mais referir que esta correcção se situa não só no nível sintático mas também no semântico (cf. Modo Assistido).

Muitos autores não se preocupam com a especificação da leitura dos argumentos, centrando-se no modo como é feita a selecção da operação a executar e considerando que a leitura é feita de um modo pré-definido (pela ordem em que estão na operação, por qualquer ordem, etc.). No entanto, neste trabalho considera-se que a possibilidade de especificar essa componente do diálogo é tão importante, ou mais até, que a segunda. Consideremos a operação referida: enquanto o argumento **sig** não está sujeito a qualquer restrição, **pal** não pode pertencer ao dicionário (cf. a informação semântica dada pela pré-condição de INSPAL). Neste caso, é desejável que **pal** seja lido primeiro que **sig**, pois deste modo é mais rapidamente detectada uma possível incorrecção semântica da frase que se está a construir.

Torna-se, então, necessário especificar a ordem pela qual os argumentos devem ser lidos. Dado que se pretendia uma notação simples e compacta, mas que permitisse a especificação de diálogo concorrente, optou-se por uma abordagem baseada numa notação de descrição dos traços possíveis de eventos, do estilo CSP [Hoa85] ou CCS [Wal87]. Consideraram-se os seguintes operadores:

- · (sequência) - **a.b** corresponde a **a** seguido de **b**;
- || (paralelismo síncrono) - **a**||**b**, **a** e **b** decorrem paralelamente mas devem sincronizar no final (a expressão só termina quando terminarem os dois);
- | (paralelismo assíncrono) - a expressão **a**|**b** termina quando terminar um dos dois;
- + (alternativa) - a expressão **a+b** representa que o utilizador terá que decidir entre prosseguir o diálogo segundo **a** ou segundo **b**;
- ∗ (repetição) - a expressão **a*** representa que o diálogo segue segundo **a**, podendo ser repetido.

Para especificar que a ordem de leitura deverá ser **pal** seguido de **sig** escrevemos:

input(pal)·input(sig)

No entanto, o primeiro argumento tem a restrição semântica de não poder pertencer ao dicionário. É, então, também necessário poder associar a cada evento uma condição de contexto que verifica se ele é válido. Por outro lado, a expressão apresentada especifica apenas o diálogo na direcção utilizador-sistema. Para especificar o diálogo inverso duas possibilidades foram consideradas: utilizar uma abordagem semelhante às Gramáticas *Multiparty* em que na mesma notação se descrevem as duas linguagens, ou uma abordagem semelhante às normalmente utilizadas em DTE's em que as reacções da aplicação são colocadas nas transições como resposta às acções do utilizador. Dada a natureza predominantemente reactiva das interfaces por comandos (cf. controlo *Dialogue Dominant*), por um lado, e a complexidade que a especificação em simultâneo de duas linguagens inevitavelmente causaria, por outro, optou-se pela segunda abordagem. Vamos, então, associar também a cada evento uma acção correspondente à resposta do sistema. A notação a utilizar será:

evento: *condição* ⇒ *acção* EXCEP *acções de erro*

Podemos então escrever a primeira versão de GInsPal.

Guião 1 (GInsPal 1ª versão)

```
DefGI GInsPal
  Declarations
    TYPE SYNTH
    SYMBOL {GInsPal, inspal}
    ARGS pal: Pal;
         sig: Sig
  Behaviour
    EVSEQ input(pal).input(sig)
    TRANS input(pal): not(EXISTPAL(pal)) => EXCEP out("Erro!")
    EXEC INSPAL(pal, sig)
EndGI
```

Como se pode verificar a definição de um Guião de Interacção está dividida em duas partes. Na primeira, são feitas as declarações do tipo do Guião, dos símbolos que o identificam e dos argumentos da operação a invocar. Na segunda, caracteriza-se o comportamento do mesmo.

Tal com está, a especificação obriga, no entanto, a que o utilizador efectue sempre todo o diálogo. Se, em dado momento, ele mudar de ideias e não quiser inserir a palavra não tem possibilidades de voltar para trás! Este problema é resolvido pela introdução de eventos assíncronos, a que chamaremos Comandos, que descrevem características predefinidas do diálogo. Nesta fase, vamos considerar os seguintes Comandos:

- CANCEL - Cancelar o diálogo especificado pelo GI; quando presente, este comando está sempre disponível.
- OK - Quando este comando está presente, a expressão descrita em EXEC não é automaticamente avaliada depois de terminada EVSEQ, sendo necessário primeiro selecciona-lo; só está disponível depois de percorrida toda a expressão EVSEQ.
- RESET - Corresponde a reinicializar[6] o diálogo especificado pelo Guião; sempre disponível.
- APPLY - Equivalente a OK mas o Guião é reinicializado, permitindo assim a repetição do diálogo.

Note-se que outros comandos podem ser considerados, dependendo das características que se pretendem especificar.

É também importante poder atribuir valores iniciais aos argumentos. Para tal é incluída uma cláusula INIT que contém instruções a serem executadas aquando da activação do GI. A especificação do diálogo passa então a ser a apresentada no Guião 2.

Guião 2 (GInsPal 2ª versão)

```
DefGI GInsPal
  Declarations
    TYPE SYNTH
    SYMBOL {GInsPal, inspal}
```

[6]iniciar: começar qualquer coisa; inicializar: colocar no estado inicial.

```
    ARGS pal: Pal;
         sig: Sig
  Behaviour
    INIT pal = "";
         sig = ""
    EVSEQ input(pal).input(sig)
    TRANS input(pal): (pal != "") && not(EXISTPAL(pal)) =>
                                    EXCEP pal=="" -> out("Palavra nula!");
                                          pal!="" -> out("Erro!")
          OK:
          CANCEL:
    EXEC INSPAL(pal, sig)
EndGI
```

Agora, **pal** e **sig** são inicializadas a *string* vazia e são incluídos os Comandos OK e CANCEL. Foi também alterada a condição de contexto de **pal**, passando a testar se a palavra recebida não é a *string* vazia. As acções de excepção passaram então a ser uma lista guardada, executando-se as acções correspondentes à guarda que se verificar.

Consideremos, agora, o GI GRemPal para especificar o diálogo referente à operação REMPAL. Neste caso, a pré-condição não tem a ver só com os argumentos mas também com o estado do sistema. Não faz sentido invocar a remoção se o dicionário estiver vazio. Torna-se, então, necessário especificar que o comando (e consequentemente o Guião) só pode ser utilizado se não se verificar essa condição. Tal é conseguido pela cláusula CONTEXT (ver Guião 3).

Guião 3 (GRemPal)

```
DefGI GRemPal
  Declarations
    TYPE SYNTH
    SYMBOL {GInsPal, inspal}
    ARGS pal: Pal
    VAR-UI sig: Sig
  Behaviour
    CONTEXT not(EMPTYDIC())
    INIT pal = "";
         sig = ""
    EVSEQ input(pal)
    TRANS input(pal): EXISTPAL(pal) => sig = CONSPAL(pal)
                                          EXCEP out("Erro!")
          OK:
          CANCEL:
    EXEC REMPAL(pal)
EndGI
```

É aqui introduzida a utilização de variáveis que não são argumentos da operação. Trata-se, neste caso, de uma variável declarada como sendo do Modelo da Apresentação (**VAR-UI**) e que é utilizada para apresentar o significado da palavra seleccionada para remoção (ver acção na cláusula **TRANS**). Para além dos argumentos, existem outros três tipos de variáveis num Guião:

- Variáveis da Apresentação (**STATE-UI**) - são variáveis que pertencem à apresentação e a que o GI acede para alteração dos seus valores;
- Variáveis Locais (**STATE-CTRL**) - são variáveis locais ao Guião;
- Variáveis da Aplicação (**STATE-APL**) - são variáveis da aplicação a que o GI acede apenas para consulta.

Na cláusula **STATE-CTRL** pode ser utilizada a forma de declaração:

var1 $\Leftarrow$ **var2**

Neste caso está a declarar-se que o valor da variável da aplicação **var2** é copiado para a variável local **var1**.

Embora o GI utilize mais do que uma operação para a sua definição (**EMPTYDIC** em CONTEXT, **EXISTPAL** em TRANS e **REMPAL** em EXEC), é muito importante notar que apenas a operação colocada em EXEC deverá poder alterar o estado da aplicação. Doutro modo, as condições de contexto necessárias para a validade semântica da operação poderiam deixar de se verificar[7].

Para uma caracterização mais rigorosa dos Guiões recorremos à especificação por modelos. A informação até agora descrita pode ser modelada formalmente como o tuplo que de seguida se apresenta:

```
GIDescr :: SYNOMS:  GISym-set
           VARS:    VarId -> VarDecl
           CONTEXT: BoolExp
           INIT:    Code
           EVSEQ:   ExprComp
           TRANS:   TransId -> TransDescr
           EXEC:    NIL | ExecDescr;
```

Em VARS são agrupadas todas as variáveis, a definição da classe a que pertencem é feita em VarDecl[8]:

```
VarDecl == VarUI | VarAPL | VarAPLcopy | VarCTRL;
VarUI :: DREF: TypeId;
VarAPL :: TYP:    TypeId;
VarAPLcopy :: TYP:    TypeId
              APLVAR: OpcVarId;
VarCTRL :: DREF: TypeId;
```

As transições são tuplos com a guarda, a acção a executar e a lista de acções de excepção e respectivas guardas. A expressão a executar consiste no nome da operação e respectiva lista de argumentos.

```
TransDescr :: COND:   BoolExp
              ACTION: Code
              EXCEP:  ExcepDescr-list;
```

[7]Suponhamos, por exemplo que, por hipótese absurda, a operação EXISTPAL para além de verificar a existência da palavra, alterava o estado da aplicação removendo-a. Quando REMPAL fosse invocado essa invocação seria incorrecta já que se estava a tentar remover uma palavra que já não existia no dicionário!

[8]Os argumentos e as variáveis da Apresentação estão agrupadas

```
ExcepDescr :: COND:   BoolExp
              ACTION: Code;
ExecDescr :: OP:   STR
             ARGS: VarId-list;
```

Consideremos, agora, que pretendíamos especificar o diálogo necessário à consulta do significado de uma palavra. A utilização pura e simples de um Guião SYNTH não resolve o problema pois este não prevê o tratamento do resultado da operação. A solução está em ter dois GI's; um, que especifica a invocação da operação e prevê a devolução do seu resultado a quem o activou e outro, que o utiliza para especificar a apresentação do resultado e que poderá ser do tipo SYNTH. Supondo que o primeiro Guião referido se chama DoConsPal o segundo será o Guião ViewConsPal.

Guião 4 (ViewConsPal)

```
DefGI ViewConsPal
  Declarations
    TYPE SYNTH
    SYMBOL {ViewConsPal, conspal}
    SUBGI DoConsPal
    VAR-UI sig: Sig
  Behaviour
    CONTEXT not(EMPTYDIC())
    INIT sig = ""
    EVSEQ DoConsPal(sig)
    TRANS OK:
  SubGI
    DefGI DoConsPal
      ...
    EndGI
EndGI
```

Note-se que este GI não tem cláusula EXEC já que não é utilizado para sintetizar um comando mas apenas para permitir a visualização da variável **sig**. A sequência de eventos possíveis é apenas a execução do GI DoConsPal cujo resultado é colocado em **sig**.

É introduzida, aqui, a cláusula SUBGI que indica estar a ser utilizado um outro GI para a definição da sequência de eventos deste. GI esse que é definido localmente, aparecendo assim também a componente de declaração de subGI's. Se DoConsPal fosse definido fora de ViewConsPal, deveria ser declarado não nesta cláusula mas na EXTERNAL. Ao ser subguião, DoConsPal herda as variáveis de ViewConsPal.

Aos GI's, anteriormente referidos, que permitem a invocação de uma operação da aplicação para cálculo de um valor, vamos chamar VALSYNTH. Eles são semelhantes aos SYNTH tendo apenas que declarar o tipo do resultado que produzem (que será o tipo da operação referida em EXEC) e a restrição de não poderem referir operações que provoquem alterações no estado da aplicação.

O Guião DoConsPal será:

Guião 5 (DoConsPal)

```
DefGI DoConsPal: Sig
  Declarations
    TYPE VALSYNTH
    SYMBOL {DoConsPal}
    ARGS pal: Pal
  Behaviour
    CONTEXT not(EMPTYDIC())
    INIT pal = ""
    EVSEQ input(pal)
    TRANS input(pal): EXISTPAL(pal) => EXCEP out("Erro!")
          OK:
          CANCEL:
    EXEC CONSPAL(pal)
EndGI
```

GIDescr é então redefinido para passar a ser:

```
GIDescr :: SYNOMS:  GISym-set
           TYP:     NIL | Type
           EXTERN:  GISym-set
           SUBGI:   GISym-set
           VARS:    VarId -> VarDecl
           CONTEXT: BoolExp
           INIT:    Code
           EVSEQ:   ExprComp
           TRANS:   TransId -> TransDescr
           EXEC:    OpcExecDescr;
```

5.3 Escolha da Operação

Falta apenas especificar o modo como é feita a selecção das operações a utilizar. Ao tipo dos Guiões que efectuam essa tarefa foi dado o nome de DECISION. Eles são, de facto, uma simplificação dos GI's SYNTH, não podendo declarar argumentos e só podendo utilizar os operadores $+$ e $*$ em conjunto com GI's do tipo DECISION ou SYNTH para a definição de EVSEQ.

Um GI para a especificação da selecção das operações do dicionário poderia ser o Guião MENU.

Guião 6 (Menu)

```
DefGI Menu
  Declarations
    TYPE DECISON
    SYMBOL {Menu, menu}
    SUBGI Fim
  Behaviour
    EVSEQ (GInit + GinsPal + GRemPal + ViewConsPal)* + Fim
  SubGI
```

```
    DefGI Fim
      Declarations
        TYPE DECISION
      Behaviour
    EndGI
EndGI
```

O Guião Fim corresponde ao Guião nulo. Quando é activado termina imediatamente, e é utilizado por Menu como o evento que terminará o diálogo.

A não utilização de operadores que expressem concorrência nas expressões de comportamento dos GI's DECISION - os quais permitiriam especificar diálogo em que seria possível sintetizar mais do que um comando em simultâneo - tem a ver com a consistência semântica do sistema. Suponhamos a seguinte hipótese: que os operadores existiam; que o utilizador seleccionava a operação CONSPAL; que indicava uma palavra a consultar e que o GI verificava que esta era válida e que, entretanto, e antes do comando anterior ser enviado à aplicação, o utilizador seleccionava a operação REMPAL e removia essa palavra do dicionário. Verificada a hipótese, quando o primeiro comando fosse enviado à aplicação seria inválido pois entretanto o estado da mesma tinha sido alterado! Deste modo e para garantir sempre a consistência semântica, não pode nunca ser permitida a síntese simultânea de dois comandos que alterem o estado da aplicação.

Para completar a definição formal dos Guiões de Interacção falta apenas incluir a declaração do seu tipo. Tal será feito do modo que a seguir se apresenta.

```
GIdef = DECISION | SYNTH | VALSYNTH;
DECISION :: GIDescr;
SYNTH :: GIDescr;
VALSYNTH :: GIDescr;
```

Como se torna aqui evidente, os três tipos de Guiões de Interacção são, de facto, variações sobre o mesmo tema. Na realidade, bastava termos apenas Guiões sem especificar qualquer tipo especial; a sua divisão em SYNTH, VALSYNTH e DECISION tem como objectivo estruturar e regrar a sua utilização.

5.4 Especificação do Comportamento

Para uma especificação mais rigorosa da semântica dos Guiões de Interacção foram utilizadas *Petri Nets*. A escolha deste formalismo ficou a dever-se fundamentalmente à facilidade com que possibilita a representação de processos concorrentes, dado possuir uma linguagem gráfica e uma semântica bem definida e estudada [Rei].

Descreve-se de seguida o tipo de redes utilizado, o modo como cada operador é especificado e como pode ser feita a composição dos diversos operadores para representar uma expressão mais complexa.

5.4.1 *Petri Nets*

Uma *Petri Net* é constituída por um conjunto de nodos (Lugares) e um conjunto de arcos dirigidos entre esses nodos (Transições)[9]. Em cada Lugar podem estar colocados um ou mais *tokens*. Cada Transição retira um determinado número de *tokens* dos Lugares

[9] Cada Transição pode ter mais do que um Lugar de Partida e/ou mais do que um Lugar de chegada)

de Partida e coloca um determinado número de *tokens* nos Lugares de Chegada. A uma determinada distribuição de *tokens* pelos Lugares chama-se uma Marcação. Casos especiais são a Marcação Inicial (estado inicial da rede) e a Marcação Final (uma vez atingida, o processo definido pela rede termina).

Dependendo do número de *tokens* que um Lugar pode conter e do modo como são interpretadas as Transições, os Lugares e os *tokens*, podemos ter vários tipos de *Petri Nets*. As escolhidas para a definição dos operadores foram as *Condition-Event Nets* (ou redes C/E).

Numa rede C/E cada Lugar pode apenas conter um *token*. As Transições retiram os *tokens* dos Lugares de Partida e colocam-nos nos Lugares de Chegada. A cada Transição está associado um determinado Evento que provoca o disparo da Transição, estando convencionado que uma transição sem Evento associado é disparada automaticamente. É importante notar que estes Eventos são atómicos, pelo que quando os *tokens* são retirados dos Lugares de Partida são simultaneamente colocados nos Lugares de Chegada. O Evento só pode ocorrer, no entanto, se todos os Lugares de Partida tiverem um *token* e todos os Lugares de Chegada não o tiverem. Deste modo, podemos interpretar os Lugares como Condições que se devem (ou não) verificar para que um determinado Evento possa ocorrer.

Para permitir incluir a especificação das condições de contexto associadas aos GI's nas cláusulas CONTEXT e TRANS, assim como as acções definidas nesta última, expandiu-se o modelo original associando a cada Transição uma expressão booleana - que é necessário verificar-se para que ela possa acontecer - e uma acção que será executada quando a transição se der. Neste novo modelo, cada Transição será então composta por uma condição, pelo evento que a "dispara" e por uma acção a executar. A estas redes chamaremos redes C/E guardadas.

5.4.2 Guiões de Interacção

O comportamento de um Guião de Interacção (neste caso chamado X) pode ser descrito pela *Petri Net* apresentada na fig. 12.

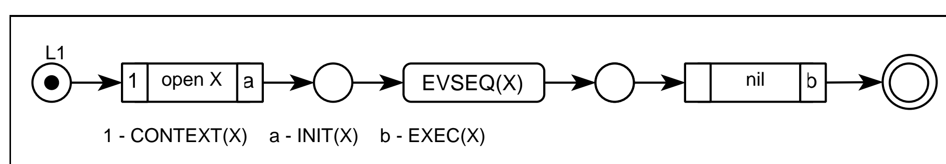


Figura 12: Rede de um Guião

A marcação apresentada é a marcação inicial da rede; as Condições que compõem a marcação final estão assinaladas com um duplo círculo[10].

Para representar a activação da rede vamos considerar o evento **open**. Quando a transição a ele associada se dá, passa a ser possível a execução de EVSEQ, finda a qual é efectuada a transição nula que provoca a execução de EXEC e o fim da rede.

Note-se que a figura é um esquema da rede representativa do Guião, uma vez que EVSEQ(X) não é verdadeiramente uma transição, mas uma "abstracção" da rede que

[10] Esta regra será mantida para todas as redes que de seguida se apresentam.

lhe corresponde (ver secção 5.4.4). Para podermos construir a rede relativa a EVSEQ necessitamos da definição dos operadores e comandos em termos de *Petri Nets*. Todas as definições que de seguida se apresentam assumem que estamos a falar de EVSEQ(X).

5.4.3 A Expressão Mais Simples

Se não considerarmos a Expressão de Comportamento vazia, que pode ser representada por uma rede com uma única Condição e nenhuma Transição, a expressão mais simples que podemos ter é a que inclui um único Guião (ou a instrução **input**):

gi(var)

Esta dá origem à rede apresentada na fig. 13.

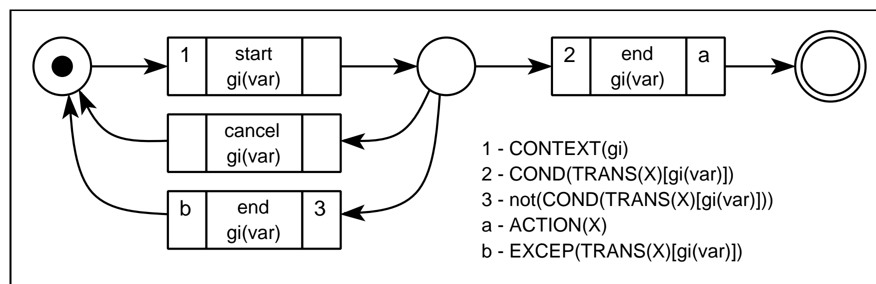


Figura 13: Rede para gi(var)

Como a execução de um GI não é atómica, torna-se necessário decompô-la em três Eventos. Quando é iniciada a execução de **gi** dá-se o evento **start** e, consequentemente, é disparada a transição a ele associada. A esta está associada a cláusula CONTEXT de **gi**, pelo que a transição só é possível se ela se verificar[11]. Se a execução for cancelada dá-se o evento **cancel**, à transição a ele correspondente não está associada nenhuma condição ou acção. Caso **gi** termine com sucesso dá-se o evento **end**. Existem duas transições correspondentes a este evento. A uma são associadas a condição e a acção de TRANS relativas a **gi(var)**. Esta transição corresponde ao fim normal da expressão. À outra são associadas a negação da condição anterior e a acção EXCEP de TRANS relativa a **gi(var)**. Esta transição corresponde à situação em que a condição de TRANS não se verifica e o diálogo segundo **gi** é anulado.

Por uma questão de simplificação da composição, vamos considerar sempre redes com marcações final e inicial de uma só condição, o que é facilmente conseguido pela utilização de transições nulas.

5.4.4 Sequenciação

A representação da sequenciação

exp1·exp2

[11]Como é evidente, se **gi** não for um Guião mas **input** não existe cláusula de contexto e, desde que a marcação o permita, a transição será sempre válida.

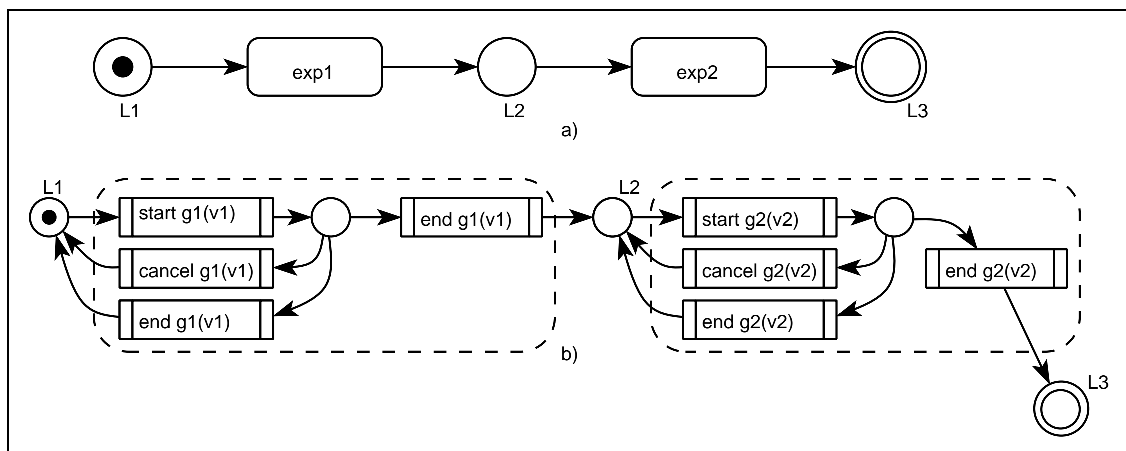


Figura 14: Rede para o Operador de Sequenciação

é apresentada na fig. 14 a) e consiste simplesmente em ligar as duas redes, fazendo coincidir a marcação final da primeira com a marcação inicial da segunda. Tal como na secção 5.4.2, a figura 14 a) é apenas um esquema genérico da rede, dado que não sabemos à partida quais são as expressões. Se **exp1** e **exp2** fossem **g1(v1)** e **g2(v2)**, respectivamente, teríamos a rede da fig. 14 b). O esquema de substituição será o mesmo para todos as definições apresentadas em seguida.

5.4.5 Concorrência Síncrona

A rede originada pela expressão

exp1||exp2

é a da fig. 15. As duas expressões podem decorrer paralelamente mas a transição nula final garante que a rede só termina quando as duas expressões tiverem terminado.

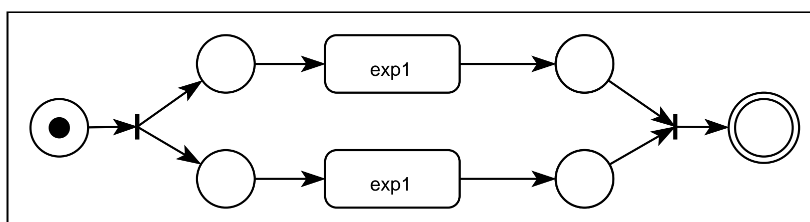


Figura 15: Rede para o Operador de Concorrência Síncrona

5.4.6 Concorrência Assíncrona

A concorrência assíncrona expressa em

exp1|exp2

é representada pela rede da fig. 16. Neste caso, as duas expressões podem decorrer igualmente em paralelo mas a rede termina quando terminar uma delas, já que foi feita uma unificação das marcações finais das redes das duas expressões.

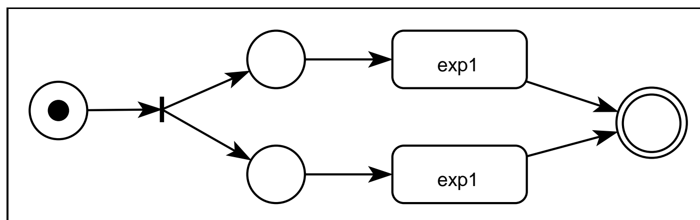


Figura 16: Rede para o Operador de Concorrência Assíncrona

5.4.7 Alternativa

A expressão de alternativa

exp1+exp2

é representada pela rede da fig. 17. Também neste caso as marcações finais correspondentes a cada uma das expressões foram unificadas

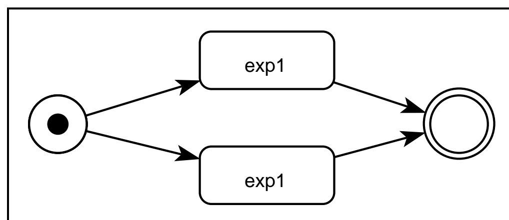


Figura 17: Rede para o Operador de Alternativa

5.4.8 Repetição

Finalmente, a repetição

exp∗

é representada por uma rede em que é adicionada uma transição da marcação final de **exp** para a sua marcação inicial (rede da fig. 18).

Para expressar o facto de que a expressão não termina, esta rede não tem marcação final. Deste modo, nas expressões

exp1∗ ||exp2
exp1∗+exp2

não é efectuada a unificação entre as marcações finais de **exp2** e de **exp1∗**(que não existe), sendo a marcação final das expressões a de **exp2**.

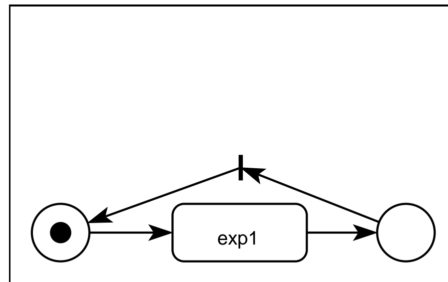


Figura 18: Rede para o Operador de Repetição

5.4.9 Alguns Casos Particulares

Algumas combinações dos operadores não se comportam bem quando efectuadas com as regras acima descritas. Vamos de seguida apresentar esses casos e as soluções propostas.

Dado que os operadores de sequência e de paralelismo síncrono necessitam que as marcações finais das duas expressões envolvidas se verifiquem para a expressão global poder terminar, as expressões

exp1∗·**exp2**
exp1∗ ||**exp2**

nunca terminariam se a definição da repetição fosse a apresentada na fig. 18. Mesmo que · e || considerassem a segunda condição da rede como marcação final, como a transição nula retira automaticamente o **token** quando ele lá é colocado, o problema mantém-se. A solução é redefinir, nestes casos, o operador de repetição para passar a ser o apresentado na fig. 19, considerando os operadores de sequência e de paralelismo síncrono a segunda condição como a marcação final dessa rede.

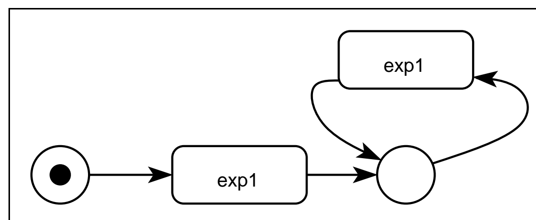


Figura 19: Rede para o Operador de Repetição (versão 2)

As expressões dos tipos

g1(v1)+(g2(v2)|g3(v3))
g1(v1)+(g2(v2)||g3(v3))

apresentam também problemas pois a transição nula inicial dos operadores | e || impede que **g1(v1)** possa ocorrer. Este problema pode ser solucionado reescrevendo a rede do modo que está apresentado na fig. 20.

Finalmente existe o caso da expressão

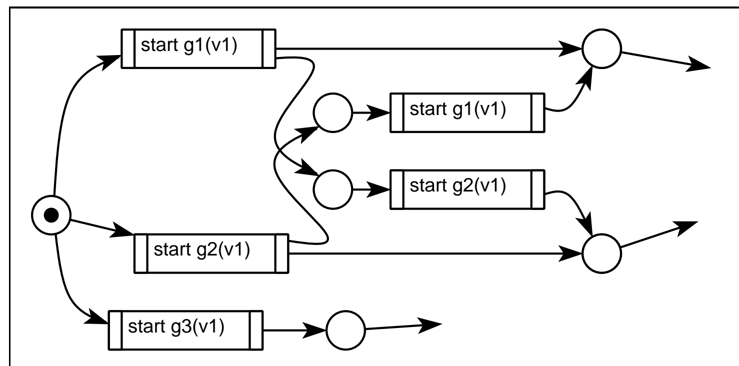


Figura 20: Rede para os operadores de Soma e Concorrência

(exp1|exp2)$*$

O problema aqui é que quando uma das expressões do operador | terminar, deverá terminar a expressão global **exp1|exp2** para que possa reiniciar-se. O que acontece, no entanto, é que na outra expressão ainda existem *tokens* que é então necessário retirar. A única forma de o conseguir será incluir tantas transições nulas quantas as necessárias de modo a que, quando uma das expressões terminar, todos os *tokens* da rede da outra sejam removidos. A determinação dessas transições pode, no entanto, não ser trivial.

5.4.10 Comandos

Falta apenas definir a semântica dos diferentes comandos. Quando o comando OK está presente, a última transição da rede representativa do Guião deixa de ser nula para passar a ocorrer quando o comando acontece (ver fig. 21 a)). O comando CANCEL é especificado na fig. 21 b). Como pode ver-se, a transição a ele correspondente fica em paralelo com EVSEQ pelo que está sempre disponível.

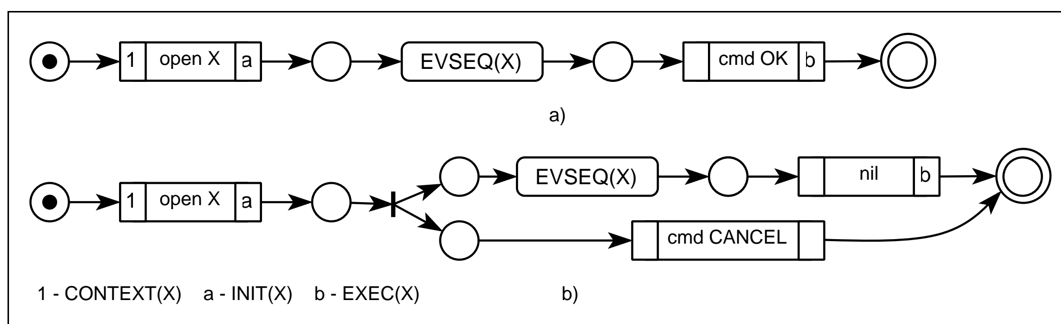


Figura 21: Rede para os Comandos OK e CANCEL

Os comandos APPLY e RESET apresentam também o problema de implicarem uma reinicialização da rede (cf. expressão **(exp1|exp2)**$*$). Como a introdução de todas as transições nulas necessárias implicaria uma rede demasiado complexa, optou-se por lhes

dar as definições de OK e CANCEL, respectivamente, sendo da responsabilidade do controlador de diálogo saber que deve recomeçar o diálogo com a mesma expressão.

Note-se que como uma *Petri Net* termina sempre na mesma marcação final, o único modo de saber se a expressão terminou com sucesso, ou não, é analisando a transição que provocou o seu fim. Se foi uma transição associada a CANCEL, o diálogo foi cancelado, senão o diálogo terminou com sucesso.

5.5 Guiões de Interacção e a Linguagem CAMILA

Os sistemas tradicionais, ao preocuparem-se apenas com a validade sintáctica das frases, necessitam apenas de informação relativa às assinaturas das operações para a especificação do Controlador de Diálogo (cf. MIKE). Ao estender a noção de validade até ao nível semântico, os Guiões de Interacção integram também informação presente na especificação das mesmas operações. Tentaremos agora ver até que ponto uma especificação CAMILA proporciona a informação necessária para a geração automática dos Guiões.

Podemos separar a informação necessária para uma especificação com GI's em dois níveis. Um, a informação mais ligada ao nível sintáctico da linguagem a especificar a qual é necessária para definir o modo como os Guiões DECISION deverão agrupar as operações em grupos lógicos e, relativamente a cada operação, qual a melhor ordem de leitura dos seus argumentos. Outro, a informação semântica para definir as cláusulas de contexto de cada Guião (CONTEXT) e as condições a que cada argumento deve estar sujeito (TRANS).

A obtenção de um critério aceitável para a definição de grupos de operações aparenta ser impossível. Por um lado, critérios simplistas como, por exemplo, o agrupamento com base no tipo do resultado produzido, não são satisfatórios; critérios mais elaborados envolvem, necessariamente, uma análise das pós-condições das operações, o que não se apresenta como tarefa fácil. Por outro lado, este tipo de critério é sempre subjectivo, sendo portanto pouco dado a automatizações. A solução deverá então passar pela utilização, numa primeira fase, de um critério simples para a geração de uma primeira versão, seguindo-se a edição da especificação gerada para a obtenção da estrutura final.

Já no que diz respeito à definição da ordem de leitura dos argumentos, será mais fácil obter uma primeira versão aceitável através da análise das pré-condições. Como vimos na secção 5.2, é desejável que os argumentos com maiores restrições semânticas sejam lidos em primeiro lugar. Assim, analisando a pré-condição de uma dada operação pode definir-se uma hierarquia para os seus argumentos que depois deverão ser lidos por essa ordem. Tal estratégia não elimina, no entanto, a necessidade de posterior edição para "afinação".

No que diz respeito às cláusulas CONTEXT e TRANS, a informação para elas necessária será retirada também das pré-condições. Em CONTEXT deverão ser colocadas todas as condições que sejam decidíveis apenas com informação relativa ao estado; em TRANS, as condições que necessitem de informação relativa a cada um dos argumentos. Assim, na operação

```
CONSPAL(pal) = { sigma!=[] && pal in dom(sigma) => ...
```

teremos que em CONTEXT ficaria

```
sigma != []
```

e em TRANS

input(pal): pal in dom(sigma) => ...

Note-se que embora a pré-condição seja redundante[12], defini-la deste modo não é despropositado se pensarmos na sua posterior utilização para geração da interface. A presença da primeira condição possibilita a geração da cláusula CONTEXT passando o GI a não ser sequer seleccionável se não existirem palavras no dicionário. Infelizmente, está a pedir-se a quem desenvolve a camada computacional que pense também na componente interactiva. Uma forma de evitar esta situação será incorporar no sistema de geração automática um conjunto de regras que cubram estas situações. Não é difícil deduzir que se **pal** deve pertencer ao domínio de **sigma**, então, à partida **sigma** não pode ser vazio!

Finalmente, é necessário dar atenção às condições em que estejam envolvidos mais do que um argumento. Neste caso, elas deverão ser colocadas em TRANS de tal modo que só sejam testadas quando todos os argumentos tiverem sido lidos, por exemplo associadas a um argumento que é obrigatoriamente lido depois deles, ou ao comando OK.

5.6 Guiões de Interacção e Tipos de Dados

Como se pôde ver nos exemplos de Guiões já apresentados, os tipos de dados definidos na camada computacional podem ser utilizados nos GI's. Como tipos pré-definidos existem Inteiros (INT), *Strings* (STR) e Booleanos (Bool).

É importante notar que, dado pretender-se um nível de abstracção elevado na especificação do Controlador de Diálogo, os valores são sempre considerados como *tokens* léxicos, independentemente da sua estrutura. A abordagem aqui tomada é a de que, assim como ao nível da Apresentação existem métodos pré-definidos para a apresentação de valores (*scales*, *radio boxes*, etc.), também ao nível do Controlador de Diálogo deverão existir métodos pré-definidos para a leitura dos diferentes tipos de modelos (Funções Finitas, Listas, Conjuntos, etc.). Deste modo, liberta-se o especificador da interface da tarefa de especificar como devem ser lidos esses modelos.

5.7 Guiões de Interacção e Outros Formalismos

A comparação entre os Guiões de Interacção e outros formalismos para a especificação de diálogo pode ser efectuada ao nível da sua capacidade para fazer reflectir o estado da aplicação na interface vs. a separação entre componente de diálogo e componente computacional, ao nível da capacidade expressiva da notação utilizada e ao nível do modo como é feita a ligação à componente de apresentação.

5.7.1 Ligação à Componente Semântica

A possibilidade de especificar diálogo sensível ao contexto da aplicação é um factor primordial na obtenção de interfaces de boa qualidade. Podemos considerar a informação necessária para tal a dois níveis: condições relativas ao estado da aplicação (as quais devem ser testadas *a priori*) e condições relativas aos argumentos das operações (que serão testadas *a posteriori*).

Na maioria dos casos os formalismos existentes não contemplam o tratamento simultâneo das duas situações. O TBNF, as Gramáticas *Multiparty*, ou a metodologia

[12] Bastaria colocar apenas a segunda condição.

proposta por Jacob em [Jac86], por exemplo, prevêem apenas a validação das acções do utilizador *a posteriori*. Os UIP's possibilitam a especificação de condições que se devem verificar para que eles possam ser activados, mas não a validação posterior das acções do utilizador. O formalismo proposto por Abowd em [Abo92] inclui o tratamento dos dois casos. No entanto, a separação entre camada de diálogo e camada computacional não é respeitada, já que o diálogo e a camada computacional são especificados em conjunto. Os Guiões de Interacção, respeitando o Princípio da Separação, permitem a especificação dos dois tipos de condições, através das cláusulas CONTEXT e TRANS, procurando-se, deste modo, alcançar as vantagens da separação entre as duas camadas, por um lado, e de uma ligação forte (mas metódica) entre elas, por outro.

5.7.2 Expressividade

A este nível podemos considerar, por um lado, quais os tipos de diálogo que é possível especificar, por outro, até que ponto a notação é de fácil utilização e leitura.

Ao permitirem a especificação de diálogo concorrente, os Guiões de Interacção são, à partida, superiores às metodologias tradicionais baseadas em Gramáticas e em Diagramas de Transição de Estados. Ao estarem orientados para a descrição de um tipo particular de interfaces (para linguagens de comandos) não têm a flexibilidade expressiva apresentada por abordagens mais genéricas como as propostas em [Jac86] ou [Abo92]. Em contrapartida, o elevado nível de abstracção a que a especificação é efectuada (cf. secção 5.7.3) e a utilização de uma notação para descrição dos traços possíveis de eventos - em conjunto com a utilização de subdiálogos que se compoêm para formar o diálogo global - tornam a notação dos Guiões compacta e simples, oferecendo uma relação expressividade/complexidade favorável. A utilização de notações gráficas apresentaria, aparentemente, vantagens. No entanto, como normalmente essas notações necessitam ser traduzidas para formas textuais, essas vantagens perdem-se, sendo, muitas vezes, as notações textuais utilizadas de difícil leitura.

5.7.3 Ligação à Apresentação

De um modo geral, os formalismos existentes especificam o diálogo ao nível do toque de tecla, botão de rato, etc. (cf. [Jac86]) e muitas vezes incluem a definição de características léxicas da interface (cf. [Shn82]). Deste modo, especificam conjuntamente aspectos relacionados com o nível sintáctico e com o nível léxico da linguagem. Na definição dos Guiões de Interacção procurou-se trabalhar a um nível mais abstracto, considerando que a estrutura sintáctica do diálogo é independente do modo como ele é efectivamente efectuado. Trata-se, no fundo, de aplicar o Princípio da Separação em relação à componente léxica do sistema interactivo. Assim, ao nível dos Guiões, não nos preocupamos em saber de que modo os tokens léxicos recebidos são construídos, ficando isso a cargo da componente léxica do sistema. Torna-se, então, possível alterar o aspecto da interface, mantendo a estrutura sintáctica, por simples substituição da componente léxica.

6 Descritores da Apresentação

A especificação do Modelo da Apresentação deverá descrever o aspecto visual de cada um dos Guiões de Interacção do Controlador de Diálogo. Para tal iremos utilizar Descritores de Apresentação (DA). Para apresentar a sua definição, vamos utilizar, a título de exemplo, os Guiões Menu e ViewConsPal apresentados no Capítulo 5.

No caso do Guião Menu, trata-se de escolher uma de um grupo de opções disponíveis, a escolha óbvia é utilizar um menu. Temos, então, como primeiro tipo de descritor o tipo MENU[13]. Podemos então escrever:

Descritor 1 (Menu)

```
DefDA Menu
  TYPE MENU
  NAME "Principal"
  OPTIONS GInit        "Dicionario Vazio"
          GInsPal      "Inserir Palavra"
          GRemPal      "Remover Palavra"
          ViewConsPal  "Consultar"
          Fim          "Sair"
EndDA
```

Estamos, assim, a definir que o GI Menu deverá ser apresentado como um menu chamado "Principal" e a dar um nome a cada um dos Guiões presentes em EVSEQ. Outras características léxicas que podem ser especificadas são a posição (POS), cor (COLOR) e cor da opção seleccionada (SEL).

Como é evidente este tipo de Descritor destina-se fundamentalmente aos Guiões do tipo DECISION, uma vez que não prevê a apresentação de variáveis ou comandos. Para a descrição da apresentação de ViewConsPal, no entanto, tal torna-se necessário. Com esse fim existem os Descritores do tipo DB (de *Dialog Box*). Para o GI referido poderemos ter:

Descritor 2 (ViewConsPal)

```
DefDA ViewConsPal
  TYPE DB
  NAME "Consulta"
  VARS sig (TYPE: HimSelf,
            NAME: "Significado",
            COLOR: "blue")
  GIS DoConsPal(sig) (OPEN: true,
                      NAME: "Ler Palavra")
  CMDS OK (NAME: "Terminar")
EndDA
```

A cada variável/Guião/Comando é associada uma série de características léxicas. De momento está prevista a especificação do nome, da cor e da posição. Todas elas são

[13] Para uma maior flexibilidade da interface poder-se-ia fazer a distinção entre menus *popup* e *pulldown*. De momento vamos apenas considera-los de uma forma global.

especificadas como *strings*, sendo da responsabilidade da componente de *runtime* do Modelo da Apresentação a sua interpretação.

Para os Guiões pode ainda ser definida a característica OPEN. Quando tal comportamento é possível, o Modelo da Apresentação deverá apresentar o GI se o seu valor for *true* e apenas um botão, que quando seleccionado origina a apresentação do GI, se o valor for *false*[14].

Às variáveis é necessário associar o seu tipo léxico. Actualmente estão previstos os seguintes:

- **HimSelf** - leitura simples;
- **RadioBox** - uma *radio box*, ou o mais próximo que a interface puder apresentar;
- **OptionMenu** - um *option menu*;
- **Scale** - uma *scale*.

No caso das *RadioBox* e dos *OptionMenu*, é então necessário listar os valores que deverão apresentar. Se quisermos apresentar a variável inteira **var** como uma *RadioBox* podemos escrever:

Descritor 3 (Uma *RadioBox*)

```
VARS var (TYPE: RadioBox,
          VALUES: (1 "Um", 2 "Dois", 3 "Tres", 4 "Quatro", 5 "Cinco"),
          NAME: "Um a Cinco")
```

Para a *Scale* é necessário especificar os valores máximo (MAX) e mínimo (MIN).

Quando uma variável é de um tipo definido utilizando modelos, a sua apresentação não pode ser definida como *HimSelf* ou *Scale*; podendo, no entanto, ser especificado o nome do Guião[15] que irá controlar a sua leitura. Por sua vez, esse terá um descritor que especificará a sua apresentação.

Ao conjunto de Descritores de Apresentação que descrevem o aspecto léxico de todos os Guiões de Interacção de uma dada especificação chamamos uma *View* dessa especificação. Uma vez que o funcionamento do Controlador de Diálogo e do Modelo da Apresentação é independente podemos definir várias *Views* permitindo ao utilizador seleccionar, mesmo em *runtime*, a que pretende utilizar. Deste modo, o aspecto gráfico da interface, e o modo como os valores são lidos/apresentados, pode ser radicalmente alterado, mesmo durante a execução da aplicação. Para tal basta definir:

Descritor 4 (Definir Múltiplas *Views*)

```
DefView "Port"
  DefDA ViewConsPal
    TYPE DB
    NAME "Consulta"
    VARS sig (TYPE: HimSelf,
              NAME: "Significado",
              COLOR: "blue")
```

[14] Tal característica pode ser utilizada, por exemplo, para a apresentação de painéis de ajuda (*help*).

[15] De um dos tipos que serão introduzidos na secção 7.2.

```
    GIS DoConsPal(sig) (OPEN: true,
                        NAME: "Ler Palavra")
    CMDS OK (NAME: "Terminar")
  EndDA
  ...
EndView
DefView "Ingl"
  DefDA ViewConsPal
    TYPE DB
    NAME "Search"
    VARS sig (TYPE: HimSelf,
              NAME: "Meaning",
              COLOR: "blue")
    GIS DoConsPal(sig) (OPEN: true,
                        NAME: "Read Word")
    CMDS OK (NAME: "Quit")
  EndDA
  ...
EndView
```

e passaremos a ter a possibilidade de optar entre uma Interface em Português ou em Inglês.

Para completar a definição da apresentação falta apenas declarar, para cada variável de cada Guião, qual o seu tipo e os sinónimos de cada GI. Esta informação poderia ser pedida ao Controlador de Diálogo mas tal iria sobrecarregar as comunicações; por outro lado, deste modo é possível definir, no Modelo da Apresentação, variáveis de que o Controlador de Diálogo não tem conhecimento. A estrutura completa da especificação da Interface será então:

Descritor 5 (Definição dos tipos das variáveis)

```
DefVars ViewConsPal
  sig: Pal
EndVars
...
DefSynoms
  GInsPal: inspal;
  ...
EndSynoms
DefView "Port"
  ...
EndView
DefView "Ingl"
  ...
EndView
```

7 O Sistema GAMA-X

Com o projecto GAMA-X pretende-se, desenvolvendo as ideias inicialmente abordadas no projecto GAMA [MCCM90], complementar a linguagem de especificação CAMILA [BA91] e o método de refinamento a ela associado, com um UIMS que permita especificar e gerar interfaces, em Modo Assistido, tanto para os protótipos (os textos CAMILA são compilados para XMetoo [Gro90] para posterior execução) como para as aplicações finais resultantes do processo de refinamento da especificação. O sistema pode ser dividido numa componente de geração semi-automática (o Módulo de Geração de Interfaces - MGI), responsável pela geração e manutenção da especifição da interface, e por uma componente de *runtime* (o Módulo de Interacção com o Utilizador - MIU) que animará a interface a partir das especificações produzidas pela primeira, gerindo todos os aspectos do diálogo entre o Utilizador e a Componente Computacional do Sistema Interactivo.

Neste capítulo é apresentada a arquitectura geral do sistema GAMA-X, propondo-se notações para a especificação de cada um dos componentes do MIU.

7.1 Arquitectura do Módulo de Interacção com o Utilizador

A missão do Módulo de Interacção com o Utilizador é fazer a ponte entre o utilizador e o sistema, auxiliando o primeiro a construir frases semanticamente válidas para o segundo e apresentando as respostas deste ao primeiro.

Sendo tão importante o conceito de validade semântica, a separação rígida proposta pelo Modelo de Seeheim entre componentes Léxica, Sintáctica e Semântica tornam-no inadequado para ser utilizado como Abstracção Arquitectural de uma interface em Modo Assistido. Deste modo, a arquitectura proposta para o Módulo de Interacção com o Utilizador, gerado pelo GAMA-X, não o irá respeitar totalmente. Embora existam um Modelo da Aplicação, um Controlador de Diálogo e um Modelo da Apresentação (ver fig. 22), eles terão tarefas ligeiramente diferentes do proposto no modelo, estando a semântica da aplicação presente nos três.

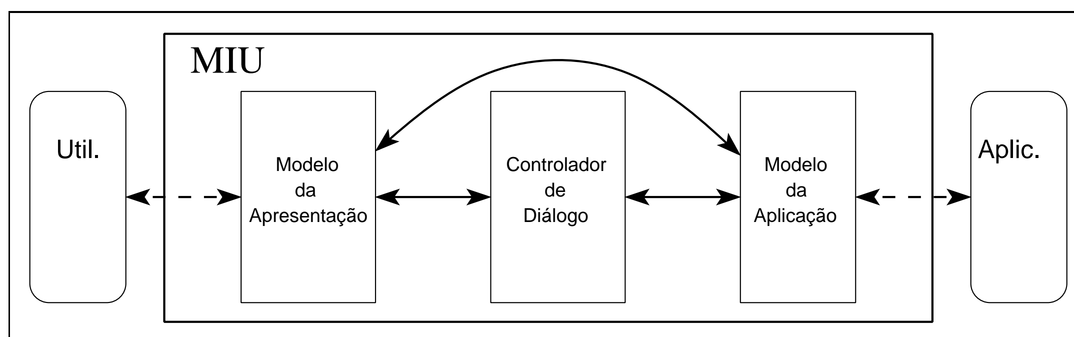


Figura 22: Arquitectura do MIU

O Modelo da Apresentação, para além da tarefa usual de controlar o aspecto visual da interface, é responsável pela validação semântica dos símbolos que passa ao Controlador de Diálogo. Caso esses símbolos sejam valores para os argumentos das funções, deverão ser validados face aos invariantes dos tipos respectivos e, para tal, ele recorre ao Modelo da Aplicação; caso se trate de um símbolo pertencente à linguagem definida pelos Guiões,

deverá estar assegurada a sua validade e, para tal, é necessário recorrer a informação fornecida pelo Controlador de Diálogo.

Este é responsável pela validação semântica das frases que constrói. Para tal, recorre ao Modelo da Aplicação para verificação das condições especificadas nos Guiões. Deverá também informar o Modelo da Apresentação de quais os símbolos válidos em cada momento.

Por sua vez, o Modelo da Aplicação recebe os pedidos dos outros dois componentes e providencia a sua execução pela camada computacional.

A arquitectura resultante destes requisitos é apresentada na fig 22. Cada um dos componentes apresentados na figura será um processo independente, sendo a comunicação entre eles efectuada por meio de protocolos a definir. Deste modo, torna-se possível especificar uma interface que acompanhará o desenvolvimento da aplicação, desde a fase do protótipo (XMetoo) até à implementação final. Basta existirem Modelos da Aplicação para cada uma das linguagens utilizadas.

7.2 Especificação do Controlador de Diálogo

Para a especificação dos Controladores de Diálogo serão utilizados os Guiões de Interacção. Como foi já avançado na secção 5.5, uma primeira versão da especificação do Controlador poderá ser gerada a partir da especificação CAMILA da camada computacional, através da geração de um Guião de Interacção SYNTH para cada operação dessa camada e de Guiões DECISION que os agrupem, devendo depois ser possível a sua edição para manipulação da especificação. Um editor/compilador de Guiões encontra-se, neste momento, em fase terminal de desenvolvimento [Roc93].

Em relação ao que foi já descrito relativamente aos Guiões de Interacção, há apenas a acrescentar a introdução de cinco novos tipos de Guiões, relacionados com a leitura dos modelos utilizados para a definição dos tipos de dados (a sua necessidade será explicada na secção 8.2.4):

- FFSYNTH - para especificar que o GI se destina à leitura de uma Função Finita;
- RELSYNTH - para Relações;
- LISTSYNTH - para Listas;
- SETSYNTH - para Conjuntos;
- TUPSYNTH - para os Tuplos.

Qualquer Guião definido como sendo de um destes tipos não deverá conter nenhuma outra cláusula, pois o seu comportamento é pré-definido. Para indicar que o Guião **leFF** se destina a controlar a leitura de Funções Finitas escrevemos, então:

Guião 7 (Leitura de Funções Finitas)

```
DefGI LeFF
  Declarations
    TYPE FFSYNTH
EndGI
```

Os GI's dos tipos FFSYNTH e RELSYNTH declaram implicitamente os argumentos **dom** e **ran** e os comandos OK e CANCEL. Os dos tipos LISTSYNTH e SETSYNTH declaram, para além dos mesmos comandos, o argumento **elem**. Guiões do tipo TUPSYNTH declaram também os mesmos comandos e argumentos com os nomes dos selectores do tuplo.

7.3 Especificação do Modelo da Apresentação

A especificação do Modelo da Apresentação é feita utilizando os Descritores apresentados no Capítulo 6.

O gerador de Descritores de Apresentação deverá criar um para cada Guião de Interacção existente. Os nomes a utilizar poderão ser, nesta primeira fase, os próprios nomes das variáveis, comandos e Guiões. Para a definição dos tipos léxicos analisar-se-ão as definições dos tipos da especificação e seus invariantes. Para valores de tipos que estejam definidos entre limites pré-estabelecidos poderá ser utilizada uma *Scale*. Para tipos discretos com uma cardinalidade baixa uma *RadioBox* ou um *OptionMenu*. No caso geral utilizar-se-á *HimSelf* (para valores de tipos simples) ou indicar-se-á o Guião responsável pelo controlo desse subdiálogo (para modelos). Como tal, esta fase poderá dar origem a novos Guiões, influenciando a especificação do Controlador de Diálogo.

7.4 Especificação do Modelo da Aplicação

A informação necessária para especificar o Modelo da Aplicação consiste na definição dos tipos da aplicação em termos de modelos e a indicação dos respectivos invariantes; na associação das variáveis da aplicação referidas nos Guiões a operações que devolvam o seu valor; dos identificadores das operações também utilizadas nos Guiões às operações da componente computacional.

Para especificar o Modelo da Aplicação do Dicionário poderíamos escrever:

```
DefType
  Pal = STR;
  Sig = STR
EndType
DefOp
  INIT = INIT;
  INSPAL = INSPAL;
  REMPAL = REMPAL;
  CONSPAL = CONSPAL;
  EMPTYDIC = EMPTYDIC;
  EXISTPAL = EXISTPAL
EndOp
```

Neste caso, como não é acedida nenhuma variável da aplicação, essa componente não aparece na especificação, será da forma:

```
DefVar
  nomevar = nomeop
EndVar
```

Não é, também, definido nenhum invariante; tal seria feito conforme o esquema-tipo:

```
DefType
  nome = def WITH invariante
EndType
```

A notação da associação dos identificadores de operações utilizados no Controlador de Diálogo a operações da camada computacional pode parecer um pouco estranha e inconsequente. Para a perceber é necessário ter em atenção que do lado direito estão *strings* que são identificadores e do lado esquerdo estão os nomes das operações reais, que, dependendo da linguagem em que a aplicação está implementada, terão diferentes interpretações. Assim, por exemplo, em XMetoo serão os símbolos referentes às operações, em C serão apontadores para essas operações.

Toda esta informação poderá ser obtida da especificação CAMILA, com a ressalva de que em CAMILA os invariantes não são explicitamente associados aos tipos respectivos. Se esta característica não vier a ser incluída em versões futuras da linguagem, ter-se-á que encontrar um esquema alternativo, fazendo, por exemplo, uma associação por nome[16].

7.5 Arquitectura do GAMA-X

A arquitectura global do sistema será, assim, a apresentada na fig. 23.

Como pode observar-se, o MGI funciona em duas fases. Numa primeira fase, são geradas versões base da especificação com os geradores; de seguida é feita a sua edição para obtenção da versão final e posterior compilação.

Note-se que as estruturas a gerar pelos compiladores irão, muito provavelmente, depender da implementação particular dos componentes (Modelo da Apresentação, Controlador de Diálogo e Modelo da Aplicação) que se pretende utilizar. Em relação à identificação das operações da aplicação, por exemplo, enquanto para XMetoo, o compilador necessita apenas gerar uma expressão que identifique o símbolo da mesma[17], para C será, em princípio, necessário gerar código que proceda é invocação das mesmas [MCCM90].

Em relação ao MGI, está em fase final de desenvolvimento o Compilador/Editor de Guiões de Interacção. A implementação do MIU será descrita no Capítulo 8.

[16] Ao tipo **Data** estaria associado o invariante **invData**.

[17] (quote symop) ou 'symop.

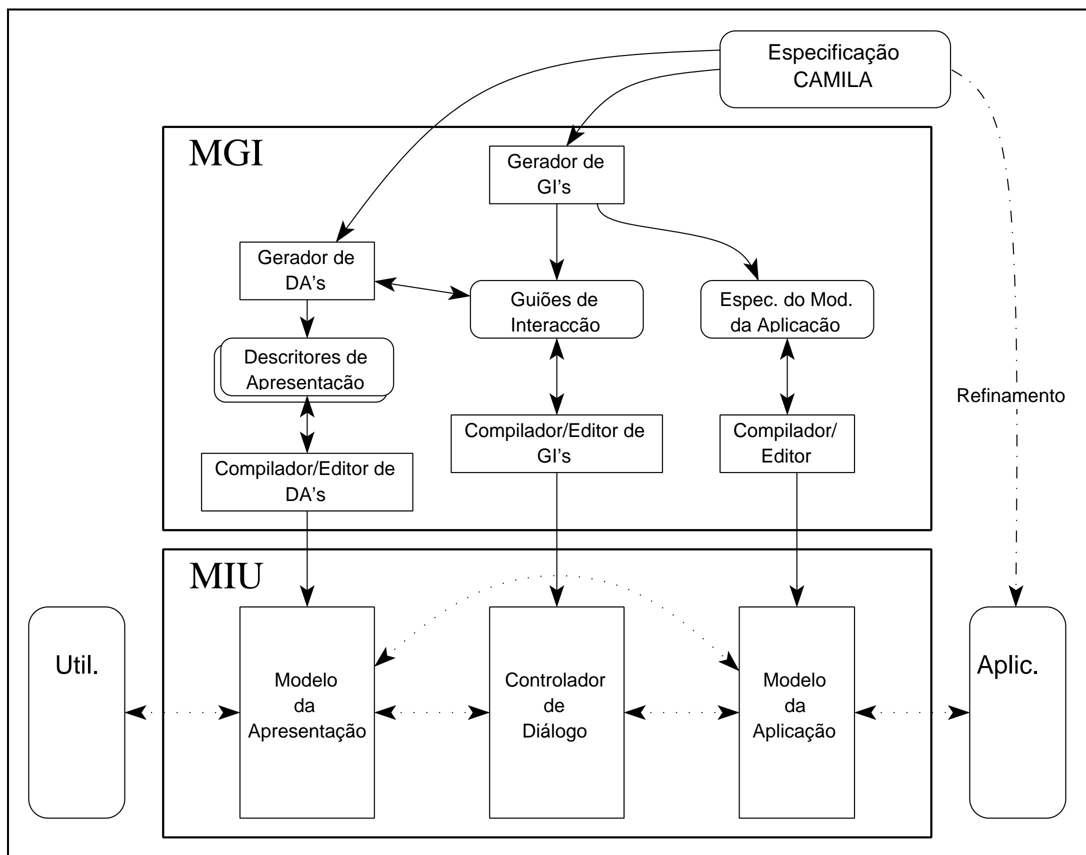


Figura 23: Arquitectura do GAMA-X

8 Implementação do MIU

Neste Capítulo é apresentada a componente de *runtime* do GAMA-X, escrita em NYAGSL[18] [ABCM89], demonstrando-se, deste modo, a viabilidade dos Guiões de Interacção como formalismo para a especificação de Interfaces Humano-Computador.

8.1 Perspectiva Global

Sendo óbvio que a representação dada aos dados na aplicação é inaceitável para ser manipulada pelo utilizador e não pode ser utilizada no Controlador de Diálogo[19], um valor poderá ter até três representações distintas dentro do sistema interactivo. Uma, a representação dos dados na aplicação, a que chamaremos representação semântica a qual é da responsabilidade da camada computacional; outra, a representação no Controlador de Diálogo, a que chamaremos representação sintáctica dos valores; outra, ainda, a representação dos valores na apresentação, a qual será especificada no Modelo da Apresentação (sendo uma de um conjunto de representações possíveis) e a que chamaremos representação léxica. Torna-se então necessário efectuar a tradução entre as diversas representações de modo a que um valor na aplicação possa ser passado para a apresentação e vice-versa. A tradução entre os níveis semântico e sintáctico é naturalmente feita pelo Modelo da Aplicação, já que a representação semântica depende da aplicação em causa. As funções de tradução serão as funções de refinamento e *retrieve* obtidas durante o processo de refinamento da especificação [Oli90]. A tradução entre os níveis léxico e sintáctico será, de modo equivalente, feita pelo Modelo da Apresentação, que neste caso recorre ao Modelo da Aplicação para validar semanticamente os valores lidos.

Convém, ainda, referir aqui o modo como os diferentes Guiões de Interacção vão sendo activados. Consideremos a expressão EVSEQ do Guião ViewConsPal apresentado na secção 5.2:

DoConsPal(sig)

Como foi visto, este evento dá origem a três transições: **start(DoConsPal(sig))**, **end(DoConsPal(sig))** e **cancel(DoConsPal(sig))**. Quando o utilizador selecciona o evento, é enviado ao Guião ViewConsPal o **start** do mesmo. Torna-se agora necessário activar o GI DoConsPal. Duas hipóteses foram consideradas:

- A activação ser efectuada pelo Controlador de Diálogo, que a comunicará ao Modelo da Apresentação. Neste caso o valor de **sig** não necessita de passar por este último.
- A activação ser feita pelo Modelo da Apresentação. Neste caso ele enviará o **start** do evento a ViewConsPal, dará ordem para activação de DoConsPal e quando este terminar, enviará ao primeiro o **end** do evento com o valor devolvido pelo segundo.

Seja qual for o responsável, ele precisará de manter informação sobre a hierarquia de Guiões activados.

Duas ordens de factores levaram a optar pela segunda solução. Primeiro, porque como a verificação dos invariantes é feita ao nível da Apresentação, é necessário que o valor sintetizado pelos Guiões lhe seja comunicado para se preceder à referida verificação. Depois, como o Modelo da Apresentação necessita saber a hierarquia dos objectos léxicos

[18] Uma versão não modelar do CAMILA

[19] A representação final dos dados dependerá do processo de refinamento.

criados e dos respectivos Guiões que os controlam, à partida já preenche o requisito mencionado anteriormente.

8.2 O Controlador de Diálogo

O Controlador de Diálogo será construído sobre um animador de Guiões de Interacção e controlará a construção de frases com símbolos recebidos do Modelo da Apresentação enviando-as ao Modelo da Aplicação, bem como fará o necessário *feedback*, para o utilizador, de mudanças na aplicação resultantes das suas acções. Como tal, receberá eventos do Modelo da Apresentação, processa-los-á e gerará outros eventos, tanto para o Modelo da Aplicação como para o da Apresentação.

8.2.1 Protocolos de Comunicação

O Controlador de Diálogo deve comunicar com o Modelo da Apresentação e o Modelo da Aplicação. Vamos agora descrever os protocolos através dos quais essa comunicação é implementada. No Apêndice I é apresentada a representação esquemática das relações entre as diferentes mensagens que de seguida se descrevem, a notação utilizada é a desenvolvida para EVSEQ.

Antes de mais, o CD deve ter a capacidade de receber do Modelo da Apresentação informação sobre quando e como activar e desactivar cada um dos Guiões. Vendo os GI's como objectos, e já que o mesmo Guião pode ser utilizado em simultâneo para fins diferentes[20], a activação de um GI corresponderá à activação de uma instância do mesmo. Existirá, então, a mensagem

```
CreateMsg :: DEST: OpcInstId
             GI:   GISym;
```

que instrui o CD para criar uma instância de **GI** tendo como "pai" (estando na cláusula SUBGI/EXTERNAL de) **DEST**, a mensagem

```
OpenMsg :: DEST: InstId;
```

que o instrui para activar uma determinada instância e a mensagem

```
KillMsg :: DEST: InstId;
```

que o informa que deverá eliminar a instância indicada.

Como pode ver-se na fig. 13, cada evento de EVSEQ origina três eventos na *Petri Net*: **start**, **end** e **cancel**; eventos esses que o Modelo da Apresentação deve comunicar ao CD. Deste modo teremos mais três mensagens, uma para cada um dos eventos. A mensagem

```
StartMsg :: DEST: InstId
            ID:   EvId;
```

informa o CD da ocorrência do **start** de um determinado evento (**ID**) na instância **DEST**;

```
EndMsg :: DEST: InstId
          ID:   EvId
          VAL:  NIL | Value;
```

[20] Considere a expressão **GetVal(v1)||GetVal(v2)**

indica que o evento **ID** terminou com sucesso e, opcionalmente, qual o valor por ele devolvido; finalmente

```
CancelMsg :: DEST: InstId
             ID:   EvId;
```

corresponde ao cancelamento do evento indicado.

Sendo os comandos tratados à parte, em relação aos eventos de EVSEQ (só originam uma transição na rede), existe a mensagem

```
CmdMsg :: DEST: InstId
          ID:   CmdId;
```

para comunicar a selecção de um determinado comando numa dada instância.

Finalmente, a Mensagem

```
SetValMsg :: DEST: InstId
             VAL:  Value;
```

permite ao Modelo da Apresentação enviar um valor à instância indicada. A utilidade desta mensagem será explicada na secção 8.2.4.

O protocolo de comunicação no sentido Modelo da Apresentação-Controlador de Diálogo é então

```
MesgLxSt = CreateMsg | OpenMsg | StartMsg | EndMsg |
           CancelMsg | CmdMsg | KillMsg | SetValMsg;
```

Quando recebe uma mensagem **CreateMsg**, o Controlador de Diálogo cria, como foi dito, uma nova instância para o Guião indicado e deverá comunicar ao Modelo da Apresentação qual a instância que foi criada, para que este possa mais tarde referir-se a ela. Para tal existe a mensagem

```
InstMsg :: SOURCE: InstId;
```

De cada vez que um GI recebe uma mensagem e muda de estado, o CD tem que comunicar ao Modelo da Apresentação quais os eventos de EVSEQ e quais os comandos que são válidos. Para esse fim utiliza as mensagens

```
EnableMsg :: SOURCE: InstId
             ID:     TransId;
DisableMsg :: SOURCE: InstId
              ID:     TransId;
```

Durante o processamento das instruções (em INIT ou TRANS), uma instrução **output** dá origem a uma mensagem

```
OutMsg :: SOURCE: InstId
          TEXT: STR;
```

a alteração de uma variável da Apresentação, a uma mensagem

```
ShowMsg :: SOURCE: InstId
           ID:     VarId
           VAL:    Value;
```

Para o CD poder informar o Modelo da Apresentação que detectou o final de um Guião, existem as mensagens

```
StopMsg :: SOURCE: InstId
           VAL:    OpcValue;
```

se o GI terminou com sucesso e

```
AbortMsg :: SOURCE: InstId;
```

se foi cancelado.

Quando termina o processamento de uma mensagem o Controlador indica ao Modelo da Apresentação que pode continuar o diálogo com a mensagem

```
GoMsg :: SOURCE:  InstId
         CMDLINE: OpcCmdLineDescr;
```

O protocolo no sentido Controlador de Diálogo-Modelo da Apresentação é

```
MesgStLx = InstMsg | OutMsg | EnableMsg | DisableMsg | ShowMsg |
           GoMsg | StopMsg | AbortMsg;
```

Na direcção oposta, o Controlador de Diálogo necessita comunicar com o Modelo da Aplicação. Neste caso, o protocolo é mais simples. Os únicos pedidos que o CD faz são para a obtenção do valor de uma dada variável da Aplicação

```
GetVarMsg :: SOURCE: InstId
             ID:     VarId;
```

para a obtenção da definição de um dado tipo

```
GetTypeMsgSt :: SOURCE: InstId
                TYP:    TypeId;
```

e para a execução de um dado comando

```
CallMsg :: SOURCE: InstId
           OPR:    STR
           ARGS:   Value-list
           RET:    Bool;
```

O campo **RET** é utilizado para indicar se o CD vai ficar, ou não, à espera de um resultado. Adicionalmente existe a mensagem

```
HaltMsg :: STATUS: INT;
```

que o Controlador utiliza para informar o Modelo da Aplicação de que deve terminar a execução. O protocolo Controlador de Diálogo-Modelo da Aplicação é, assim,

```
MesgStSm = GetVarMsg | CallMsg | GetTypMsgSt | HaltMsg;
```

Na sentido inverso existem apenas as mensagens **SetValMsg** que o Modelo da Aplicação utiliza para enviar valores ao CD, quer como resposta a **GetValMsg**, quer a **CallMsg** quando a operação produza um resultado e ele seja pedido e **DefTypMsg** que o Modelo da Aplicação utiliza para responder ao pedido da definição de um tipo.

```
MesgSmSt = SetValMsg | DefTypMsg;
DefTypMsg :: SOURCE: InstId
             DEF:    TypeDef;
```

8.2.2 Animador de *Petri Nets*

Sendo o comportamento dos Guiões de Interacção modelado por *Petri Nets*, o Controlador de Diálogo foi desenvolvido sobre um animador dessas redes que aqui se apresenta.

No desenvolvimento de qualquer sistema *software*, está sempre presente a necessidade de estabelecer um compromisso entre a complexidade das estruturas de dados e a dos algoritmos. No caso presente optou-se por simplificar as *Petri Nets*, passando alguma da informação especificada nas redes do Capítulo 5 para a alçada do controlador de diálogo. Deste modo, toda a informação referente às cláusulas TRANS não é incluida na definição da rede, sendo da responsabilidade do Controlador o seu tratamento. A rede referente à expressão

gi(var)

não necessita, deste modo, de duas transições para **end** (ver fig. 13) passando a existir apenas a referente à situação em que se verificam as condições de TRANS. Quando o Controlador detecta que essas condições não se verificam, substitui o evento **end** por um **cancel** e executa as acções de EXCEP. Deste modo o comportamento do sistema é idêntico. Também as condições de CONTEXT não são incluídas. O animador da rede invoca uma operação do Controlador que faz a sua verificação. Deste modo consegue-se, por um lado simplificar o animador das redes e a geração das mesmas e, por outro, optimizar a partilha de informação. Na versão original das redes, seria fácil termos as mesmas condições e instruções repetidas em várias transições. Deste modo a informação está centralizada nas definições dos Guiões.

Formalmente as rede utilizadas podem ser modeladas da seguinte forma:

```
PetriNet :: B : Case
            Ev: Events
            Fi: Flow
            Fo: Flow
            Cb: Case
            Ce: Case;
Conditions = CId -> Bool;
Events = Event-set;
Event = StartEvent | CancelEvent | EndEvent | CmdEvent |
        NilEvent;
StartEvent :: DREF: EvId;
CancelEvent :: DREF: EvId;
EndEvent :: DREF: EvId;
CmdEvent :: DREF: CmdId;
NilEvent :: DREF: STR;
Flow = Event -> Case;
Case = CId-set;
```

B é o conjunto de todos os lugares da rede, **Ev** é o conjunto de todas as transições, **Fi** e **Fo** são os fluxos de entrada e saída dessas transições (de que lugares retiram *tokens* e em que lugares os colocam) e **Cb** e *Ce* são as marcações inicial e final, respectivamente. Como a definição da *Petri Net* será partilhada por todas as instâncias de um Guião ela não inclui a marcação actual, devendo cada instância conservar a sua.

A utilização do animador é feita por meio das seguintes funções:

startpn: PetriNet × InstId → Conditions

que inicializa a rede cuja descrição lhe é passada como parâmetro, devolvendo a marcação daí resultante. A necessidade de indicar a instância prende-se com a verificação das condições necessárias para que as transições se possam efectuar;

firepn: PetriNet × Conditions × Event × InstId → Conditions

que implementa o disparo de uma transição. Neste caso é passada a descrição da rede, a marcação actual, qual o evento e a instância, sendo devolvida a marcação resultante da transição, e

endpn: PetriNet × Conditions → Bool

que serve para verificar se uma dada marcação é a marcação final da rede referida.
Adicionalmente existe a função:

valideventspn: PetriNet × Conditions × InstId → Event-set

para calcular todos os eventos possíveis para uma dada marcação da rede.

8.2.3 O Interpretador de Instruções

Outra tarefa, bem definida, que o Controlador de Diálogo deve realizar, é a execução das instruções e cálculo das expressões colocadas em INIT e TRANS. Para o efeito foi desenvolvido um pequeno interpretador de instruções que, no entanto, facilmente pode ser ampliado.

Na sua versão actual estão previstas expressões envolvendo atribuições, invocações de operações (com ou sem resultado), a instrução **out**, as instruções condicional, **if...then...else...** e cíclica, **while...do...**. Foi ainda incluído o tratamento de expressões do tipo inteiro, *string* e booleano.

A utilização do interpretador é feita através das operações

docode: Code × InstId →

para executar uma sequência de instruções e

dotypedexp: TypedExp × InstId × OpcGISym → Value

para calcular o valor de uma expressão. O identificador de instância passado às operações é o da instância à qual se referem as expressões a calcular e é necessária para se fazer o acesso às variáveis do Guião. No caso em que se invoca **dotypedexp** para verificar a cláusula CONTEXT de um Guião cuja a instância ainda não foi criada, passa-se a instância "pai" e o símbolo do Guião a que se refere a expressão a calcular. No caso normal o terceiro parâmetro terá o valor NIL.

8.2.4 A Inclusão dos Tipos

Como foi dito na secção 5.6, ao nível da especificação os valores dos argumentos são sempre considerados como *tokens* léxicos, independentemente do seu tipo. Na implementação, no entanto, torna-se necessário controlar a leitura desses valores. Se os tipos básicos (inteiros, *strings*, etc.) não apresentam problemas, o mesmo já não se pode dizer dos que são definidos à custa dos modelos. A leitura de uma Função Finita ou de um

Conjunto não é uma tarefa trivial, existem vários valores a serem lidos e condições que devem ser verificadas[21]. Coloca-se, então, o problema de decidir quem/como controlar esse diálogo.

Numa primeira fase pretendeu-se atribuir esta responsabilidade ao Modelo da Apresentação. Se os valores são *tokens* léxicos, então deverá ser a camada léxica a proceder à sua construção. Tal solução mostrou-se pouco adequada, principalmente por "furar" o esquema de funcionamento já existente, nomeadamente no que diz respeito à leitura concorrente de valores. Digamos que se introduzia uma (grande) singularidade no Modelo da Apresentação.

A ideia seguinte foi colocar o controlo da leitura no Controlador de Diálogo. Assim, quando uma variável é de um tipo definido com recurso a modelos estruturados, o seu tipo léxico é substituído pela indicação do GI que deverá controlar a interacção a ela referente. No Controlador de Diálogo, por sua vez, esse Guião é definido como sendo de um tipo especial e o controlador "desvia" o processamento dos eventos a ele dirigidos para um módulo de tratamento apropriado. Este processo é transparente uma vez que o protocolo de comunicação continua a ser o mesmo.

Este modo de funcionamento permite ter vários métodos de leitura para cada um dos modelos, bastando para tal definir tipos de Guiões e os respectivos módulos de processamento associados. Actualmente estão definidos tipos de Guiões para a leitura de Funções Finitas (FFSYNTH), Relações (RELSYNTH), Listas (LISTSYNTH), Conjuntos (SETSYNTH) e Tuplos (TUPSYNTH). A definição dos módulos de processamento para cada um destes Guiões baseou-se, em parte, na experiência adquirida em [CM92].

Os Guiões do tipo FFSYNTH e RELSYNTH geram um diálogo em que os pares domínio/contradomínio (variáveis **dom** e **ran**) são apresentados/lidos um a um, sendo disponibilizados, para além de OK e CANCEL, os comandos NEW (inicializar a vazio), UP (par anterior), DOWN (par seguinte) e DEL (apagar o par actual)[22]. Internamente, o valor é representado por quatro listas de valores:

- **dup** - valores do domínio anteriores ao valor actualmente apresentado;
- **rup** - valores do contradomínio correspondentes;
- **ddown** - na cabeça está o valor actualmente apresentado e na cauda os restantes;
- **rdown** - valores do contradomínio correspondentes.

Os Guiões do tipo SETSYNTH e LISTSYNTH são semelhantes aos anteriores mas a variável utilizada para apresentar os valores é **elem**.

Por último, os Guiões TUPSYNTH geram um diálogo correspondente à expressão

input(sel1) || input(sel2) || ... || input(seln)

com os comandos OK e CANCEL e em que **sel1** a **seln** são os selectores do tuplo. Importa notar que a representação sintática utilizada para os tuplos é uma função finita de nome do selector para valor correspondente.

[21] Por exemplo, no caso das Funções Finitas, não podem existir repetições no domínio.
[22] Aqui está um caso em que são acrescentados novos comandos aos inicialmente existentes.

8.2.5 Estrutura Geral do Controlador

A informação necessária ao Controlador de Diálogo consiste na definição dos Guiões de Interacção, na descrição das instâncias existentes e nos canais de comunicação com os outros componentes de MIU. Temos então:

```
CD :: DEFS:  GISym -> GIdef
      INSTS: InstId -> InstDescr
      OUTSM: CHAN
      INSM:  CHAN
      INLX:  CHAN
      OUTLX: CHAN;
GIdef = DECISION | SYNTH | VALSYNTH | FFSYNTH | RELSYNTH |
        SISTSYNTH | SETSYNTH | TUPSYNTH;
FFSYNTH :: DREF: TypeId;
InstDescr :: FATHER:  NIL | InstId
             VARS:    VarId -> Value
             EVSEQ:   Conditions
             CMDLINE: NIL | CmdLineDescr;
```

O seu funcionamento consiste basicamente em:

- criar as instâncias, quando tal é pedido;
- idem para a sua remoção;
- validar e efectuar, na *Petri Net* da instância indicada, as transições relativas aos eventos que lhe são comunicados;
- indicar ao Modelo da Apresentação quais os eventos válidos em cada momento;
- indicar, também, alterações nos valores das variáveis conhecidas pelo Modelo da Apresentação;
- construir e enviar a frase ao Modelo da Aplicação para avaliação e receber o resultado, passando-o ao Modelo da Apresentação.

Na fig. 24 são apresentados os ficheiros que compõem o Controlador de Diálogo. Note-se que as setas representam uma relação de inclusão textual, pelo que podem existir (e existem de facto) referências cruzadas entre os ficheiros.

O ficheiro **cd.n** contém o corpo principal do Controlador. Em **cdcom.n** estão as funções de comunicação e este por sua vez inclui **msgs.n** onde são definidos os protocolos de comunicação; **petrinet.n** é o animador de *Petri Nets* e **mkpn.n** um gerador das mesmas a partir de descrições de EVSEQ utilizando as seguintes definições:

```
PNDescr = NIL | SeqDescr | RepDescr | ConcSDescr | ConcADescr |
          OpcDescr | EvId;
SeqDescr :: P: PNDescr                /* Sequenciacao     */
            S: PNDescr;
RepDescr :: DREF: PNDescr;            /* Repeticao        */
ConcSDescr :: A: PNDescr              /* Conc. Sincrona   */
              B: PNDescr;
```

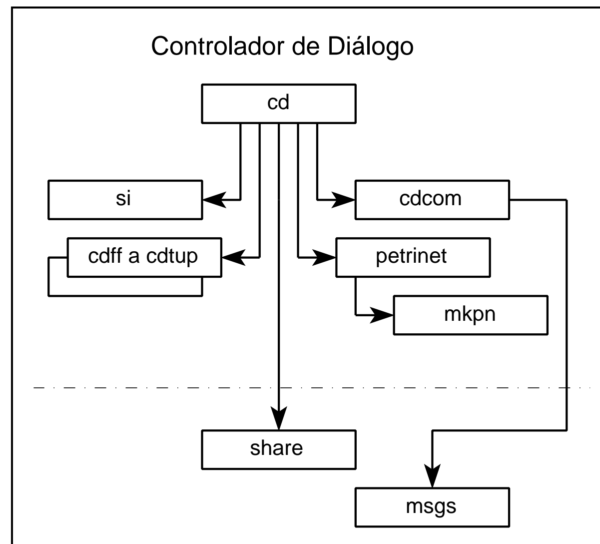


Figura 24: Composição do Controlador de Diálogo

```
ConcADescr :: A: PNDescr            /* Conc. Assincrona */
              B: PNDescr;
OpcDescr :: A: PNDescr              /* Alternativa      */
            B: PNDescr;
```

si.n é o interpretador de instruções e **cdff.n** a **cdtup.n** são os módulos de tratamento dos modelos.

Finalmente, em **share** são feitas definições globais aos três componentes do MIU. Tanto este ficheiro como **msgs.n** são partilhados pelos três componentes do sistema.

8.3 O Modelo da Apresentação

O Modelo da Apresentação pode ser dividido num módulo genérico (ModApr) de processamento de eventos - cuja principal responsabilidade é implementar as comunicações com o Controlador de Diálogo - e um *frontend* que implementará o aspecto gráfico da interface e a tradução dos diferentes eventos de e para o utilizador, tanto em Modo Menu como em Modo Comando.

É em ModApr que é colocada a informação resultante da compilação dos Descritores de Apresentação, ele conterá ainda informação relativa aos canais para comunicação com o Controlador de Diálogo e o Modelo da Apresentação.

A Descritores do tipo MENU são modelados pelos tuplos:

```
MENU :: NAME:  STR
        POS:   STR
        COLOR: STR
        SEL :  STR
        OPCS:  Option-list;
Option :: GI:   EvId
          NAME: STR;
```

os DB por:

```
DB :: NAME:  STR
      POS:   STR
      COLOR: STR
      VARS:  VarId -> ViewType
      GIS:   EvId -> DBGIView
      CMDS:  CmdId -> LxDef;
```

Na descrição gráfica das variáveis, se elas forem de um dos tipos pré-definidos, poder-se-á utilizar um dos métodos de visualização também pré-definidos:

- **HimSelf**;
- **RadioBox**;
- **OptionMenu**;
- **Scale**.

Todos eles partilham um conjunto de definições léxicas descritas no tipo **LexDef**. Actualmente, estão contempladas a posição, o nome, e a cor. Mais características podem ser introduzidas sem perturbar o funcionamento de *frontends* já existentes uma vez que cada um utiliza apenas aquelas que consegue implementar ou de que tem conhecimento.

```
ViewType = PreDefView | GISym;
PreDefView = HimSelf | RadioBox | OptionMenu | Scale;
HimSelf :: DEF: LxDef;
RadioBox :: VALS: STR -> Value
            DEF:   LxDef;
OptionMenu :: VALS: STR -> Value
              DEF:   LxDef;
Scale :: MIN: Value
         MAX: Value
         DEF:  LxDef;
DBGIView :: DEF:   LxDef
            OPEN:  Bool;
LxDef :: POS:   STR
         NAME:  STR
         COLOR: STR;
```

Caso se trate de uma variável cujo tipo é definido utilizando modelos, é feita a indicação do Guião de Interacção responsável pelo seu controlo. Nesta situação é criada uma instância desse Guião e o diálogo referente à variável é desviado para ela. O valor da variável é enviado à instância referida através da mensagem **SetValMsg** e a instância é activada; quando termina, o valor devolvido por ela é enviado de novo para o Controlador de Diálogo, à instância onde a variável está declarada, como valor lido para a variável em causa.

A definição léxica dos GI's contém, para além das definições léxicas normais, o valor booleano que indica se a descrição léxica do Guião deve ser apresentada automaticamente ou não[23].

[23] Neste caso será apresentado, por exemplo, um botão que, quando seleccionado, desencadeará a apresentação da descrição do GI.

Os comandos são descritos utilizando simplesmente **LexDef**.
Podemos então definir ModApr:

```
ModApr :: GLOBAL:  GISym -> GlobalInfo
          VIEWS:   ViewId -> ViewDescr
          DEFAULT: ViewId
          MAIN:    GISym
          INST:    CHAN
          INSM:    CHAN
          OUTST:   CHAN
          OUTSM:   CHAN;
GlobalInfo = VarId -> TypeId;
ViewDescr = GISym -> GIView;
GIView = MENU | DB;
```

Como é evidente, as definições léxicas descritas são relevantes apenas quando o *frontend* está a funcionar em Modo Menu, uma vez que o Modo Comando terá uma sintaxe fixa[24]. Pelo mesmo motivo, em Modo Comando não é necessário recorrer aos Guiões de síntese de modelos.

Para estudar a implementação de um *frontend* procedeu-se ao desenvolvimento de um para XMetoo. Dada a quase inexistência de facilidades de *input/output* da linguagem[25], o resultado obtido não prima pela "*user frendliness*", no entanto possui todas as características necessárias a um *frontend* mais poderoso (em X11 por exemplo).

A principal tarefa que lhe compete é, por um lado, apresentar ao utilizador a implementação da descrição gráfica dos diferentes elementos da interface, por outro, gerir o *input/output* de valores. Para tal, é mantida uma tabela das instâncias actualmente criadas onde se regista, para além da hierarquia em que elas estão organizadas (selectores FATHER e CHILDREN), os valores das suas variáveis e qual a variável a que o resultado da sua conclusão deve ser atribuído se ele existir (VALUES e VAR) e ainda uma máscara onde estão definidos os eventos válidos (EVMASK) e a indicação se a instância já foi activada (ACTIVE).

```
InstInfo :: FATHER:   OpcInstId
            EVMASK:   EvMask
            VAR:      OpcVarId         /* Vari'avel do Pai */
            VALUES:   VarId -> OpcValue
            CHILDREN: EvId -> InstId
            ACTIVE:   Bool;
```

Quando em Modo Menu, em cada instante o utilizador deve escolher de entre todas as instâncias activas qual a que pretende utilizar (corresponderá à colocação do rato sobre uma janela). De seguida, e dentro dessa instância, pode interactuar com todos os seus elementos que geram eventos válidos nesse contexto.

Em Modo Comando é utilizada uma sintaxe semelhante à do XMetoo. Os Guiões e os argumentos são vistos como funções, os comandos não são utilizados sendo as frases

[24] Na realidade, esta afirmação não é totalmente verdadeira. Mesmo quando em Modo Comando, se um argumento está definido como **Scale** o seu valor é comparado com os limites da definição, se está definido como **OptionMenu** ou **RadioBox** é testado se o valor é um dos enumerados na definição.

[25] O meu agradecimento ao Eng. José João pela cedência das suas funções de I/O.

confirmadas com **Return**. A invocação do Guião **GInsPal** é feita do seguinte modo (neste caso está a utilizar-se o sinónimo do Guião):

```
cmd> (inspal (pal "Ola") (sig "Hello"))
```

Caso seja detectado um erro na frase (quer sintático quer semântico) o Modelo da Apresentação muda automaticamente para Modo Menu, preservando o estado em que o erro ocorreu. Assim, se no exemplo anterior em vez de **sig** tivesse sido escrito **sgi** - ou se em vez de uma *string* tivesse sido escrito um inteiro - a interface passava para Modo Menu apresentando a *Dialog Box* de **GInsPal**, tendo o argumento **pal** o valor **"Ola"**.

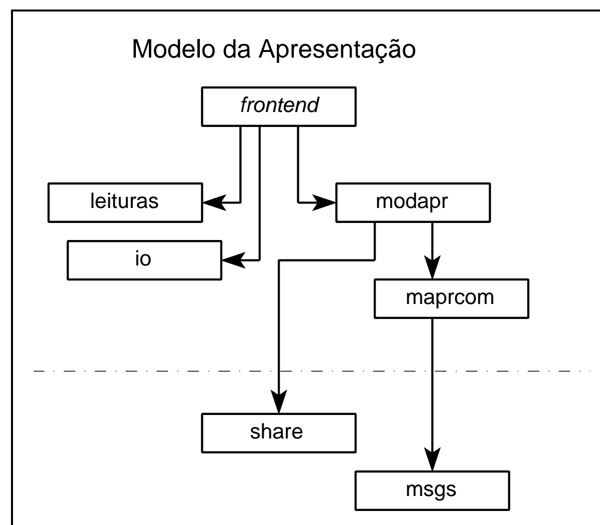


Figura 25: Composição do Modelo da Apresentação

Na fig. 25 são apresentados os ficheiros que compõem o Modelo da Apresentação. Os ficheiros **io.n** e **leituras.n** implementam o *input/output*, neste caso os métodos pré-definidos são considerados apenas para leitura (a apresentação do valor é sempre feita directamente) à excepção dos métodos **HimSelf** e **Scale** que são traduzidos para leituras simples (embora no caso do segundo os limites sejam verificados), todos os outros são implementados como menus. O ficheiro **fexm.n** é o corpo principal do *front-end* e **modapr.n** o do ModApr, **maprcom.n** implementa as comunicações.

8.4 O Modelo da Aplicação

O Modelo da Aplicação é composto por uma parte fixa, que será aqui apresentada, e pelas funções de tradução de valores que devem ser fornecidas pela equipa de desenvolvimento da camada computacional, que, por serem dependentes da aplicação em causa, não serão abordadas. Deste modo, quando referirmos o Modelo da Aplicação, estaremos apenas a falar da componente referida em primeiro lugar.

Esta componente da interface é a mais simples das três, limitando-se a ser um servidor de pedidos feitos pelas outras duas. A informação que armazena reflecte o tipo de pedidos a que responde:

- pedidos para obtenção das definições dos tipos (pedido: **GetTypMsg**; resposta: **DefTypMsg**);

- pedidos para verificação de invariantes (pedido: **InvMsg**; resposta: **SetValMsg**);
- pedidos para invocação de operações (pedido: **CallMsg**; resposta: **SetValMsg**);
- pedidos para obtenção dos valores de variáveis da aplicação (pedido: **GetTypMsgSt** e **GetTypMsgLx**; resposta: **SetValMsg**).

Deste modo, o Modelo da Aplicação necessita saber, como vimos, as definições dos tipos (em termos de modelos) e respectivos invariantes e a que operações da aplicação estão as variáveis da aplicação e as operações do Controlador de Diálogo associadas (cf. secção 7.4). Naturalmente, necessita também de canais para efectuar as comunicações.

```
ModApl :: TYPS:  TypeId -> TypeInfo
          VARS:  VarId -> PTR
          LINKS: STR -> PTR
          IN: CHAN
          OUTST: CHAN
          OUTLX: CHAN;
TypeInfo :: DEF: TypeDef
            INV: OpcPTR;
```

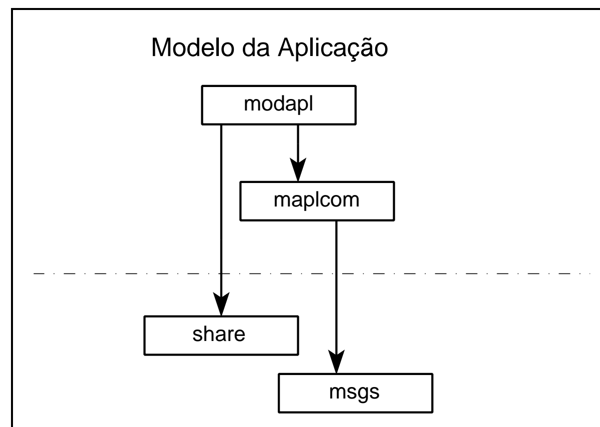


Figura 26: Composição do Modelo da Aplicação

Como a fig. 26 mostra, a estrutura de ficheiros reflecte esta simplicidade. São apenas necessários, para além dos ficheiros comuns a todas as componentes do MIU, o corpo principal do servidor (**modapl.n**) e as rotinas de comunicação (**maplcom.n**).

8.5 Utilização e Exemplos

Vamos agora apresentar, como exemplo, a utilização do sistema descrito para a geração de uma interface para o dicionário apresentado no Apêndice A (para um exemplo mais completo ver [Cam93]).

Como já foi visto, a especificação da interface é feita a três níveis:

- Especificação do Modelo da Apresentação;

- Especificação do Controlador de Diálogo;
- Especificação do Modelo da Aplicação.

8.5.1 Modelo da Apresentação

Não estando ainda disponível o compilador de Descritores de Apresentação para a geração do Modelo da Apresentação, torna-se necessário especificá-lo directamente em NYAGSL utilizando os modelos descritos na secção 8.3. Esta especificação é apresentada no Apêndice F (no Apêndice C são apresentados os Descritores correspondentes).

Como pontos mais salientes importa notar que uma vez que o *frontend* XMetoo não trata as definições léxicas posição e cor, elas não são sequer especificadas; importa também referir que é feita a definição de duas *views*, a chamada "portugues", que define nomes em português para as variáveis, comandos e guiões, e a "ingles" que os define em inglês.

8.5.2 Controlador de Diálogo

A maior parte dos Guiões necessários para a especificação do Controlador de Diálogo foram já apresentados ao longo do Capítulo 5; no Apêndice B é apresentada toda a especificação.

Está em desenvolvimento um compilador de Guiões de Interacção. Porém, enquanto ele não está disponível, no entanto, é necessário especificar os GI's directamente em NYAGSL. Tal especificação é apresentada no Apêndice E.

8.5.3 Modelo da Aplicação

O Modelo da Aplicação limita-se a definir os tipos **Pal** e **Sig** e a associar aos identificadores das operações o respectivo símbolo.

Também neste caso é ainda necessário especificar o Modelo da Aplicação directamente em NYAGSL. Nos Apêndices D e G é apresentada a especificação utilizando a notação apresentada na secção 7.4 e em NYAGSL, respectivamente.

8.5.4 Utilização

Quando se inicia a interacção são lançadas três janelas:

- Aplicação - corresponde ao processo onde correm o Modelo da Aplicação e a componente computacional e nela são apresentadas as mensagens que o modelo envia e recebe;
- Controlador de Diálogo - corresponde ao processo homónimo em que são apresentadas também as mensagens por ele recebidas e enviadas;
- Apresentação - é através desta janela, onde executa o Modelo da Apresentação, que é efectuado o diálogo com o utilizador.

Inicialmente é apresentado o Menu que se pode ver na fig. 27. Note-se que é acrescentada uma opção às especificadas no GI Menu. A opção adicionada destina-se a permitir a alteração da *view* a utilizar.

Se for seleccionada a opção "Inserir Palavra", é apresentado um novo menu, agora relativo à inserção e que está a ser controlado pelo GI GInsPal. Inicialmente, estão

disponíveis a leitura da palavra e o cancelamento da interacção (variável **pal** e comando **Cancel** respectivamente, cf. Modelo da Apresentação para o Guião GInsPAl), como se pode ver na fig. 28.

Depois de introduzida a palavra e respectivo significado e seleccionado o comando **OK** (que entretanto fica disponível) o GI termina e o diálogo volta a ser controlado por Menu, sendo apresentado o menu da fig. 29. Como se pode constatar, passaram a estar disponíveis também as opções Consulta e Remover, que anteriormente não estavam pois as suas condições de contexto indicam que elas só são válidas quando o dicionário não está vazio (cf. Guiões GInsPal e GViewConsPal).

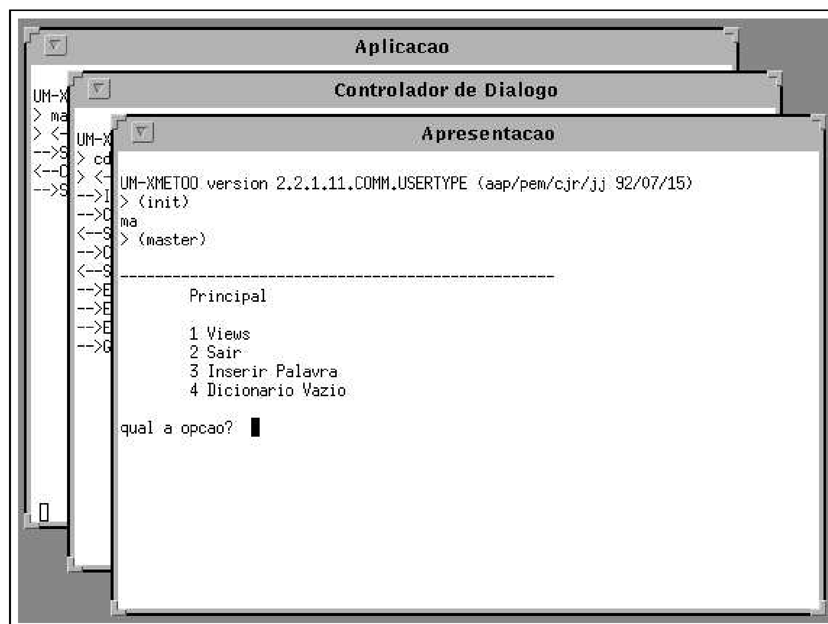


Figura 27: Menu Inicial

8.6 Versão X11 do *Frontend*

Está neste momento em desenvolvimento uma versão para X11 do Modelo da Aplicação. Através de um ambiente misto C/XMetoo [AR93] torna-se possível desenvolver um *frontend* que, por um lado, lê as estruturas XMetoo que descrevem a apresentação e, por outro, utiliza o ModApr previamente apresentado para comunicar com o Controlador de Diálogo. Problemas com as comunicações neste ambiente, no entanto, têm atrasado o desenvolvimento do *frontend*, não sendo ainda possível incluir aspectos dinâmicos (comunicação com o Controlador de Diálogo).

A título de exemplo são apresentadas as figuras 30 e 31 em que pode observar-se o efeito da alteração da *View*. Note-se que, para a obtenção desta interface, se procedeu apenas à substituição do *frontend*. Todo o restante MIU é o mesmo utilizado anteriormente bem como as especificações.

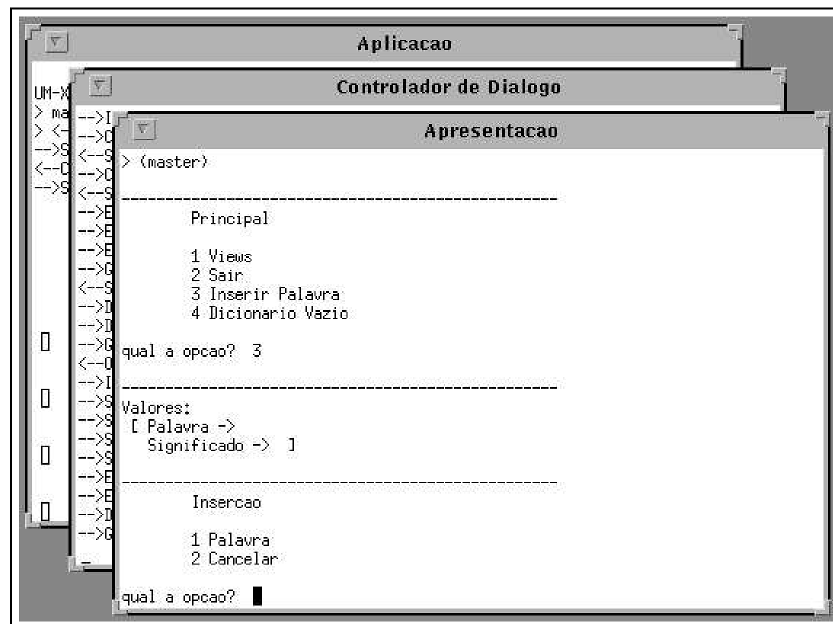


Figura 28: Inserção

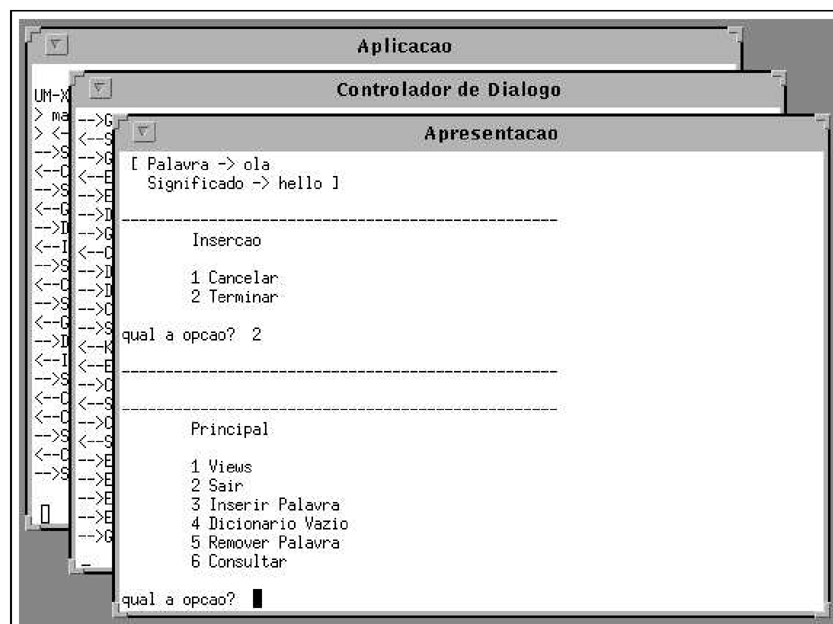


Figura 29: Menu Inicial depois da Inserção

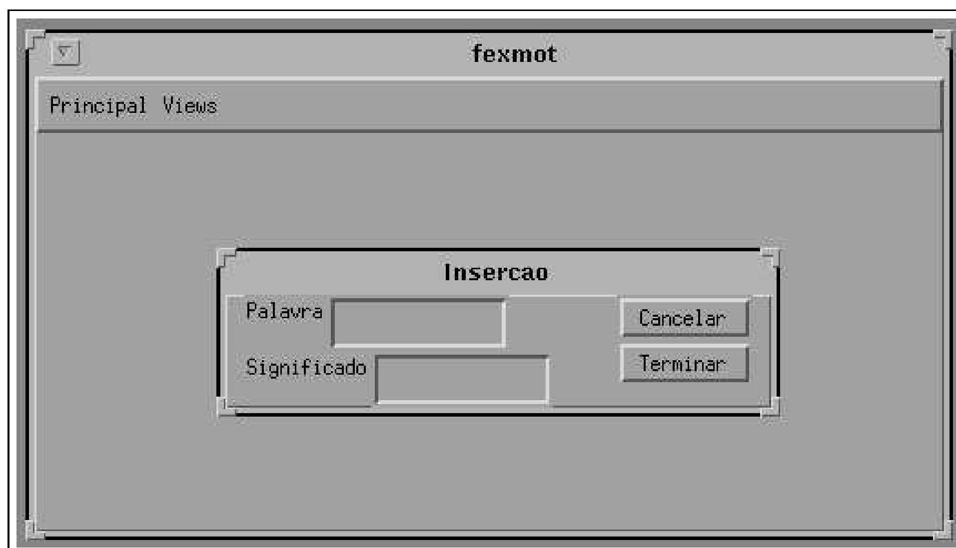


Figura 30: *View* "Portugues"

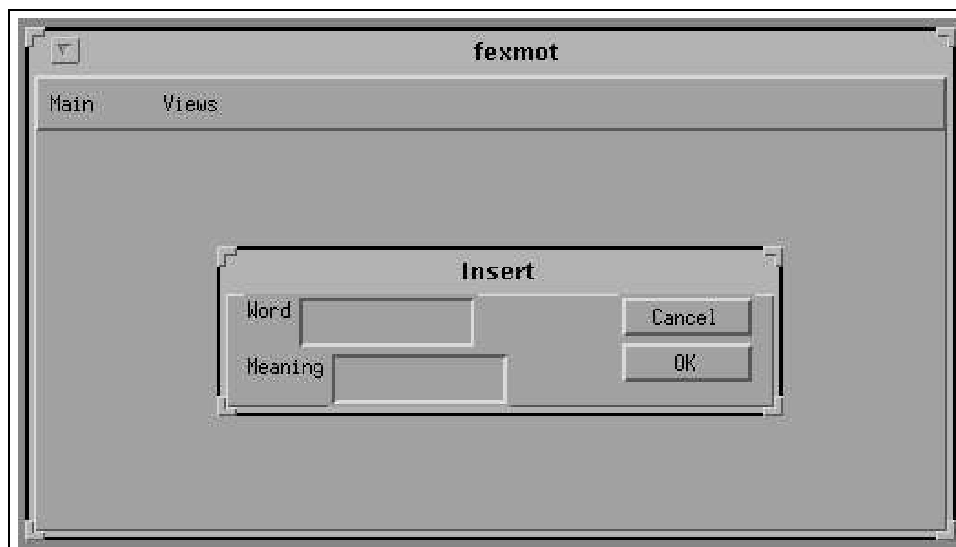


Figura 31: *View* "Ingles"

9 Conclusões

"... o que foi feito está feito
e ficou tanto por fazer."

Sétima Legião - in *"Tão Só"*

É hoje ponto assente que uma boa Interface Humano-Computador é um factor fundamental para o sucesso de qualquer sistema *software* interactivo.

Para além da complexidade inerente à concepção de um sistema *software*, o desenvolvimento de uma interface humano-computador, deve ainda ter em conta o factor humano com as suas características de não-determinismo, imprecisão, etc. O conceito de Interface em Modo Assistido surge desta necessidade. Uma Interface Assistida age de um modo preventivo: evita que o utilizador cometa erros, solicita informação necessária ainda não fornecida, etc.

O objectivo principal deste trabalho foi o desenvolvimento e implementação de um formalismo para a especificação da componente de controlo de diálogo (tradicionalmente dita sintáctica) para Interfaces em Modo Assistido: os Guiões de Interacção. Ao permitirem a especificação, não só da sintaxe da linguagem de interacção, mas também das condições semânticas a que essa linguagem deve obedecer, os Guiões de Interacção permitem desenhar um diálogo em que o utilizador, embora tendo a iniciativa (cf. Controlo *Dialogue Dominant*) é guiado através da interface (cf. Modo Assisitido).

Depois da apresentação dos Guiões de Interacção e dos Descritores de Apresentação, utilizados para caracterizar lexicamente os Guiões, foi feito o seu enquadramento no âmbito mais global do sistema GAMA-X - um UIMS para a linguagem CAMILA (descrita no Capítulo 4) - sendo apresentada a arquitectura proposta para o sistema. Em consequência da sua modularidade, o GAMA-X consegue acompanhar a aplicação desde o estado de protótipo até uma eventual implementação, com as óbvias vantagens de consistência da Interface, ao longo das diversas fases de desenvolvimento.

Finalmente, foi descrita a implementação da componente de *runtime* do GAMA-X em NYAGSL.

A divisão dos Guiões em três tipos com objectivos bem definidos, se por um lado guia o desenho de interfaces orientadas ao comando (como era objectivo), por outro, dificulta a especificação de outro tipo de interfaces. A utilização de um único tipo de Guião (que, *grosso modo*, corresponderia ao VALSYNTH) flexibilizaria a especificação. No entanto, deixaria de ser possível estabelecer, com tanta clareza, regras relativas à manutenção da validade semântica do contexto de interacção. Poderemos então dizer que as interfaces assistidas deixariam de ser suportadas para passarem, apenas, a ser possíveis. Este problema poderia, por sua vez, ser resolvido se fosse implementado um mecanismo de detecção e aviso de alterações no estado da aplicação[26]. Tal poderia originar, no entanto, interfaces com um comportamento pouco desejável: a meio da execução de uma tarefa ela poderia deixar de ser válida sendo cancelada por iniciativa do próprio sistema.

A utilização de *Petri Nets* C/E (aumentadas com condições e acções nas transições) permitiu especificar, de uma forma simples e rápida, a semântica dos Guiões de Interacção. A utilização de um animador de redes, por outro lado, demonstrou que elas são, também, um modelo adequado para a implementação dos Guiões de Interacção. Apenas em casos onde se torna necessário proceder à reinicialização global das redes surgiram

[26] Algo semelhante ao método **changed:** do SmallTalk.

problemas, dada a complexidade que a inclusão de todas as transições necessárias para o efeito acarretaria. A solução encontrada consistiu em colocar a responsabilidade da reinicialização no animador das redes. Note-se que esta solução é satisfatória em termos de implementação mas, relativamente à especificação formal, em rigor, seria necessário fazer o cálculo das transições referidas.

A actual versão dos Descritores de Apresentação (utilizados para especificar o Modelo da Apresentação), contempla, ainda, poucas características léxicas. Outras poderão ser, no entanto, adicionadas sem perturbar grandemente o funcionamento do sistema, uma vez que cada Apresentação tratará aquelas características para as quais possui capacidade e das quais tem conhecimento. O desenvolvimento de *frontends* para diversos ambientes encarregar-se-á, certamente, de fazer surgir a necessidade de outras características.

Em relação à implementação da componente de *runtime*, foi tomada a decisão de deixar a cargo do Modelo da Apresentação a activação dos diferentes Guiões conforme vão sendo necessários para controlar a Interface. Seria talvez interessante estudar a outra possibilidade apresentada, colocando essa responsabilidade no Controlador de Diálogo. Deste modo, será possível ao Modelo da Aplicação intrometer-se, mais facilmente, no diálogo Modelo da Apresentação/Controlador de Diálogo e assim simular interacção. Tal permitirá, por exemplo, que o Modelo da Aplicação receba directamente invocações de operações e faça *feedback* para o utilizador das mesmas.

A actual versão do Modelo da Apresentação, a correr em XMetoo, não é muito interessante do ponto de vista estrito de utilização. Uma versão para X11 está a ser implementada, mas problemas com as comunicações no ambiente híbrido C/XMetoo, têm atrasado a sua conclusão. Está ainda a ser considerada a hipótese de transportar o sistema GAMA-X para Xlisp, o que permitirá a utilização do Winterp [May90][27] - um sistema para a geração de interfaces X11 - para a implementação do Modelo da Apresentação.

Por último, em relação ao sistema GAMA-X, está já disponível, como vimos, uma versão para XMetoo do Módulo de Interacção com o Utilizador, estando a ser desenvolvida a versão X11 como foi dito. Relativamente ao Módulo de Geração de Interfaces, está praticamente acabado o compilador de Guiões de Interacção.

Para que o sistema se possa considerar completo, falta então estudar e implementar:

- o gerador de Guiões de Interacção a partir de especificações CAMILA (algumas pistas para esse estudo são apresentadas neste trabalho),
- os geradores e compiladores para os Descritores de Apresentação e desenvolver Modelos da Aplicação para as diferentes linguagens em que a aplicação pederá ser implementada e Modelos da Apresentação para outros ambientes, nomeadamente NeXTStep.

[27] O meu agradecimento ao Pina Miranda por me ter chamado a atenção para este sistema.

Referências

[ABCM89] J. J. Almeida, J. B. Barros, P. M. Castro, and F. C. Madeira. Pré-processador para YARPT. Relatório jnict:pmct:87/66:jja1, Departamento de Informática, Universidade do Minho, Outubro 1989.

[Abo92] Gregory D. Abowd. Using formal methods for the specification of user interfaces. In *Proceedings of the Second Irivine Software Symposium - ISS'92*, Março 1992.

[AR93] José João Almeida and Jorge Gustavo Rocha. Interface C XMETOO: manual de utilização. Technical report, Universidade do Minho, Junho 1993.

[BA91] L. Barbosa and J. J. Almeida. CAMILA by Example. Relatório interno, DI/INESC, Universidade do Minho, 1991.

[BCMR90] M. Bordegoni, U. Cugini, M. Motta, and C. Rizzi. An Environment for User Interface Development Based on the ATN and Petri Nets Notations. In *Eurographics/Esprit User Interface Management Systems and Environments*, pages 177–192, Junho 1990.

[BP81] H. G. Borufka and G. Pfaff. The design of a general-purpose command interpreter for a graphical man-machine communication. In T. Sata and E. Warman, editors, *Man-Machine Communication in CAD/CAM*. North-Holland, 1981.

[BW84] I. Benbasat and I. Wand. A structured approach to designing human-computer dialogs. *Int. Journal of Man-Machine Studies*, 21:105–126, 1984.

[Cam93] J. Creissac Campos. GAMA-X Manual de Utilização (NYAGSL). Technical report, Universidade do Minho/Departamento de Informática, 1993.

[CC90] Niels Vejrup Carlsen and Niels Jørgen Christensen. Modelling User Interface Software. In *Eurographics/Esprit User Interface Management Systems and Environments*, pages 21–35, Junho 1990.

[CM92] J. Creissac Campos and F. Mário Martins. IAPF - Interfaces Assistidas para Protótipos Funcionais. Technical report, Universidade do Minho/INESC, 1992.

[Dig87] Digitalk Inc. *Smalltalk/V Tutorial and Programming Handbook*, Agosto 1987.

[DR85] Alan Dix and Colin Runciman. Abstract Models of Interactive Systems. In *People and Computers: Designing the Interface - Proc. of the BCS-HCI Spec. Group*, pages 13–22, Setembro 1985.

[FN85] A. J. Fountain and M. A. Norman. Modelling User Behaviour with Formal Grammar. In *People and Computers: Designing the Interface - Proceedings of the BCS-HCI Spec. Group*, pages 3–12, Setembro 1985.

[GR90] Joachim Grollmann and Christoph Rumpf. Some Comments on the Future of User Interface Tools. In *Eurographics/Esprit User Interface Management Systems and Environments*, pages 3–20, Junho 1990.

[Gre86] Mark Green. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):243–275, Julho 1986.

[Gro90] The CAMILA Group. XMETOO - User's Manual. Technical report, DI/INESC, Universidade do Minho, 1990.

[Hay90] Clive Hayball. Dialogue Specification for Knowledge Based Systems. In *Eurographics/Esprit User Interface Management Systems and Environments*, pages 107–117, Junho 1990.

[HH89] H. Rex Hartson and Deborah Hix. Human-Computer Interface Development: Concepts and Systems for Its Management. *ACM Computing Surveys*, 21(1):5–92, Março 1989.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, UK, Ltd., 1985.

[HS90] P. J. W. ten Hagen and D. Soede. Assembling a User Interface out of Communicating Processes. In *Eurographics/Esprit User Interface Management Systems and Environments*, pages 81–86, Junho 1990.

[Jac83] Robert J. K. Jacob. Using Formal Specifications in the Design of a Human-Computer Interface. *Communications of the ACM*, 26(4):259–264, Abril 1983.

[Jac86] Robert J. K. Jacob. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283–317, Outubro 1986.

[LJBS78] Harold W. Lawson Jr., Miquel Bertran, and Javier Sanagustin. The Formal Definition Of Human/Machine Communication. *Software-Practice and Experience*, 8:51–58, 1978.

[MAH90] F. Mário Martins, J. João Almeida, and Pedro R. Henriques. Mecanismos para a Especificação e Prototipagem de Interfaces Utilizador-Sistema. Universidade do Minho, 1990.

[Mar92] F. Mário Martins. Interacção Homem-Máquina. Universidade do Minho, 1992.

[May90] Niels P. Mayer. The WINTERP Widget INTERPreter - An application prototyping and extension environment for OSF/Motif. Hewlett-Packard Laboratories, Outubro 1990.

[MCCM90] Filipe C. Madeira, J. Creissac Campos, Pedro M. Castro, and F. Mário Martins. GAMA - Geração Automática de Modo Assisitido. Relatório interno, DI/INESC, Universidade do Minho, 1990.

[MF93] F. Mário Martins and Adalberto G. Ferreira. GAIA - Gerador Automático de Interfaces Adaptativas. In *5º Encontro Português de Computação Gráfica*, pages 211–225, Fevereiro 1993.

[MO90] F. Mário Martins and J. Nuno Oliveira. Archetype Oriented User Interfaces. *Computer & Graphics*, 14(1):17–28, 1990.

[MVM90] Margarita Martínez, Bonifacio Villalobos, and Pedro de Miguel. GMENUS: an Ada Concurrent User Interface Managment System. In *Eurographics/Esprit User Interface Management Systems and Environments*, pages 37–48, Junho 1990.

[OJ86] Dan R. Olsen Jr. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics*, 5(1):318–344, Outubro 1986.

[Oli90] J. N. Oliveira. A Reification calculus for Model-Oriented Software. Technical report, Universidade do Minho/INESC, Março 1990.

[QO90a] Valerie Quercia and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual.* O'Reilly & Associates, Inc., 2.rd edition, Setembro 1990.

[QO90b] Valerie Quercia and Tim O'Reilly. *X Window System User's Guide.* O'Reilly & Associates, Inc., 3.rd edition, Maio 1990.

[Rad90] Mark J. C. Radcliffe. DESQview/X Extending DOS into de 21st Century - A Technical Perspective. Technical report, Quarterdeck Office Systems, Agosto 1990.

[Rei] W. Reisig. *Petri Nets - An Introduction.* Springer-Verlag.

[Rei81] Phyllis Reisner. Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Transactions on Software Engineering*, 7(2):229–240, Março 1981.

[Roc91] Jorge Gustavo Rocha. CAMILA Modular - Um Pré-processador para YARPTC. Universidade do Minho, 1991.

[Roc93] Fernando Rocha. Um Editor/Compilador de Guiões de Interacção. Relatório de estágio, Universidade do Minho/Departamento de Informática, 1993. (em preparação).

[Ros87] David S. H. Rosenthal. Window System Implementations. In *EUROGRAPHICS (UK) - The* 5th *Annual Conference*, Abril 1987. Notes for Tutorial A.

[Shn82] Ben Shneiderman. Multiparty Grammars and Related Features for Defining Interactive Systems. *IEEE Transactions on Systems, Man and Cybernetics*, 12(2):148–154, Março/Abril 1982.

[Ste87] Hal L. Stern. Comparison of Window Systems. *BYTE*, pages 265–272, Novembro 1987.

[Stu90] Peter Sturm. IUICE An Interactive User Interface Construction Environment. In *Eurographics/Esprit User Interface Management Systems and Environments*, pages 89–106, Junho 1990.

[Sun90] Sun Microsystems, Inc. *SunView Programmer's Guide*, Maio 1990.

[Too91] Roger Took. The Active Medium: A Reference Architecture for Direct Manipulation. In *A Collection of Papers on HCI.* University of York - Department of Computer Science, 1991.

[Wal87] David Walker. Introduction to a Calculus of Communicating Systems. Department of Computer Science, University of Edinburgh, 1987.

[Was85] Anthony J. Wasserman. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, 11(8):699–713, Agosto 1985.

A Especificação CAMILA de um Dicionário

```
interface DIC =
    sorts Dic, Pal, Sig, Bool;
    state Dic;
   events INIT:->,
          INSPAL:PalxSig->,
          REMPAL:Pal->,
          CONSPAL:Pal->Sig,
          EMPTYDIC:->Bool,
          EXISTPAL:Pal->Bool;
end;

module Dicionario: DIC
 sorts Dic = Pal -> Sig;
       Pal = STR;
       Sig = STR;
       Bool = SYM;
events INIT =
         sigma' = [];
       INSPAL(pal, sig) =
         { ~EXISTPAL(pal) => sigma' = sigma + [pal -> sig];
       REMPAL(pal) =
         { EXISTPAL(pal) => sigma' = sigma\{pal};
       CONSPAL(pal, sig') =
         { EXISTPAL(pal) => sig' = sigma[pal];
       EMPTYDIC(b') =
         b' = sigma == [];
       EXISTPAL(pal, b') =
         b' = pal in dom(sigma);
end;
```

B Guiões de Interacção para o Dicionário

```
DefGI Menu
  Declarations
    TYPE DECISON
    SYMBOL {Menu, menu}
    SUBGI Fim
  Behaviour
    EVSEQ (GInit + GinsPal + GRemPal + ViewConsPal)* + Fim
  SubGI
    DefGI Fim
      Declarations
        TYPE DECISION
      Behaviour
    EndGI
EndGI

DefGI GInit
  Declarations
    TYPE SYNTH
  Behaviour
    EXEC INIT()
EndGI

DefGI GInsPal
  Declarations
    TYPE SYNTH
    SYMBOL {GInsPal, inspal}
    ARGS pal: Pal;
         sig: Sig
  Behaviour
    INIT pal = "";
         sig = ""
    EVSEQ input(pal).input(sig)
    TRANS input(pal): (pal != "") && not(EXISTPAL(pal)) =>
                               EXCEP pal=="" -> out("Palavra nula!");
                                     pal!="" -> out("Erro!")
          OK:
          CANCEL:
    EXEC INSPAL(pal, sig)
EndGI

DefGI GRemPal
  Declarations
    TYPE SYNTH
    SYMBOL {GRemPal, rempal}
    ARGS pal: Pal
    VAR-UI sig: Sig
```

```
  Behaviour
    CONTEXT not(EMPTYDIC())
    INIT pal = "";
         sig = ""
    EVSEQ input(pal)
    TRANS input(pal): EXISTPAL(pal) => sig = CONSPAL(pal)
                                              EXCEP out("Erro!")
          OK:
          CANCEL:
    EXEC REMPAL(pal)
EndGI

DefGI ViewConsPal
  Declarations
    TYPE SYNTH
    SYMBOL {ViewConsPal, conspal}
    SUBGI DoConsPal
    VAR-UI sig: Sig
  Behaviour
    CONTEXT not(EMPTYDIC())
    INIT sig = ""
    EVSEQ DoConsPal(sig)
    TRANS OK:
  SubGI
    DefGI DoConsPal: Sig
      Declarations
        TYPE VALSYNTH
        SYMBOL {DoConsPal}
        ARGS pal: Pal
      Behaviour
        CONTEXT not(EMPTYDIC())
        INIT pal = ""
        EVSEQ input(pal)
        TRANS input(pal): EXISTPAL(pal) => EXCEP out("Erro!")
              OK:
              CANCEL:
        EXEC CONSPAL(pal)
    EndGI
EndGI
```

C Descritores de Apresentação para o Dicionário

```
DefVars GInsPal
  pal: Pal;
  sig: Sig
EndVars
DefVars GRemPal
  pal: Pal;
  sig: Sig
EndVars
DefVars ViewConsPal
  sig: Sig
EndVars
DefVars DoConsPal
  pal: Pal
EndVars
DefSynoms
  Menu: menu;
  GInsPal: inspal;
  GRemPal: rempal;
  ViewConsPal: conspal;
EndSynoms
DefView "Portugues"
  DefDA Menu
    TYPE MENU
    NAME "Principal"
    OPTIONS GInit       "Dicionario Vazio"
            GInsPal     "Inserir Palavra"
            GRemPal     "Remover Palavra"
            ViewConsPal "Consultar"
            Fim         "Sair"
  EndDA
  DefDA Fim
    TYPE DB
    NAME "Fim"
  EndDA
  DefDA GInsPal
    TYPE DB
    NAME "Insercao"
    VARS pal (TYPE: HimSelf
              NAME: "Palavra")
         sig (TYPE :HimSelf
              NAME: "Significado")
    CMDS Cancel (NAME: "Cancelar")
         OK     (NAME: "Terminar")
  EndDA
  DefDA GRemPal
    TYPE DB
```

```
    NAME "Remocao"
    VARS pal (TYPE: HimSelf
              NAME: "Palavra")
         sig (TYPE: HimSelf
              NAME: "Significado")
    CMDS Cancel (NAME: "Cancelar")
         OK     (NAME: "Terminar")
  EndDA
  DefDA ViewConsPal
    TYPE DB
    NAME "Consulta",
    VARS sig (TYPE: HimSelf
              NAME: "Significado")
    GIS DoConsPal(sig) (OPEN: true
                        NAME: "Ler Palavra")
    CMDS OK     (NAME: "Terminar")
  EndDA
  DefDA DoConsPal
    TYPE DB
    NAME "Ler Palavra"
    VARS pal (TYPE: HimSelf
              NAME: "Palavra")
  EndDA
  DefDA GInit
    TYPE DB
    NAME "Vazio"
  EndDA
EndView
DefView "Ingles"
  DefDA Menu
    TYPE MENU
    NAME "Main"
    OPTIONS GInit       "Empty Dictionary"
            GInsPal     "Insert Word"
            GRemPal     "Delete Word"
            ViewConsPal "Search"
            Fim         "Quit"
  EndDA
  DefDA Fim
    TYPE DB
    NAME "End"
  EndDA
  DefDA GInsPal
    TYPE DB
    NAME "Insert"
    VARS pal (TYPE: HimSelf
              NAME: "Word")
         sig (TYPE: HimSelf
```

```
              NAME: "Meaning")
    CMDS Cancel (NAME: "Cancel")
         OK     (NAME: "OK")
  EndDA
  DefDA GRemPal
    TYPE DB
    NAME "Delete"
    VARS pal (TYPE: HimSelf
              NAME: "Word")
         sig (TYPE: HimSelf
              NAME: "Meaning")
    CMDS Cancel (NAME: "Cancel")
         OK     (NAME: "OK")
  EndDA
  DefDA ViewConsPal
    TYPE DB
    NAME "Search",
    VARS sig (TYPE: HimSelf
              NAME: "Meaning")
    GIS DoConsPal(sig) (OPEN: true
                        NAME: "Read Word")
    CMDS OK     (NAME: "OK")
  EndDA
  DefDA DoConsPal
    TYPE DB
    NAME "Read Word"
    VARS pal (TYPE: HimSelf
              NAME: "Word")
  EndDA
  DefDA GInit
    TYPE DB
    NAME "Empty"
  EndDA
EndView
```

D Especificação do Modelo da Aplicação para o Dicionário

```
DefType
  Pal STR;
  Sig STR
EndType
DefOp
EndType
DefOp
  INIT = INIT;
  INSPAL = INSPAL;
  REMPAL = REMPAL;
  CONSPAL = CONSPAL;
  EMPTYDIC = EMPTYDIC;
  EXISTPAL = EXISTPAL
EndOp
```

E Especificação NYAGSL do Controlador de Diálogo para o Dicionário

```
FUNC init((cd)):(cd)
STATE
let(pnd1 = SeqDescr(EvId(NIL, "pal"),
                    EvId(NIL, "sig")
                   ),
    codexec1 = ExecDescr("INSPAL", <"pal", "sig">),
    trans11 = TransDescr(And(Ne("pal", StrExp("")),
                             Not(FunCall("EXISTPAL", "", <"pal">))
                            ),
                         <>,
                         <ExcepDescr(Eq("pal", StrExp("")),
                                     <Out("Palavra nula!")>
                                    ),
                          ExcepDescr(Ne("pal", StrExp("")),
                                     <Out("Erro!")>
                                    )
                         >
                        ),
    gi1 = SYNTH({"GInsPal"}, NIL, {}, {},
                ["pal"->VarUI("Pal"),
                 "sig"->VarUI("Sig")
                ],
                true,
                <Atrib("pal", ""), Atrib("sig", "")>, pnd1,
                [EvId(NIL, "pal") -> trans11,
                 CmdId("$Cancel") -> TransDescr(true, <>, <>),
                 CmdId("$OK") -> TransDescr(true, <>, <>),
                ], codexec1),
    pnd2 = EvId(NIL, "pal"),
    codexec2 = ExecDescr("REMPAL", <"pal">),
    trans21 = TransDescr(FunCall("EXISTPAL", "", <"pal">),
                         <Atrib("sig", FunCall("CONSPAL", "", <"pal">))>,
                         <ExcepDescr(true,
                                     <Out("Erro!")>
                                    )
                         >
                        ),
    gi2 = SYNTH({"GRemPal"}, NIL, {}, {},
                ["pal"->VarUI("Pal"),
                 "sig"->VarUI("Sig")
                ],
                Not(FunCall("EMPTYDIC", "", <>)),
                <Atrib("pal", ""), Atrib("sig", "")>, pnd2,
                [EvId(NIL, "pal") -> trans21,
```

```
             CmdId("$Cancel") -> TransDescr(true, <>, <>),
             CmdId("$OK") -> TransDescr(true, <>, <>),
            ], codexec2),
gi3 = SYNTH({"ViewConsPal"}, NIL, {}, {"DoConsPal"},
            ["sig"->VarUI("Sig")
            ],
            Not(FunCall("EMPTYDIC", "", <>)),
            <Atrib("sig", "")>,
            EvId("DoConsPal", "sig"),
            [EvId(NIL, "pal") -> TransDescr(true, <>, <>),
             CmdId("$OK") -> TransDescr(true, <>, <>),
            ], NIL),
pnd4 = EvId(NIL, "pal"),
codexec4 = ExecDescr("CONSPAL", <"pal">),
gi4 = VALSYNTH({"DoConsPal"}, "Sig", {}, {},
               ["pal"->VarUI("Pal")
               ],
               true,
              <Atrib("pal", "")>, pnd2,
              [EvId(NIL, "pal") -> trans21], codexec4),
pnd5 = OpcDescr(EvId("Fim", NIL),
                RepDescr(OpcDescr(EvId("GRemPal", NIL),
                                  OpcDescr(EvId("GInsPal", NIL),
                                           OpcDescr(EvId("GInit", NIL),
                                                    EvId("ViewConsPal", NIL)
                                                   )
                                          )
                                 )
                        )
               ),
gi5 = DECISION({"Menu"}, NIL, {}, {"Fim"},
               [],
               true,
               <>, pnd5,
               [], NIL),
gi6 = DECISION({"Fim"}, NIL, {}, {"Fim"},
               [],
               true,
               <>, NIL,
               [], NIL),
gi7 = SYNTH({"GInit"}, NIL, {}, {"Fim"},
            [],
            true,
            <>, NIL,
            [], ExecDescr("INIT", <>)),    c1 = channel("lxst"),
c2 = channel("stsm"),
c3 = channel("stlx"),
c4 = channel("smst")
```

```
   )
in cd <- CD(["GInsPal" -> gi1,
             "GRemPal" -> gi2,
             "ViewConsPal" -> gi3,
             "DoConsPal" -> gi4,
             "Menu" -> gi5,
             "Fim" -> gi6,
             "GInit" -> gi7
            ], [], c2, c4, c1, c3);
```

F Especificação NYAGSL do Modelo da Apresentação para o Dicionário

```
FUNC init((ma)):(ma)
STATE
let(global = ["Menu"        -> GlobaInfo([], {"menu"}),
              "GInsPal"     -> GlobalInfo(["pal" -> "Pal", "sig" -> "Sig"],
                                          {"inspal"}
                                         ),
              "GRemPal"     -> GlobalInfo(["pal" -> "Pal", "sig" -> "Sig"],
                                          {"rempal"}
                                         ),
              "ViewConsPal" -> GlobalInfo(["sig" -> "Sig"],
                                          {"conspal"}
                                         ),
              "DoConsPal"   -> GlobalInfo(["pal" -> "Pal"], {})
             ],
    gis1 = ["Menu"   -> MENU("Principal",
                             <Option(EvId("GInit", NIL), "Dicionario Vazio"),
                              Option(EvId("GInsPal", NIL), "Inserir Palavra"),
                              Option(EvId("GRemPal", NIL), "Remover Palavra"),
                              Option(EvId("ViewConsPal", NIL), "Consultar"),
                              Option(EvId("Fim", NIL), "Sair")
                             >
                            ),
            "Fim"     -> DB("Fim",
                            [],
                            [],
                            []
                           ),
            "GInsPal" -> DB("Insercao",
                            ["pal" -> HimSelf(LxDef("", "Palavra", "")),
                             "sig" -> HimSelf(LxDef("", "Significado", ""))
                            ],
                            [],
                            [CmdId("$Cancel") -> LxDef("", "Cancelar", ""),
                             CmdId("$OK")     -> LxDef("", "Terminar", "")
                            ]
                           ),
            "GRemPal" -> DB("Remocao",
                            ["pal" -> HimSelf(LxDef("", "Palavra", "")),
                             "sig" -> HimSelf(LxDef("", "Significado", ""))
                            ],
                            [],
                            [CmdId("$Cancel") -> LxDef("", "Cancelar", ""),
                             CmdId("$OK")     -> LxDef("", "Terminar", "")
                            ]
```

```
                         ),
          "ViewConsPal" -> DB("Consulta",
                          ["sig" -> HimSelf(LxDef("", "Significado", ""))
                          ],
                          [EvId("DoConsPal", "sig") ->
                                  DBGIView(LxDef("", "Ler Palavra", ""),
                                           true
                                          )
                          ],
                          [CmdId("$OK")     -> LxDef("", "Terminar", "")
                          ]
                         ),
          "DoConsPal" -> DB("Ler Palavra",
                          ["pal" -> HimSelf(LxDef("", "Palavra", ""))],
                          [],
                          []
                         ),
          "GInit"     -> DB("Vazio",
                          [],
                          [],
                          []
                         )
         ],
   gis2 = ["Menu"   -> MENU("Main",
                            <Option(EvId("GInit", NIL), "Empty Dictionary"),
                             Option(EvId("GInsPal", NIL), "Insert Word"),
                             Option(EvId("GRemPal", NIL), "Delete Word"),
                             Option(EvId("ViewConsPal", NIL), "Search"),
                             Option(EvId("Fim", NIL), "Quit")
                            >
                           ),
          "Fim"     -> DB("Fim",
                          [],
                          [],
                          []
                         ),
          "GInsPal" -> DB("Insert",
                          ["pal" -> HimSelf(LxDef("", "Word", "")),
                           "sig" -> HimSelf(LxDef("", "Meaning", ""))
                          ],
                          [],
                          [CmdId("$Cancel") -> LxDef("", "Cancel", ""),
                           CmdId("$OK")     -> LxDef("", "OK", "")
                          ]
                         ),
          "GRemPal" -> DB("Delete",
                          ["pal" -> HimSelf(LxDef("", "Word", "")),
                           "sig" -> HimSelf(LxDef("", "Meaning", ""))
```

```
                          ],
                          [],
                          [CmdId("$Cancel") -> LxDef("", "Cancel", ""),
                           CmdId("$OK")     -> LxDef("", "OK", "")
                          ]
                         ),
          "ViewConsPal" -> DB("Search",
                          ["sig" -> HimSelf(LxDef("", "Meaning", ""))
                          ],
                          [EvId("DoConsPal", "sig") ->
                                  DBGIView(LxDef("", "Read Word", ""),
                                           true
                                          )
                          ],
                          [CmdId("$OK")     -> LxDef("", "OK", "")
                          ]
                         ),
          "DoConsPal" -> DB("Read Word",
                          ["pal" -> HimSelf(LxDef("", "Word", ""))],
                          [],
                          []
                         ),
          "GInit"     -> DB("Empty",
                          [],
                          [],
                          []
                         )
         ],
    views = ["Portugues" -> gis1, "Ingles" -> gis2],
    canal = channel("lxst"),
    canal1 = channel("stlx"),
    canal2 = channel("stsm"),
    canal3 = channel("smlx")
   )
in ma <- ModApr(global, views, "Portugues", "Menu", canal1, canal3, canal,
             canal2);
```

G Especificação NYAGSL do Modelo da Aplicação para o Dicionário

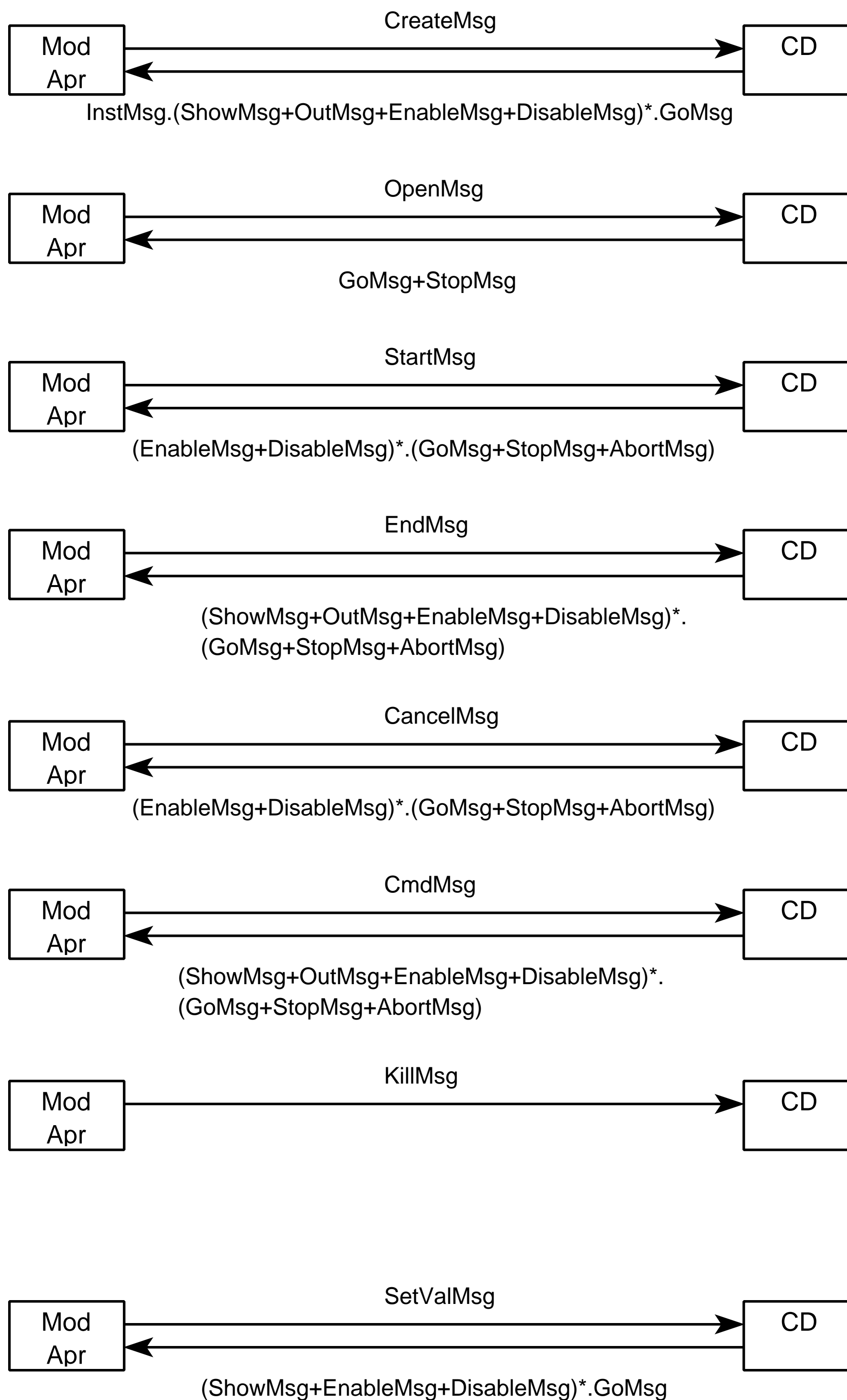
```
FUNC init((ma)):(ma)
STATE
ma<- ModApl(["Pal" -> TypeInfo("STR", NIL),
             "Sig" -> TypeInfo("STR", NIL)
            ],
            [], ["INIT" -> 'INIT,
                 "INSPAL" -> 'INSPAL,
                 "REMPAL" -> 'REMPAL,
                 "CONSPAL" -> 'CONSPAL,
                 "EMPTYDIC" -> 'EMPTYDIC,
                 "EXISTPAL" -> 'EXISTPAL,
                ], channel("stsm"), channel("smst"), channel("smlx"));

#include dic.n
```

H Definições Comuns

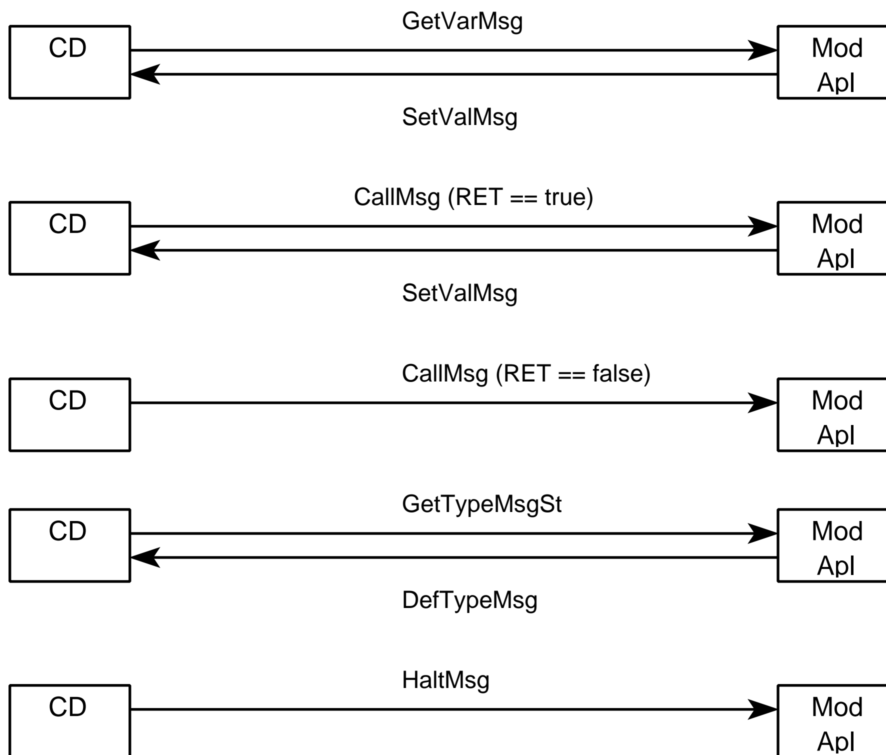
```
; GAMA-X - Share - 1.0 --- jfc 8/93

TYPE
  GISym = STR;
  TypeId = STR;
  TypeDef = TypeId | (TypeDef)-list;
  VarId = STR;
  OpcVarId = NIL | VarId;
  EvId :: GI:  OpcGISym               /* NIL se input */
          VAR: OpcVarId;
  OpcGISym = NIL | GISym;
  InstId :: SYMB: GISym
            INST: Nat0;
  Nat0 = INT;
  OpcInstId = NIL | InstId;
  OpcCmdLineDescr = NIL | CmdLineDescr;
  CmdLineDescr :: OP:   GISym     /* Podia so ser colocado no fim */
                  ARGS: ArgDef-list;
  ArgDef :: NAME:  VarId
            VALUE: OpcValue;
  OpcValue = NIL | Value;
  TransId = EvId | CmdId;
  CmdId :: DREF: STR;
  Value = INT | STR | Bool | Value-set | Value->Value |
          Value-list | Value<->Value;
  Bool = SYM;
ENDTYPE
#includemetoo kill.met
```

I Comunicações

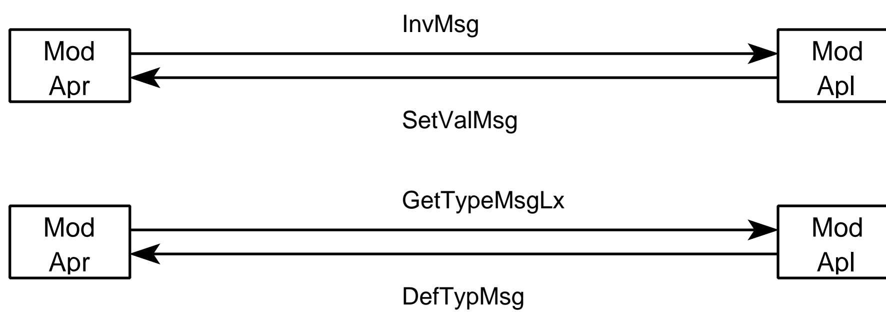
Modelo da Apresentação - Controlador de Diálogo



Controlador de Diálogo - Modelo da Aplicação



Modelo da Apresentação - Modelo da Aplicação



J Mensagens

```
; GAMA-X - Mensagens - 1.0 --- jfc 8/93
TYPE
; CD -> ModApr
  MesgStLx = InstMsg | OutMsg | EnableMsg | DisableMsg | ShowMsg |
             GoMsg | StopMsg | AbortMsg;
  InstMsg :: SOURCE: InstId;
  OutMsg :: SOURCE: InstId
            TEXT: STR;
  EnableMsg :: SOURCE: InstId
               ID:     TransId;
  DisableMsg :: SOURCE: InstId
                ID:     TransId;
  ShowMsg :: SOURCE: InstId
             ID:     VarId
             VAL:    Value;
  GoMsg :: SOURCE:  InstId
           CMDLINE: OpcCmdLineDescr;
  StopMsg :: SOURCE: InstId
             VAL:    OpcValue;
  AbortMsg :: SOURCE: InstId;

; ModApr -> CD
  MesgLxSt = CreateMsg | OpenMsg | StartMsg | EndMsg | CancelMsg | CmdMsg |
             KillMsg |
             SetValMsg;
  CreateMsg :: DEST: OpcInstId
               GI:   GISym;
  OpenMsg :: DEST: InstId;
  StartMsg :: DEST: InstId
              ID:   EvId;
  EndMsg :: DEST: InstId
            ID:   EvId
            VAL:  Value;
  CancelMsg :: DEST: InstId
               ID:   EvId;
  CmdMsg :: DEST: InstId
            ID:   CmdId;
  KillMsg :: DEST: InstId;
  SetValMsg :: DEST: InstId
               VAL:  Value;

; CD -> ModApl
  MesgStSm = GetVarMsg | CallMsg | GetTypMsgSt | HaltMsg;
  GetVarMsg :: SOURCE: InstId
               ID:     VarId;
  CallMsg :: SOURCE: InstId
```

```
             OPR:    STR
             ARGS:   Value-list
             RET:    Bool;
  GetTypMsgSt :: SOURCE: InstId
                 TYP:    TypeId;
  HaltMsg :: STATUS: INT;

; ModApl -> CD
  MesgSmSt :: SetValMsg | DefTypeMsg;
  DefTypMsg :: SOURCE: InstId
               DEF:    TypeDef;

; ModApr -> ModApl
  MesgLxSm = InvMsg | GetTypMsgLx;
  InvMsg :: SOURCE: InstId
            TYP:    TypeId
            VAL:    Value;
  GetTypMsgLx :: SOURCE: InstId
                 TYP:    TypeId;

; ModApl -> ModApr
  MesgSmLx = SetValMsg | DefTypMsg;

ENDTYPE
```

K Controlador de Diálogo

```
; GAMA-X - Controlador de Dialogo - 1.0 --- jfc 8/93
#include share.n
#include cdcom.n
#include petrinet.n
#include si.n
#include cdff.n
#include cdset.n
#include cdlist.n
#include cdrel.n
#include cdtup.n
TYPE
  CD :: DEFS:  GISym -> GIdef
        INSTS: InstId -> InstDescr
        OUTSM: CHAN
        INSM:  CHAN
        INLX:  CHAN
        OUTLX: CHAN;
  GIdef = DECISION | SYNTH | VALSYNTH | FFSYNTH | SETSYNTH |
          LISTSYNTH | RELSYNTH | TUPSYNTH;
  DECISION :: GIDescr;
  SYNTH :: GIDescr;
  VALSYNTH :: GIDescr;
  GIDescr :: SYNOMS:  GISym-set
             TYP:     OpcType
             EXTERN:  GISym-set
             SUBGI:   GISym-set
             VARS:    VarId -> VarDecl
             CONTEXT: BoolExp
             INIT:    Code
             EVSEQ:   CDPetriNet
             TRANS:   TransId -> TransDescr
             EXEC:    OpcExecDescr;
  FFSYNTH :: DREF: TypeId;
  SETSYNTH :: DREF: TypeId;
  LISTSYNTH :: DREF: TypeId;
  RELSYNTH :: DREF: TypeId;
  TUPSYNTH :: DREF: TypeId;
  CDPetriNet = PetriNet | PNDescr;
  OpcType = NIL | TypeId;
  VarDecl = VarUI | VarAPL | VarAPLcopy | VarCTRL;
  VarUI :: DREF: TypeId;              /* Tb. ARGS  */
  VarAPL :: TYP:    TypeId
            VAL:    Value;
  VarAPLcopy :: TYP:    TypeId
                APLVAR: OpcVarId;
  VarCTRL :: DREF: TypeId;
```

```
  TransDescr :: COND:   BoolExp
                ACTION: Code
                EXCEP:  ExcepDescr-list;
  ExcepDescr :: COND:   BoolExp
                ACTION: Code;
  OpcExecDescr = NIL | ExecDescr;
  ExecDescr :: OP:   STR
               ARGS: VarId-list;
  InstDescr :: FATHER:  OpcInstId
               VARS:    VarId -> Value
               EVSEQ:   Conditions
               ENABLED: TransId-set
               CMDLINE: OpcCmdLineDescr;
  Mesg = MesgLxSt | MesgSmSt;
ENDTYPE

; O P E R A C O E S
; Estado: cd: CD

; Master loop -----------------------------------------------------------------
FUNC master((cd)): (cd)
RETURN
let(msg = getlx(),                    /* ler mensagem    */
    sfx = domesg(msg)                 /* tratar mensagem */
   )
in if INSTS(cd) != [] then master()
   else progn(sendsm(HaltMsg(0)),
              quit()                  /* terminou        */
             );

; Tratar as Mensagens ---------------------------------------------------------
FUNC domesg(msg: Mesg, (cd)): (cd)
RETURN
if (is-SetValMsg(msg) -> putvalue(VAL(msg), DEST(msg)),
    is-CreateMsg(msg) -> creategi(msg),
    is-OpenMsg(msg)   -> opengi(msg),
    is-KillMsg(msg)   -> killgi(DEST(msg))
   )
otherwise dotransmsg(msg);

; Colocar valor em Instancias de Modelos --------------------------------------
FUNC putvalue(val: Value, instid: InstId, (cd)): (cd)
RETURN
let(defn=DEFS(cd)[SYMB(instid)]
   )
in if(is-FFSYNTH(defn) -> putff(val, instid),
      is-SETSYNTH(defn) -> putset(val, instid),
      is-LISTSYNTH(defn) -> putlist(val, instid),
```

```
      is-RELSYNTH(defn) -> putrel(val, instid),
      is-TUPSYNTH(defn) -> puttup(val, instid)
     );

; Criar Instancias ---------------------------------------------------------------
FUNC creategi(msg: Mesg, (cd)): (cd)
RETURN
let(def=DEFS(cd)[GI(msg)]
   )
in if(type(def) in {'DECISION, 'SYNTH, 'VALSYNTH} -> newgi(GI(msg), DEST(msg)),
      is-FFSYNTH(def) -> newff(GI(msg), DEST(msg)),
      is-SETSYNTH(def) -> newset(GI(msg), DEST(msg)),
      is-LISTSYNTH(def) -> newlist(GI(msg), DEST(msg)),
      is-RELSYNTH(def) -> newrel(GI(msg), DEST(msg)),
      is-TUPSYNTH(def) -> newtup(GI(msg), DEST(msg))
     );

; Criar uma Instancia (nao de modelos)
FUNC newgi(sym: GISym, father: InstId, (cd)): (cd)
STATE
cd <- let(gi = DEFS(cd)[sym],
          evseq = EVSEQ(gi)
         )
      in if is-PetriNet(evseq) then
          cd                             /* PN ja gerada */
         else                            /* gerar PN     */
          let(cmds = {id | id<-dom(TRANS(gi)): is-CmdId(id)},
              newevseq = makepn(evseq, cmds),
              newgi = if(is-DECISION(gi) ->
                           DECISION(SYNOMS(gi), TYP(gi), EXTERN(gi),
                                    SUBGI(gi), VARS(gi), CONTEXT(gi),
                                    INIT(gi), newevseq, TRANS(gi), EXEC(gi)
                                   ),
                        is-SYNTH(gi)    ->
                           SYNTH(SYNOMS(gi), TYP(gi), EXTERN(gi), SUBGI(gi),
                                 VARS(gi), CONTEXT(gi), INIT(gi), newevseq,
                                 TRANS(gi), EXEC(gi)
                                ),
                        is-VALSYNTH(gi) ->
                           VALSYNTH(SYNOMS(gi), TYP(gi), EXTERN(gi),
                                    SUBGI(gi), VARS(gi), CONTEXT(gi),
                                    INIT(gi), newevseq, TRANS(gi), EXEC(gi)
                                   )
                       )
             )
          in CD(DEFS(cd) + [sym -> newgi], INSTS(cd), OUTSM(cd), INSM(cd),
                INLX(cd), OUTLX(cd)
               )
```

```
RETURN
let(instid = InstId(sym, newninst(1, sym)),      /* novo identificador  */
    inst = createinst(sym, instid, father),      /* criar instancia     */
    defn = DEFS(cd)[sym],
    sidefx1 = sendlx(InstMsg(instid)),           /* enviar ident ao lx  */
    sidefx2 = docode(INIT(defn), instid),        /* executar INIT       */
    sidefx3 = showall(VARS(defn), instid),       /* enviar vars ao lx   */
    enabled = {DREF(id)| id<-valideventspn(EVSEQ(defn), EVSEQ(inst), instid)},
    sidefx4 = setenable(enabled, instid),        /* eventos permitidos  */
    sidefx5 = setdisable(dom(TRANS(defn))-enabled, instid),
    sidefx6 = putenabled(enabled, instid)        /* registar permitidos */
   )
in sendlx(GoMsg(instid, CMDLINE(inst)));         /* fim                 */

; Calcular numero de instancia
FUNC newninst(ninst: INT, sym: GISym, (cd)): INT
RETURN
if ninst in {INST(id)| id<-dom(INSTS(cd)): SYMB(id)==sym} then
  newninst(add(ninst, 1), sym)
else
  ninst;

; Criar e registar instancia
FUNC createinst(sym: GISym, instid: InstId, father: EvId, (cd)): InstDescr (cd)
STATE
cd <- let(defn = DEFS(cd)[sym],
          vars = [id->defval(VARS(defn)[id], id, instid) |
                  id<-dom(VARS(defn)) : ~is-VarAPL(VARS(defn)[id])
                 ],
          evseq = startpn(EVSEQ(defn), instid),
          cmdline = if(EXEC(defn)==NIL) then
                      NIL
                    else
                      CmdLineDescr(OP(EXEC(defn)),
                                   <ArgDef(id, NIL)| id<-ARGS(EXEC(defn))>
                                  ),
          inst = InstDescr(father, vars, evseq, {}, cmdline)
         )
      in CD(DEFS(cd), INSTS(cd)+[instid -> inst], OUTSM(cd), INSM(cd),
            INLX(cd), OUTLX(cd)
           )
RETURN
INSTS(cd)[instid];

; Valores por defeito Built in
FUNC defval(v: VarDecl, id: VarId, instid: InstId, (cd)): Value
RETURN
if is-VarAPLcopy(v) then
```

```
  VAL(sendgetsm(GetVarMsg(instid, APLVAR(v))))
else
  if (DREF(v) == "INT" -> 0,
      DREF(v) == "STR" -> "",
      DREF(v) == "Bool" -> true
     )
  otherwise
    NIL;

; Mostrar variaveis
FUNC showall(vars: VarId, instid: InstId, (cd)):
RETURN
if vars == [] then NIL
else let(id = choice(dom(vars)),
         sidefx = if is-VarUI(vars[id]) then
                    sendlx(ShowMsg(instid, id, VARS(INSTS(cd)[instid])[id]))
        )
     in showall(vars\{id}, instid);

; Enviar permitidos
FUNC setenable(trs: TransId-set, instid: InstId, (cd)):
RETURN
if trs == {} then NIL
else let(id = choice(trs),
         sidefx = sendlx(EnableMsg(instid, id))
        )
     in setenable(trs-{id}, instid);

; Enviar nao permitidos
FUNC setdisable(trs: TransId-set, instid: InstId, (cd)):
RETURN
if trs == {} then NIL
else let(id = choice(trs),
         sidefx = sendlx(DisableMsg(instid, id))
        )
     in setdisable(trs-{id}, instid);

; Registar eventos permitidos
FUNC putenabled(en: Transid-set, instid: InstId, (cd)): (cd)
STATE
cd <- let(inst = INSTS(cd)[instid],
          newinst = InstDescr(FATHER(inst), VARS(inst), EVSEQ(inst),
                              en, CMDLINE(inst)
                             )
         )
      in CD(DEFS(cd), INSTS(cd) + [instid -> newinst], OUTSM(cd), INSM(cd),
                 INLX(cd), OUTLX(cd)
           );
```

```
; Abrir uma Instancia ---------------------------------------------------------
FUNC opengi(msg: OpenMsg, (cd)): (cd)
RETURN
let(sym = SYMB(DEST(msg)),
    instid = DEST(msg),
    inst = INSTS(cd)[instid],
    defn = DEFS(cd)[sym]
   )
in if type(defn) in {'DECISON, 'SYNTH, 'VALSYNTH} then
     if endpn(EVSEQ(defn), EVSEQ(inst)) then
       dostop(instid, true)           /* Marcacao Final Atingida */
     else
       sendlx(GoMsg(instid, CMDLINE(inst)))
   else
     sendlx(GoMsg(instid, NIL));

; Terminar Instancia
FUNC dostop(instid: InstId, ok: Bool, (cd)):
RETURN
if ok then
  let(inst = INSTS(cd)[instid],
      defn = DEFS(cd)[SYMB(instid)]
     )
  in if type(defn) in {'DECISION, 'SYNTH, 'VALSYNTH} then
       if CMDLINE(inst) != NIL then                      /* se tem EXEC */
         let(defn = DEFS(cd)[SYMB(instid)],
             args = <VALUE(ar)| ar<-ARGS(CMDLINE(inst))>,
             res = if TYP(defn) != NIL then
                     VAL(sendgetsm(CallMsg(instid, OP(CMDLINE(inst)),
                                       args, true
                                      )
                               )
                     )
                   else
                     let(sidefx = sendsm(CallMsg(instid, OP(CMDLINE(inst)),
                                                 args, false
                                                )
                                        )
                     )
                     in NIL
            )
         in sendlx(StopMsg(instid, res))
       else
         sendlx(StopMsg(instid, NIL))
     else
       let(res = collectvalue(defn, inst)
          )
```

```
       in sendlx(StopMsg(instid, res))
else
  sendlx(AbortMsg(instid))
STATE resetgi(instid);    /* para futuras utilizacoes da instancia! */

; Calcular valor de modelos
FUNC collectvalue(defn: GIDef, inst: InstDescr, (cd)): Value
RETURN
if(is-FFSYNTH(defn) -> collectff(inst),
   is-SETSYNTH(defn) -> collectset(inst),
   is-LISTSYNTH(defn) -> collectlist(inst),
   is-RELSYNTH(defn) -> collectrel(inst),
   is-TUPSYNTH(defn) -> collecttup(inst)
  );

; Fazer reset de uma Instancia
FUNC resetgi(instid: InstId, (cd)): (cd)
RETURN
let(oldinst = INSTS(cd)[instid],
    sym = SYMB(instid),
    defn = DEFS(cd)[sym]
   )
in if type(defn) in {'SYNTH, 'VALSYNTH} then
     let(inst = createinst(sym, instid, FATHER(oldinst)),
         sidefx2 = docode(INIT(defn), instid),
         sidefx3 = showall(VARS(defn), instid),
         enabled = {DREF(id)| id<-valideventspn(EVSEQ(defn), EVSEQ(inst), instid)},
         sidefx4 = setenable(enabled, instid),
         sidefx5 = setdisable(dom(TRANS(defn))-enabled, instid),
         sidefx6 = putenabled(enabled, instid)
        )
     in sendlx(GoMsg(instid, CMDLINE(inst)))
    else
      sendlx(GoMsg(instid, NIL));

; Eliminar uma instancia ------------------------------------------------
FUNC killgi(instid: InstId, (cd)): (cd)
STATE
cd <- CD(DEFS(cd), INSTS(cd)\{instid}, OUTSM(cd), INSM(cd), INLX(cd),
         OUTLX(cd)
        );

; Mensagem de Transicao -------------------------------------------------
FUNC dotransmsg(msg: MesgLxSt, (cd)): (cd)
RETURN
let(transid = ID(msg),
    dest = DEST(msg),
    defn = DEFS(cd)[SYMB(dest)],
```

```
    inst = INSTS(cd)[dest]
   )
in if type(defn) in {'DECISION, 'SYNTH, 'VALSYNTH} then
     let(oldenab = ENABLED(inst),              /* Permissoes anteriores */
         newinst = if (is-StartMsg(msg)  -> startgi(transid, dest),
                       is-EndMsg(msg)    -> doend(transid, VAL(msg), dest),
                       is-CancelMsg(msg) -> docancel(transid, dest),
                       is-CmdMsg(msg)    -> startcmd(transid, dest)
                      ),
         newenab = {DREF(id)| id<-valideventspn(EVSEQ(defn),
                                                EVSEQ(newinst), dest
                                               )
                   },
         sidefx1 = setenable(newenab-oldenab, dest),
         sidefx2 = setdisable(oldenab-newenab, dest),
         sidefx3 = putenabled(newenab, dest)
        )
     in if endpn(EVSEQ(defn), EVSEQ(newinst)) then
          dostop(dest, endok(msg))
        else
          sendlx(GoMsg(dest, CMDLINE(newinst)))
   else                                        /* Para modelos          */
     let(end = if(is-FFSYNTH(defn) -> domsgff(msg),
                  is-SETSYNTH(defn) -> domsgset(msg),
                  is-LISTSYNTH(defn) -> domsglist(msg),
                  is-RELSYNTH(defn) -> domsgrel(msg),
                  is-TUPSYNTH(defn) -> domsgtup(msg)
                 )
        )
     in if end then
          dostop(dest, endok(msg))
        else
          sendlx(GoMsg(dest, NIL));

; Terminou sem ser com Cancel?
FUNC endok(msg: MsgLxSt): Bool
RETURNS
if (is-StartMsg(msg)  -> true,
    is-EndMsg(msg)    -> true,
    is-CancelMsg(msg) -> true,
    is-CmdMsg(msg)    -> if (ID(msg)==CmdId("$Cancel") -> false
                            )
                         otherwise true
   );

; Mensagem Start ---------------------------------------------------------------
FUNC startgi(id: EvId, instid:InstId, (cd)): InstDescr (cd)
STATE
```

```
cd <- let (defn = DEFS(cd)[SYMB(instid)],
           inst = INSTS(cd)[instid],
           newpn = firepn(EVSEQ(defn), EVSEQ(inst), StartEvent(id), instid),
           newinst = InstDescr(FATHER(inst), VARS(inst), newpn, ENABLED(inst),
                               CMDLINE(inst)
                              )
          )
       in
       CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd), INSM(cd),
          INLX(cd), OUTLX(cd)
         )
RETURN
INSTS(cd)[instid];

; Mensagem End -------------------------------------------------------------
FUNC doend(id: EvId, v:Value, instid: InstId, (cd)): InstDescr (cd)
STATE
cd <- let (sidefx1= if VAR(id) != NIL then alteravar(VAR(id), v, instid),
           defn = DEFS(cd)[SYMB(instid)]
          )
      in if id in dom(TRANS(defn)) then            /* se em TRANS */
           if dotypedexp(COND(TRANS(defn)[id]), instid, NIL) then
             let (sidefx2 = docode(ACTION(TRANS(defn)[id]), instid),
                  inst = INSTS(cd)[instid],
                  newpn = firepn(EVSEQ(defn), EVSEQ(inst), EndEvent(id),
                                 instid
                                ),
                  cmdline = if VAR(id) != NIL then
                              chngcmdln(VAR(id), v, instid)
                            else
                              CMDLINE(inst),
                  newinst = InstDescr(FATHER(inst), VARS(inst), newpn,
                                      ENABLED(inst), cmdline
                                     )
                 )
             in
             CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd),
                INSM(cd), INLX(cd), OUTLX(cd)
               )
           else           /* se nao verifica condicao de TRANS faz Cancel */
             let (sidefx2 = doexcep(EXCEP(TRANS(defn)[id]), instid),
                  inst = INSTS(cd)[instid],
                  newpn = firepn(EVSEQ(defn), EVSEQ(inst), CancelEvent(id),
                                 instid
                                ),
                  cmdline = if VAR(id) != NIL then
                              chngcmdln(VAR(id), v, instid)
                            else
```

```
                              CMDLINE(inst),
                  newinst = InstDescr(FATHER(inst), VARS(inst), newpn,
                                      ENABLED(inst), cmdline
                                     )
                 )
              in
              CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd),
                 INSM(cd), INLX(cd), OUTLX(cd)
                )
          else                        /* nao esta em TRANS */
            let (inst = INSTS(cd)[instid],
                 newpn = firepn(EVSEQ(defn), EVSEQ(inst), EndEvent(id), instid),
                 cmdline = if VAR(id) != NIL then
                             chngcmdln(VAR(id), v, instid)
                           else
                             CMDLINE(inst),
                 newinst = InstDescr(FATHER(inst), VARS(inst), newpn,
                                     ENABLED(inst), cmdline
                                    )
                )
            in CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd),
                  INSM(cd), INLX(cd), OUTLX(cd)
                 )
RETURN
INSTS(cd)[instid];

; Actualiza variavel
FUNC alteravar(id: VarId, v:Value, instid: InstId, (cd)): (cd)
STATE
cd <- let (inst = INSTS(cd)[instid],
           newvars = VARS(inst)+[id -> v],
           newinst = InstDescr(FATHER(inst), newvars, EVSEQ(inst),
                               ENABLED(inst), CMDLINE(inst)
                              )
          )
       in CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd), INSM(cd),
             INLX(cd), OUTLX(cd)
            );

; actualiza CMDLN
FUNC chngcmdln(id: VarId, v:Value, instid: InstId, (cd)): PhrDescr
RETURN
if CMDLINE(INSTS(cd)[instid]) != NIL then
  let(oldcmd = CMDLINE(INSTS(cd)[instid]),
      newparams = putparam(ARGS(oldcmd), id, v)
     )
  in CmdLineDescr(OP(oldcmd), newparams)
else
```

```
  NIL;

; Coloca parametro
FUNC putparam(params: ArgDef-list, id: VarId, v: Value): ArgDef-list
RETURN
if params == <> then <>
else
  let(h = head(params),
      t = tail(params)
     )
  in if NAME(h) == id then <ArgDef(id, v):t>
     else <h:putparam(t, id, v)>;

; Tratamento de Excepcoes
FUNC doexcep(trl: ExcepDescr-list, instid: InstId, (cd)): (cd)
RETURN
if trl == <> then NIL
else let(tr = head(trl)
        )
     in if dotypedexp(COND(tr), instid, NIL) then
          docode(ACTION(tr), instid)
        else
          doexcep(tl(trl), instid);

; Mensagem Cancel ----------------------------------------------------------------
FUNC docancel(id: CmdId, instid: InstId, (cd)): InstDescr (cd)
STATE
cd <- let (defn = DEFS(cd)[SYMB(instid)],
           inst = INSTS(cd)[instid],
           newpn = firepn(EVSEQ(defn), EVSEQ(inst), CancelEvent(id), instid),
           newinst = InstDescr(FATHER(inst), VARS(inst), newpn,
                               ENABLED(inst), CMDLINE(inst)
                              )
          )
       in
       CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd), INSM(cd),
          INLX(cd), OUTLX(cd)
         )
RETURN
INSTS(cd)[instid];

; Mensagem Cmd --------------------------------------------------------------------
FUNC startcmd(id: CmdId, instid: InstId, (cd)): InstDescr (cd)
STATE
cd <- let (defn = DEFS(cd)[SYMB(instid)]
          )
      in if dotypedexp(COND(TRANS(defn)[id]), instid, NIL) then
           let (sidefx = docode(ACTION(TRANS(defn)[id]), instid),
```

```
                inst = INSTS(cd)[instid],
                newpn = firepn(EVSEQ(defn), EVSEQ(inst), CmdEvent(id), instid),
                newinst = InstDescr(FATHER(inst), VARS(inst), newpn,
                                    ENABLED(inst), CMDLINE(inst)
                                   )
               )
            in
            CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd), INSM(cd),
               INLX(cd), OUTLX(cd)
              )
         else                                   /* se nao verifica condicao */
           let (sidefx = doexcep(EXCEP(TRANS(defn)[id]))
               )
           in cd
RETURN
INSTS(cd)[instid];

; Validar o Contexto de um GI e inicializar Variaveis APL (Petri Nets) --------
FUNC validcontext(sym: GISym, father: InstId, (cd)): Bool (cd)
STATE
cd <- if sym != NIL then
        let(defn = DEFS(cd)[sym],
            vars = [id -> getvarapl(defn, id) | id<-dom(VARS(defn)) :
                                                is-VarAPL(VARS(defn)[id])
                   ],
            gitype = type(defn),             /* DECISION, SYNTH, VALSYNTH */
            newdef = (eval(gitype))(SYNOMS(defn), TYP(defn), EXTERN(defn),
                                    SUBGI(defn), VARS(defn)+vars,
                                    CONTEXT(defn), INIT(defn), EVSEQ(defn),
                                    TRANS(defn), EXEC(defn)
                                   )
           )
        in CD(DEFS(cd)+[sym -> newdef], INSTS(cd), OUTSM(cd), INSM(cd),
              INLX(cd), OUTLX(cd)
             )
      else
        cd
RETURN
if sym != NIL then
  dotypedexp(CONTEXT(DEFS(cd)[sym]), father, sym)
else
  true;

; Calcular valor de variavel da aplicacao
FUNC getvarapl(defn: GIDescr, id: VarId, (cd)): VarAPL
RETURN
VarAPL(TYP(VARS(defn)[id]), VAL(sendgetsm(GetVarMsg(0, id))));
```

```
; Alterar Valor de Variavel (SI) -----------------------------------------
FUNC putvar(id: VarId, v: Value, instid: InstId, (cd)): (cd)
STATE
cd <- let(inst = INSTS(cd)[instid],
          defn = DEFS(cd)[SYMB(instid)]
         )
      in if type(defn) in {'DECISION, 'SYNTH, 'VALSYNTH} then
           if id in dom(VARS(defn)) then
             let(newvars = VARS(inst) + [id -> v],
                 sidefx = if is-VarUI(VARS(defn)[id]) then
                             sendlx(ShowMsg(instid, id, v)),
                 newinst = InstDescr(FATHER(inst), newvars, EVSEQ(inst),
                                     ENABLED(inst), CMDLINE(inst)
                                    )
                )
             in CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd),
                   INSM(cd),INLX(cd), OUTLX(cd)
                  )
           else
             let(sidefx = putvar(id, v, FATHER(inst))
                )
             in cd
         else
           let(newvars = VARS(inst) + [id -> v],
               sidefx = sendlx(ShowMsg(instid, id, v)),
               newinst = InstDescr(FATHER(inst), newvars, EVSEQ(inst),
                                   ENABLED(inst), CMDLINE(inst)
                                  )
              )
           in CD(DEFS(cd), INSTS(cd)+[instid -> newinst], OUTSM(cd),
                 INSM(cd),INLX(cd), OUTLX(cd)
                );

; ------------------------------------------------------------------------------
```