ARIAS

Universidade Estadual de Campinas
Instituto de Matemática, Estatística e Ciência da Computação
Departamento de Ciência da Computação

# *EDITOR TOPOLÓGICO*

# *PARA A LINGUAGEM DE ESPECIFICAÇÃO DE COMPUTAÇÕES*

# *LEGOSHELL*

**Este exemplar corresponde a redação final da tese devidamente corrigida defendida pelo Sr.Hernán Piñón Arias e aprovada pela Comissão Julgadora.**

Campinas, 11 de janeiro de 1991

Orientador: Prof. Dr. Rogério Drummond

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação
como parte dos requisitos exigidos
para a obtenção do título de Mestre em Ciência da Computação.

*A mi mujer Maria de Fátima.*

*A mis hijos María,Antonio Francisco,*

*João Víctor y Florencia María.*

*A mis Padres.*

# AGRADECIMENTOS

Desejo agradecer a todos aqueles que fizeram possível que o presente trabalho chegasse a bom porto.

Em primeiro lugar a toda minha familia que me apoiou continuamente nesta empreitada.

Desejo agradecer aos membros da banca examinadora Doutor João Antônio Zuffo, do LSI da Poli - USP e Doutor Paulo Lício de Geuss, do DCC da UNICAMP, quem com suas agudas perguntas e observações puseram a prova meus conhecimentos sobre o tema. Ao Professor Doutor Rogério Drummond, orientador e amigo, com quem discutimos, em muitas oportunidades acaloradamente, todos os temas aquí descritos.

Aos meus colegas e amigos do projeto A_HAND em especial Alicia de Sarno, André Vincent, Antônio G. Figueiredo Filho, Carlos Alexandre Polanczyk, Carlos Alberto Furuti, Camem Satie Hara (Carmencita), Cassius Di Cianni, Karsten Hauten e Lídia Yamamoto, que participaram ativamente no projeto e na implementação do Editor de LegoShell, e sem cuja companhia não teria conseguido chegar.

Ao Professor Doutor Hans Kurt Liesenberg e a Maria Helena Pereira Dias, compadres, que tiveram a infinita paciência de corrigir meu "portunhol".

Finalmente à Armada Argentina a quem devo dezenove anos e dois meses de formação, à UNICAMP que me albergou como estudante da Pós durante toda a duração do Mestrado, ao CNPq, CAPES, IBM e OEA que me deram respaldo financeiro através da bolsa que deles recebí.

# SUMÁRIO

O presente trabalho descreve a implementação do editor topológico para as computações de LegoShell. A LegoShell é uma das linguagens do Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados (A_HAND), atualmente sendo desenvolvido no Departamento de Ciência da Computação da UNICAMP.

Ela possibilita a composição de programas através da conexão de suas portas de entrada e saída. Ela estende da noção de "pipe" do UNIX®, restrita a uma dimensão só, para grafos multidimensionais compostos de programas, dispositivos periféricos, arquivos, conectores, etc. Estes grafos, chamados computações, podem ser abstraídos como programas e depois utilizados nas computações indistintamente de programas reais. Qualquer dos componentes de uma computação pode residir ou ser executado em qualquer uma das máquinas da rede local, possibilitando assim, a especificação de computações distribuídas.

Pela sua naturesa, a LegoShell é inerentemente uma linguagem de estrutura topológica. Um editor apropriado deve ser capaz de manipular estruturas topológicas e representá-las de forma gráfica. Este editor será o "front-end" principal do ambiente A_HAND. Por isto ele é o principal responsável pela determinação do "look-and-feel" da interface com o usuário do ambiente.

A interface com o usuário foi implementada usando os widgets do X Windows, uma máquina de "statecharts" foi utilizada para a definição do núcleo semântico do editor e finalmente os objetos complexos manipulados internamente foram implementados seguindo o paradigma de tipos abstratos de dados.

# ABSTRACT

This work describe the implementation of the topologic editor for the LegoShell computations. LegoShell is one of the languages of the Software Development Environment based on Hierarchies of Differentiated Levels of Abstraction (A_HAND), being developed at the Computer Science Department of UNICAMP.

It allows program composition through the connection of program input and output ports. It extends the notion of UNIX pipes, which is restricted to a single dimension, to multidimensional graphs composed of programs, peripheral devices, files, connectors, etc. These graphs, called computations, can be abstracted and used afterwards in other computations without any difference from real programs. Any of the components could run locally or in any of the Local Area Network machines, this way allowing the specification of distributed computations.

Because of its own nature, LegoShell is intrinsically a topologically structured language. A suitable editor has to be able to manipulate topological structures and to represent them in a graphical manner.

This editor will be the principal front-end of A_HAND. This it is the reason why the editor is mainly reponsible for the look-and-feel of the user interface in this environment. The user interface has been implemented using X Windows' widgets, a statechart machine has been used to define the semantic core of the editor and finally the complex objects which are internally manipulated were implemented following the abstract data tipe paradigm.

# CONTEÚDO

# LISTA DE FIGURAS

# Capítulo 1

# INTRODUÇÃO

## 1.1. Definição do Contexto

A literatura técnica recente evidencia uma acentuada evolução dos ambientes de programação e das interfaces com o usuário, como pode ser visto em [Pen88, Con86, Sh87, Hix89, BS84]. Inicialmente os computadores eram programados em linguagem de máquina e a interface com o usuário consistia de resultados impressos ou perfurados. Com o surgimento dos sistemas operacionais que suportavam múltiplos usuários e dos terminais de vídeo, avançou-se neste aspecto em termos de sofisticação. Mas mesmo assim ainda não se cogitava poder representar figuras com realismo nem oferecer uma interface de alta performance, devido ao custo do hardware envolvido e a restrições tecnológicas, que limitavam a velocidade das máquinas.

O barateamento do hardware continua em ritmo acelerado [Do78], e atualmente é possível ter sobre a mesa uma estação de trabalho com desempenho equivalente ao de uma máquina de grande porte ao custo de alguns poucos milhares de dólares. Como resultado disto o mercado de estações de trabalho vem crescendo vertiginosamente nos últimos anos, muito além do resto do mercado de informática, sempre em frenética expansão.

As estações de trabalho, apesar da grande variedade de modelos e marcas, guardam entre si muitas semelhanças: alta performance (mais de 8 MIPS), vasta memória (pelo menos 8 Mbytes) e monitores de alta resolução (1 Mpixels ou mais) monocromáticos ou coloridos. O software oferece serviços básicos de memória virtual, multitarefa e multiusuário e abstrações de mais alto nível como sistemas de janelas e de arquivos distribuídos (X-Windows®, Network File System®), conexão em rede Ethernet e gerenciadores de interface com o usuário (OpenLook® e Motif®). Elas abrangem a faixa de preço entre US$ 5.000 e US$ 100.000 e a cada ano, pelo mesmo preço, duplicam o desempenho oferecido.

O crescimento do uso de estações de trabalho ligadas em rede, com alto nível de padronização, associado a uma explosiva expansão da quantidade de usuários e a uma intensa diversificação do tipo dos mesmos, constitui um panorama que poderíamos qualificar como revolucionário, do ponto de vista de suas implicações futuras na área de computação [Ba90]. Além do mais, aplicações de software antes proibitivas devido a seu tamanho ou complexidade tornam-se tecnicamente viáveis. Nos referimos aqui a aplicações ditas não-convencionais, tais como sistemas de engenharia e de automação industrial. Visto o alto custo de desenvolvimento de tais sistemas, é necessário para se tornarem economicamente factíveis que sua vida útil seja mais longa, através de características como portabilidade, adaptabilidade e "manutenibilidade" [DLi87b]. Portabilidade é a facilidade com que um software pode ser transportado para máquinas diferentes; adaptabilidade é a qualidade do sistema em adequar-se a novos requisitos; e "manutenibilidade" corresponde à facilidade com que eventuais erros possam ser corrigidos e novas caracter'ísticas adicionadas. Novas técnicas de apoio, bem como ferramentas, se fazem necessárias para explorar de forma mais eficiente as facilidades providas pelo hardware.

Vários autores [Sch86, Pot87, Ra82] têm abordado o tema sugerindo que os novos ambientes de programação devam alterar de forma substancial a interação com o usuário e a funcionalidade computacional, para levar em conta este novo vetor na demanda.

Por outro lado, podemos ver que a aceitação ou rejeição sofrida por um novo produto de software que chega ao mercado, está diretamente associada, entre outros motivos, à qualidade da sua interface com o usuário. Sistemas de grande funcionalidade, mas de interface pobre, têm conseguido pouca penetração no mercado, especialmente com o grande público, como foi inicialmente o caso do próprio sistema operacional UNIX®. Uma substancial e crescente parcela dos investimentos em software tem sido alocada no estudo e desenvolvimento da interface com o usuário para tornar os produtos mais "amigáveis", com interface mais padronizada e, portanto, mais fáceis de serem usados por usuários não- especialistas.

A busca de um alto grau de padronização e portabilidade não constitui simplesmente uma política mercadológica circunstancial; ela é uma exigência dos usuários. O governo norte-americano, por exemplo, que é o maior comprador e usuário mundial de software, está investindo grandes quantias na determinação de padrões, particularmente no caso do sistema operacional UNIX. Não obstante isto, há quem critique a padronização extrema da interface com o usuário [Pot89, Le89]

Para garantir a incorporação destas características nos sistemas, é preciso oferecer aos projetistas de software ferramentas mais poderosas e melhor elaboradas, além dos tradicionais editores de texto e processadores de linguagens. Desta forma, começaram a surgir aplicativos como editores orientados a sintaxe, geradores de diagramas, geradores de esqueletos ("templates") de programas e muitos outros. Veio então à tona a idéia de se automatizar o próprio ambiente de desenvolvimento de software (ADS). Uma vez que grande parte do esforço de desenvolvimento de software consiste em tarefas mecânicas como, por exemplo, verificar se todos os módulos dependentes de um módulo que foi alterado foram recompilados, justifica-se a utilização do computador para automatizar seu próprio ambiente. Além disso, o surgimento de estações de trabalho de baixo custo ligadas em rede viabilizou ADS's, permitindo o desenvolvimento distribuído de sistemas, além de fornecer facilidades individuais com interfaces gráficas e amigáveis e bom tempo de resposta.

## 1.2. Apresentação do Trabalho

O objetivo deste trabalho é apresentar um novo paradigma de especificação de computações complexas e o editor que implementa a ferramenta de manipulação destas computações. Assim, pretendemos mostrar a viabilidade das idéias contidas nestes novos conceitos e dispor de um protótipo que permita fazer as correções e ajustes necessários à especificação do ambiente final.

A linguagem para a qual está orientado este editor, a ser detalhada a seguir, é a LegoShell [Dru89], que é suportada pelo A_HAND (Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados) [DLi87a], atualmente em desenvolvimento no Departamento de Ciência da Computação da UNICAMP. Segundo Meyer [Mey87], a especificação e desenvolvimento de software reutilizável é um problema técnico. Isto é, além de facilidades para recuperação dos módulos, as próprias linguagens devem facilitar a fatoração de partes comuns em todos os níveis de abstração do objeto. Como o próprio nome indica, a criação de níveis diferenciados de abstração é uma linha mestra na concepção do A_HAND.

A apresentação do trabalho está disposta na seguinte seqüência: o capítulo 2 descreve as características mais importantes do ADS A_HAND e da linguagem Cm [SLD88, DSi88, Fur90a]. A definição detalhada do ambiente A_HAND foi fruto do trabalho em conjunto com Carmem Satie Hara. No capítulo 4 de [Har90] há uma descrição geral do A_HAND. Nós entraremos em mais detalhe naqueles aspectos do ambiente e da LegoShell relevantes ao presente trabalho. No capítulo 3 descrevemos a linguagem LegoShell, suas potencialidades e particularidades mais importantes, e no capítulo 4 enunciamos as características mais importantes que deve possuir uma interface amigável com o usuário, justificamos a escolha do sistema de

janelas X-Windows e dos Estadogramas, definindo por último a arquitetura do Editor da LegoShell. No capítulo 5 definiremos a representação interna, as funções que permitem manipulá-la e faremos algumas considerações sobre aspectos particulares da implementação do Núcleo Topológico. No capítulo 6 descreveremos em detalhe a semântica do editor, os seus estados, usando "statecharts", e faremos algumas considerações sobre aspectos particulares da implementação do Núcleo de Edição Gráfica e da máquina de "statecharts". No capítulo 7 definiremos em detalhe a interface com o usuário do protótipo e discutiremos alguns aspectos particulares da implementação, e finalmente no capítulo 8 faremos um pequeno histórico de como se desenvolveu o trabalho de pesquisa e relatamos nossas conclusões.

# Capítulo 2

# O AMBIENTE A_HAND

Este capítulo descreve as características mais importantes do ADS A_HAND e da linguagem Cm. A definição detalhada do ambiente A_HAND foi fruto do trabalho em conjunto com Carmem Satie Hara. No capítulo 4 de [Har90] há uma descrição geral do A_HAND. Nós entraremos em mais detalhe naquelas partes do ambiente e da LegoShell relevantes ao presente trabalho.

## 2.1. O Ambiente de Desenvolvimento de Software(ADS)

Se analisarmos quais são as caraterísticas comuns de ambientes para suporte de desenvolvimento de software com aqueles usados no desenvolvimento de hardware (CAD, CAM, CAE), vemos que ambos manipulam objetos complexos normalmente organizados em estruturas hierárquicas. Encontramos este traço característico tanto nos circuitos integrados e objetos manufaturados quanto nos programas, no modelo lógico de uma base de dados, ou no formalismo para especificação por refinamentos sucessivos.

Um "chip" pode ser visualizado como um conjunto de áreas, cada uma delas associada a uma função determinada, subdivididas por sua vez em subáreas, até o nível de transistor, ou como um conjunto de funções, cada uma delas associada a um conjunto de subfunções, e assim por diante até chegar ao nível de porta lógica.

Do mesmo modo, a programação estruturada e as técnicas de engenharia de software ascendentes ou descendentes baseiam-se na idéia de criar hierarquicamente um objeto por meio dos processos de refinamento e de agregação. Para poder apoiar adequadamente este processo de criação é necessário preservar toda a informação contida na estrutura do objeto.

No desenvolvimento de sistemas complexos, que requerem um esforço de dezenas ou centenas de homem/ano, é imperativo dar especial atenção aos aspectos de reusabilidade, portabilidade e "manutenibilidade" dos mesmos. A preocupação com o conjunto destes aspectos é uma das características principais do ADS proposto .

Deseja-se automatizar as funções de administração do ambiente, de forma tal que o próprio usuário não seja obrigado a gerenciar estes objetos complexos [HBe85]. Pretende-se dotar o ambiente de uma interface que permita uma interação agradável. Os recursos gráficos presentes nas estações de trabalho e utilizados na interface possibilitarão a visualização das estruturas hierárquicas dos objetos manipuláveis dentro do ADS.

O que propomos, então, é uma ferramenta que, aproveitando às caraterísticas propiciadas pelas novas gerações de equipamentos, permita ao usuário manipular objetos complexos com naturalidade e de maneira mais transparente. Esta ferramenta está subjacente ao gerenciador de interface com o usuário, como mostramos na Figura 2.1, e que, de acordo com o tipo de objeto que o usuário deseja manipular, adequará o sentido semântico das ações do diálogo, ativando um novo universo de operações possíveis ao objeto em questão.

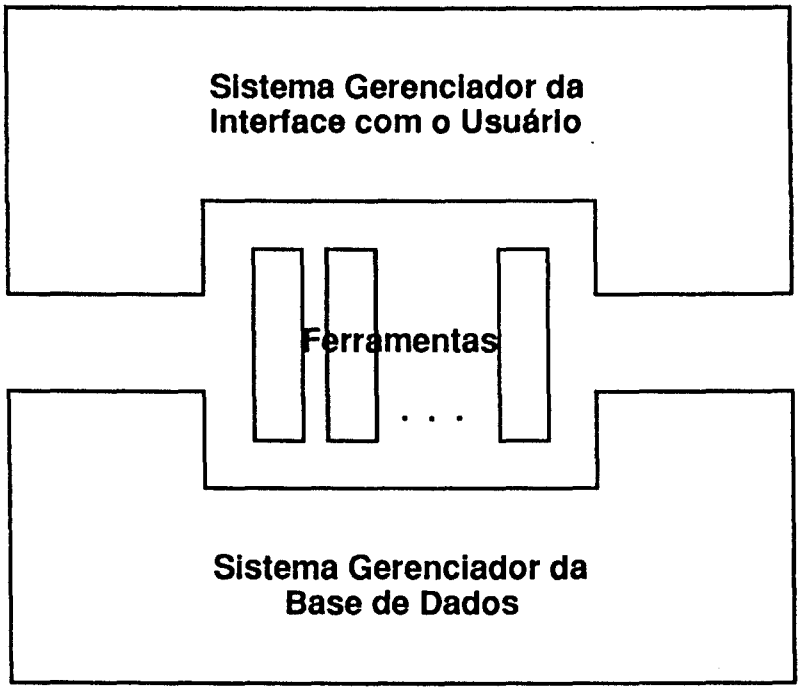



**Figura 2.1** : Arquitetura de um ADS Genérico

Todos os objetos conhecidos no ADS são manipulados por meio de editores específicos, que entendem da estrutura destes objetos. Podemos agrupá-los em duas classes: os objetos textuais e os de estrutura topológica. Para os primeiros usamos editores sensíveis à sintaxe, como o Editor Sensível à Sintaxe do Cm [Osmaira] que está sendo desenvolvido como parte do A_HAND, baseado no Cornell Synthesizer Generator. Para os outros, como é o caso da LegoShell, usaremos um editor topológico.

O editor topológico foi concebido para ser reconfigurável, manipulando em cada caso particular um conjunto determinado de objetos, cada um deles com suas próprias regras de consistência ou restrições, e que, de forma gráfica, permita a visualização dos objetos em seus diferentes níveis de abstração.

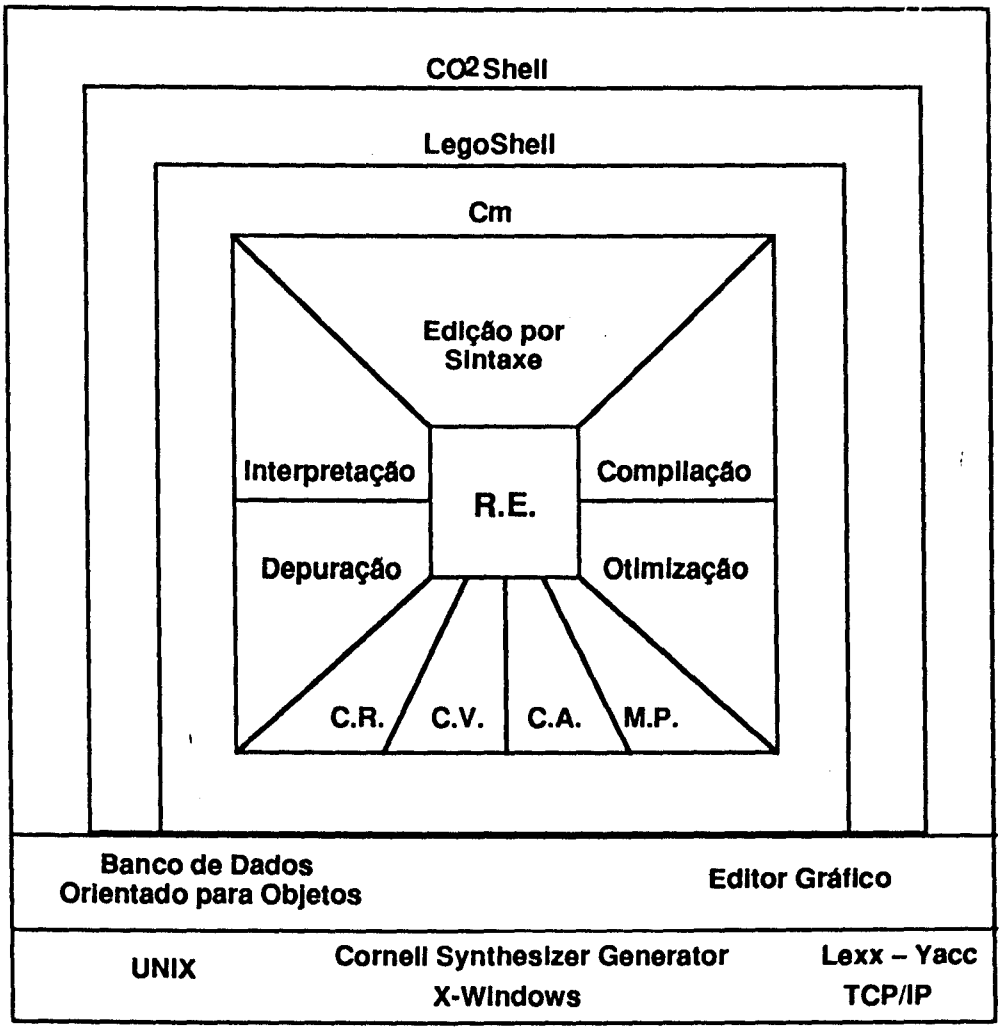
Isto possibilitará ao usuário, no futuro, passar da edição da LegoShell à de circuitos integrados, de um Diagrama de Fluxo de Dados (DFD) ou de um Modelo de Entidades e Relacionamentos (MER). Neste caso o ambiente apenas mudaria externamente os ícones disponíveis para o usuário, fazendo com que a interação se processasse de maneira similar à anterior.

É importante distinguir entre o objeto propriamente dito e sua visualização. Os editores são responsáveis pela visualização dos objetos, e podem nos apresentar um mesmo objeto com várias visualizações diferentes, como no exemplo do "chip" que descrevemos mais acima. Existe também uma representação interna dos objetos, que denominaremos Representação Essencial. Ela constitui a estrutura na qual o ambiente armazena os objetos e através da qual os manipula. A longo prazo pretende-se uma uniformização das Representações Essenciais dos diversos Objetos. Idealmente teríamos uma única Representação Essencial parametrizável para todos os Objetos manipuláveis pelo ADS.

Para alcançar o alto grau de uniformidade proposto é necessário que o ADS conte com a capacidade de integração de ferramentas.

Descreveremos a seguir, com maior detalhe, o ADS proposto.

## 2.2. O Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados (A_HAND).




**Legenda:** C.R. : Catalogação e Recuperação
C.V. : Controle de Versão
C.A. : Controle de Acesso
M.P. : Manutenção da Consistência de Programas Fonte e Compilados
R.E. : Representação Essencial

**Figura 2.2** : Arquitetura Geral do ADS A_HAND

O ADS A_HAND faz parte de um projeto mais abrangente, chamado AIDSH (Ambiente Integrado de Desenvolvimento de Software e Hardware), atualmente em andamento no Departamento de Ciência da Computação da UNICAMP. Uma de suas características principais é a de permitir que os objetos manipulados

no ambiente possam ser compostos de objetos mais simples. Assim, cada objeto representa, na realidade, uma hierarquia de objetos, provendo cada um um nível de abstração. Daí o nome A_HAND.

Como qualquer outro ADS, seu objetivo final é dar suporte à atividade de produção de software, livrando o usuário de processos mecânicos e repetitivos do ciclo de desenvolvimento, além de facilitar o gerenciamento da produção de sistemas grandes e complexos. Idealmente, o ambiente também deve dar suporte ao desenvolvimento distribuído. Pretende-se, dessa forma, aumentar a produtividade e eficiência desta atividade e, ao mesmo tempo, diminuir o seu custo. O que diferencia as diversas propostas de ADS é o direcionamento dos esforços no sentido de atingir estes objetivos. Uma das linhas mestres do A_HAND é a manutenção da homogeneidade da interface nos diversos níveis de abstração. Dessa forma, o usuário não sente dificuldade para adaptar-se às mudanças de nível, bem como seu tempo de aprendizado sobre todo o ambiente é reduzido. Esta interface deve, preferencialmente, ser gráfica e amigável. (Figura 2.2 )

Outro ponto importante, que já mencionamos na Introdução, é a reutilização de software. De acordo com Meyer [Mey87], a viabilização desta técnica depende, principalmente, da existência de facilidades para o desenvolvimento de módulos reutilizáveis, apesar desta abordagem envolver fatores administrativos tais como o gerenciamento e recuperação de uma biblioteca de módulos. Atualmente, os programas são desenvolvidos para solucionar problemas específicos, sem a preocupação de fatorar as semelhanças. Assim, além de continuamente estarmos "reinventando" soluções, seu custo é elevado por não aproveitarmos partes já utilizadas e testadas dentro de outros sistemas. Neste sentido, houve uma evolução nas linguagens de programação, culminando com as linguagens orientadas a objetos, que apresentam mecanismos que facilitam a construção de código reutilizável, tais como:

- **tipos abstratos de dados:** criam uma interface de operações bem definida para cada módulo e encapsulam a sua implementação interna;
- **modularidade:** padrão de unidade de programação para fins de reutilização de código, especificação das interdependências de partes de um programa e da sua administração. No Cm, descrito a seguir, os construtores de tipos abstratos e módulos são unificados num único conceito;
- **generalidade:** é a capacidade do módulo ser definido com parâmetros genéricos que representam tipos, permitindo ao implementador escrever um único módulo para um tipo abstrato de dados, aplicáveis a vários tipos de objetos (por exemplo, um mesmo módulo *Pilha* para pilha de inteiros, reais, cadeia de caracteres, etc.);
- **herança:** permite a estruturação do sistema em classes de objetos com diferentes níveis de abstração, através da herança das propriedades e operações de um objeto de nível mais alto para aqueles que foram especializados a partir do primeiro.

Em suma, softwares reutilizáveis devem ter características semelhantes às de "chips" eletrônicos: funcionalidade encapsulada, interfaces bem definidas e alto grau de qualidade. Reunindo as facilidades citadas, o A_HAND utiliza a linguagem Cm (C modular e polimórfico) [SLD88, Fu90a], uma derivação da linguagem C [KRi78], para seu nível de programação a nível de módulo. O ambiente possui, ainda, duas outras linguagens: LegoShell e $CO^2$ Shell. Se para a linguagem Cm os objetos derivados são programas ou classes Cm, para a LegoShell os objetos resultantes são computações. A LegoShell é uma linguagem gráfica para configurar programas. Ela consiste em estender o conceito unidimensional do "pipe" de UNIX para mais dimensões, através da adição de conectores do tipo "mailbox" e "broadcast" às conexões entre programas, arquivos, dispositivos e computações formando assim novas computações executáveis a nível da LegoShell. A semelhança da criação de computações com jogos de montar para crianças como o Lego® deu origem ao nome da linguagem. A interface padrão dos objetos manipulados na LegoShell são portas de entrada e saída, ligadas por meio dos conectores, que permitem a especificação do fluxo de dados entre os objetos.

O último nível das linguagens, a $CO^2$ Shell, corresponde a uma linguagem de comandos orientada para objetos, cuja especificação ainda não está terminada. Ela seguirá a mesma linha das linguagens de comandos do UNIX

e servirá a dois propósitos: interface entre o usuário e o sistema operacional e linguagem de prototipagem rápida.

Segundo a classificação proposta por Dart em [Dar87], podemos dizer que o A_HAND é um ambiente baseado em uma linguagem, uma vez que suas características mais marcantes são as linguagens que o compõe. Porém, estas linguagens compartilham uma mesma macro-estrutura, sua representação essencial, os objetos serão manipulados então por editores sensíveis à sintaxe, o que caracteriza ambientes baseados na estrutura. O ambiente suporta ainda outras ferramentas para tarefas como compilação, interpretação, depuração, catalogação e recuperação de objetos e gerenciamento de configurações.

É por isto que foi considerado como fundamental para o êxito do Projeto A_HAND, associar a um núcleo conceitualmente poderoso uma interface com o usuário visualmente atrativa e de fácil uso.

## 2.3. Programas Cm

A linguagem Cm - C modular e polimórfico - foi projetada para ser a linguagem básica do ambiente A_HAND. Sua descrição detalhada pode ser encontrada em [SLD88, DSi88, Fur90a]. Ela é derivada da linguagem de programação C, cujos operadores e comandos foram preservados, incrementando-a com:

- **verificação rigorosa de tipos;**
- **modularidade;**
- **suporte à noção de hierarquia de tipos;**
- **tratamento uniforme de tipos.**

Além disso, é introduzida a noção de classes, que constituem a característica básica da linguagem. As classes definidas pelo usuário são construtores de tipos. Uma vez fornecidos os parâmetros reais de uma classe, ela define um tipo específico. Classes sem parâmetros são um caso particular que correspondem, por si só, a um tipo. Em essência, uma classe é um tipo abstrato de dados constituído de:

- parâmetros formais,
- lista de classes importadas (cláusula **import**),
- lista de classes usadas (cláusula **use**),
- lista de classes herdadas (cláusula **inherit**),
- estrutura de dados primitivos,
- portas,
- operadores privados,
- operadores exportáveis.

Os parâmetros podem ser de qualquer tipo, inclusive do tipo **type**, o que torna a classe um construtor de tipos. No exemplo abaixo [SLD88], temos uma única definição da classe Array. Ela pode ser instanciada em Array de inteiros, cadeias de caracteres, ou mesmo de Array, bastando para isso passar o tipo dos elementos como valor do parâmetro Tipo. Outra característica é a possibilidade de se definir valores "default" para os parâmetros, como é o caso de **int** para Tipo. Estes valores são utilizados caso eles não sejam especificados por ocasião da criação de uma instância.

```
class Array (type Tipo = int; int Size)
          Tipo Vet[Size];

export Tipo Index (int n)
          return (Vet[n]);
```

```
export void PutN (int n; Tipo Elem)
            Vet[n] = Elem;
```

A definição da classe `complex` abaixo [Dru90] introduz um novo construtor de tipos de nome `Complex`. Este construtor possibilita a especificação de tipos abstratos de dados `Complex(Z)` para todo tipo `Z` que tenha os operadores de `t` utilizados na implementação da classe `Complex`, neste caso atribuição (=) e adição (`add`).

Os operadores de classe são ativados segundo a sintaxe:

```
objeto.nome_do_operador (parâmetro)
```

O objeto é considerado um parâmetro do operador como por exemplo `a.add(b)`. Um tipo é uma coleção de objetos como constantes, tipos, variáveis e operadores. O tipo `Complex(t)` contém variáveis `real` e `imag` e operadores `complex()` e `add()`. Uma instância é criada na declaração de uma variável ou por alocação dinâmica. A referência a um objeto do tipo é realizada por seleção na forma `instância.objeto` ou `tipo.objeto`. Muitos dos operadores exigem uma instância como seletor; esta instância é um parâmetro implícito do operador. O operador de atribuição é pré-definido para todo tipo e portanto ele não precisa de seleccionador.

A definição funcional do operador `add` é:

$\texttt{add}: (t, t) \rightarrow (t)$

onde o primeiro parâmetro corresponde à instância seletora. A instância corrente pode ser selecionada dentro da classe por **`self`**, um objeto `obj` por **`self`**`.obj` ou simplesmente por `obj`.

```
class complex (type t=float)
            t real, imag;

export mytype Complex(t r; t i)
{
            real = r;
            imag = i;
}

export mytype add(complex b)
{
            mytype local;
            local.real = real.add(b.real);
            local.imag = imagg.add(b.imag);
            return(local);
}
```

As variáveis `x`, `y` e `z` abaixo são de tipos derivados da classe `Complex`.

```
Complex() x;            /* número complexo comum */
Complex(int) y;         /* número complexo integral */
Complex(complex()) z;   /* número complexo composto de complexos */
```

As dependências de uma classe em relação a outras são explicitamente definidas através das listas de importação. Já que o Cm não permite a definição de classes aninhadas, a relação de importação é a forma de indicar as classes que serão utilizadas dentro da classe corrente. Desta forma, elas podem ser utilizadas como um construtor de tipos para a definição de objetos locais. Existem três maneiras de importar uma classe externa utilizando as cláusulas **import**, **use** e **inherit**. A cláusula **import** *list_of_classes* é a mais simples delas: ela apenas torna conhecidas no escopo da classe corrente as classes em *list_of_classes*. A cláusula **use** *list_of_classes* implica na importação da *list_of_classes* e na "instanciação" de um objeto (normalmente anônimo) para esta classe na *list_of_classes*. Estes objetos são compartilhados por várias classes de um mesmo programa. A cláusula **use** é utilizada para a definição de instâncias globais ao programa. Toda classe que quiser compartilhar de uma instância global deve incluir um **use** para a classe apropriada. Isto poderia ser alcancado fazendo com que todas as classes envolvidas passassem como parâmetro o objeto declarado na classe que é "raiz" do programa. No entanto, isto é tedioso e muitas vezes forçaria algumas classes que não necessitam do objeto a ser compartilhado a ter um parâmetro adicional somente para poder passá-lo para as instâncias das classes importadas que o utilizam. A cláusula **use** resolve este problema criando para cada tipo especificado, um único objeto que é compartilhado por todas as classes que "usam" este tipo [DSi88].

Outro conceito incorporado pela linguagem é a herança de classes. A herança de uma classe implica na importação implícita da mesma. Sua idéia central é que novas unidades de software possam ser criadas como extensão de unidades já existentes sem que estas sejam alteradas. Como mencionado anteriormente, esta facilidade auxilia a construção de código reutilizável. A linguagem suporta herança múltipla, isto é, uma classe pode herdar características de várias outras classes, que passam a ser suas superclasses. Dessa forma, os atributos e operadores exportáveis (declaradas como visíveis para outras classes) das superclasses são incorporadas à classe corrente. Quando algum operador herdado possui o mesmo nome de um operador definido pela classe, a operador da classe prevalece sobre a operador da superclasse, o mesmo acontecendo com os operadores dos tipos herdados declarados em uma mesma cláusula **inherit**, onde as classes mais à esquerda redefinem as funções das classes mais à direita. Porém, se desejarmos dispor de ambas, é possível redefinir a operador da superclasse com um outro nome através da cláusula **rename**, como ilustrado no exemplo abaixo [SLD88].

```
class Device()

export Open (int Descriptor)
...  implementação de Open ...

...  outras funções de Device ...

class Terminal()
inherit Device();
rename GenericOpen = Open;

Open (int Descriptor)
...  nova implementação de Open ...
```

A estrutura de dados privativa de uma classe é composta de declarações de constantes, tipos e variáveis, com sintaxe semelhante à da linguagem C. O Cm incorpora todas as facilidades de C, além de permitir a definição de variáveis do tipo derivados de classes, com parâmetros reais definidos. Isto é, classes são construtores de tipos, classes com parâmetros definidos são tipos específicos, e variáveis de tipos específicos derivados de classes podem ser declarados da mesma forma que variáveis de tipos básicos, como inteiros e caracteres. Chamamos isto de "instanciação" da classe.

Além das classes de armazenamento permitidas na linguagem C, as variáveis definidas dentro de uma classe podem ser variáveis de classe ou de instância. Para o primeiro tipo, existe apenas um exemplar deste elemento compartilhado por todas as instâncias da classe, enquanto que, para o segundo, existe um exemplar para cada instância. As variáveis de instância descrevem o estado de uma instância. Uma classe Cm pode ter, ainda, variáveis da classe **`Port`**, que é pré-definida na linguagem. É por meio de instâncias desta classe que as entradas e saídas são realizadas. Ela difere de outras classes importadas porque suas instâncias são sempre conhecidas externamente a nível de programa, já que é necessário ligá-las a outras portas. Estas podem pertencer a arquivos ou outros programas, formando, assim, um "pipe" entre os mesmos. Isto é efetuado a nível da LegoShell, que será detalhada no capítulo seguinte. Outro detalhe é que, como as portas de um programa compartilham um mesmo espaço de nomes, não é possível existir duas portas com mesmo nome em um mesmo programa. Os parâmetros desta classe são dois: o tipo de operação da porta e o tipo de dados a ser comunicado pela porta. O tipo de operação pode ser de entrada, saída ou ao mesmo tempo de entrada e saída. Quanto ao tipo de dados transmitidos, é possível declará-los como bytes (**`Byte`**), blocos de tamanho fixo (**`Block`**) ou blocos de tamanho variável (**`VSBlock`**). Em uma extensão futura, pretende-se criar portas com parâmetro do tipo **`type`**, que permitirão a comunicação de dados tipados, preservando a estrutura dos seus tipos, além de possibilitar sua verificação.

Os operadores de uma classe podem ser exportáveis ou privados (locais), dependendo se são ou não acessíveis externamente à classe. Ou seja, os operadores exportáveis podem ser utilizados pelas classes que o herdam, importam ou "usam", enquanto os privados só são utilizados para executar operadores internos à classe. Assim como as variáveis, os operadores podem ser também de classe ou instância, aplicando-se a mesma semântica. Os operadores Cm, assim como as classes, permitem a definição de valores "default" aos seus parâmetros.

Finalmente, qualquer classe em Cm pode ser utilizada como um programa. Qualquer dos operadores exportáveis de uma classe pode ser invocado para execução. Primeiro é necessário fornecer os eventuais parâmetros atuais da classe selecionando assim um dos seus tipos abstratos derivados. A execução de um operador da classe implica na "instanciação" automática de um objeto deste tipo abstrato que servirá de contexto de execução para o operador. A maioria das classes projetadas para utilização como programa contem somente um operador de nome `main()` que é o operador "default" para execução.

## Aspectos de Implementação

Um tradutor da linguagem Cm para o C padrão está atualmente sendo desenvolvido [Fur90b]. Esta estratégia de implementação foi escolhida a fim de obter maior portabilidade, uma vez que será possível executar programas escritos em Cm em todas as máquinas onde C é disponível. Alguns aspectos com relação a implementação deste tradutor são importantes para a compreensão de alguns detalhes da modelagem do ambiente, bem como de algumas decisões de projeto da LegoShell, o "configurador" de programas.

Em alguns casos, a tradução de uma classe Cm para C só é possível após a definição dos valores dos seus parâmetros formais porque, dependendo destes valores, uma mesma classe Cm pode gerar diferentes programas C. A verificação desta possibilidade nem sempre é trivial. Por exemplo, um parâmetro do tipo inteiro pode ser apenas o valor de inicialização de uma variável, como também o tamanho de um array em sua declaração. Portanto, inicialmente, teremos uma tradução para cada conjunto de parâmetros definidos. Tomemos como ilustração o parâmetro do tipo **`type`**, que é o mais característico na geração de múltiplas traduções. Considerando a classe A do exemplo abaixo.

```
class A ()
import B, C;
B(10, int) i;
```

```
class B (int Size; type Tipo)
import D;
D(Tipo) x;
```

```
B(10, float) x, y;              ...operadores de B ...
C()  w;
...operadores de A ...
```

A Figura 2.3a mostra uma representação das dependências desta classe. Porém, devido às múltiplas traduções de B e D, as dependências reais são como representadas na Figura 2.3b. Assim, antes de realizar a tradução da classe propriamente dita, é necessário que o tradutor verifique se as classes das quais ela depende já estão compiladas para os tipos necessários. Se não o foram, é preciso compilá-las antes da compilação da classe corrente. Este processo é semelhante àquele executado pela ferramenta *make* do sistema UNIX, porém com as dependências definidas internamente ao programa.

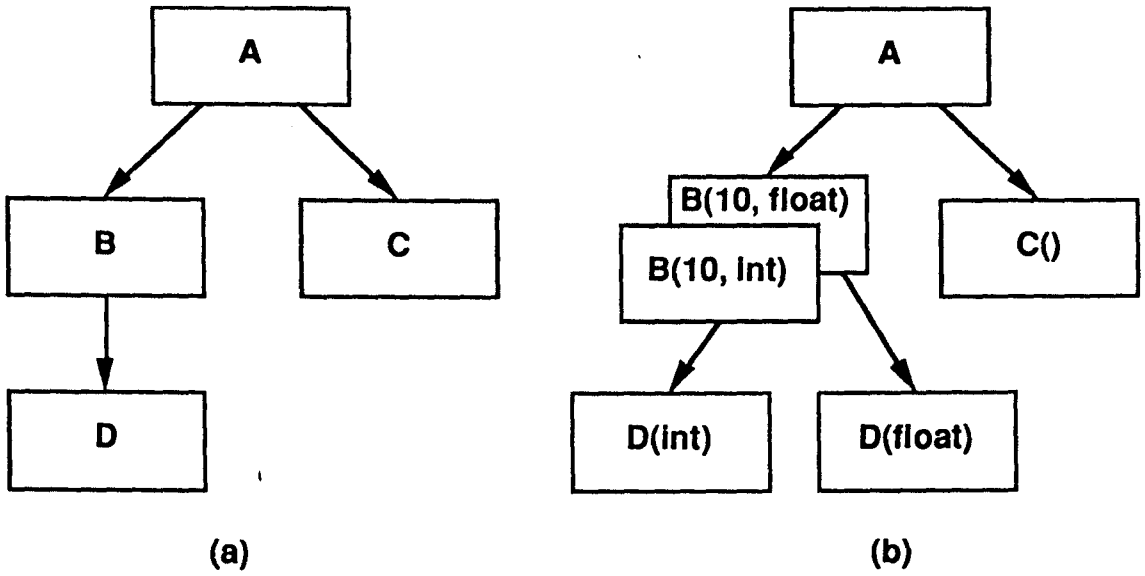



**Figura 2.3 :** (a) Dependências da Classe A. (b) Dependências vistas pelo tradutor

Existem muitos outros aspectos importantes na implementação do compilador Cm, mas que fogem ao escopo do nosso trabalho.

Uma classe Cm, compilada com todas as suas dependências resolvidas, é potencialmente um programa. Porém, como já vimos, as suas entradas e saídas estão, a princípio, em aberto, sendo necessário conectá-las a outros objetos. Dessa forma, é possível combinar as classes a fim de criar um objeto com um nível de abstração maior que, por sua vez, pode ser utilizado para criar um outro objeto. Para expressarmos de forma simples estas "computações", foi concebida a linguagem LegoShell, que descreveremos no capítulo seguinte.

# Capítulo 3

# A LINGUAGEM DE COMPUTAÇÕES LEGOSHELL

Neste capítulo descrevemos as características da linguagem LegoShell, suas potencialidades e particularidades mais importantes.

## 3.1. A Linguagem LegoShell

A LegoShell é uma linguagem gráfica de especificação de comandos complexos chamados computações. A idéia é que a construção destes objetos seja tão simples quanto um jogo de montar, onde as peças básicas são:

- **programas Cm;**
- **arquivos;**
- **dispositivos periféricos;**
- **conectores: "pipe", "broadcast" e "mailbox".**

Todos estes objetos possuem portas de comunicação associadas. Os programas Cm e conectores definem suas próprias portas e os arquivos e dispositivos periféricos têm portas de entrada e saída. Através de cada uma delas fluirá um único tipo de dado em um único sentido. A construção de computações consiste em interligar as portas dos objetos utilizando os conectores. A principal vantagem desta abordagem é que ela constitui uma extensão natural do paradigma de comunicação do UNIX ("Stream Oriented Communication"), facilmente implementável no sistema operacional, e cuja especificação visual é feita de forma muito simples. Cabe destacar que o "Stream Oriented Communication" está sendo adotado como modelo por outros sistemas operacionais [Bal86]. A LegoShell é, então, uma linguagem gráfica que implementa processos que se comunicam, segundo o paradigma de "streams", generalizando o conceito de "pipe" de UNIX através de ligações multidimensionais (conectores). Na Figura 3.1, temos um exemplo da computação *MSort*, constituída de dois programas *Sort* e um programa *Merge*. Note que esta computação pode ser abstraída, tornando-se componente e peça básica para outras computações. Dessa forma, podemos construir computações sem restrições quanto ao número de níveis de abstrações.

Computações como *MSort*, que não possuem todas as suas portas conectadas, são ditas incompletas. Estas portas em aberto devem, então, ser conectadas a uma borda da abstração, onde uma porta é automaticamente criada. Um exemplo de computação completa é ilustrado no exemplo da Figura 3.2. É possível, caso se queira desprezar os dados associados a uma porta, conectá-la a um "buraco negro". Seu propósito é "fechar" uma porta cujo acesso deva ser negligenciado, como na saída de dados duplicados do programa *Sort*.

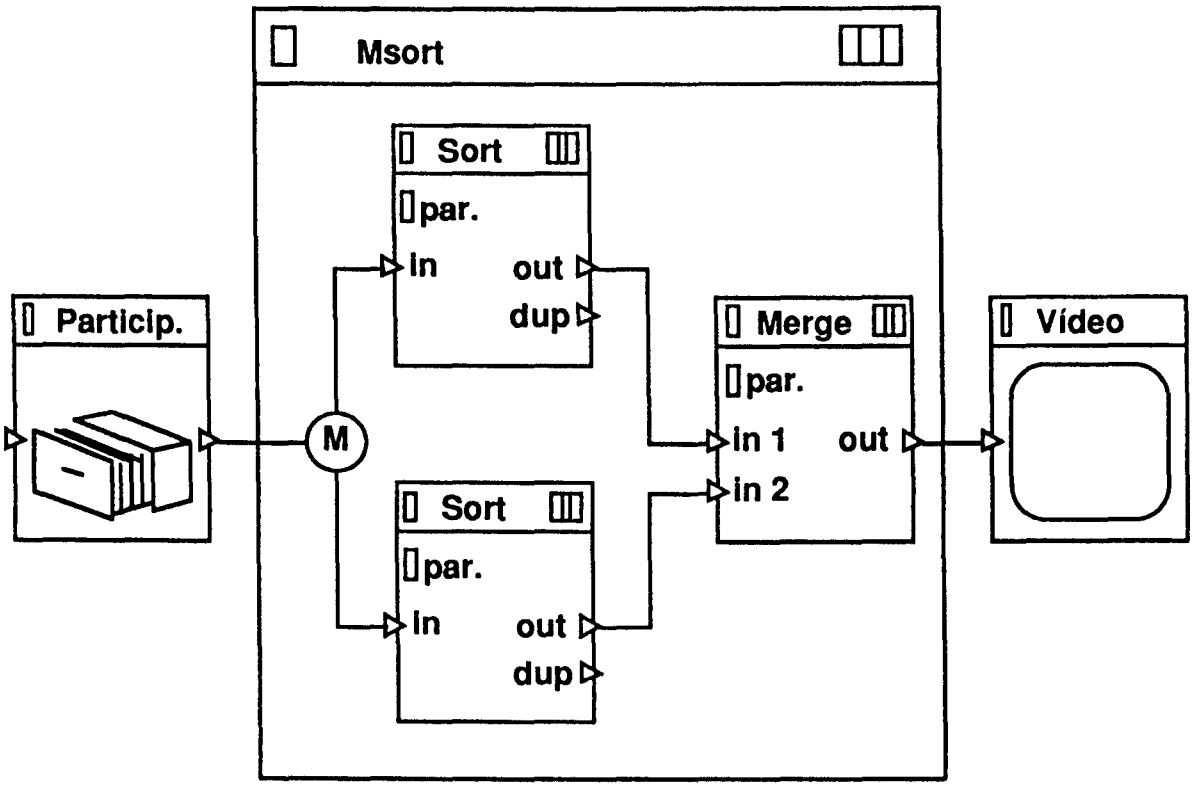



**Figura 3.1** : Exemplo de abstração de computação

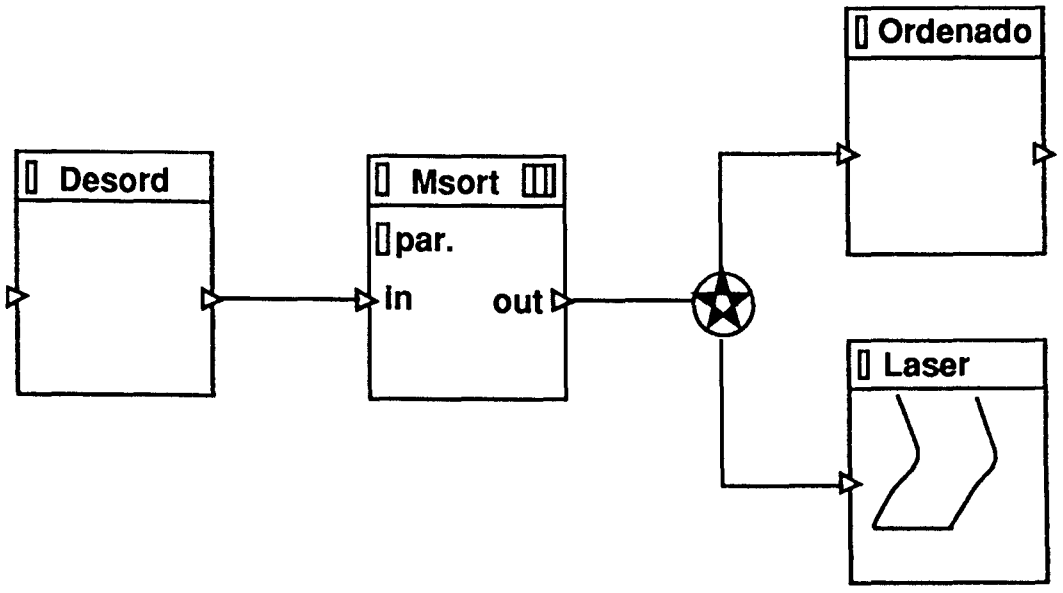



**Figura 3.2** : Exemplo de computação completa

## Os Objetos da LegoShell

Os objetos da LegoShell podem ser agrupados segundo três classificações:

- **composto / primitivo:** se um objeto é construído a partir de outros objetos ou envolve outros em sua definição, ele é dito composto. Por outro lado, objetos primitivos são imutáveis, não possuem dependências e têm um mapeamento direto para objetos físicos.
- **ativo / passivo:** objetos ativos são geradores e/ou consumidores de dados, sob sua própia requisição, enquanto que objetos passivos são fornecedores e/ou receptores e trabalham sob demanda externa.
- **objetos com conteúdo semântico / conectores:** esta classificação distingue objetos que possuem conteúdo semântico por si mesmos daqueles que só têm significado dentro de uma computação da LegoShell, interligando os demais objetos. Na Figura 3.3 apresentamos, de forma tabular, como os objetos se classificam segundo os critérios acima citados.

| Comput. | CmPRG | Perif. | File | Pipe | Mailbox | Broadcast |
|---|---|---|---|---|---|---|
| Objeto com conteúdo semântico | | | | Conector | | |
| Ativo | | | Passivo | | | |
| Composto | Primitivo | | | | | |

**Figura 3.3 :** Classificação dos objetos da LegoShell

Na seqüência, detalharemos cada objeto básico da LegoShell.

## Programas Cm

Na LegoShell, as partes visíveis de um programa Cm restringem-se às suas portas e aos parâmetros da classe. Os parâmetros de um programa Cm são definidos e/ou alterados durante a edição da computação que a contém.

## Arquivos

Um arquivo pode ser visto como um depósito de dados. Ele possui portas de entrada e saída de acordo com a permissão de leitura e escrita que o usuário que o utilizará possuir. Estas portas são virtuais, isto é, elas só passam a existir realmente quando são ligadas a um conector. Dessa forma, quando uma porta de saída é utilizada, uma nova porta virtual de saída é criada. Por outro lado, somente um processo pode escrever em um arquivo (associamos o concepto "processo" ao de "programa em execução"). Portanto, um arquivo possui no máximo uma porta de entrada. Na Figura 3.2 temos a representação gráfica de arquivos em dois exemplos: arquivo desordenado *Desord* e arquivo ordenado *Ordenado*.

## Dispositivos Periféricos

Os dispositivos periféricos podem ser vistos como objetos produtores ou consumidores de dados e detentores de portas de entrada e saída pré-definidas de acordo com seu tipo. Assim, um dispositivo de entrada (i. e. um teclado) será um processo que terá uma porta de saída associada, através da qual fluirão os dados que gerou.

Um dispositivo de saída (i. e. uma impressora) será um processo que terá uma porta de entrada associada, pela qual chegam os dados que deve consumir.

Assim como os programas Cm, os periféricos possuem parâmetros, cujos valores são definidos em sessões da LegoShell. No protótipo, todos os dispositivos terão a mesma representação gráfica, como ilustrado pelo periférico "Tela" da Figura 3.2. Futuramente, haverá a facilidade de associar ícones distintos para cada tipo de periférico.

## Conectores

Os conectores são objetos concentradores e distribuidores de dados. Cada conector terá associado um "buffer" no qual armazena os dados que consumiu e ainda não distribuiu. Enviar um dado a uma porta de conector equivale a colocá-lo no final do "buffer" que o implementa e, receber um dado de um conector, equivale a retirar seu primeiro elemento. O tamanho do "buffer" é um parâmetro que pode ser alterado. Se ele for igual a zero, dizemos que o conector é síncrono, caso contrário, ele é dito assíncrono.

Durante o segundo semestre de 1988 trabalhamos em conjunto com Alicia Noemi Di Sarno, para especificar qual seria a semântica mais adequada para os conectores. Em [SDr88] encontramoas um análise semântica dos conectores, da política de consumo e distribuição dos conectores e considerações sobre a sua implementação.

Um conector "broadcast" ou "mailbox" pode ser visto, e também implementado, como um programa ou computação com significado peculiar. Não obstante distingue-se dos demais objetos com conteúdo semântico, porque, na LegoShell, eles só têm significado dentro de uma computação, interligando os demais objetos (os conectores).

Na primeira versão da LegoShell, os dados transmitidos pelos conectores são apenas seqüências de bytes. Porém, assim como Balzer [Bal86] defende que a próxima geração de sistemas operacionais fará a comunicação dos processos através de um banco de dados, com a vantagem de terem comunicação de dados "tipados", pretende-se, futuramente, que os dados transmitidos pelos conectores tenham tipos associados.

Os conectores previstos pela LegoShell são:

- **Pipe:** define um canal de comunicação unidirecional. Em muitas aplicações, o conceito de canal bidirecional é necessário, o que é equivalente a dois canais unidirecionais. Neste caso, o usuário indicará estas ligações, que serão representadas graficamente como um único canal bidirecional.
- **Broadcast:** é semanticamente equivalente a uma rede de disseminação ("broadcast") sem colisões. Os dados recebidos pelas portas de entrada são distribuídos para todas as portas de saída. Todas as portas de saída receberão a mesma seqüência de dados. As portas de um conector "broadcast" são ilimitadas em número, virtuais e invisíveis. Elas só se tornam visíveis quando conectadas a um "pipe".
- **Mailbox:** define uma caixa postal, isto é, ao contrário do "broadcast", as mensagens recebidas pelo "mailbox" são consumidas quando requisitadas. Portanto, uma mensagem canalizada via "mailbox" atinge apenas um consumidor acoplado a este conector. Suas portas são também ilimitadas, virtuais e invisíveis.

Os conectores "broadcast" e "mailbox" generalizam o conceito de "pipe" do UNIX, sendo esta uma das principais características da linguagem LegoShell.

Na próxima seção mostraremos como estes objetos podem ser utilizados para a construção de computações que estão além do alcance limitado do "pipe" de UNIX [Dru89].

## Computações da LegoShell

No exemplo mostrado na Figura 3.1 vemos um Programa Cm chamado *Sort* que ordena os dados de sua porta de entrada reproduzindo-os na sua porta de saída. O programa *Merge* produz em sua porta de saída a intercalação de seqüências ordenadas oriundas de suas duas portas de entrada. O conector "mailbox" é utilizado para distribuir o conteúdo do arquivo de entrada para as duas instâncias do programa *Sort*. Cada uma receberá um conjunto distinto de dados. Elas estão competindo pelo mesmo recurso (dados do arquivo de entrada) e o "mailbox" arbitra a competição. O "mailbox" lê do arquivo sob demanda dos programas *Sort*, enviando cada dado ao *Sort* que o requisitou há mais tempo. Os programas são objetos ativos e conseqüentemente suas portas de entrada e saída também são ativas, fazendo requisições aos conectores associados. O arquivo é um objeto passivo e suas portas esperam por requisições para produzir um dado.

Os conectores tipo "pipe", que interligam dois outros conectores ou um conector e um arquivo ou programa, têm sempre "buffer" de tamanho zero. Só podem ter "buffer" de tamanho maior de zero quando interligam dois extremos ativos. Os conectores que interligam os *Sorts* ao *Merge* podem ter "buffer" maior que zero e neste caso servem para equilibrar variações de velocidade dos processos.

Uma computação é dita completa se todas as suas portas reais estão devidamente conectadas, e somente computações completas podem ser executadas. O buraco negro deve ser utilizado para conectá-lo às portas que devem ser ignoradas, para se chegar a uma computação completa.

O próximo exemplo, retirado do cotidiano, nos mostrará como usar a LegoShell para especificar a execução distribuída de uma computação.

## Distribuição

Os arquivos em UNIX já são distribuídos na rede, de forma transparente ao usuário, pelo NFS ou outro sistema semelhante. A LegoShell permite também que cada computação ou programa Cm possa ser executado em uma máquina diferente da rede local. Para tal usaremos um mecanismo próprio de UNIX ("sockets" ou "streams").

Na Figura 3.4 definimos uma agência bancária usando computações da LegoShell. Determinadas funções estão associadas ao atendimento dos clientes, são as computações chamadas *Ponto de Atendimento*. A estas computações estão associados os periféricos *CAIXA*, representados por estações de trabalho dedicadas. Determinadas operações geram requerimentos, por exemplo de talões de cheque, que são enviados através da porta *REQ* da computação *Ponto de Atendimento*. Todos os requerimentos deste tipo são centralizados no conector "mailbox" (M na figura) que por sua vez os distribui para as computações $GERENTE_i$. $GERENTE_i$ atenderá sempre a primeira solicitação da fila determinando ou não a sua autorização. *ALMOXarifado* atenderá todas as solicitações autorizadas pelos gerentes e concentradas pelo conector "broadcast" (Estrela na figura), atualizando simultaneamente o estoque e gerando, quando necessário, os pedidos de mais talões de cheque, que podem ser derivados para uma computação que implementa as funções do departamento de produção gráfica do banco. O leitor pode aqui verificar a facilidade com que se faz a especificação de computações com a LegoShell.

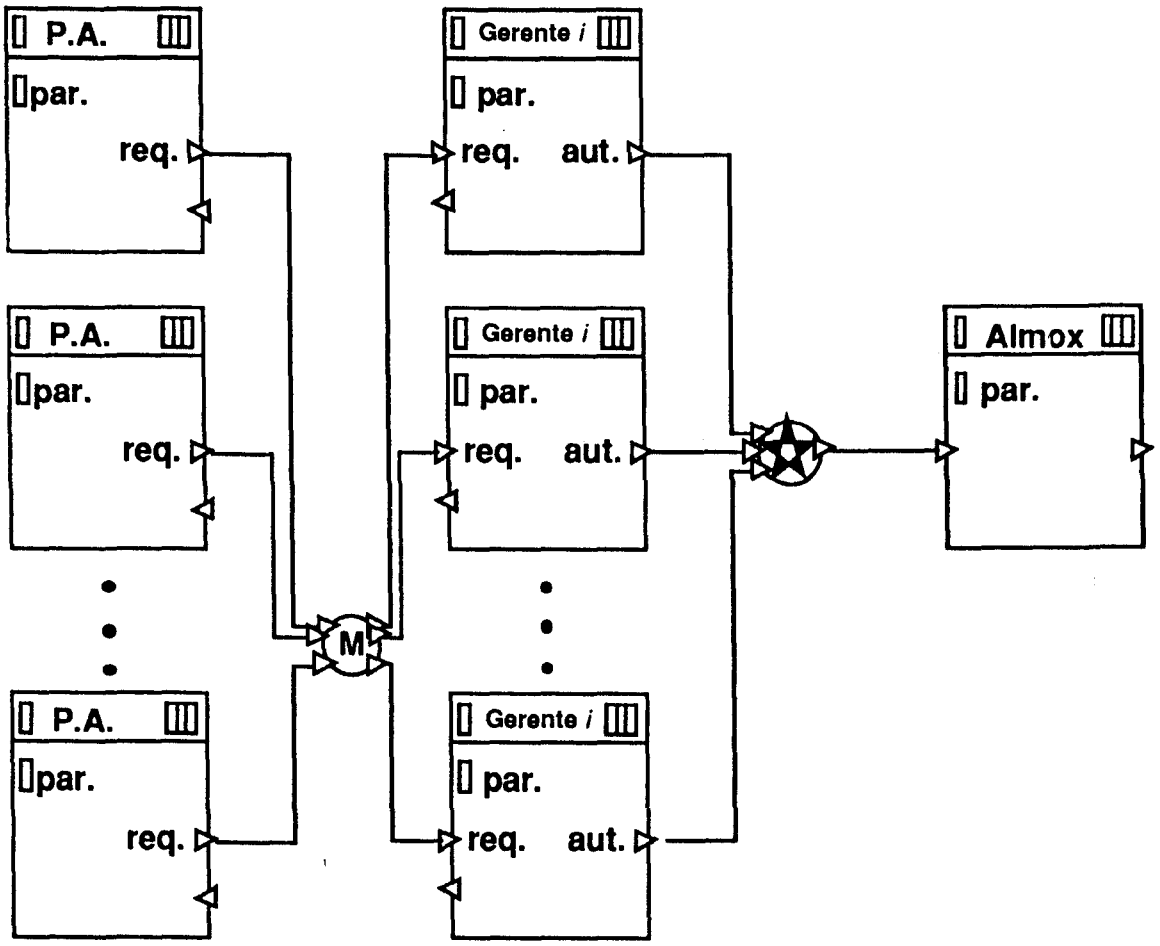



**Figura 3.4** : Modelo (parcial) de um banco usando LegoShell

Vemos que as computações podem ser executadas num único processador ou podem ser distribuídas de forma tal que a comunicação se faça através da rede, ficando totalmente transparente para o usuário qual das opções foi escolhida. A estação onde um programa será executado é um parâmetro da computação associada à instância do programa. Veremos na seção seguinte como são especificados estes parâmetros. Veremos também que a LegoShell permite fazer um gerenciamento dos recursos da rede, através da distribuição da localidade dos processos que implementam as diferentes computações, de maneira muito simples e direta.

## 3.2. Parâmetros dos Objetos

Como mencionamos previamente, as partes visíveis de um programa Cm restringem-se às suas portas e aos parâmetros da classe. É dentro de uma edição de computação, da qual um programa Cm é componente, que os valores de seus parâmetros são definidos. Porém, é possível fazer com que parâmetos dos componentes passem a ser parâmetros da computação a ser abstraída. Dessa forma, os parâmetros do componente passam a ser uma referência a um parâmetro da computação. Por exemplo, na Figura 3.1, se um dos parâmetros dos programas *Sort* e *Merge* for o tipo dos elementos a serem ordenados, é interessante que o mesmo passe a ser um parâmetro da computação *MSort*. Assim, além de evitar que exista um *MSort* para cada tipo de dado, quando este fizer parte de uma outra computação, é preciso definir apenas uma vez qual o tipo dos elementos e, automaticamente, os programas *Sort* e *Merge* serão instanciados para o tipo correto.

Um programa Cm, assim como uma computação, possui ainda parâmetros de execução, que devem ser definidos para que este possa ser executado dentro de uma computação. Um exemplo deste tipo de parâmetro é a máquina onde ele será executado, visto que o ambiente A_HAND foi idealizado para uma configuração composta de estações de trabalho interligadas em rede. Se analisarmos a sintaxe dos comandos de uma Shell do UNIX que permite execução remota, veremos que seu formato é semelhante ao seguinte:

```
on UNIX3 ' cat -l 100 -f /dev/tty01 '
```

onde *on* é o comando "executar em remoto", UNIX3 corresponde ao nome do hospedeiro, `cat` ao comando, `-l -f` são opções de execução e `100` e `/dev/tty01` são os argumentos. Assim, distinguimos no ambiente três conjuntos de parâmetros:

- **parâmetro de classe**, que são os parâmetros definidos pelas classes Cm;
- **chaves de seleção**, que correspondem às opções e
- **parâmetros de execução**, que envolvem, além dos argumentos, o nome do hospedeiro, prioridade de execução e outros.

No UNIX, somente os dois últimos tipos de parâmetros são suportados. Os parâmetros são passados para os programas na forma de "strings" sem qualquer pré-processamento. O programa é responsável pela análise, conversão e verificação de consistência destes parâmetros. A LegoShell, por outro lado, possibilita o pré-processamento e a verificação de tipos dos parâmetros. O número de parâmetros dos comandos é em geral grande e a maioria é opcional, com valores "default" apropriados para simplificar o seu uso. Cada usuário conhece apenas algumas das opções dos comandos que usa com mais freqüência. Não é incomum que usuários experientes descubram, após anos de uso de um comando, que ele tem uma opção, até então ignorada, e que poderia ter lhe simplificado a vida se conhecida anteriormente. A abundância de comandos e opções com a interface criptográfica colabora com que os usuários se restrinjam a um subconjunto de comandos e opções. O resultado final disto é que se usa mal o UNIX. A LegoShell apresenta as opções de forma tabular como um formulário onde os campos de dados podem ser alterados. (Figura 3.5). Só a facilidade de visualizar os parâmetros e opções com seus valores de forma compacta já induz ao usuário a usar novas opções. Ademais, como os parâmetros e opções serão apresentados ao usuário quando requerido, não é mais necessário que ele decore os nomes das opções.

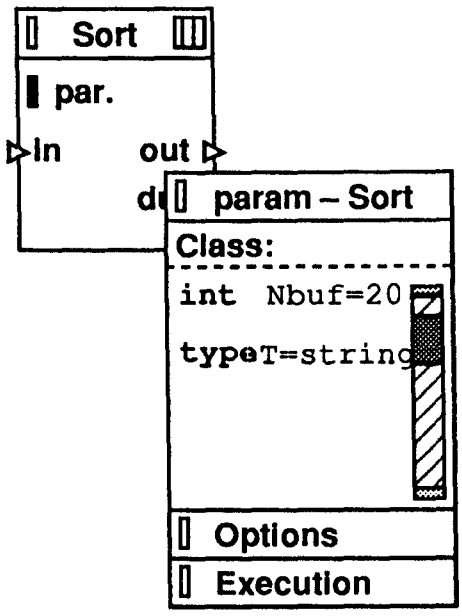



**Figura 3.5 :** Janela de edição de parâmetros

Veremos a seguir uma forma muito interessante de se criar novas computações.

## 3.3. Abstração de Computações

Abstrair uma computação significa criar uma nova computação a partir de um subconjunto dos componentes de algum objeto. Esta computação contem os componentes do subconjunto e suas portas são as portas que conectam estes componentes com o resto dos componentes. Através deste mecanismo de abstração, as propriedades polimórficas e de encapsulamento das classes Cm são preservadas a nível de computações da LegoShell.

No momento da abstração teremos 4 tipos de portas:

a. Porta com "pipe" que a conecta a outro componente selecionado para abstração.
b. Porta com "pipe" que a conecta a um componente não selecionado.
c. Porta sem pipe.
d. Porta conectada a Buraco Negro.

As portas dos tipos **a** e **d** ficarão encapsuladas e não serão acessíveis a partir do exterior da computação criada pela Abstração. As portas do tipo **b** automaticamente se transferem para o perímetro da nova computação. Para as portas do tipo **c** teremos que decidir se as fazemos visíveis ao exterior ou não. Neste último caso, deveremos conectá-las a buracos negros.

### Expansão

Inversamente à Abstração, Expansão significa substituir, dentro do objeto em edição, uma computação por seus componentes. Como conseqüência disto, os "pipes" ligados a ela são conectados aos respectivos componentes. Esta ação não implica na exclusão da computação do ambiente; ela é simplesmente expandida no objeto em edição.

## Verificação de Consistência

Ao falar em consistência de uma computação de LegoShell nos referimos ao nível do objeto em edição. Isto é, verificamos que todos os objetos ou portas estão conectados, que os dados que transitam pelas portas são do tipo adequado para aquela porta, que os parâmetros do objeto em edição foram adequadamente definidos, bem como os parâmetros dos componentes, etc.

Existe na LegoShell uma função que nos permite verificar a consistência do objeto em edição, internamente e somente no seu nível. A consistência é um atributo do objeto em edição. Ela é equivalente a compilar, mas sem geração de código.

Um objeto só pode ser incluído em uma computação de LegoShell se for consistente. O ambiente verificará isto quando se tenta incluir este objeto como componente de uma computação, corroborando o estado do atributo consistência.

Por outro lado, já mencionamos em várias oportunidades que o ambiente permitirá que se armazenem objetos incompletos, e entre eles objetos inconsistentes. É por isto que a ação de consistir é delegada ao usuário como uma ação voluntária. Um objeto pode ser armazenado incompleto se e somente se ele for de propriedade do usuário (particular) e só pode ser "congelada" (i.e. salva e feita visível pelo ambiente) a versão de um objeto consistente. Mas a exceção à regra se dá quando se produz a abstração de um objeto: imediatamente antes de produzida a criação do novo objeto, verifica-se sua consistência para permitir que seja incluído no objeto em edição.

A nível de ambiente, o sistema de controle de versão permite manter a consistência da configuração dos objetos. Também é necessário verificar a consistência de um objeto antes de ser congelado e liberado para outros usuários. Portanto, além da já mencionada, existem duas funções que implicam no congelamento de objetos complexos, e portanto requerem prévia verificação de consistência dos mesmos: são as funções de efetivação e liberação.

Reunindo as facilidades de distribuição, mencionadas na seção anterior, às facilidades de abstração, teremos uma poderosa ferramenta para configuração e monitoração lógica de complexas redes de computadores, com capacidade de abstração e hierarquização. Isto é possível graças às funções da LegoShell que permitem ligar, desligar e suspender a execução dos processos que implementam uma computação e seus componentes a qualquer momento. Discutiremos estas funções com mais detalhe no capítulo 6.

Descreveremos a seguir o sistema de controle de versões e configurações do A_HAND.

## 3.4. Controle de Versões e Configurações do Ambiente

O modelo de controle de versões do ambiente A_HAND foi proposto por Rogério Drummond e descrito por Victorelli em [VMD89]. Ele distingue três tipos de versões:

- **edição:** é uma versão ainda em progresso e particular a cada projetista.
- **efetiva:** é criada por promoção de uma edição para ser testada dentro de um grupo de projeto.
- **liberada:** é uma versão efetiva aprovada e, portanto, validada para utilização por todos os usuários do sistema.

Tanto versões liberadas como efetivas não podem ser atualizadas, isto é, estão congeladas. Quando algum projetista deseja atualizar uma destas versões, deve antes derivar uma versão particular da mesma e, então, trabalhar sobre ela.

Os objetos desenvolvidos no ambiente e que portanto posuem versões são os programas Cm e computações da LegoShell. Como já vimos, um programa Cm é um objeto composto, construído a partir de relacionamentos de importação, "uso" e herança. Uma vez que os programas possuem versões, é necessário a definição de um

"plano de configuração" que ligue uma versão específica deste objeto composto a versões específicas de seus componentes, ou seja, às classes das quais depende. Portanto, quando um programa participa de uma computação, sua identificação apenas não basta. É preciso uma especificação da versão deste programa e a configuração que resolve suas dependências. Da mesma forma, existem configurações de computações LegoShell que, quando utilizadas dentro de outra computação, sofrem o mesmo processo de seleção de versão e configuração.

Devemos distinguir neste ponto uma definição de configuração (dc) de uma visão de execução (ve). Uma visão de execução é composta por um conjunto S de pares do tipo (objeto de projeto, número de versão), ou (OP, v), onde "v" identifica univocamente uma versão. O número da versão é determinado a partir das entradas que compõem uma definição de configuração. Em outras palavras, a definição de configuração determina como o número das versões é encontrado e a visão de execução origina-se da resolução de uma definição de configuração para um determinado objeto composto. Neste modelo, as entradas de uma definição de configuração podem ser de três tipos:

- **Direta (OP, i):** este tipo de entrada especifica de forma explícita a versão que será copiada em S. "i" pode ser um número de versão ou um rótulo. No primeiro caso, o par (OP, i) é diretamente inserido no conjunto S. Para o segundo caso, os rótulos possíveis são: ULTLIB, ULTEFET e CORRENTE. Caso seja especificado ULTLIB, o número de versão correspondente à última versão liberada é procurada e o par (OP, número de versão) é inserido em S; o rótulo ULTEFET corresponde ao número de versão da última versão efetiva e o rótulo CORRENTE corresponde à última edição do usuário envolvido.
- **Indireta (OP, dc):** a versão do objeto OP a ser inserida em S é aquela determinada utilizando a definição de configuração dc para a sua resolução.
- **de Inclusão (dc, p):** as entradas de inclusão determinam uma prioridade p para a definição de configuração dc. São construídos conjuntos de pares S', S", ..., para cada entrada de inclusão. É feito, então, um "merge" destes conjuntos da seguinte forma: para cada objeto de projeto OP, será escolhido o número de versão correspondente à entrada de inclusão de maior prioridade. Esta versão só será inserida no conjunto S, caso ainda não exista em S nenhuma entrada para este objeto de projeto.

Para os componentes cuja versão não foi resolvida pela definição de configuração, é utilizada uma definição de configuração "default". Ela baseia-se na propriedade dos objetos componentes e pode ser alterada segundo as necessidades do projetista. Inicialmente, o ambiente A_HAND adotará a seguinte configuração "default":

- para os objetos pertencentes ao projetista, será utilizado o rótulo CORRENTE;
- para os objetos não pertencentes ao projetista, mas a algum membro de seu projeto ou ao projeto propriamente dito, será utilizado o rótulo ULTEFET;
- para todos os demais, será associado o rótulo ULTLIB.

Porém, esta definição poderá ser alterada, tanto a nível de ambiente, como de projeto e usuário. Isto é, será possível definirmos diferentes configurações "default" para cada projeto, bem como diferentes configurações "default" para cada usuário membro destes projetos. A prioridade dentre estas definições será a de mais baixo nível, ou seja, a do usuário.

Em sua primeira versão, o ambiente não dará suporte a versões alternativas, permitindo concorrência somente no último nível. Portanto, o histórico de versões típico de um objeto terá a seqüência como mostra a Figura 3.6.

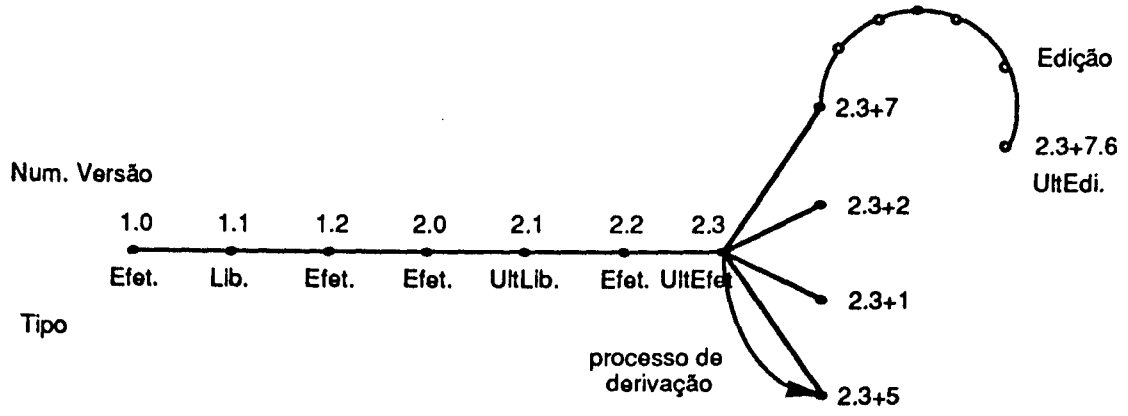



**Figura 3.6** : Histórico de versões do A_HAND

Os números de versão utilizados têm a seguinte notação:

**E.I + U.Ed,** onde

- **E:** é o número de versão externa ou de interface, que é acrescido de um quando a interface é alterada;
- **I:** é o número de versão interna, que é acrescido de um quando o objeto é alterado somente internamente. Quando o campo E é acrescido de um, o campo I é automaticamente zerado;
- **+:** indica derivação de versão. Significa que a versão corrente é derivada da versão indicada por E.I;
- **U:** é a identificação do usuário. Cada usuário pode ter uma versão particular de um determinado objeto (seu objeto em edição). É somente nesse nível que a concorrência pode existir;
- **Ed:** identifica o número da edição feita pelo usuário. Estas edições funcionam como aquelas feitas em um editor de textos usual. O ambiente permitirá que cada usuário defina o número de edições que serão mantidas, porém a responsabilidade de gerenciamento de espaço será responsabilidade de seu proprietário.

Quando um dos usuários consegue efetivar uma de suas versões particulares (edições), as versões particulares dos demais usuários passam automaticamente a ser derivadas desta nova versão efetiva. No esquema de Figura 3.6, se 2.3+7.6 é efetivada, ela passa a ser a última efetiva, com número 2.4 (ou 3.0). As demais versões derivadas de 2.3 passam de 2.3+20 para 2.4+20, de 2.3+18 para 2.4+18 e assim por diante. Esta notação de versões não será necessariamente mantida internamente. Ela apenas representa o esquema lógico do modelo para facilitar sua compreensão.

A criação e resolução de configurações são funções que serão desempenhadas pelo sistema controlador de versão (cv) através do editor da LegoShell. Para cada edição de computação existe uma configuração ativa. Ao derivarmos uma nova edição de uma computação, adotaremos como "default" de configuração ativa aquela que era a ativa no objeto do qual efetuamos a derivação. Deste modo sempre manteremos a consistência dentro do ambiente.

O sistema controlador de versão (cv) tem dois componentes básicos: controlador de acesso e controlador de configuração.

O primeiro oferece funções de armazenamento e recuperação de versões de objetos. Ele é responsável pela proteção e visibilidade dos objetos, pelo esquema lógico de armazenamento (em um banco de dados ou no sistema de arquivos), pela sua forma (arquivos diferenciais, compactação, etc.), pela determinação dos números das versões "default" em função do usuário e da sua relação com o projeto ao qual o módulo pertence.

O controlador de configuração é responsável por manter a associação dos objetos e suas tabelas de configuração. Ele inclui um editor de configuração que possibilita a edição, criação e remoção de tabelas de configuração para os objetos do A_HAND.

O **cv** é um módulo que pode ser facilmente acoplado às ferramentas de desenvolvimento cujos objetos devam ter versões. Ele será utilizado pelo tradutor Cm → C [Fur90b], sistema **ht** [Pol90] e pelos editores sensíveis a estrutura de LegoShell , de Cm [Tro90], de $CO^2$ e de "statecharts" [Fig90].

O **cv** ainda está em desenvolvimento e sua interface funcional foi definida no âmbito do presente trabalho. Como o editor da LegoShell é a ferramenta do A_HAND com a interface mais elaborada, ela herdou a responsabilidade de estabelecer o padrão de interface do ambiente. Decorre disto o nosso interesse em especificar a interface com o usuário do sitema **cv**. Nos capítulos seguintes, ao referirmos a funções que interagem ou dependem do **cv** suporemos, para clarificar nossa exposição, que ambos os sistemas estão integrados.

Discutiremos a filosofia de implementação do editor no capítulo seguinte.

# Capítulo 4

# FILOSOFIA DE IMPLEMENTAÇÃO

Neste capítulo enunciamos as características mais importantes que deve possuir uma interface amigável com o usuário, justificamos a escolha do sistema de janelas X-Windows e dos Estadogramas, definindo por último a arquitetura do Editor da LegoShell.

## 4.1. Como Funciona o Ambiente

Sendo uma característica da LegoShell possuir uma notação bidimensional, a escolha lógica para um editor neste nível é um editor gráfico. Este editor será também o front-end para o ambiente, isto é, ele deverá permitir a manipulação dos objetos da LegoShell e simultaneamente executar as funções inerentes ao ambiente. Diversas ferramentas serão colocadas à disposição do usuário, com as quais os objetos criados poderão ser manipulados simultaneamente. Pretende-se atingir a integração do ambiente através de uma interface com o usuário o mais padronizada possível, a qual chamaremos de interface genérica.

### Interface Genérica

A maioria dos autores, ao se referirem ao projeto de uma interface com o usuário, dizem que esta é a parte mais difícil de uma aplicação interativa. Ele requer o conhecimento profundo da aplicação e dos pacotes gráficos, e do software e do hardware a serem usados, e ainda uma grande sensibilidade para perceber as necessidades e habilidades do futuro usuário [Spr]. Isto se deve ao fato de que não existem métodos ou regras fixas para o projeto de uma interface com o usuário. Este é um campo multidisciplinar, onde interagem a Ergonomia, a Psicologia Aplicada, a Psicologia do Conhecimento e a Ciência da Computação, cujo objeto de estudo é o Homem e seu comportamento [GC87]. Os resultados de cada experiência não têm, normalmente, significado fora do seu contexto, e por isto são de difícil generalização [Re86, LSi87, DA79].

Adotaremos como definição de interface com o usuário a elaborada por Hill [Hi86]: "é a parte de um Sistema Interativo que apresenta informação ao usuário, e escolhe comandos do usuário para passá-los às rotinas de aplicação para sua execução". Dentro deste contexto utilizaremos o modelo de interface com o usuário proposta por Seeheim, que a divide em três componentes (Figura 4.1). O componente de representação, que manipula diretamente os dispositivos de entrada e saída, pode ser considerado como o nível léxico da interface. O componente de controle do diálogo, responsável pelos comandos e o diálogo empregado pelo usuário, pode ser visto como o nível sintático da interface com o usuário. E, finalmente, o nível do Modelo de interface com a Aplicação define a interface com o resto do programa e manipula as chamadas aos procedimentos da aplicação. Estes três componentes podem ser vistos como três processos independentes comunicando-se através de mensagens, o que constitui a vantagem do modelo.

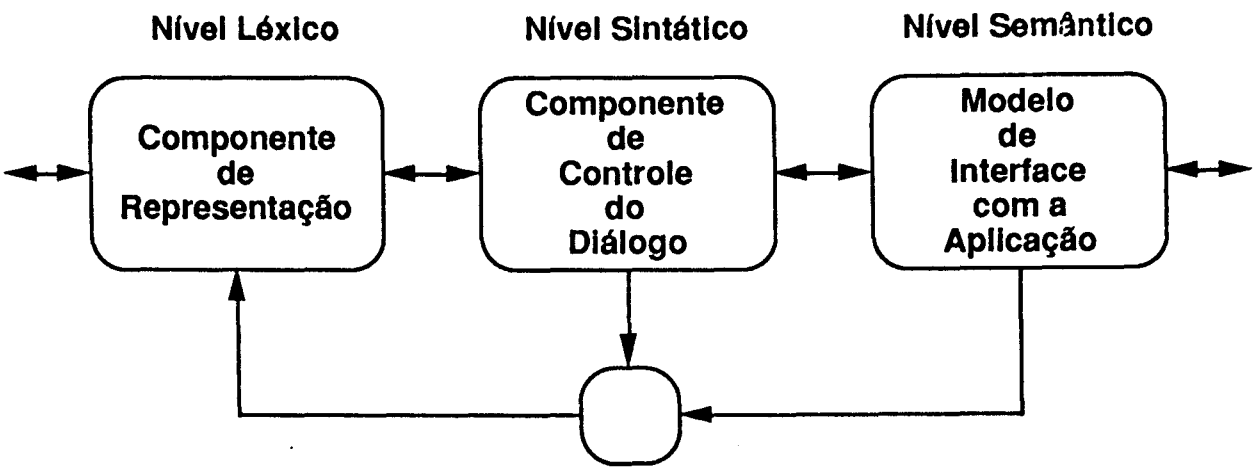



**Figura 4.1** : Modelo de interface com o usuário de Seeheim

Atualmente a parte que melhor se conhece é o componente de controle do diálogo, já que existem três grandes classes de formalismos diferentes, mas de poder descritivo equivalente, para se expressar formalmente o diálogo. São eles as Redes de Transição, as Gramáticas Livres de Contexto e os Eventos [Gr86]. Um de nossos objetivos foi determinar qual destes formalismos, ou de suas variações [SMa84, Ja86, Pn86, Ha87, Ha88], se adequa melhor às necessidades do tipo de interface com o usuário que desejávamos definir [SHB86, SIKV82, Bar86, Hi86, HK86].

Ao desenvolver a interface com o usuário visávamos obter velocidade de uso, rapidez na adaptação ao uso (para o usuário experiente) e facilidade de aprendizado e uso (para o usuário inexperiente) [DN85]. Reisner [Re86] considera que a forma mais adequada para se obter estas qualidades são a prototipação, a iteração do processo de projeto para correção do protótipo, a modelagem da interface com o usuário, e a proposta de testes (benchmarking) para comprovar os resultados. Este é um dos motivos pelos quais escolhemos o caminho da prototipação.

Para definir a interface genérica ideal a ser atingida pelo ambiente A_HAND, deve se levar em conta que a velocidade com a qual um novo usuário se adapta ao ambiente tem relação direta com a quantidade de seqüências que ele deve memorizar, com a padronização dos comandos para executar uma mesma operação nos diferentes níveis e, fundamentalmente, com a coerência do conjunto. Uma mesma seqüência não deve ter significados opostos em dois contextos diferentes.

Outro fator muito importante relacionado com a produtividade num ambiente de programação é que uma seqüência de comandos possua o mesmo significado em todos os níveis do ambiente, permitindo que o usuário se mecanize no seu uso. Esta característica, chamada de "modeless interface" junto com a padronização da interface, é a pedra fundamental da facilidade de uso da interface do Macintosh® [UIG84].

Para que o usuário se sinta motivado a usar o ambiente, ele deve ser de uso prazeroso e deve induzir o seu uso, tanto no sentido de facilitar as tarefas de desenvolvimento através dele, quanto no sentido de impedir que tarefas sensíveis ao ambiente sejam feitas fora dele. Como exemplo disto último, diremos que não deve ser possível manipular os arquivos contendo dados dos objetos do ambiente sem a utilização da sua interface.

Descrevemos na seqüência outras características desejáveis em todo ambiente, que propomos como características básicas da interface Genérica "Ideal" do ambiente.

Um dado importante a ser levado em conta é que, se todas as ferramentas disponíveis no ambiente são usadas através de uma interface genérica padronizada, o acesso a uma nova ferramenta, desconhecida até esse momento pelo usuário, requerirá de pouco ou nenhum treinamento específico, já que o usuário terá a sensação de já conhecê-la.

## HELP – Auxílio Eletrônico

Como um dos objetivos do ambiente é a ausência de manuais, uma das formas de atingi-lo é contarmos com um procedimento de auxílio "on-line". Esta ajuda estará sempre relacionada com o contexto da edição e pode referir-se ao significado do estado atual, à próxima ação a ser executada, a condições de erro ou ao significado de parâmetros e variáveis. O HELP será implementado usando o sistema de hipertexto (ht) proposto por Carlos Alexandre Polanczyk [Pol90].

No futuro, será possível ajustar o ambiente a um nível de auxílio de acordo com o conhecimento que o usuário possui sobre o mesmo. Assim, usuários mais experientes usariam simplificações de comandos ao invés de passar por vários menus para atingir a função desejada.

## Tailoring – Personalização

Definidas as funções que a interface genérica deve proporcionar ao usuário, a possibilidade deste adequar a interface ao seu estilo particular de trabalho é considerada muito importante: há quem goste mais de "pop-up menus" enquanto outros gostam mais de "pulldown menus" e há ainda um terceiro grupo de usuários que adota uma combinação não trivial de ambos.

No protótipo, será permitido ao usuário redefinir os "defaults" do ambiente, assim como definir qual é o estado do ambiente que deseja ser guardado de uma sessão de edição para outra. Além disto, nosso ambiente permitirá, numa segunda fase, que o usuário reconfigure o ambiente ao seu gosto, sem alterar a sua filosofia básica.

Outra função de adequação ao usuário será a de incluir novas classes pré-definidos como escolhas disponíveis no menu de ícones. O ambiente permitirá também que o usuário escolha o nível de ajuda que deseja que o ambiente lhe proporcione.

Descreveremos a seguir qual será a interface com o usuário do protótipo.

## A Interface com o Usuário do Protótipo

O editor da LegoShell, além de permitir a manipulação dos objetos na composição de computações, servirá de interface para invocar funções inerentes ao ambiente. Diversas ferramentas serão colocadas à disposição do usuário através deste editor, obtendo a integração do ambiente através de uma interface padronizada.

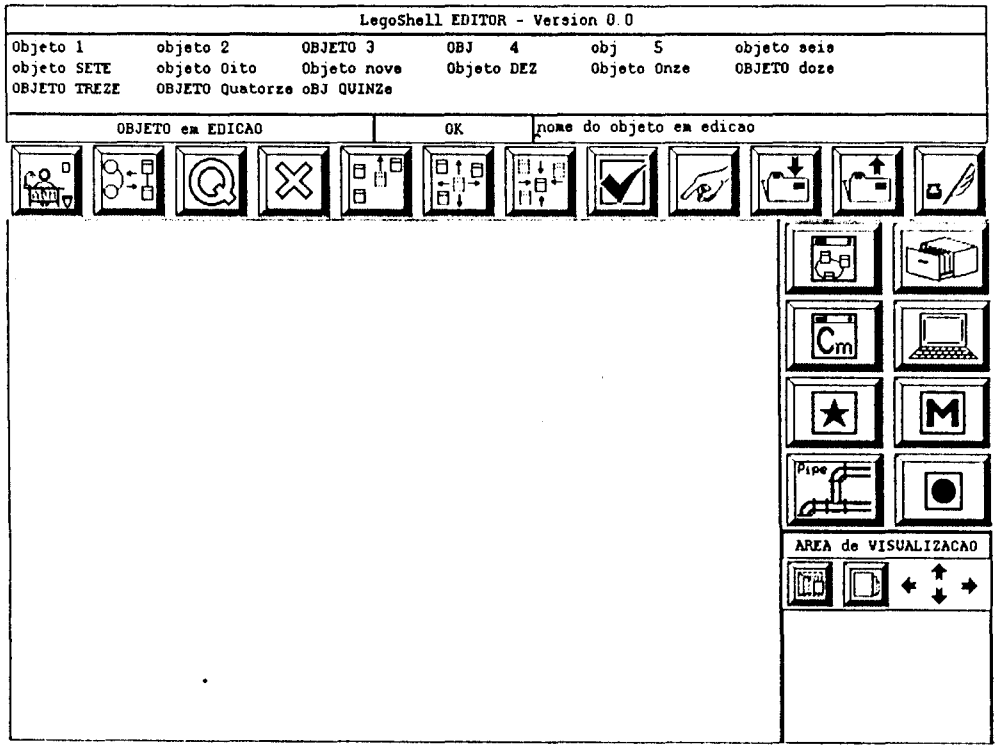



**Figura 4.2** : A interface com o usuário do protótipo

No protótipo do editor, a interface será como ilustrado na Figura 4.2. Distinguimos nela cinco áreas:

- **cabeçalho:** ela é composta por uma pilha contendo o nome dos objetos sendo editados. Como é possível suspender temporariamente uma edição, iniciando outras, para depois retomá-la do ponto em que foi abandonada, os nomes na pilha indicam as edições suspensas no momento. Quando da suspensão, dizemos que descemos um nível de edição, enquanto que quando a retomamos dizemos que subimos um nível de edição.
- **menu de barras:** contém as principais funções do editor.
- **área de trabalho:** nela será desenvolvido o trabalho de edição.
- **menu de ícones:** nesta área estão dispostos os ícones dos objetos do ambiente que podem ser selecionados e incluídos na área de trabalho durante uma edição. Em uma extensão futura, será possível associar ícones a objetos utilizados com muita freqüência no ambiente para facilitar sua inclusão.
- **área de visualização:** corresponde a uma janela de visualização total do objeto, contendo uma janela de zoom, que mostra a parte do objeto sendo representada na área de trabalho.

Descreveremos a seguir o Sistema de Janelas X Windows que foi utilizado na implementação da interface com o usuário do editor.

## 4.2. O Sistema de Janelas X-Windows

Entre os diferentes requisitos que serão necessários satisfazer no Ambiente, salientamos os relacionados aos ADSs distribuídos [DLi87a,b]:

- Suporte para o desenvolvimento distribuído de sistemas;
- Suporte para multitarefas e para a execução concorrente de programas;
- Suporte para a execução distribuída; e
- Portabilidade.

Para alcançar estes requisitos e ainda proporcionar uma interface com as características descritas anteriormente, necessita-se um sistema de janelas que:

- Associe uma ou mais janelas a cada processo, permitindo que um usuário desenvolva múltiplas tarefas e que múltiplas aplicações apresentem informação numa mesma tela concorrentemente;
- Seja transparente à rede de comunicações, para permitir a execução distribuída e o desenvolvimento distribuído de sistemas;
- Possibilite que uma aplicação permaneça independente de dispositivo proporcionando portabilidade;
- Possa suportar diferentes gerenciadores de interface e uma variedade de programas de aplicação, permitindo um uso generalizado da ferramenta.

Uma discusão atualizada sobre a tecnologia aplicada em diferentes gerenciadores de janelas pode ser encontrada em [HFW86].

O Projeto A_HAND selecionou o Sistema de Janelas X-Windows [SGe86, Mu88], desenvolvido no MIT a partir de 1984 para o Projeto Athena e o Sistema Argus, num grande esforço de padronização, com o apoio da Digital Equip. Corp. e da IBM. Posteriormente outras entidades se juntaram ao projeto, entre elas a HP, a SUN Microsystems, a Universidade de Berkeley e a de Stanford, formando o X-Consortium [Tre88].

Além das características mencionadas como necessárias, o X-Windows proporciona as seguintes:

- Foi implementado em uma ampla variedade de monitores gráficos;
- Permite usar janelas superpostas, e enviar dados de saída para janelas parcialmente ocultas;
- Suporta uma hierarquia de janelas redimensionáveis, e permite que uma aplicação controle várias janelas simultaneamente;
- Provê um suporte de alta qualidade e alto desempenho para texto, gráficos sintéticos 2D, e imagens;
- É extensível;
- Permite trabalhar em diferentes níveis de abstração na definição de uma interface com o usuário; e
- Execução distribuída das aplicações com independência de máquina e de sistema operacional, permitindo assim que as janelas sejam apresentadas em qualquer dispositivo de visualização (monitor) da rede local.

O X-Windows tem-se convertido no sistema de janelas padrão para estações de trabalho UNIX, e está sendo adotado pela grande maioria dos fabricantes de estações de trabalho e das grandes "software houses", como o protocolo padrão para a definição e o gerenciamento de janelas [Tre88], inclusive para as versões multitarefa do DOS®[Moo90, DES90].

Salientamos que desde o ano de 1987, quando foi oficializada a criação do X-Consortium, são distribuídas uma ou duas vezes por ano novas versões ou "releases" oficiais do X-Windows, com fonte e documentação, totalizando mais de 110 Mbytes de código fonte. Atualmente o X-Consortium incentiva o transporte do sistema para outras arquiteturas, através da distribuição gratuita dos fontes.

X-Windows (X) define dois tipos de programas: servidores e clientes. Um servidor de X é o programa que manipula as imagens no monitor e que concentra as ações do usuário no teclado e no "mouse". Os clientes são as aplicações do usuário que usam as facilidades de X e os programas utilitários de X. Neste esquema sevidor-cliente os clientes enviam comandos aos servidores e deles recebem eventos, correspondentes às ações do usuário. Existe um cliente especial chamado de Gerenciador de Janelas que determina qual das janelas receberá as entradas, em caso de não ser a apontada pelo cursor do mouse, além de oferecer ao usuário acesso à movimentação e redimensionamento das janelas.

O sistema X-Windows não impõe nenhum estilo de interface com o usuário, mas foram desenvolvidas diversas interfaces gráficas com o usuário (IGU) a partir das facilidades do X. A combinação de um conjunto de objetos complexos de interface com o usuário, chamados "Widgets", e da biblioteca de métodos específicos para estes "widgets", chamada "Intrinsics", foram os chamados "toolkits", que utilizam o paradigma de programação orientada para objetos. Entre os "toolkits" mais conhecidos podemos mencionar:

- **Athena Toolkit**, desenvolvido pelo projeto Athena no MIT e que foi o adotado para o desenvolvimento da IGU do Editor;
- **Andrew Toolkit**, desenvolvido na Carnegie Mellon University;
- **OSF/Motif**, desenvolvido pela Open Software Foundation e apoiado pela IBM, que pode se converter no padrão de IGU para X;
- **OpenLook**, desenvolvido em conjunto pela AT&T e SUN. O OpenWin, através do Xview "toolkit", é a implementação deste paradigma de IGU realizado pela SUN e o Xol "toolkit" é a implementação da AT&T.
- **XVT Toolkit**, é um conjunto de bibliotecas que permitem portar aplicações geradas para X para outros sistemas como o MacOS, o Windows ou o OS/2. Permite mapear chamadas genéricas ao "toolkit" em funções usadas pelo sistema alvo.

### A Biblioteca de "Intrinsics"

Esta biblioteca provê um conjunto de rotinas das quais os "widgets" dependem. Administra a criação, remoção e gerenciamento dos "widgets" assim como a manipulação de mensagens de eventos. O Athena Toolkit implementa seu "Intrinsics" na biblioteca Xt, a qual proporciona todos os métodos para administrar os "widgets", e determina a sua aparência na tela.

### A Biblioteca de "widgets"

Esta biblioteca de objetos abstratos da Version 11 Release 4 possui os seguintes "widgets":

Command Button, Label, Text, Scrollbar, ViewPort, Box, Vpaned, Form, Dialog, List, Grip, Menu.

Uma IGU pode ser fácil e rapidamente implementada num cliente de X, utilizando um conjunto destes "widgets".

Um panorama mais completo dos "toolkits" e suas características pode ser encontrado em [OR89].

## 4.3. Os Estadogramas ("Statecharts")

A notação adotada para descrever o funcionamento do Editor está baseada nos *estadogramas* ("statecharts") proposta inicialmente por D. Harel [Ha87] para representar especificações funcionais de sistemas reativos em termos de diagramas de estados suportando definições hierárquicas. Entre as diversas opções que tínhamos [Gr86] para representar a especificação da interface com o usuário e da semântica do editor da LegoShell, escolhemos os "statecharts" por permitir a descrição de estruturas hierárquicas e modulares.

Os "statecharts", através de sua linguagem gráfica de "blobs", consistem em uma extensão do diagrama de estados convencional abrangendo conceitos de hierarquia, concorrência e comunicação. Deste modo, implementa as noções de profundidade e ortogonalidade.

Desenvolvemos, em conjunto com Antônio G. Figueiredo Filho, a especificação de uma extensão dos "statecharts" que nos permitiu definir mais diretamente a semântica do editor da LegoShell. Ela consiste em uma transição especial chamada "History with Return", que permite, uma vez finalizado corretamente um estado, que se possa voltar ao estado anterior, desempilhando-o da lista de estados prévios [FLi90, Fig90]. Fizemos uso de estados superpostos na especificação pela vantagem de reduzir a quantidade de "blobs" necessários para especificar um conjunto de estados e usamos o motor de "statecharts" desenvolvido por Figueiredo pela sua facilidade de implementar diretamente os "blobs" que especificavam a semântica de nosso editor.

Na seção 6.1 veremos alguns exemplos da aplicação dos Statecharts para a definição dos estados e da semântica do editor. A descrição completa da semântica do editor usando "statecharts" encontra-se no apêndice A.

## 4.4. Arquitetura do Editor

Uma construção ascendente do editor nos permitiu, como em sistemas operacionais, partir de um "kernel" que atue diretamente sobre a representação interna dos objetos na memória, para construir funções mais complexas que serão invocadas pelo editor em diferentes níveis de abstração. Esses níveis são:

**a. Funções topológicas;**
**b. Funções de edição topológica;**
**c. Funções geométricas;**
**d. Funções de representação;**
**e. Funções de edição gráfica;**
**f. Primitivas do Editor;**
**g. Máquina de "statecharts";**
**i. Interface com o usuário.**

Como vemos na Figura 4.3, acima da Representação Interna temos o Núcleo Topológico que agrupa as funções **a** e **b**. Acima do Núcleo Topológico e compartilhando a Representação Interna, o Núcleo de Edição Gráfica que agrupa as funções **c**, **d** e **e**. Interagindo com este se encontra a máquina de "statecharts" **f** e as primitivas do editor **g**, que implementam a semântica e a memória de estados do editor. Finalmente a camada da interface com o usuário **i** que interage com a máquina de "statecharts".

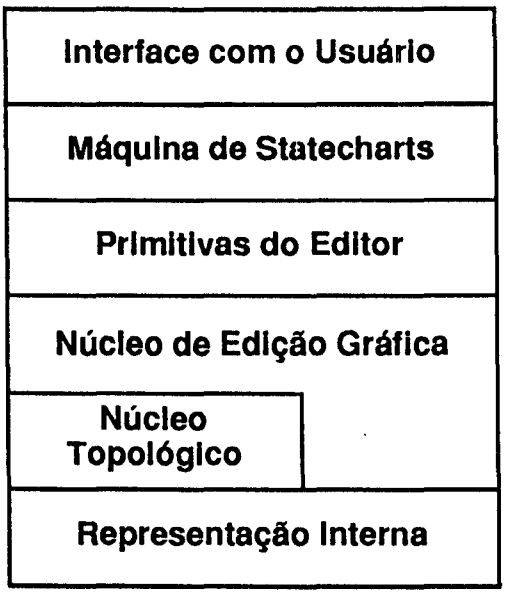



**Figura 4.3** : Arquitetura do Editor da LegoShell

No capítulo 5 descreveremos as funções do Núcleo Topológico. No capítulo 6, descreveremos também os estados do editor gráfico usando Statecharts. Indicaremos as transições em função de eventos e para descrever a semântica de cada estado usaremos as funções definidas nos níveis anteriores. No capítulo 7 definiremos a interface com o usuário usando uma associação *ação_do_usuário* com *evento*.

# Capítulo 5

# O NÚCLEO TOPOLÓGICO

Neste capítulo definiremos a representação interna, as funções que permitem manipulá-la e faremos algumas considerações sobre aspectos particulares da implementação do Núcleo Topológico.

## 5.1. Definição da Estrutura Interna

A primeira dificuldade na identificação dos objetos envolvidos em um ambiente é obter um modelo adequado. O objetivo deste modelo é formar uma base comum de conceitos que nos permita visualizar de forma mais clara o nível dos objetos manipulados em determinado contexto, além de estabelecer uma linguagem comum através da qual o núcleo topológico e as outras camadas do editor e do ambiente se comuniquem.

Utilizaremos uma estrutura de tipos baseada na proposta de Cardelli [Car89], que divide os objetos em: valores, tipos e categorias, ou níveis 0, 1 e 2, respectivamente. Categorias são meta-tipos, isto é, descrevem famílias de tipos, assim como tipos, em linguagens de programação, descrevem coleções de valores. Para mantermos a

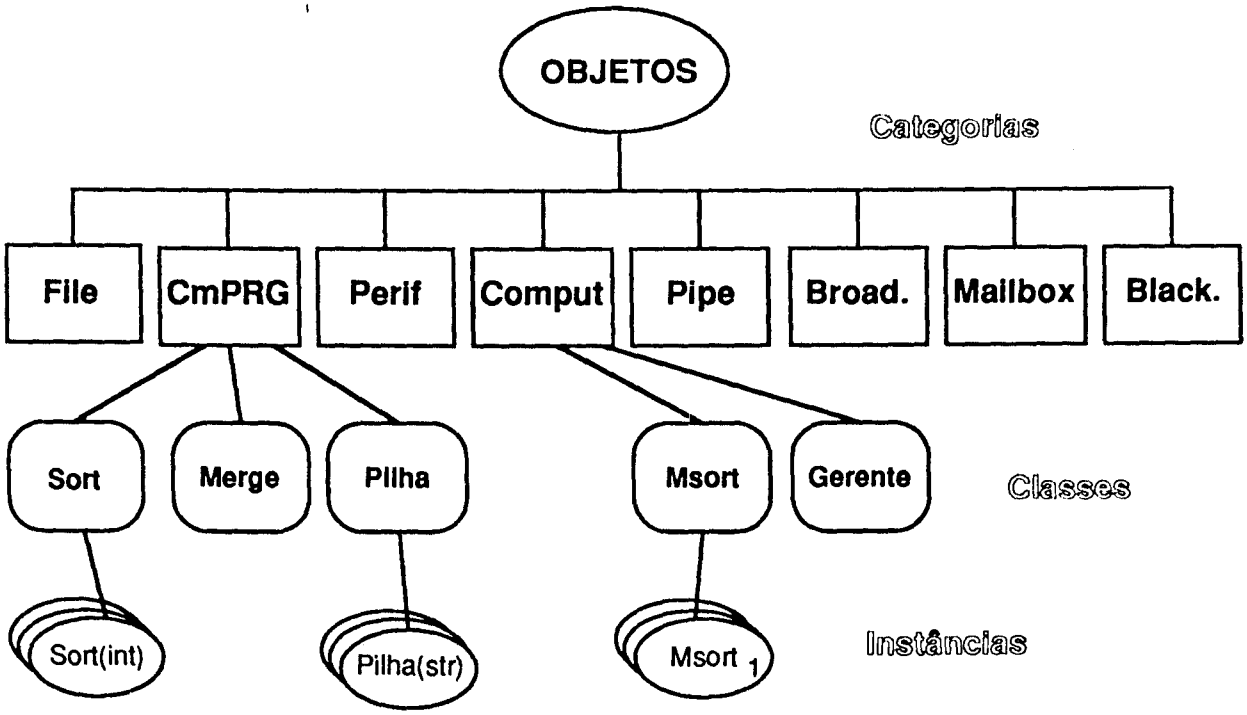



**Figura 5.1**: Categorias e Classes dos objetos manipulados pelo ambiente.

coerência em relação ao projeto como um todo, adotaremos a nomenclatura usual do paradigma de orientação a objetos, de instâncias, classes e categorias, porém mantendo a semântica proposta por Cardelli.

Segundo Cardelli, a existência do terceiro nível "é importante porque inclui a noção de computação de tipos". Analogamente, podemos dizer que os objetos manipulados por um sistema são categorias e que suas computações são os relacionamentos que as interligam. Assim, modelar o ambiente em um nível bastante abstrato seria identificar seus objetos primitivos, isto é, suas categorias. As classes seriam, então, obtidas através de especializações dessas categorias, e as instâncias através de instanciações das classes. Por exemplo, um programa Cm é uma categoria do ambiente A_HAND. Criando uma classe `"Sort"` e outra `"Merge"`, estaremos especializando a categoria programa Cm e, conseqüentemente, criando novas classes. Supondo que estas classes possuam como parâmetros o tipo a ser ordenado, `Sort (int)` e `Sort (char)` são diferentes instâncias da classe `Sort` (Figura 5.1).

Uma vez que o ambiente comportará controle de versões e configurações, é importante situarmos o nível na qual elas trabalharão. Claramente, será no nível de classes, já que são elas que definem os objetos que realmente existirão no ambiente. Portanto, as instâncias serão criadas a partir de uma determinada versão de uma classe.

As características de instância que desejamos apreender e preservar são aquelas que nos permitem referenciá-la univocamente. Incluímos nela, portanto, as referências ou os valores que nos permitam recriar aquela instância de classe, como por exemplo a posição, representação, valores dos parâmetros e das portas etc.

Para identificar os atributos das categorias, classes e instâncias, desenvolvemos um trabalho em conjunto com Carmem Satie Hara. Analisamos as possibilidades de se utilizar um gerenciador de base de dados orientado para objetos[Har90], ou de se projetar estruturas "ad hoc" (simplificadas) para manipular os objetos da LegoShell diretamente em memória, como fizemos neste editor. Tivemos também que decidir se o ambiente guardaria uma cópia do objeto completo com todos os seus níveis de abstração ou manteria um nível só, correspondente ao objeto em edição e os seus componentes. Esta última foi a solução adotada, e ela teve implicações diretas na estrutura usada para representar os objetos.

Como estruturas primitivas definimos listas duplamente ligadas e pilhas de elementos genéricos, com as primitivas que as manipulam. Isto permitiria manter um isomorfismo a nível de chamada de função como se estivéssemos invocando funções em Cm. Adotamos, de comum acordo com Hara, a mesma divisão de atributos de classe e de instância para cada categoria, o que permitiria a implementação do editor com nossa representação interna , ou usando o BDOO Damokles [Abr88], ou qualquer outro BDOO, para o qual fosse portada nossa definição de objetos da LegoShell.

Portanto, os objetos da LegoShell têm uma estrutura que inclui:

**cabeçalho**, identificando o objeto;
**núcleo**, onde mantemos os atributos da classe;
**informação de instância**, onde mantemos os atributos da instância de classe;
**informação de representação**, que é também atributo da instância mas que colocamos em separado para facilitar a codificação das funções de apresentação.

Também diferenciamos entre a informação do objeto em edição e a dos objetos componentes, correspondendo a primeira à da classe e a segunda à das instâncias.

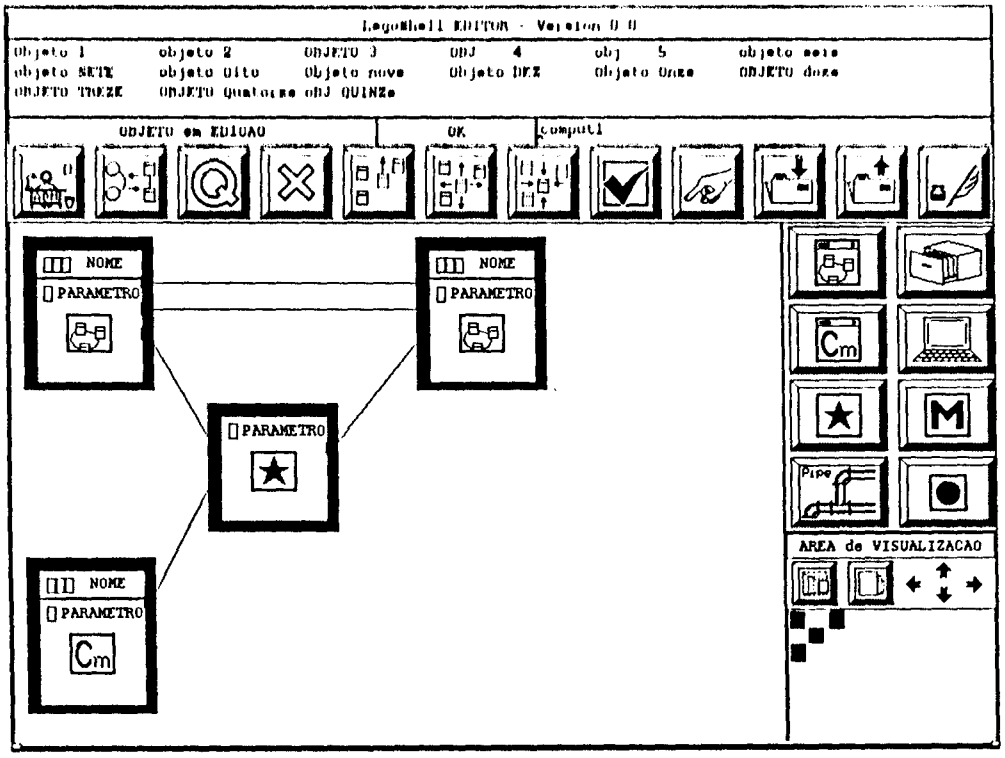



**Figura 5.2** : Um exemplo de computação sendo editada

Na Figura 5.3 vemos a representação da estrutura do exemplo da Figura 5.2. A estrutura resultante é bastante complexa pela grande quantidade de informação que deve ser guardada, especialmente a correspondente à instância. A informação de instância deve ser armazenada toda ela por referências, já que estamos manipulando um nível de abstração só.

## 5.2. Funcionalidade

Dada a representação interna que definimos acima, podemos, neste primeiro nível, criar ou destruir um objeto, dar-lhe uma identidade, e carregá-lo ou armazená-lo no disco. Se definirmos agora um componente, poderemos então incluí-lo ou excluí-lo da lista de componentes do objeto em edição, conectá-lo ou desconectá-lo (se for um "pipe"), perguntar se pertence à lista de componentes, etc.

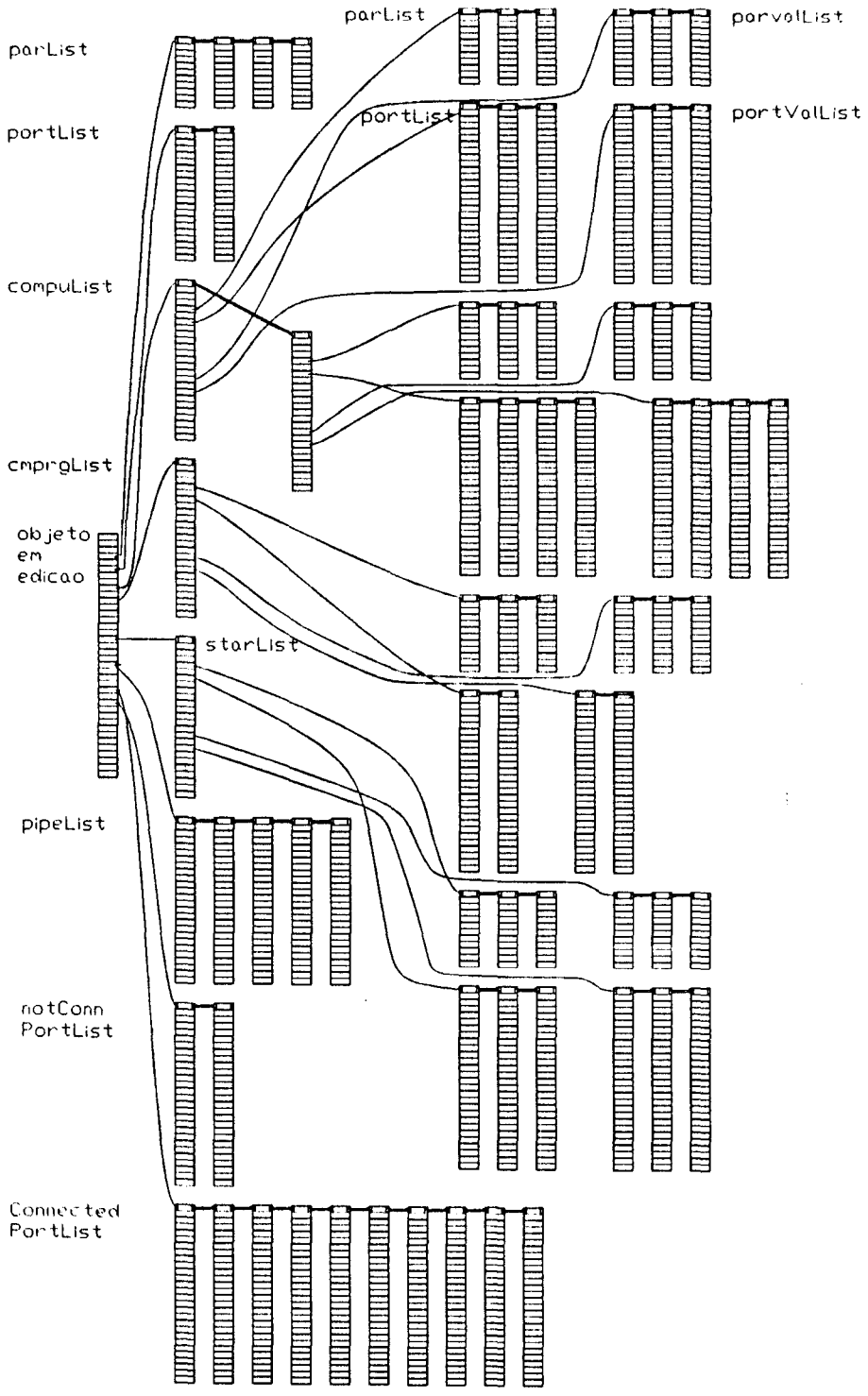



**Figura 5.3 :** Estrutura interna do objeto em edição da Figura 5.2

Para isto usamos as

## Funções topológicas:

- Create-Obj;
- Load-Obj;
- Store-Obj;
- Include-One-Compo;
- Exclude-One-Compo;
- Connect-One-to-One;
- Disconnect-One-Port;
- Get-Obj-Id;
- Get-Obj-Name;
- Get-New-Obj-Id;
- Get-Port-Id;
- Is-Connected;
- Is-Known;
- ErrorT-Handler;
- WarningT-Handler.

Até aqui só há um nível de abstração.

Se agregamos agora o universo de objetos e componentes, externo à representação interna, pertencente ao ambiente, percebemos a necessidade de adicionar os seguintes conceitos: conjuntos de componentes, parâmetros do objeto em edição e dos componentes e níveis de abstração diferenciados. Será necessário então contar com as mesmas funções definidas no nível topológico definidas agora para conjuntos, funções de edição dos parâmetros do objeto e dos componentes e funções que permitam o trânsito pela árvore de abstração dos objetos. Incluímos também neste nível a facilidade de edição que permitirá criar um novo objeto a partir de um subconjunto de componentes de um outro objeto, e sua recíproca (abstração e expansão).

Isto foi implementado nas

## Funções de edição topológica:

- Edit-One-Compo;
- Edit-One-Compo-Par;
- Edit-One-Obj-Par;
- Edit-One-Obj-Conf;
- Abstract-Computation;
- Expand-One-Computation;
- Include-Set;
- Exclude-Set;
- Expand-Set;
- Consist-Set;
- Reconnect-From-To;
- Connect-Port-Set;
- Disconnect-Port-Set;
- ExitTL;
- QuitTL;
- UndoTL;

- SaveTL-State;
- RestoreTL-State;
- Is-In-Edition.

Em cada um destes níveis foram definidas as funções de alarma e erro correspondentes.

## 5.3. Aspectos Particulares da Implementação

Todas as estruturas foram concebidas com o objetivo de implementá-las em Cm. Podemos então dizer que as funções que detalhamos previamente são polimórficas, no sentido dado a este termo em Cm, e que encapsulam a estrutura dos objetos da LegoShell, fazendo que eles só possam ser manipulados através destas funções. Por exemplo, a função Include-One-Compo tem como um de seus parâmetros a categoria de componente, o que equivale em Cm a especificar o tipo parametrizável.

Seguimos, sempre que possível, uma política de programação orientada para objetos, ou mais precisamente uma política de programação em Cm.

# Capítulo 6

# O EDITOR GRÁFICO

Neste capítulo descreveremos em detalhe a semântica do editor, os seus estados, usando "statecharts", e faremos algumas considerações sobre aspectos particulares da implementação do Núcleo de Edição Gráfica e da máquina de "statecharts".

## 6.1. Os Estados do Editor

Foi demostrado por [Gr86] que os diagramas de estado, entre os quais se encontram os "statecharts" [Ha87, Ha88], podem ser traduzidos para gramáticas de atributos e também para a especificação de eventos usando a linguagem Events. As gramáticas de atributos e os diagramas de estado estendidos possuem um poder de expressão equivalentes, mas escolhemos os "statecharts" pois facilitam visualizar as mudanças de estado assim como a completitude da especificação de cada um dos estados.

Usando então esta ferramenta definimos a semântica do editor. (Figura 6.1). As funções que definimos para o ambiente e o editor são:

| | | | |
|---|---|---|---|
| ABSTRACT | CREATE | LOAD | SELECT |
| ACTIVATE | EDIT | MOVE | STORE |
| ADM. | EXCLUDE | PAUSE | TURN OFF |
| CLEAR | EXIT | QUIT | TURN ON |
| COMPILE | EXPAND | REDRAW | UNDO |
| CONSIST | INCLUDE | | |

Analisaremos a seguir cada função em detalhe

## 6.2. Funcionalidade

O editor, ao ser invocado, nos coloca diretamente no estado de criação CREATE. A partir deste estado inicial, descreveremos quais são as alterações de estado produzidas pela invocação das diferentes funções.

### CREATE – Criação

Criar uma computação consiste conceitualmente em definir quais são seus componentes e como eles estão conectados. No processo de criação poderemos ter, portanto, ações de inclusão de novos componentes, modificação de posição dos componentes, edição da configuração do objeto, edição dos parâmetros do objeto e dos componentes e o armazenamento do objeto criado.

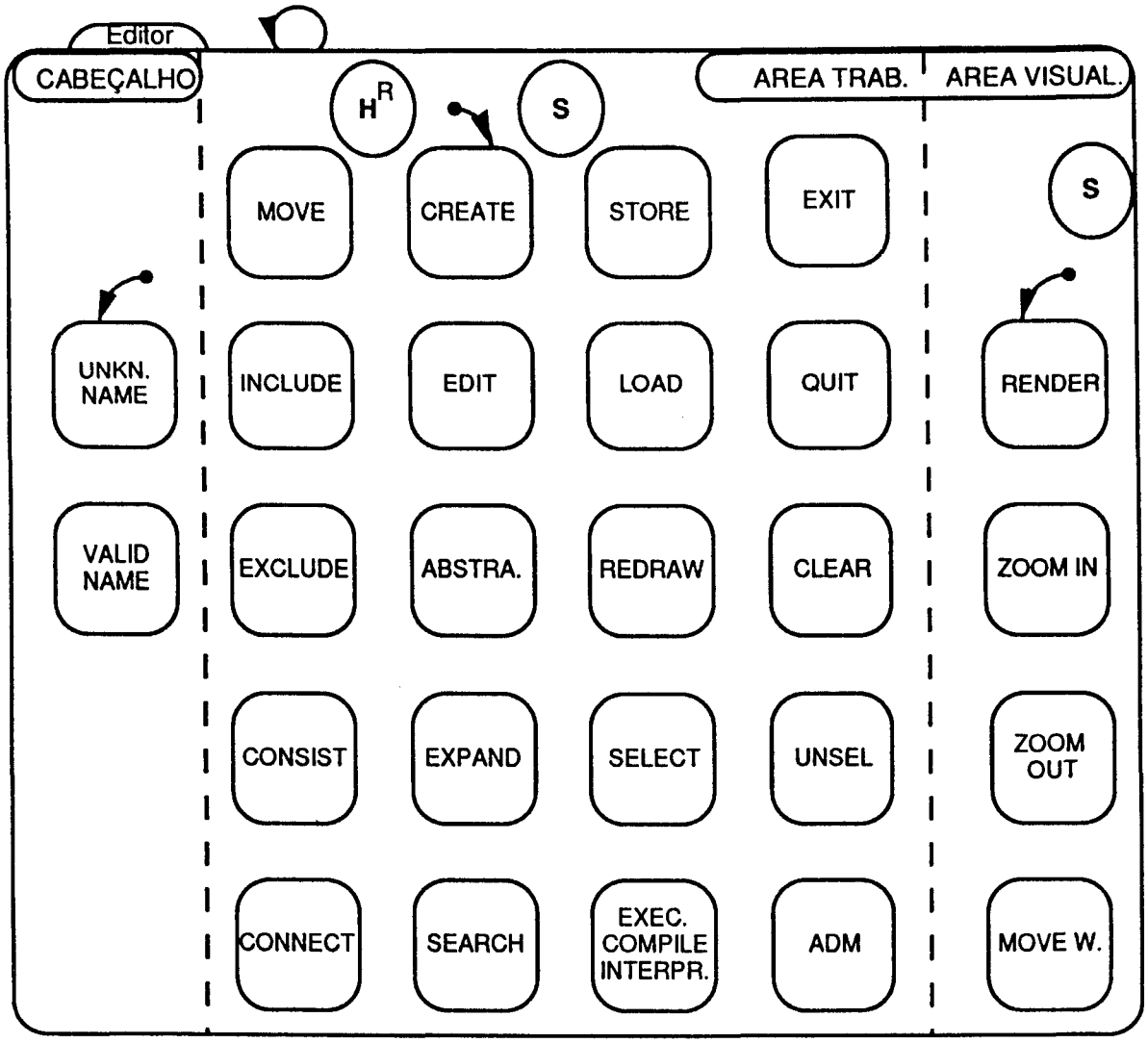



**Figura 6.1 :** "Statechart" do Editor (nível 1)

Mais formalmente, a seqüência de ações no CREATE é:

início
INCLUDE +
[MOVE ] *
[EDIT Componente] *
[EDIT Configuração] *
[EDIT Parâmetros Componente] *
[EDIT Parâmetros Objeto] *
Atribuição de NOME +
STORE
fim

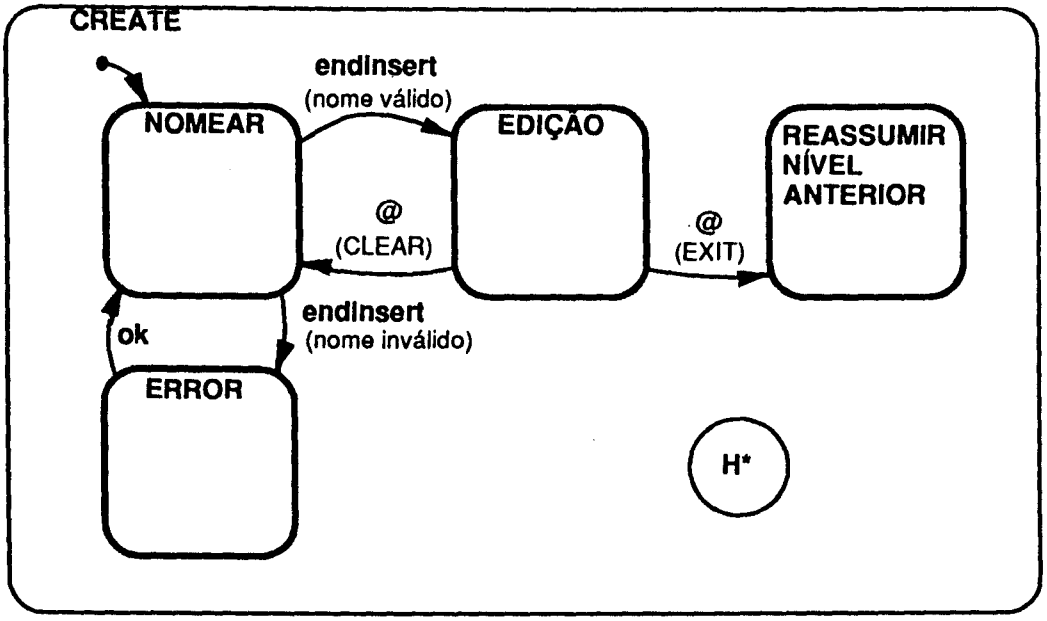



**Figura 6.2**: Statechart da função CREATE

Ao se atribuir um nome a um objeto novo, é verificado se dentro do ambiente já existe um objeto com o mesmo nome, para garantir sua unicidade. Isto foi uma decisão de projeto, pois não existem, no protótipo, versões alternativas de um objeto. A unicidade da linha de evolução histórica do objeto está então assegurada pela existência de um universo único de nomes para todos os objetos de um mesmo tipo.

Uma vez determinado que o nome é válido, o módulo cv agregará o número de versão e o número de edição. Como o objeto acaba de ser criado estes serão iguais a 1. Como regra geral o novo número de edição privada é dado quando se carrega o objeto para edição. Supondo que a última edição privada armazenada fosse a n-ésima, estaríamos sempre editando a (n+1)-ésima. A administração dos números de versões é responsabilidade do sistema controlador de versões (cv). O editor apenas invoca as funções do cv com parâmetros que informam o tipo de modificação realizada no objeto e a categoria da nova versão e têm garantida a criação das versões apropriadas.

Como a tarefa de criação ou de edição de uma computação ou de um objeto do ambiente pode durar várias sessões, veremos a seguir quais as opções de armazenamento que o ambiente oferece.

## Armazenamento

Ha três funções relacionadas com o armazenamento dos objetos do ambiente: STORE, EXIT e QUIT.

Se o usuário terminou de criar ou editar um objeto, ele pode querer armazená-lo, criando assim um novo número de edição privada, mantendo-se no mesmo nível. Para isto usará a função STORE. Para armazenar o objeto e também sair do nível de edição para um nível mais alto, ou para fora do ambiente, usará a função EXIT. E se quiser abortar a edição, abandonando este nível para um nível superior, usará a função QUIT.

Um objeto pode ser armazenado de forma incompleta, mas não pode ser congelado se sua consistência não foi verificada.

## STORE – Função de Armazenamento

Esta função de armazenamento é utilizada no caso de desejarmos guardar o estado do objeto em edição e prosseguir sua edição. Assim, um novo número de versão privada é criada e nos mantemos no mesmo nível de edição.

## EXIT – Função de Armazenamento e Saída

A função implica em armazenamento e término de edição. O nível ao qual a edição retorna é retratado pelas opções disponíveis:

EXIT do ambiente;
EXIT do nível de edição.

No primeiro caso, o estado do ambiente, incluindo toda a cadeia de sessões correntes, é armazenado para que a edição possa ser reassumida no mesmo ponto onde foi suspensa.

A segunda opção consiste em armazenar o objeto, criando uma nova versão, e subir um nível de edição. Caso não haja edições suspensas, isto é, se o nível do objeto editado for o primeiro, entramos no modo criação de um novo objeto.

## QUIT – Abortar

A função QUIT significa abortar a edição ao nível atual, sem salvar o objeto que estava sendo editado, e voltar ao nível superior. Pelas suas conseqüências, exige sempre uma confirmação.

## INCLUDE – Inclusão de Componentes

A inclusão de um componente em uma computação corresponde à escolha do tipo no menu de ícones (computação, programa, periférico, arquivo ou conector), à sua identificação, no caso de objetos de conteúdo semântico e, finalmente, ao seu posicionamento dentro da computação. Quanto à identificação dos objetos de conteúdo semântico, é possível fazê-lo diretamente fornecendo o seu nome ou utilizando a facilidade do ambiente de listar os objetos segundo a classificação abaixo:

- **Uso Geral:** objetos classificados como sendo de uso geral, como por exemplo, os Programas Cm Data, Array e Pilha.
- **Projeto (x):** objetos classificados como de projeto, cujo proprietário seja o projeto x ou um membro de x. Se o usuário que efetua a edição é um membro de x, todos os objetos que satisfazem a regra acima serão listados. Caso contrário, somente aqueles que possuem versões liberadas serão apresentados.
- **Particular:** objetos que estão sendo editados pelo usuário corrente.

Devemos ressaltar, ainda, que apenas computações e programas Cm consistentes podem ser incluídos em uma computação. A inclusão de "pipes" na computação deve ser posterior à inclusão dos objetos interligados pelo mesmo. Para isto, é necessário apontar o ícone correspondente ao "pipe" e, então, indicar as portas de entrada e saída as quais interconectará.

Durante a criação ou edição de uma computação de LegoShell teremos que incluir novos objetos, assim como se agregam tijolos para se construir uma parede. Isto será possível através da função INCLUDE.

A fim de dar flexibilidade durante a edição e criação de computações foram incluídas duas opções:

- BUSCA;
- CRIA.

A primeira é uma invocação à função de recuperação e, a segunda, uma invocação à função de criação. Como todos os conectores da mesma categoria têm um comportamento igual, não é necessário atribuir-lhes um nome. Devemos levar em conta que, para que a criação de um objeto seja válida, nele deveremos ter incluído pelo menos um objeto de conteúdo semântico.

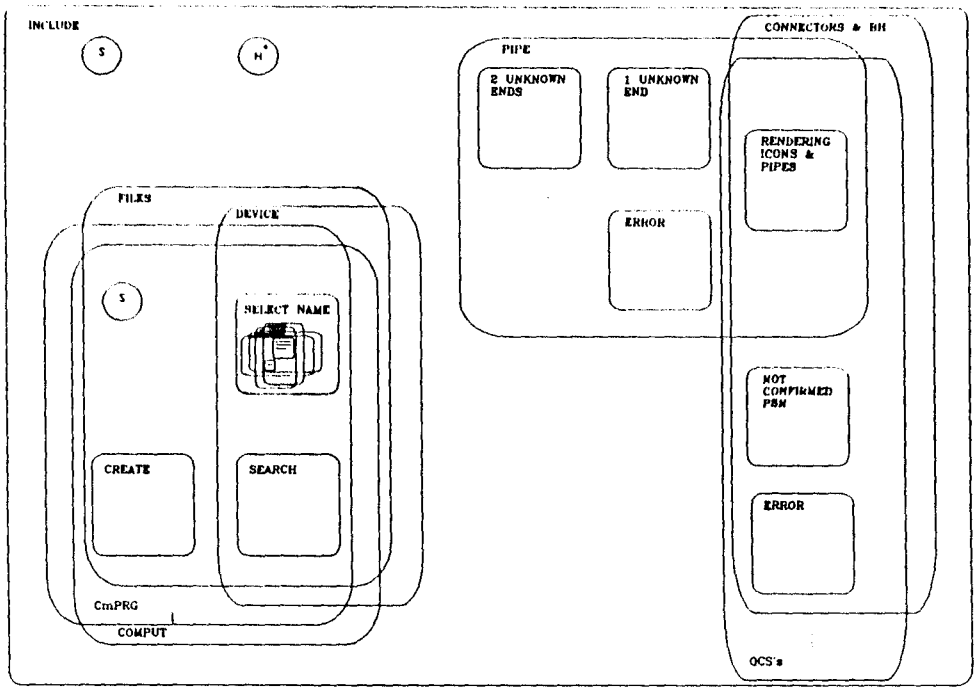



**Figura 6.3:** Statechart da função INCLUDE

A seguir veremos as funções que permitem modificar os objetos componentes de uma computação.

## Alteração

Como as computações de LegoShell possuem uma natureza topológica, a posição na qual os objetos são dispostos na tela não tem relevância para o seu significado semântico.

Durante o processo de criação, e mais genericamente de edição, há dois níveis nos quais o usuário pode alterar o objeto em edição:

- O nível geométrico;
- O nível semântico.

Pode ser necessário reposicionar um objeto depois de tê-lo incluído numa computação. Para isto teremos a função MOVE. Para efetuar alterações do significado semântico dos objetos, teremos disponíveis as diferentes opções da função EDIT.

## MOVE – Alteração Geométrica de Componentes

Esta função possibilita que a posição de objetos dentro de uma computação possa ser alterada. O "pipe" é uma exceção a esta facilidade, pois, a princípio, este será apenas uma linha reta ligando as portas relacionadas. Em função disto quando for confirmada a posição do objeto componente, se produzirá o redesenho da área de trabalho e da janela de visualização total do objeto[1].

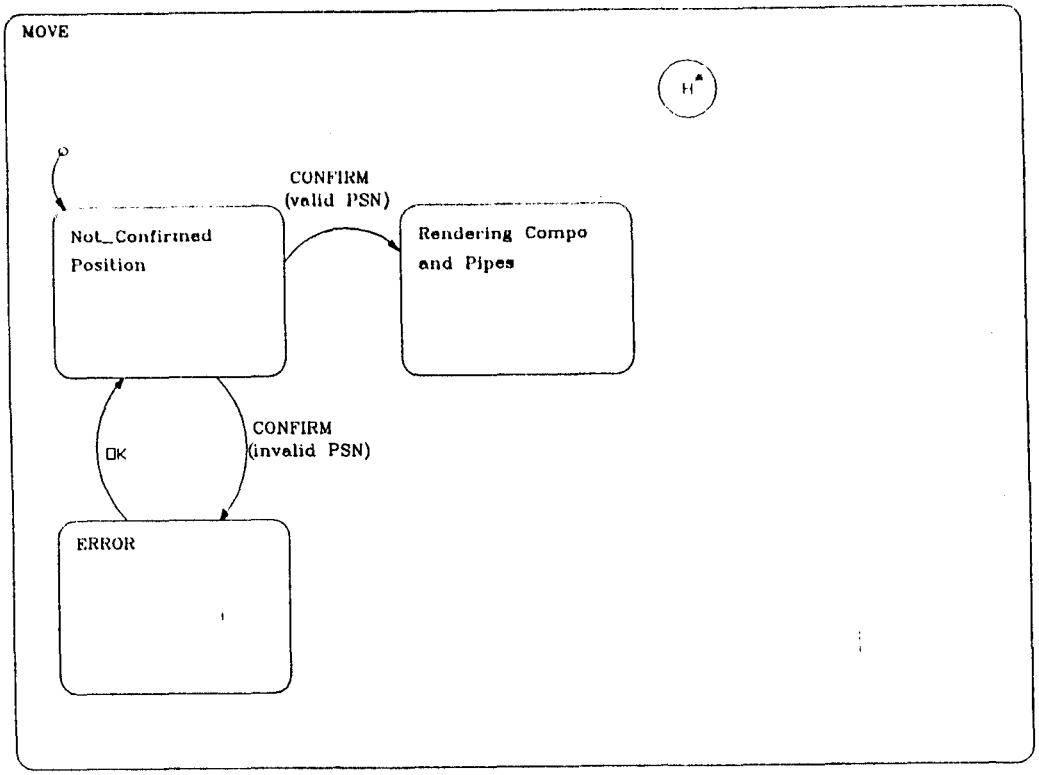



**Figura 6.4:** Statechart da função MOVE

## EDIT – Alteração Semântica de Componentes

A alteração dos componentes pode se dar nos planos de interface, conteúdo e versão. Assim, são seis as opções desta função:

- edição de um objeto componente;
- edição da configuração;
- edição dos parâmetros dos componentes;
- edição dos parâmetros do objeto em edição;
- edição do ícone do objeto em edição;
- edição do ícone de um objeto componente.

---

[1]Em breve o layout dos "pipes" será efetuado por um roteador especificamente projetado para as necessidades de um editor gráfico como o da LegoShell. Ele possibilitará o roteamento dinâmico com especificação online de restrições pelo usuário.

Esta função é complementada com a função LOAD, que permite carregar no editor um novo objeto para ser editado, habilitando um novo nível de edição.

Como veremos a seguir, ao descrevermos as opções de edição, se editarmos um componente do objeto em edição (EDIT de componente) ou um outro objeto (LOAD), desceremos um nível de edição, enquanto que nas outras opções, nos manteremos no mesmo nível de edição.

## LOAD – Edição de um Objeto Composto

A função LOAD corresponde à edição de um objeto, caso ele já exista, ou criação do mesmo, caso contrário. Este objeto pode ser um programa Cm ou uma computação. No primeiro caso, um editor sensível à sintaxe será invocado para desempenhar sua função; caso contrário, o próprio editor da LegoShell executará a edição.

A edição propriamente dita, com direito a escrita, será permitida somente se o usuário estiver utilizando a versão corrente do objeto, isto é, se a versão utilizada for uma edição. Assim, ao se retornar da edição, a versão alterada passará automaticamente a ser utilizada dentro da computação. Esta situação é bastante comum no teste ou alteração de novos objetos.

Se a versão utilizada não for uma edição, o usuário terá apenas direito de leitura sobre o objeto. Caso alguma alteração sobre o mesmo seja necessária, ele deverá utilizar a função LOAD e, então, alterar a configuração da computação para que a nova versão seja incorporada.

Os direitos de acesso de leitura e escrita definem as funções disponíveis a cada usuário. Além disso, se um objeto já estiver sendo editado por um usuário, o mesmo não pode criar uma nova sessão de edição para o mesmo objeto, podendo apenas visualizá-lo.

Como já foi dito, apenas versões do tipo edição podem ser atualizadas. Portanto, se a função LOAD for invocada e o usuário possuir direito de escrita sobre o objeto, automaticamente uma nova edição particular é criada. Neste caso, a configuração associada à computação é novamente resolvida e os objetos componentes que modificam sua versão são apresentados para que o usuário decida se estas serão ou não utilizadas. Por exemplo, se a entrada da configuração indica ULTLIB para o objeto componente A e uma nova versão da mesma é liberada, então o usuário terá a opção de incorporar esta nova versão à computação.

Durante o processo de edição, os componentes da computação serão incluídos e ligados através de conectores. Porém, na inclusão de objetos compostos, não basta qualificá-los apenas com o nome, apesar deste ser unívoco em todo o ambiente. Como sua representação gráfica depende do número de portas, é necessário que antes definamos a versão do componente a ser utilizada. Portanto, é preciso que a toda edição, mesmo que se trate de uma computação nova, esteja associada uma configuração. Convencionou-se que, no caso de criação de um objeto, a configuração a ser utilizada será a "default" de mais baixo nível. Se o objeto já existir, sendo a edição, portanto, uma derivação da última versão, a configuração a ser associada será a mesma utilizada pela versão da qual se originou.

Cabem aqui as mesmas considerações que fizemos ao descrever a função de criação, no que diz respeito à validade dos nomes e às verificações necessárias.

Ao se atribuir o NOME à variável CLASSE, será verificado se o objeto não está sendo editado. Se este for o caso, somente será permitida a visualização do mesmo.

É adequado lembrar que o ambiente proporcionará um editor orientado para sintaxe para editar Programas Cm. Se formos editar um arquivo sem uma estrutura conhecida, usaremos algum editor de texto semelhante ao GNU Emacs [St87].

Internamente o ambiente incorporará este objeto à Representação Essencial, de forma tal que ele possa ser manipulado corretamente pelo editor.

## EDIT de um Objeto Componente

A função EDIT de componente é igual à função LOAD de um objeto do qual o ambiente já conhece o NOME.

## EDIT da Configuração do Objeto em Edição

Para fazermos a alteração da versão de cada componente de uma computação ou das dependências de um programa Cm, utilizaremos a função de edição de configuração. Seguindo o modelo de controle de versões e configurações do ambiente já descrito, existem 4 tipos de entrada:

- direta,
- direta com rótulo, que pode ser ULTLIB, ULTEFET ou CORRENTE,
- indireta, e
- de inclusão,

onde os três primeiros são relacionados com um objeto componente e o último com a computação ou programa como um todo.

A configuração assim determinada passará então a ser a configuração ativa do objeto, isto é, aquela que determina a versão dos seus componentes.

Para ativar outras configurações deverá ser usada a função ACTIVATE.

## EDIT dos Parâmetros de um Componente

Para os três tipos de parâmetros existentes (parâmetros da classe, chaves de seleção e parâmetros de execução), será permitido que se defina o seu valor ou que este seja exportado como parâmetro da computação da qual é componente. Esta exportação pode se dar de três formas:

- passando a ser parâmetro da computação com o mesmo nome que possuía no objeto componente;
- passando a ser parâmetro da computação com um nome diferente do que possuía no componente, no caso de haver parâmetros com o mesmo nome, ambos exportados;
- parâmetros de objetos distintos passando a ser um único parâmetro na computação e, portanto, com um nome único.

É importante salientar que um número será associado a cada objeto componente, o que permitirá uma referência sem ambigüidades aos diferentes componentes, mesmo que eles sejam duas cópias de um mesmo objeto.

## EDIT dos Parâmetros do Objeto em Edição

É através desta função que determinamos o valor "default" dos parâmetros exportados para a computação corrente.

No fim da edição dos parâmetros de um objeto em edição, será verificada a consistência desses parâmetros entre os componentes e o objeto em edição. Como vemos, existe uma segurança extra, que nos obriga a passar sempre por este estado de verificação cada vez que alteramos os parâmetros de um componente.

## SELECT – Função de Seleção

Há casos em que desejamos efetuar uma mesma operação, por exemplo exclusão, sobre vários objetos. Para facilitar estas ações existe a função de seleção, implementada através de uma lista circular dos objetos selecionados. Assim, é possível andar sobre esta lista utilizando os comandos anterior e próximo e realizando a operação desejada. São previstas, ainda, opções para selecionar ou liberar todos os objetos da computação.

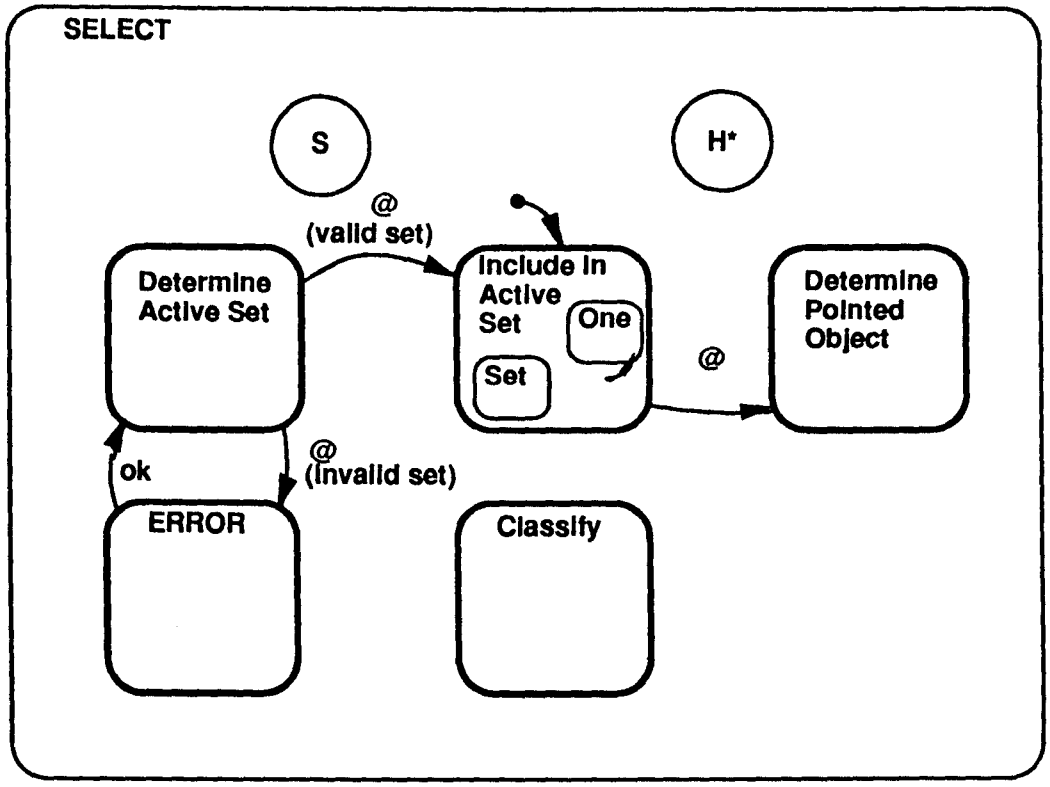



**Figura 6.5 :** Statechart da função SELECT

## EXCLUDE – Remoção de Componentes

A remoção de componentes de uma computação é feita através da função EXCLUDE. Se, eventualmente, houver "pipes" ligados ao objeto selecionado, eles serão excluídos juntamente com o objeto.

## CLEAR – Limpa a Tela

Como o nome já indica, a função consiste em excluir todos os objetos componentes da computação, mantendo-se no mesmo nível de edição, ou seja, não abandonando a sessão de edição.

## REDRAW – Re-desenhar a Tela

Genericamente significa voltar a desenhar apenas os objetos que têm que ser representados na área de trabalho. Esta função é importante durante a depuração do protótipo, já que nos permitirá eliminar objetos soltos dentro da área de trabalho.

## UNDO – Anular Última Ação

Genericamente significa voltar ao estado anterior, abandonando o estado atual e eliminando ou revertendo seus efeitos. Para cada estado particular, ela possui um significado especial.

## ABSTRACT – Abstração de Computações da LegoShell

Esta função consiste em criar uma nova computação composta pelos objetos selecionados através da função select, descrita acima, substituindo-os pela mesma na computação corrente. Ao se dar a confirmação, para cada "pipe" que ligar portas de objetos selecionados com portas de objetos não selicionados será criada uma porta na abstração preservando a ligação das portas originais. As portas que não foram ligadas serão automaticamente transferidas para o perímetro da nova computação. Desta forma, portas conectadas a objetos não selecionados, bem como portas não conectadas, passam a ser portas da nova computação (exportadas).

## EXPAND – Função de Expansão

É a função oposta da abstração, isto é, ela consiste em substituir uma computação por seus componentes dentro da computação corrente. Isto implica que a computação corrente seja redesenhada.

## CONSIST – Verificação de Consistência

A verificação de consistência através desta função é feita somente no nível de edição corrente e não inclui a geração de código. Como uma das regras de inclusão de objetos complexos em uma computação é que estes estejam consistentes, mantemos, desta forma, a abstração necessária entre os diversos níveis.

A consistência de uma computação implica, entre outras coisas, na verificação da existência de objetos ou portas não conectados, compatibilidade dos dados que transitam entre duas portas e consistência da definição de parâmetros.

A função CONSIST permitirá verificar a consistência do objeto em edição. Se o resultado da verificação for satisfatório, o atributo consistência do objeto será atualizado para verdadeiro. Se for feita qualquer modificação no objeto, seu atributo de consistência será colocado no valor falso, tendo então que repetir a operação de verificação de consistência.

## COMPILE – Função de Consistência e Geração de Código

Esta função é equivalente à função CONSIST, porém, inclui a geração de código. Esta geração não é uma operação trivial devido a alguns aspectos da execução de computações.

## Execução de Computação

Os objetos de conteúdo semântico poderão ser monitorados de maneira interativa durante a execução. Para este propósito, eles possuem dois "botões", correspondentes às quatro funções TURN ON (ligar) e TURN OFF (desligar), e PAUSE (suspender) e RESUME (ativação), as duas últimas funções compartilham alternadamente o mesmo "botão" do objeto.

Assim, no início da execução, todos os objetos estarão com seus botões TURN ON ligados, implementados através da criação de diversos processos concorrentes. Se a função PAUSE for ativada, o efeito será a suspensão da execução do processo correspondente àquele objeto, podendo ser reassumida novamente através da função RESUME. O efeito da função TURN OFF é equivalente a abortar o processo que implementa o objeto. Por exemplo, na computação *MSort* da Figura 3.1, se desligarmos um dos programas *Sort*, o processo continuará sua execução, porém apenas com um *Sort*. Os objetos desligados podem ser reinicializados através da função TURN ON, mas isto implica em criar novamente todos os processos necessários.

A definição do ambiente prevê, ainda, a possibilidade de execução de objetos através de interpretação, caso estes não tenham sido compilados. Porém, a estratégia de implementação do módulo de execução do ambiente não está totalmente acabada.

## Ativação

Para cada edição de computação existe uma configuração ativa. A configuração ativa pode ser modificada usando a função EDIT de configuração, que ao finalizar a torna a configuração ativa. Se desejamos ativar uma outra configuração para o objeto em edição podemos fazê-lo através da função ACTIVATE. Ela nos permitirá escolher uma dentre todas as definições de configuração do objeto.

Se confirmada, esta configuração fica sendo a ativa. Caso contrário, volta-se ao estado anterior, permitindo uma nova seleção.

## Adm – Funções Administrativas

As funções que permitirão o gerenciamento de projetos são:

- ativação de uma configuração já existente para o objeto em edição;
- definição da configuração "default" para um usuário, projeto ou para o todo o ambiente;
- efetivação e liberação de versões;
- funções de "tailoring", como definição do tamanho de buffer;
- "default" para os tipos de conectores e definição do número de edições particulares a serem guardadas;
- cadastramento de usuários e projetos;
- atribuição de responsabilidades;
- definição de direitos de acesso.

Voltando à definição da representação interna, e passando a um nível acima do Núcleo Topológico, descrito no capítulo 5, após definir a semântica do editor percebemos a necessidade de adicionar outros conceitos aos já definidos. O principal requerimento a este nível é a necessidade de editar os objetos através de uma representação gráfica, o que nos permite visualizar mais facilmente sua estrutura hierárquica. Temos então os conceitos de geometria, representação e as necessidades inerentes à edição gráfica.

A inclusão das funções poderia ser feita em qualquer ordem, mas escolhemos incluir em primeiro lugar aquelas que tratam a geometria, isto é a posição e as mudanças de posição.

Isto foi implementado usando as

## Funções Geométricas:

- Include-One-At;
- Include-Set-At;
- Move-One-To;
- Move-Set-To;
- Get-Obj-Position;
- Get-Port-Position.

Depois agregamos aquelas funções que tratam a representação dos objetos e componentes através de ícones.

Usamos para isto as

### Funções de Representação:

- Represent-As;
- Modify-Representation-Of;
- Iconify;
- Deiconify;
- Is-Area-Free;
- Get-Area-Occupied-By;
- Get-Object-Representation;
- Scale-Factor-Multiply;
- Iso-Expan-Repr;
- NonIso-Expan-Repr.

Finalmente no nível de edição gráfica adicionamos o cursor do apontador e o "zoom".

Para isto faremos uso das

### Funções de Edição Gráfica:

- Select-One;
- Select-Set;
- Select-All;
- Unselect-One;
- Unselect-All;
- Point-To;
- Zoom-In;
- Zoom-Out;
- Is-Available-Option.

## 6.3. Aspectos Particulares da Implementação

Um dos aspectos mais importantes decorrentes do uso da máquina de statecharts é a facilidade com que podemos armazenar os estados empilhando-os na própria estrutura definida em [FLi90]. Este mecanismo nos liberta da necessidade de manter na memoria da máquina de estados quais foram os estados anteriores ao atual visitados pela máquina. O esforço que realizamos para compatibilizar a definição de statecharts com este requerimento rendeu seus frutos, já que conseguimos um mecanismo independente e genérico para esta função. Na Figura 6.6 vemos a árvore de estados que resulta da interpretação da estrutura de "blobs" que descreve a semântica do editor.

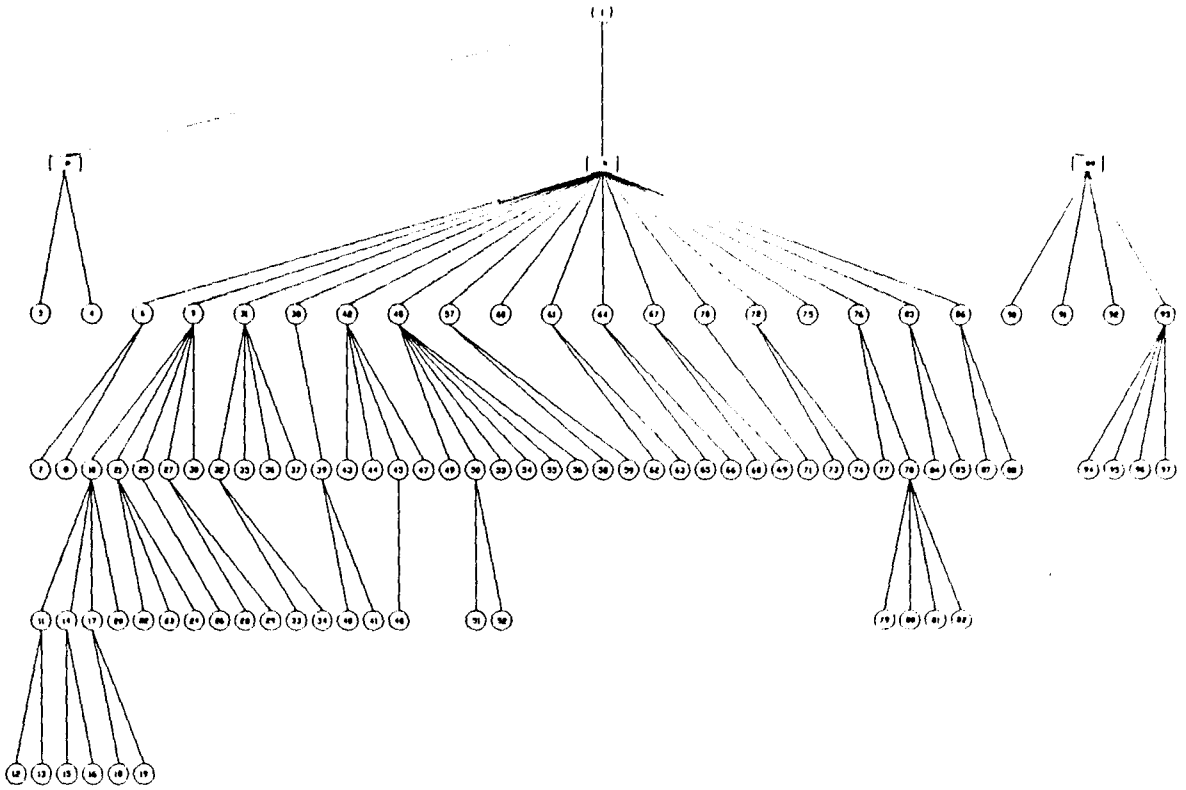


**Figura 6.6 :** Arvore de blobs do Statechart

# Capítulo 7

# A INTERFACE COM O USUÁRIO

Neste capítulo definiremos em detalhe a interface com o usuário do protótipo e discutiremos alguns aspectos particulares da implementação.

## 7.1. A Interface com o Usuário do Protótipo

Depois de avaliar diferentes interfaces com o usuário correspondentes a editores gráficos e a ambientes, adotamos a premissa de que no protótipo deveríamos implementar a interface que nos parecesse mais vantajosa, e após experimentos, tirar as conclusões que nos permitissem modificá-la convenientemente.

Outra premissa que adotamos foi que o nível de ajuda visual seria o máximo compatível com uma representação clara, já que um excesso de figuras, botões e menus confundiriam o usuário em lugar de ajudá-lo. Numa versão posterior poderíamos pensar em uma representação mais despojada, isto é, equivalente a um nível de ajuda menor.

Com este mesmo intuito, não seriam implementados os "short-cuts" que permitiriam um uso mais flexível por parte de um usuário experiente.

Adotamos então um conjunto de janelas justapostas ("tiled windows"), dispostas numa janela padrão do sistema gerenciador de janelas que estivesse sendo usado. Isto quer dizer que todas as funções do gerenciador de janelas, como por exemplo as de modificar a posição ou iconificar a janela do editor, serão preservadas. Na Figura 7.1 vemos as cinco janelas justapostas :

- **Cabeçalho:** Contém o título onde estará o nome do objeto em edição, e a pilha dos objetos sendo editados em cada um dos níveis de edição.
- **Menu de barra:** Este menu contém os botões correspondentes às principais funções do editor.
- **Area de trabalho:** Nesta janela é desenvolvido o trabalho de edição dos objetos do ambiente.
- **Menu de ícones:** Nesta área estão dispostos os ícones dos objetos do ambiente, passíveis de serem incluídos para compor uma computação.
- **Area de visualização:** Nesta área temos uma janela de visualização total do objeto, contendo uma janela de zoom, que corresponde à parte do objeto sendo representada na área de trabalho. Acima dela há um menu de ícones que permite manipular a posição e o tamanho da janela de zoom.

O indicador da posição do "mouse" ("pointing device") está representado como uma seta na área de trabalho.

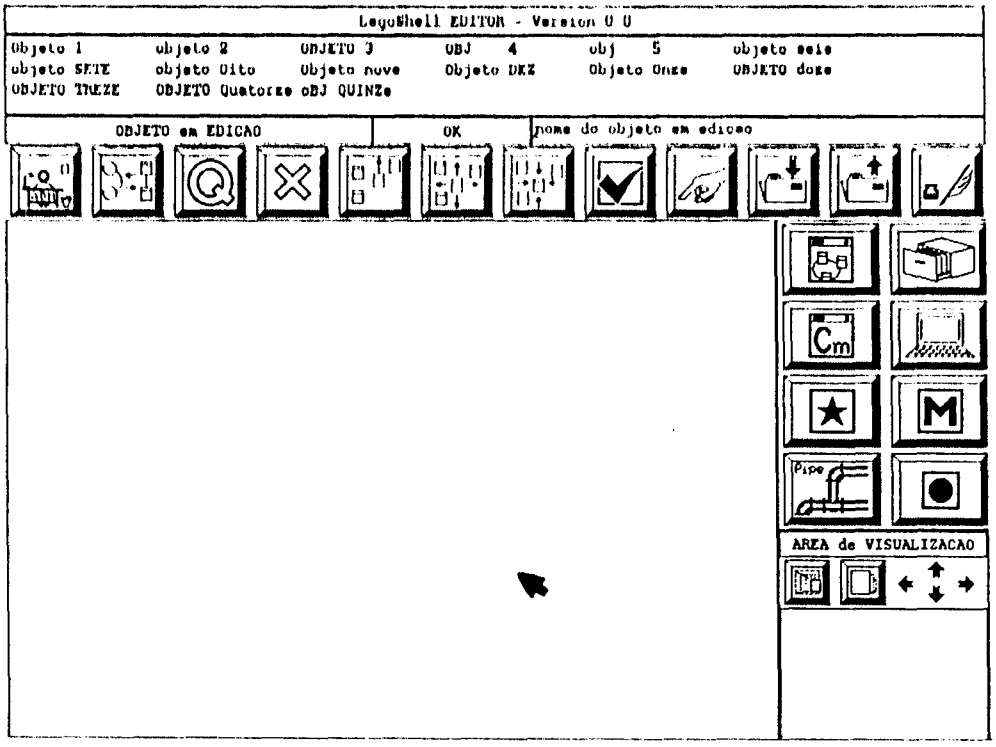



**Figura 7.1** : A interface com o usuário do protótipo

Descreveremos a seguir as funções básicas do Editor.

## 7.2. Funcionalidade

A seqüência padrão para se executar uma função segue o paradigma Seleção-Ação:

seleção

ação

[confirmação]

Quando estamos num determinado estado, nem todas as opções são válidas. Isto será salientado, por exemplo, fazendo com que as opções inválidas fiquem em um tom mais apagado (indicação de Insensível).

Analisaremos a seguir a forma de ativar cada função em detalhe

### CREATE

No processo de criação poderemos ter ações de inclusão de novos componentes, modificação de posição dos componentes, edição da configuração do objeto, edição dos parâmetros do objeto e dos componentes e o armazenamento do objeto criado.

No Cabeçalho teremos o nome do objeto sendo editado.

Quando estamos criando um objeto, é muito possível que não tenhamos dado nome ao objeto, ainda. Isto deverá ser feito no momento do armazenamento. Até lá, o nome do objeto em edição conterá o nome da última computação carregada ou estará vazio.

Veremos a seguir quais as opções de armazenamento que o Editor oferece.

## Armazenamento

Ha três funções relacionadas com o armazenamento dos objetos do ambiente: STORE, EXIT e QUIT. Se o usuário terminou de criar ou editar um objeto, ele pode querer armazená-lo, criando assim um novo número de edição privada, mantendo-se no mesmo nível. Para isto usará a função STORE. Se ele quer armazenar o objeto e também sair do nível de edição para um nível mais alto, ou para fora do ambiente, usará a função EXIT. Se ele quer abortar a edição, abandonando este nível para um nível superior, usará a função QUIT.

Um objeto pode ser armazenado de forma incompleta, mas não pode ser congelado se sua consistência não foi verificada.

## STORE

A seqüência de ações na função STORE é:

Selecionar na barra de menu

[Selecionar nos submenus ("pop-up menu")] *

Confirmar

O primeiro sub-menu que aparecerá terá duas opções relevantes:

STORE as...
YES

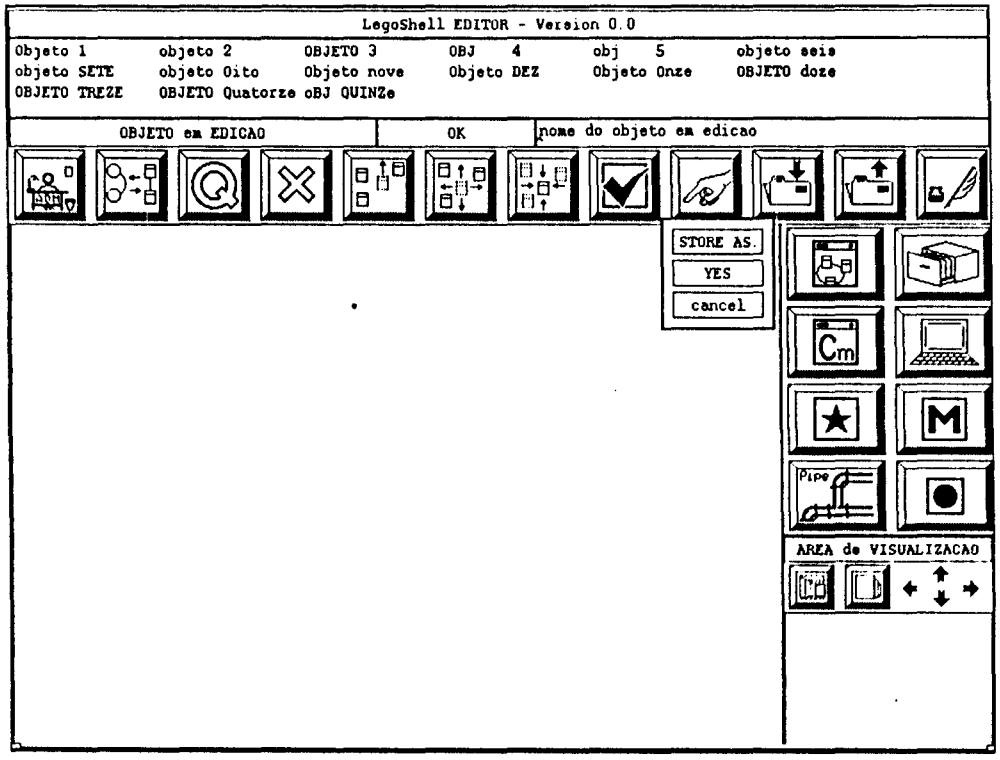



**Figura 7.2** : Pop-up menu da função STORE

Se o objeto tem um nome válido atribuído, o que pode ser verificado na barra de nível correspondente, ambas as opções estarão disponíveis. Se o nome não for válido, estaremos ante a situação de que somente a opção STORE as ... estará disponível. Se selecionada a opção STORE as ..., estará habilitada uma janela que permitirá atribuir nome ao objeto editado. Finalizamos esta função com a confirmação, através da tecla OK na janela. Produz-se então a verificação da validade do nome, e a ação correspondente ao armazenamento é realizada.

## EXIT

A seqüência de ações na função EXIT é:

Selecionar na barra de menu

[Selecionar nos submenus ("pop-up menu"s)] *

Confirmar

Ao ativar a função EXIT, aparecerá um sub-menu com duas opções:

EXIT do nível (LEVEL)
EXIT do ambiente (SESSION)

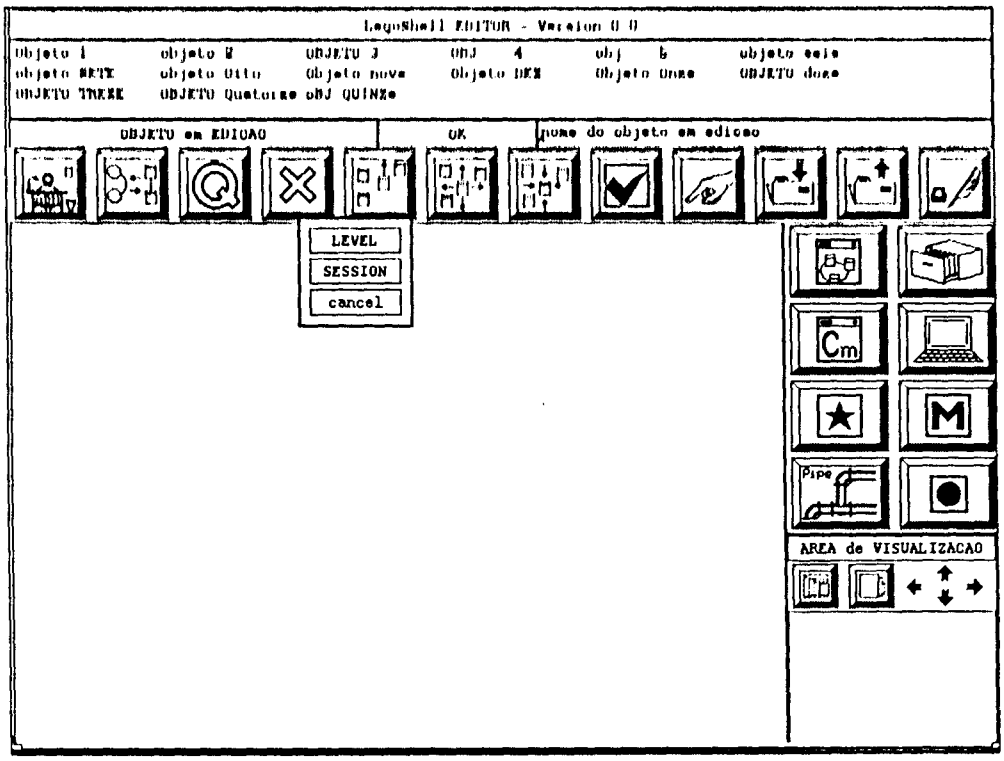



**Figura 7.3** : Pop-up menu da função EXIT

## QUIT

Aparecerão no menu duas opções YES e CANCEL.

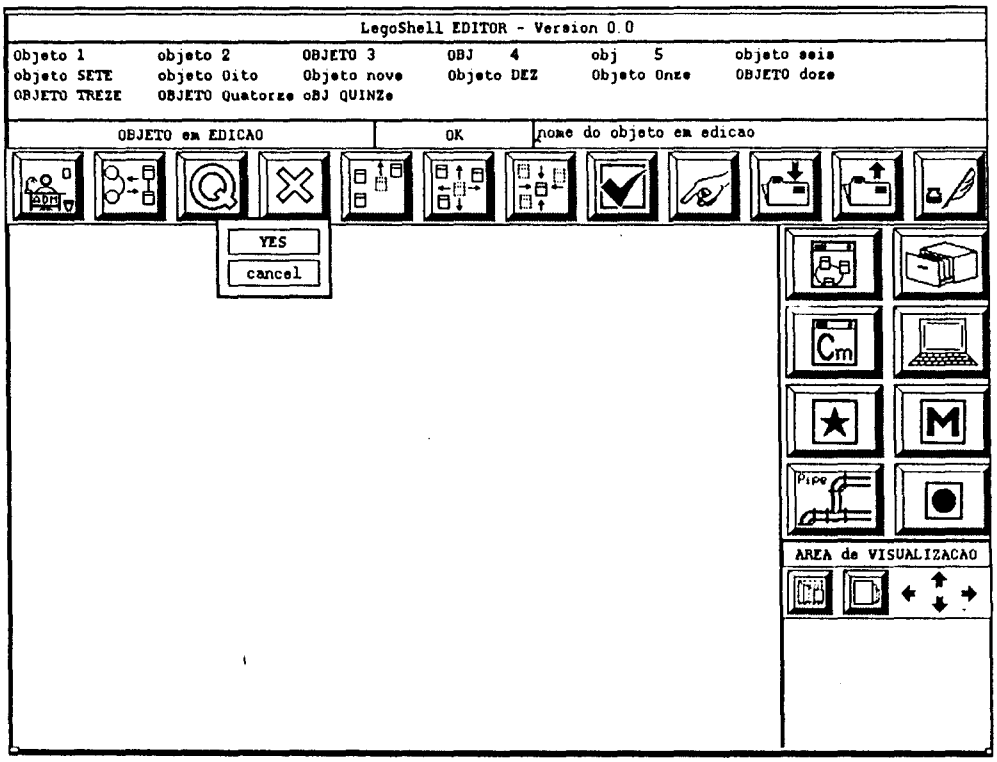



**Figura 7.4** : Pop-up menu da função QUIT

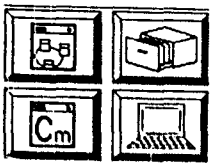
## INCLUDE

A seqüência de comandos ou operações que o usuário deverá realizar para invocar a função INCLUDE depende, como veremos a seguir, da categoria do objeto que se vai incluir:

Inclusão de Objetos de Conteúdo Semântico:

Selecionar

Escolher (Kind)




Aparece um menu de opções ("pop-up menu")

Selecionar

Escolher (Class)

Executa a ação associada recursivamente até

CLASS = NOME + CONFIRM

Posicionar

Confirmar posição

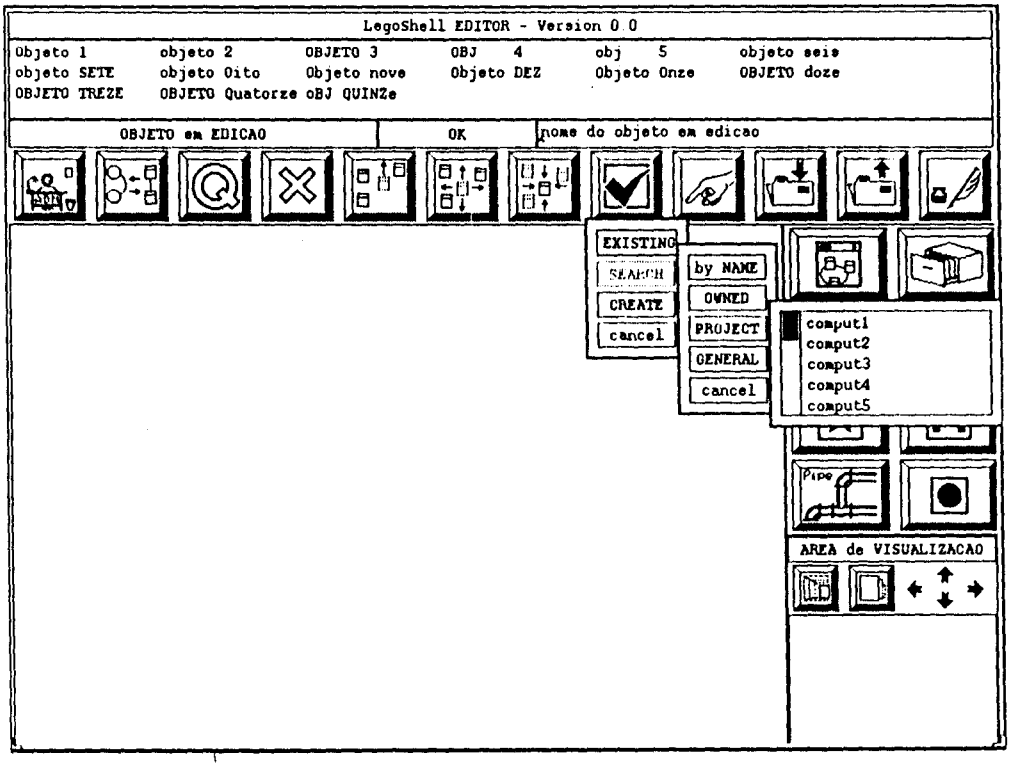
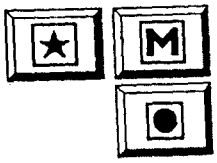



**Figura 7.5** : Pop-up menus da função INCLUDE COMPUT

Inclusão de Conectores-Derivadores e Buraco Negro:



Selecionar

Escolher (Kind)

Posicionar

Confirmar Posição

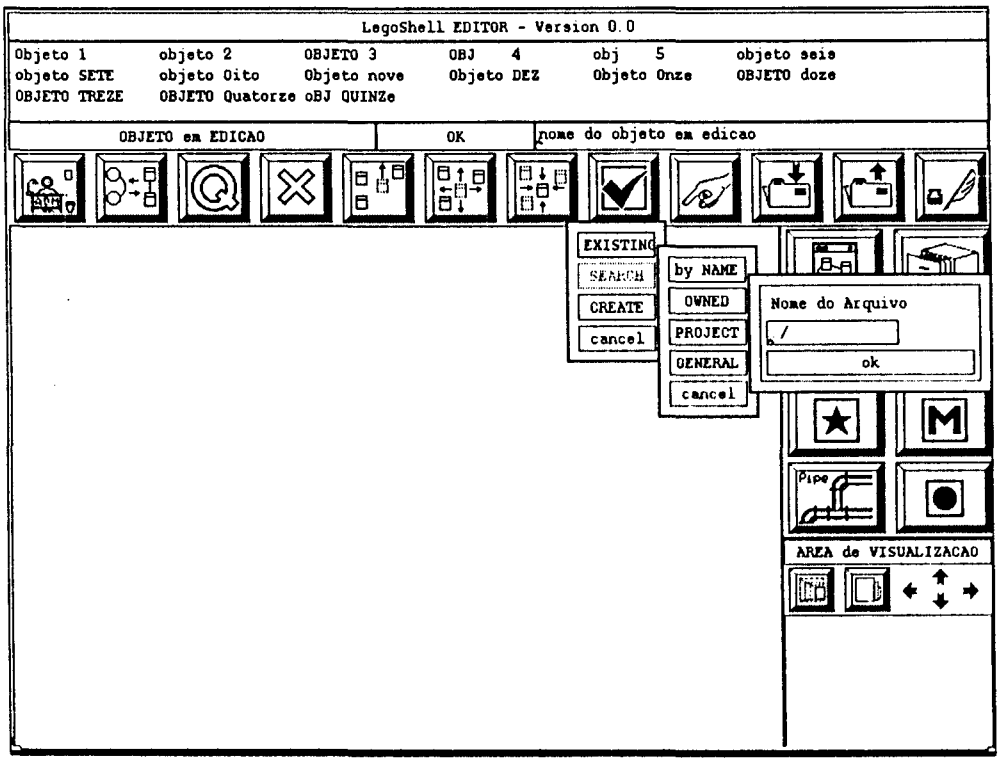
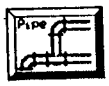



**Figura 7.6** : Pop-up menu da função INCLUDE FILE

Inclusão de Pipes




Selecionar

Escolher (Kind)

Posicionar na Primeira Port

Confirmar posição

Posicionar na Segunda Porta

Confirmar posição

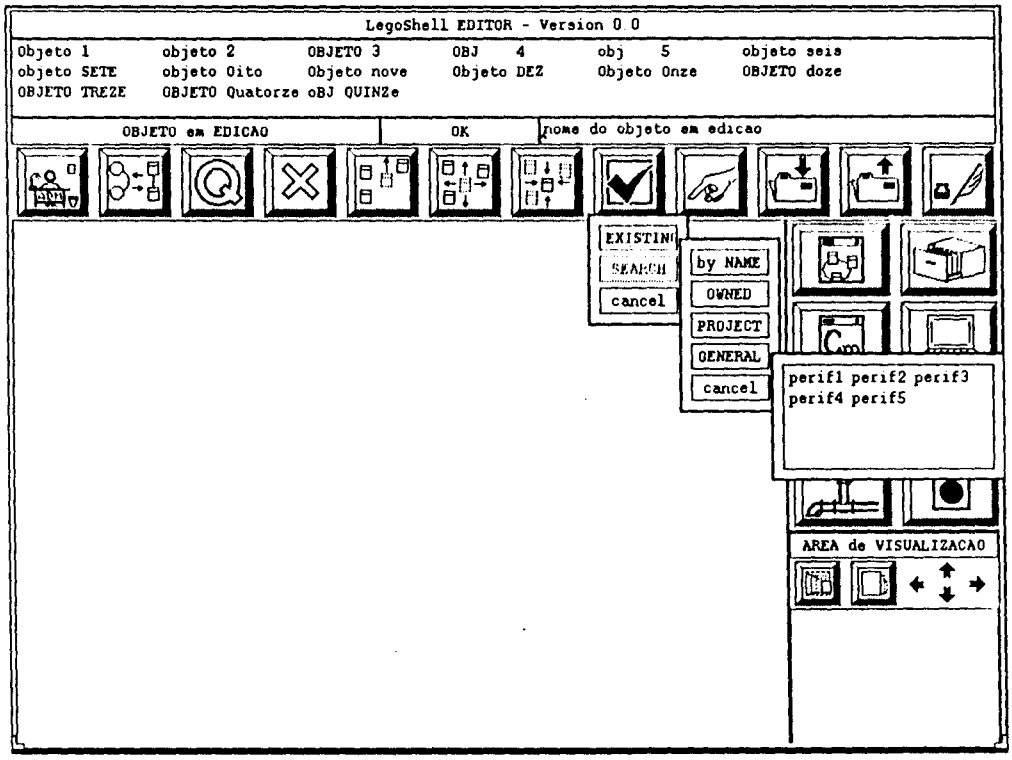



**Figura 7.7** : Pop-up menu da função INCLUDE PERIF

Nas Figuras 7.5 a 7.8 podemos ver quais os "pop-up menu's" que aparecerão no protótipo quando são selecionados objetos de conteúdo semântico para inclusão. Em cada uma das opções disponíveis, se ativará um processo de navegação para percorrer o subconjunto selecionado.

A fim de dar flexibilidade durante a edição e criação de computações foram incluídas nos "pop-up menus" duas outras opções:

SEARCH
CRIATE

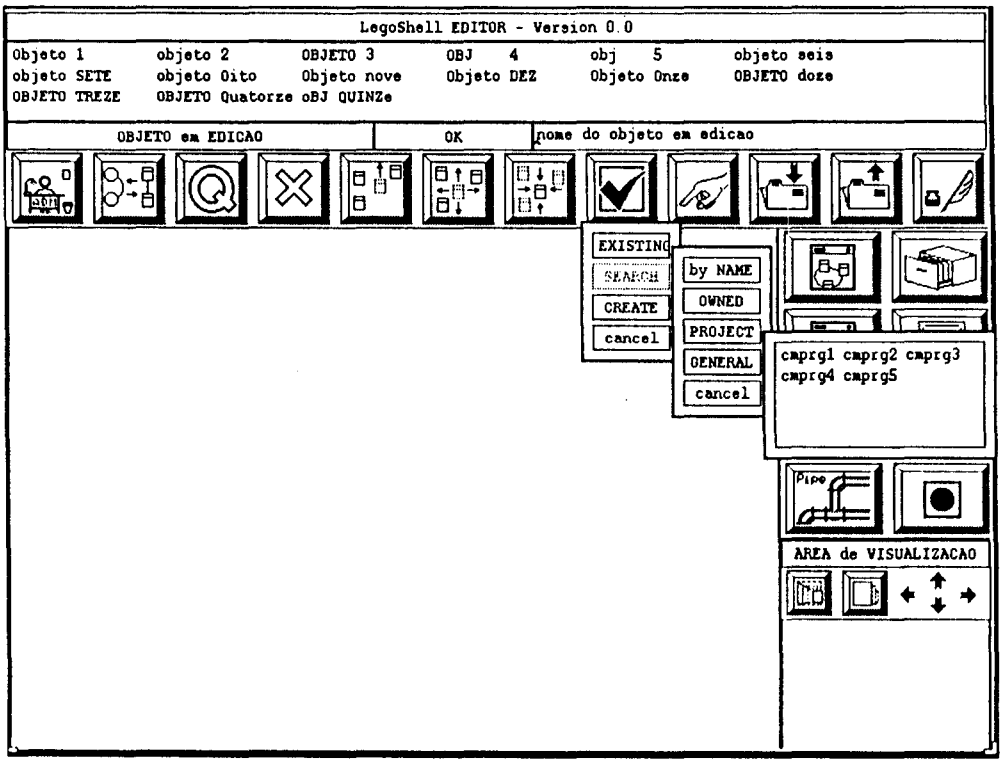



**Figura 7.8** : Pop-up menu da função INCLUDE CMPRG

A primeira opção em todos os "pop-up menu's" será EXISTING, que indica que o objeto é conhecido no ambiente. Esta será a opção pré-escolhida. A configuração associada à computação sendo editada indica que versão do objeto utilizar.

Como todos os conectores da mesma categoria têm um comportamento igual, não é necessário atribuir-lhes NOME.

A seguir veremos as funções que permitem modificar os objetos componentes de uma computação.

## Alteração

## MOVE

A seqüência das ações da função MOVE é:

Apontar

GRAB

A posição virtual do objeto é representada

Posicionar

Confirmar Posição

Poderemos, no protótipo, mover todos os objetos que se encontram dentro da área de trabalho, exceto os "pipes". Isto se deve ao fato de que foi decidido que os "pipes" seriam representados com um segmento de reta

que une as duas portas no extremo do pipe. Quando for confirmada a posição do objeto componente, se produzirá o re-desenho da área de trabalho e da janela de visualização total do objeto. Uma vez colocados os objetos na tela, os "pipes" serão desenhados unindo as portas. Se algum "pipe" tiver que passar por cima de outro "pipe" ou objeto, ele será transparente nesse percurso. Cabe, então, ao usuário a determinação da posição mais adequada para todos os componentes.

É possível movimentar um objeto ou um conjunto de objetos. Se fizermos o "grab" acima do objeto apontado, apenas este será movimentado. Se o "grab" for acima de um outro objeto do conjunto corrente movimentaremos o conjunto corrente. Se o "grab" for acima de um objeto que não pertence ao conjunto ativo apenas movimentaremos este objeto.

## Edição

A função EDIT tem seis opções:

- EDIT de um Objeto Componente
- EDIT da Configuração
- EDIT dos parâmetros de um componente
- EDIT dos parâmetros do objeto em edição
- EDIT do ícone de um componente
- EDIT do ícone do objeto em edição

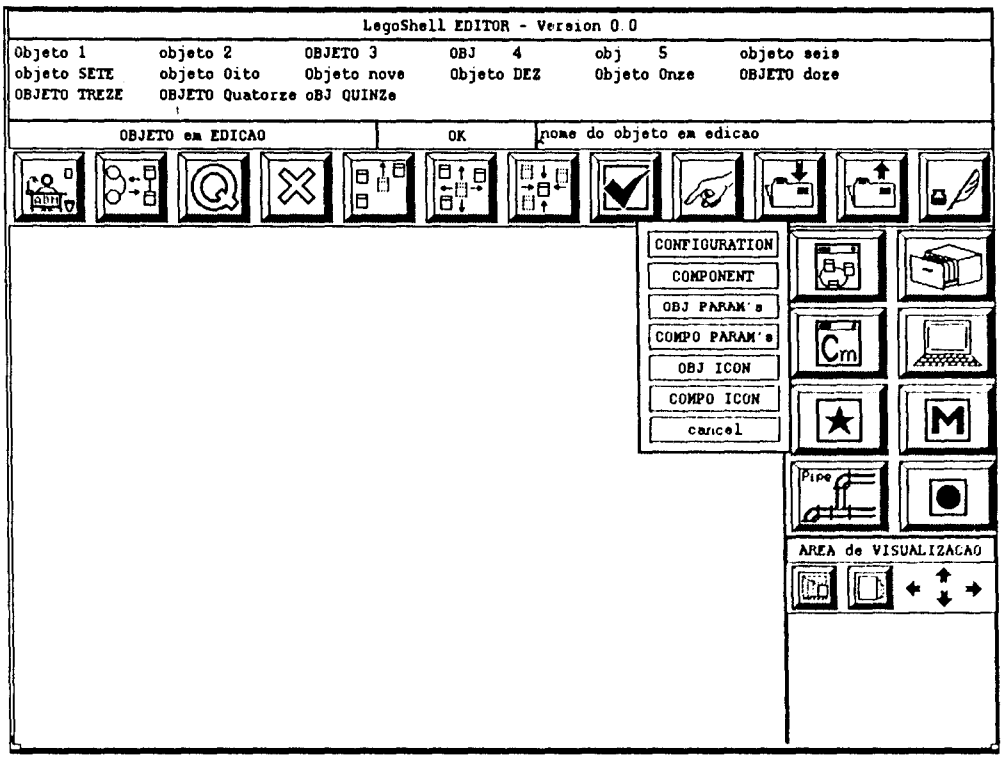



**Figura 7.9** : Pop-up menu da função EDIT

Esta função é complementada com a função LOAD, que permite carregar no editor um novo objeto para ser alterado, habilitando um novo nível de edição.

A seqüência de ações na função EDIT é:

Selecionar o Objeto

Selecionar na barra de menu

[Selecionar nos submenus ("pop-up menu"s)] *

Confirmar

## LOAD

A seqüência de ações na função LOAD é:

Selecionar a Função LOAD (na barra de menu)

[Selecionar nos submenus ("pop-up menu"s)] *

executar esta ação recursivamente até

CLASSE = NOME + versão + Confirmação

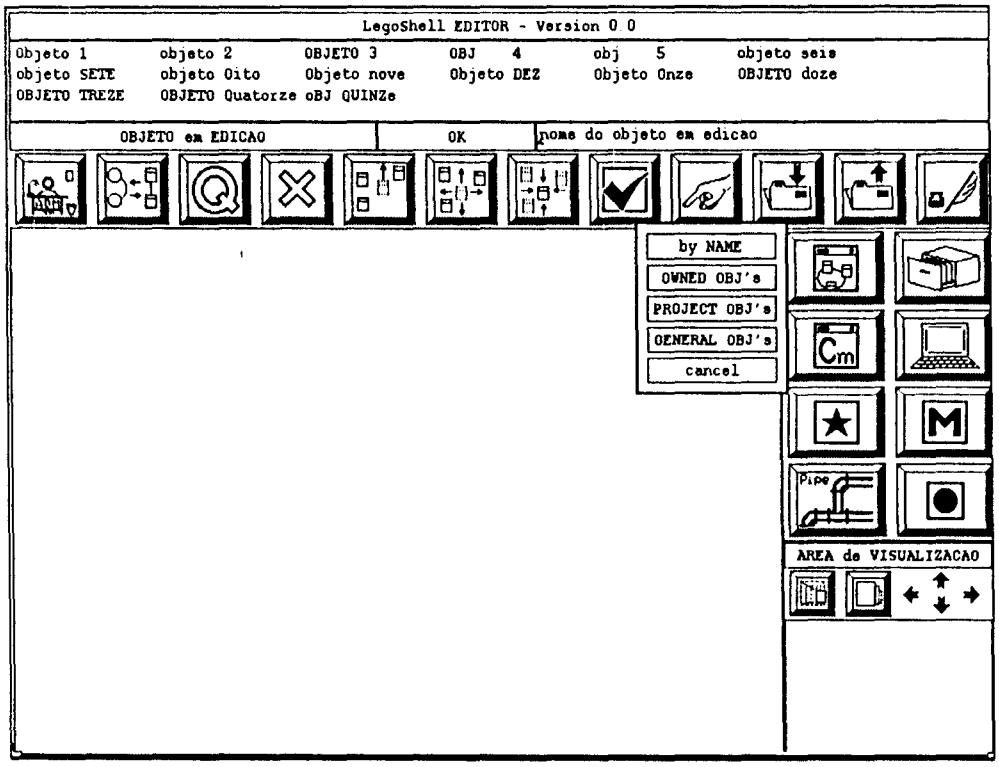



**Figura 7.10** : Pop-up menu da função LOAD

Na Figura vemos os "pop-up menus" que serão oferecidos ao usuário. A primeira seleção indicará a categoria do objeto, a segunda o subconjunto a que o objeto pertence, tendo como opção pré-escolhida a opção NAME. Cabem aqui as mesmas considerações que fizemos ao descrever a função de criação, no que diz respeito à validade dos nomes e às verificações necessárias.

É adequado lembrar que o ambiente proporcionará um editor orientado para sintaxe para Programas Cm. Se formos editar um arquivo sem uma estrutura conhecida, usaremos algum editor de texto semelhante ao GNU-Emacs.

No caso de objetos topológicos, como conseqüência da execução do LOAD, veremos que a área de trabalho e a área de representação integral do objeto são limpas, o nome do objeto em edição é colocado na pilha e ele é substituído pelo NOME do objeto carregado.

O objeto será representado com tamanho padrão na área de trabalho abrangendo três quartos da área de Visualização.

## EDIT de um Objeto Componente

A seqüência de ações na função EDIT de componente é:

Selecionar o objeto componente

Selecionar a Função EDIT (na barra de menu)

Selecionar no submenu ("pop-up menu"s) a opção COMPONENTE

## EDIT da Configuração do Objeto em Edição

Para permitir uma edição gráfica da definição de versão de cada um dos componentes, definidos na configuração ativa do objeto em edição, usaremos a função EDIT de Configuração.

A seqüência de ações na função EDIT de componente é:

Selecionar a Função EDIT (na barra de menu)

Selecionar no submenu ("pop-up menu"s) a opção CONFIGURAÇÃO

A representação na tela permanece quase inalterada. Adiciona-se agora uma janela de edição, dentro de cada objeto, contendo a definição de versão desse objeto.

Nesta janela teremos três campos:

O primeiro campo corresponde ao número
O segundo campo corresponde ao caracter indicador de tipo
O terceiro campo, eventualmente visível, é um campo auxiliar de edição.

No segundo campo usaremos a seguinte convenção:

| | |
|---|---|
| Corrente | + |
| ULTLIB | L |
| Indireta | I |
| Inclusão | * |
| Efetiva | E |
| Direta | D |

Ao estar definido o "default" para Configuração, nunca temos o caso de um objeto cuja definição de versão é inválida ou indefinida: no último caso é adotada a definição "default".

No primeiro campo será representado o número de versão, após a resolução da configuração, para o caso da escolha no segundo campo ser Corrente, ULTLIB, Inclusão ou Efetiva.

Se a escolha for Direta ou Indireta, aparecerá o terceiro campo, que nos permitirá incluir o número. Quando aceito pelo ambiente, o número da versão que o ambiente determinou será apresentado no primeiro campo. Por isto o primeiro campo não será passível de edição, atuando somente como "feed-back" para o usuário.

No canto superior esquerdo da área de trabalho aparecerá, também, uma janela que permitirá editar a lista de Inclusão. Incialmente esta lista estará vazia, se ainda nada foi especificado.

Para permitir a finalização da edição da Configuração, teremos uma tecla de OK, para confirmação.

A confirmação traz como conseqüência a criação de uma nova definição de configuração. Por "default" esta última se converte na configuração ativa para o objeto em edição.

Para ativar outras configurações é preciso utilizar a função ACTIVATE.

## EDIT dos Parâmetros de um Componente

Na representação dos objetos na LegoShell, colocou-se um botão denominado PAR (Parâmetros), que nos permitirá invocar a edição dos parâmetros, tanto de classe quanto de execução, e fazer a escolha das chaves de seleção.

A seqüência de ações na função EDIT dos parâmetros de um componente é:

Selecionar o botão PAR do objeto componente

Selecionar no submenu ("pop-up menu") opções.

Ao selecionar o botão PAR aparecerão duas janelas de edição.

A primeira estará parcialmente superposta com a representação do objeto. Ela terá três áreas: parâmetros de classe, chaves de seleção e parâmetros de execução. Esta janela permitirá que se efetue a inclusão do valor, de uma referência ou se deixe o valor "default". Também permitirá a exportação explícita e o "rename" explícito do parâmetro.

A segunda, no canto superior esquerdo da área de trabalho, conterá também a representação de três áreas, correspondentes aos parâmetros de classe, às chaves de seleção e aos parâmetros de execução do objeto em edição, respectivamente.

Usamos a seguinte convenção gráfica:

- **Rename Explícito:** `type R Δ A`. Assim desejamos expressar que adotaremos para a variável R o tipo da variável A do objeto em edição.
- **Exportação de Parâmetro:** `type T *`. No objeto em edição aparecerá uma referência explícita a este parâmetro: `type (#SORT) T ...`

É importante salientar que o número de cada objeto componente estará presente no rótulo do objeto e isto nos permitirá uma referência sem ambigüidades aos diferentes componentes, mesmo que eles sejam duas cópias de um mesmo objeto.

Para finalizar a edição dos parâmetros de um componente disporemos de uma tecla OK.

## EDIT dos Parâmetros do Objeto em Edição

A seqüência de ações na função EDIT dos parâmetros do objeto em edição é:

Selecionar a Função EDIT (na barra de menu)

Selecionar no submenu ("pop-up menu") opção PARAMETROS.

Selecionar no submenu ("pop-up menu") opção OBJETO CORRENTE.

Ao selecionar a opção de EDIT dos parâmetros do objeto em edição, aparecerá a janela no canto superior esquerdo da área de trabalho que descrevemos na seqüência anterior. Bastará então efetuar as modificações ou completar os dados que faltarem.

## EDIT do Ícone de um Componente

A seqüência de ações na função EDIT dos parâmetros do objeto em edição é:

Selecionar a Função EDIT (na barra de menu)

Selecionar no submenu ("pop-up menu") opção ICONE.

Selecionar no submenu ("pop-up menu") opção COMPONENTE.

Poderemos movimentar as portas dentro da borda do ícone do componente com a ação de MOVE fazendo um "grab" nela. Estaremos mundando unicamente os valores de posição desta instância .

## EDIT do Ícone do Objeto em Edição

A seqüência de ações na função EDIT dos parâmetros do objeto em edição é:

Selecionar a Função EDIT (na barra de menu)

Selecionar no submenu ("pop-up menu") opção ICONE.

Selecionar no submenu ("pop-up menu") opção OBJETO EM EDIÇÃO.

Ao escolher esta opção a área de trabalho terá uma borda virtual que representará os limites do ícone. Nesta borda serão representadas as portas exportadas e será possível então executar as seguintes ações:

- MOVE PORT: segue a mesma seqüência que o MOVE e permite mudar a posição de uma porta exportada dentro da borda;
- EXPORT PORT: selecionando uma porta não conectada e selecionando na borda virtual, estaremos indicando que queremos explicitamente exportar esta porta e qual a sua posição "default" na borda do ícone. A partir deste momento haverá uma linha hachurada unindo a porta exportada e sua representação na borda virtual, permitindo reconhecer visualmente a ligação;
- UNEXPORT PORT e EDIT PARAMETROS EXPORTED PORT: selecionando duas vezes numa porta que está sendo representada na borda virtual, abriremos um menu que nos permitirá explicitamente eliminá-la da lista de portas exportadas ou alterar alguns dos parâmetros da porta exportada (i.e. nome a ser representado no ícone).

## SELECT

O modo mais rápido e cômodo para selecionar objetos na área de trabalho é o cursor: colocando o cursor em cima de um objeto, usando o "mouse", e apertando o botão de seleção uma vez (click).

As vezes é necessário selecionar todos os objetos da tela. Para isto foi prevista a opção SELECT no menu de barra. Quando selecionamos esta opção aparecerá um "pop-up menu" com as opções ALL e UNSELECT ALL. Elas permitirão selecionar todos os objetos da tela e cancelar a seleção de todos os objetos, respectivamente.

É possível selecionar mais de um objeto usando o mouse. Para adicionar um objeto à lista circular, bastará fazer um "Shift-Click" acima do objeto. O mesmo processo repetido sobre um objeto selecionado o exclui da lista circular.

Se andarmos sobre a lista circular usando os comandos ANTERIOR e PROXIMO, o objeto apontado atualmente será realçado. Podemos excluir o objeto apontado da lista circular usando o comando EXCLUA (-).

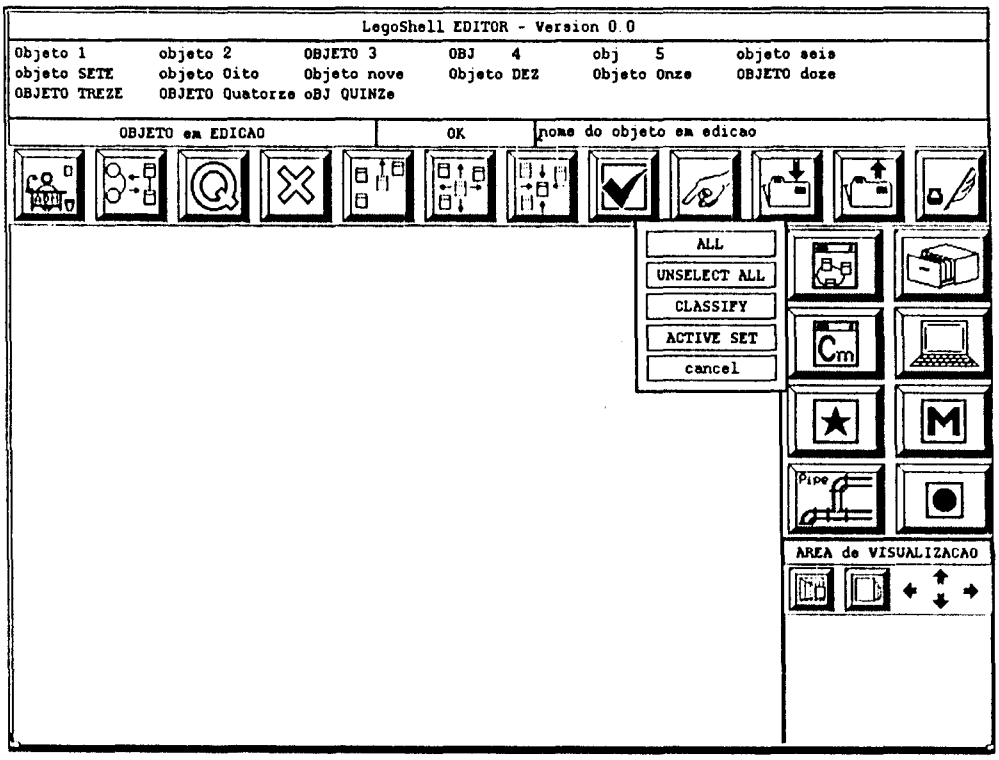



**Figura 7.11** : Pop-up menu da função SELECT

Para cancelar a seleção de todos os objetos da lista poderemos fazê-lo usando a opção UNSELECT ALL do "pop-up menu" ou selecionar sobre um espaço livre da tela ou sobre outro objeto qualquer. Neste último caso, estaremos apagando a lista anterior e inicializando uma nova lista.

Foi atribuído a cada "pipe" uma área de sensibilidade à seleção, próxima aos seus extremos. Isto é equivalente a dizer que se desejamos selecionar um "pipe" deveremos selecionar na região próxima às portas às quais estão ligadas suas extremidades.

## EXCLUDE

Quando desejamos remover um objeto componente do objeto em edição usaremos a função EXCLUDE. A seqüência de ações na função EXCLUDE é:

Selecionar o objeto

Selecionar a Função EXCLUDE (na barra de menu)

Confirmar

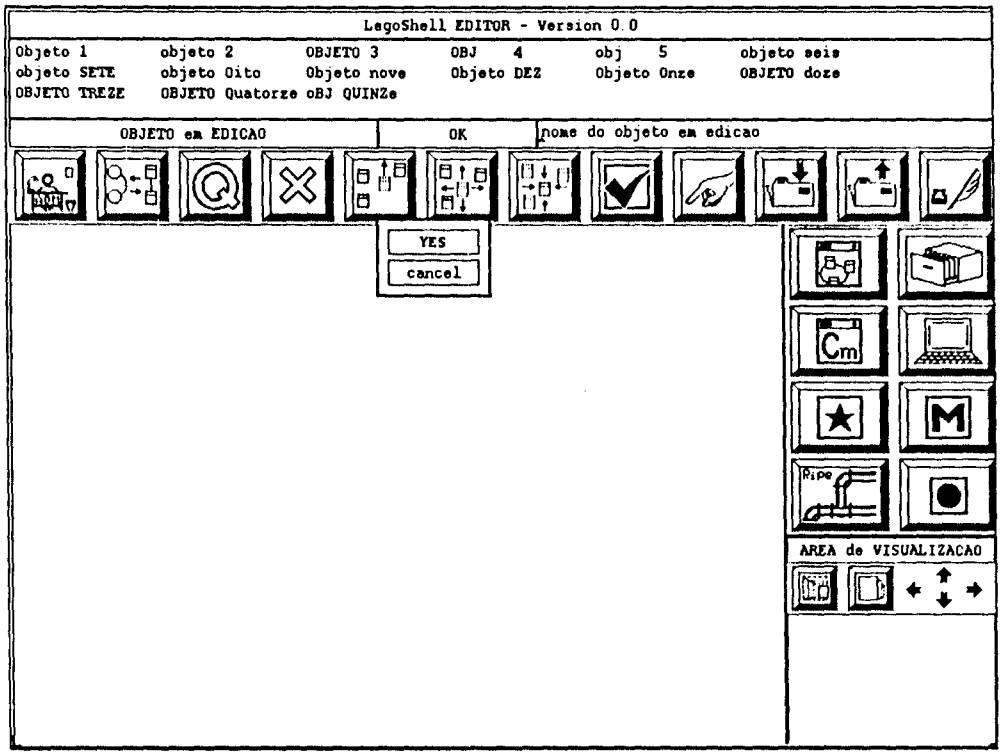



**Figura 7.12** : Pop-up menu da função EXCLUDE

## CLEAR

A seqüência de ações na função CLEAR é:

Selecionar o objeto

Selecionar a Função CLEAR (na barra de menu)

Confirmar

## REDRAW

Isto pode ser feito através de uma ação implícita ou explícita: se movimentarmos a janela de zoom na área de visualização total ou com o comando REDRAW (Alt-Click).

## ABSTRACT

A seqüência de ações na função ABSTRACT será:

Selecionar os objetos

Selecionar na barra de menu a função ABSTRACT

Atribuir nome à Computação

Confirmar (Inclui o SAVE)

Posicionar

Confirmar posição

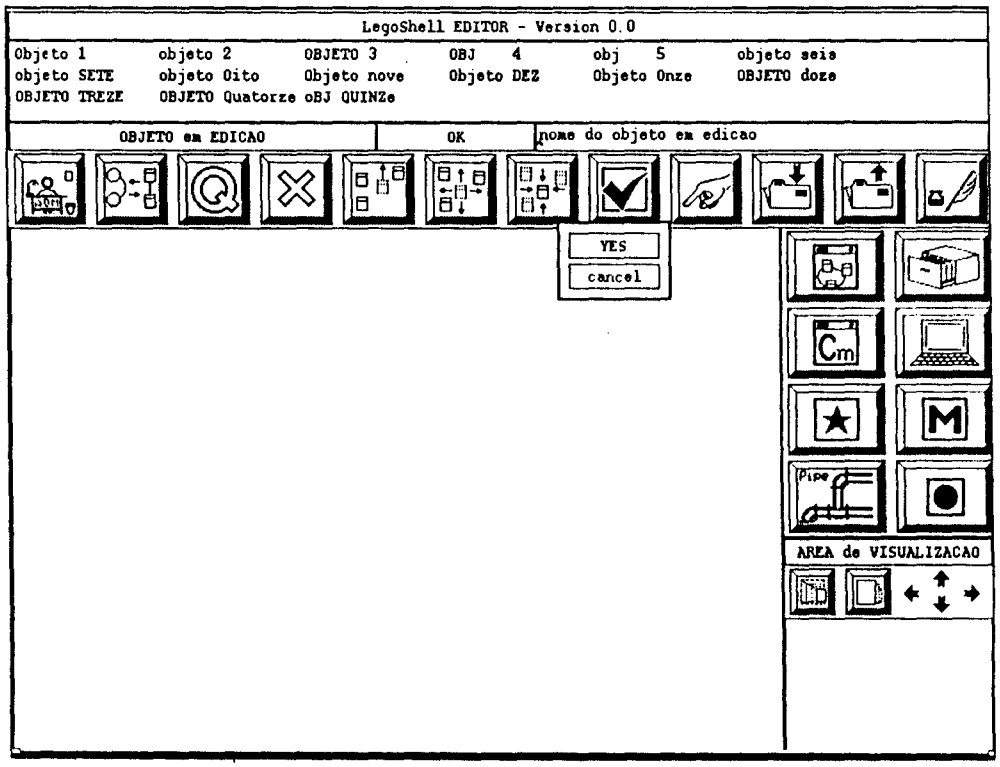



**Figura 7.13** : Pop-up menu da função ABSTRACT

Para as portas que não ficaram conectadas após a abstração, teremos que decidir se as fazemos visíveis ao exterior ou não. Neste último caso, deveremos conectá-las a buracos negros. Ao se dar a confirmação, aquelas que não foram ligadas a um buraco negro, serão automáticamente transferidas para o perímetro da nova computação (exportadas).

## EXPAND

A seqüência de ações da função EXPAND é:

Selecionar a Computação Componente

Selecionar na Barra de Menu a função EXPAND

Confirmar

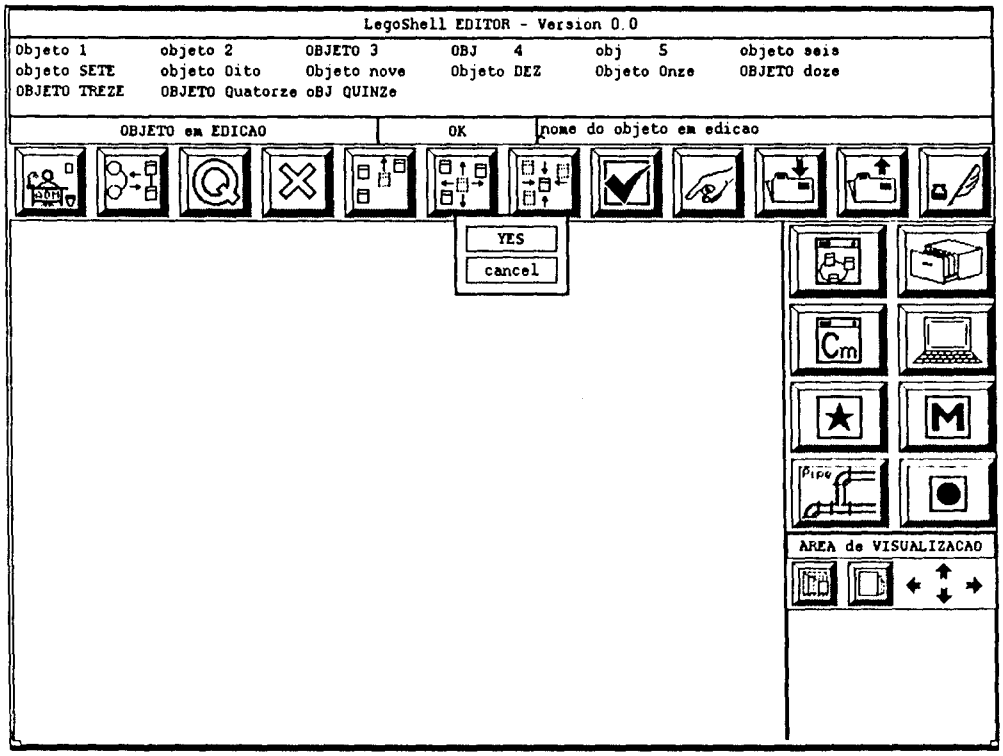



**Figura 7.14** : Pop-up menu da função EXPAND

## ACTIVATE

A seqüência de ações da função ACTIVATE é:

Selecionar na barra de menu a função ACTIVATE

Selecionar no "pop-up menu" a configuração ativa

Confirmar

Ao se fazer a seleção, será representada na área de trabalho o objeto em edição com a mesma disposição dos componentes, incluindo dentro de cada um deles uma janela onde está a definição de configuração que está sendo usada. A representação na tela é igual à da função EDIT de Configuração, só que não permite a edição.

Se confirmada, esta configuração fica sendo a ativa. Se não confirmada, volta-se ao estado anterior, permitindo-se uma nova seleção.

## CONSIST

A seqüência de ações da função CONSIST é:

Selecionar na Barra de Menu a função CONSIST

Confirmar

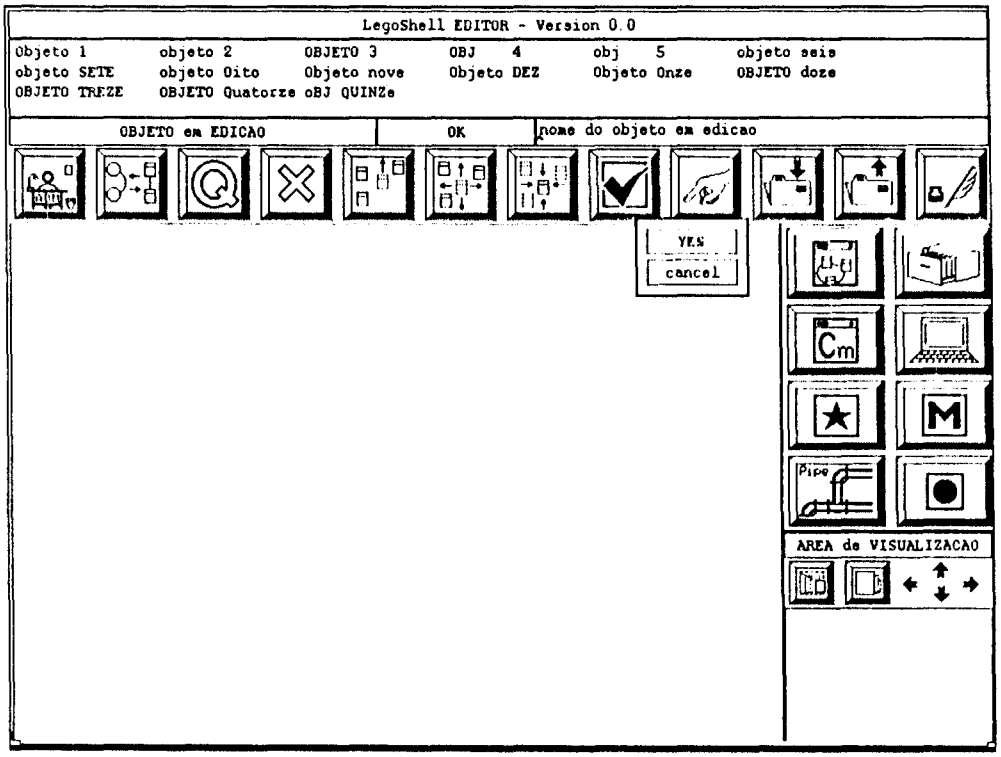



**Figura 7.15** : Pop-up menu da função CONSIST

## Execução

Os objetos de conteúdo semântico poderão ser monitorados de maneira interativa durante a execução. Para isto possuem três botões luminosos correspondentes às funções ligar (TURN-ON), desligar (TURN-OFF) e suspender e ativar(PAUSE & RESUME).

Se acionamos no botão PAUSE & RESUME com o objeto em execução, o efeito deve corresponder à suspensão da execução do processo, junto com todos os processos descendentes. Podemos, num momento posterior, continuar com a tarefa suspendida clicando novamente no botão PAUSE & RESUME.

Se acionamos no botão TURN-OFF, num determinado nó da árvore, isto é equivale a abortar o processo que o implementa, junto com todos aqueles que implementam os seus nós descendentes. Por exemplo, no MSort da Figura 3.1, se desligamos o *Sort #1*, o processo como um todo seguirá rodando, só que agora terá um *Sort* só: estamos desmembrando um ramo completo da estrutura.

Um conjuto de objetos compostos, estruturados numa computação de LegoShell, que foi desligada com TURN-OFF, poderá ser reiniciada acionando o botãoTURN-ON. Isto implica em criar novamente todos os processos necessários.

## 7.3. Aspectos Particulares da Implementação

Tudo isto foi implementado usando os "widgets" do Athena Toolkit do X-Windows, Version 11 Release 3, na estação de trabalho Interpro 220, da Intergraph, e depois portada para a Version 11 Release 4, na estação de trabalho Sparc Station 1+ da SUN Microsystems.

Durante a implementação tivemos acesso à seguinte documentação relacionada com o X Windows e o Athena "toolkit" [ Ful88, GeSN89, MAS88, NOR90, Pe89, Sun90a, Sun90b, SW88]

# Capítulo 8

# CONCLUSÕES

No dia 13 de março de 1987, culminava um longo período da minha vida. Começava minha reforma, com a graduação de Capitão de Corveta da Marinha de Guerra Argentina, após 19 anos e um mês de serviço.

Tinha chegado a um ponto de decisão, "o turning point", e queria retomar meus estudos em Computação. Já tinha sido aceito como aluno do Mestrado em Ciência da Computação em setembro do ano anterior, mas começar as aulas com três semanas de atraso, mesmo que o novo uniforme fosse "chinelos e barba", foi uma prova dura, da qual, parece-me agora, estou saindo victorioso.

## 8.1. Como se desenvolveu o trabalho

Durante 1987 participamos das reuniões do Projeto A_HAND, e discutimos diversos aspectos da linguagem Cm, de programação orientada para objetos e de catalogação e recuperação de informação.

Em 1988 procuramos toda a informação disponível sobre X-Windows e Statecharts, que por sinal era reduzidíssima no Brasil. No segundo semestre de 88 trabalhamos em conjunto com Alicia Di Sarno na especificação da semântica dos conectores da LegoShell. Tentamos portar o X-Windows para o DigiRede e para o Micro-VAX do Projeto ETHOS, sem documentação!!! (A impossibilidade desta missão só foi comprendida muito tempo depois, quando percebemos que não tinhamos os recursos gráficos necessários!!).

Em 1989 com a chegada da estação de trabalho da Intergraph, Interpro 220, fizemos a instalação (física e elétrica) no Departamento de Ciência da Computação da UNICAMP, portamos o X-Windows Version 11 Release 3 para a Interpro 220. Nesta época estava visitando a UNICAMP o Prof. Roger Hoover, um dos criadores do Cornell Synthesizer Generator, e tivemos oportunidade de intercambiar opiniões sobre a especificação formal da linguagem Cm e da LegoShell. Junto com ele descobrimos que o servidor de X da Interpro era Release 1, incompatível com as bibliotecas que havíamos gerado para a Release 3. Como esta situação se prolongou até setembro, desde maio até setembro de 89 especificamos em detalhe, em um árduo trabalho em conjunto com Carmem Satie Hara, o Ambiente A_HAND, as suas funções e o "look-and-feel" da interface do usuário do editor. De setembro em diante definimos a representação interna e estipulamos que ela deveria ser totalmente compatível com a estrutura de objetos definidos por Carmem Satie Hara no SGBDOO DAMOKLES. Definimos as funções dos Núcleos Topológico e de Edição Gráfica. Usamos o Hipertexto **ht** desenvolvido por Carlos Alexandre Polanczyk para organizar o nosso trabalho. A definição de cada função foi feita usando uma meta-linguagem e utilizamos o **ht** como ferramenta para verificar a consistência das definições e auxiliar na comunicação das duas pessoas que trabalhavam concorrentemente sobre os mesmos arquivos. A partir desta fase contamos com a colaboração de Cassius Di Ciani que codificou em C as funções do Núcleo Topológico.

De novembro de 89 a março deste ano trabalhamos em conjunto com Antônio G. Figueiredo Filho na alteração da especificação dos "statecharts" e na verificação da consistência da especificação do nosso editor de

LegoShell. Como ainda não está pronto o compilador de "blobs" que ele está desenvolvendo, toda a codificação de tabelas e a geração de código dos "statecharts" tiveram que ser feitas manualmente.

No mês de abril deste ano já tinhamos conseguido fazer a primeira versão da interface com o usuário. A falta de documentação foi um grande empecilho pois na maioria das vezes as dúvidas deviam ser solucionadas na base da tentativa e erro. A falta de um bom depurador simbólico e a concorrência no uso da Interpro também atrapalharam bastante o andamento da implementação.

Durante julho e agosto contamos com a colaboração de Karsten Hauten na codificação do algoritmo idealizado por Rogério Drummond para linearizar estruturas estáticas. Em agosto conseguimos integrar esta função com a representação interna e com a interface com o usuário, o que nos permitiu a primeira interação com objetos da LegoShell. No fim de setembro incorporamos o motor de "statecharts" ao protótipo. No mês de outubro fizemos o transporte do editor para a Sparc Station 1+ da Sun Microsystems. Isto foi relativamente rápido (72 horas sem dormir) e gratificante pelo fato de dispormos de maior velocidade e de interação através da rede o que até alí só podíamos supor que funcionaria.

No estado atual, o editor reutiliza também código desenvolvido por Lídia Yamamoto e Cassius Di Ciani para o **mtree** e o módulo de alocação de cores do **stardust** desenvolvido por Carlos A. Furuti.

## 8.2. Um olhar crítico sobre o Editor

Na proposta de trabalho que fizemos em novembro de 1988 pretendíamos:

- Completar a busca bibliográfica;
- Estudar a programação usando widgets do X Windows;
- Definir a arquitetura do protótipo;
- Modelar a Representação Essencial;
- Modelar o diálogo;
- Modelar a representação de restrições;
- Implementar o protótipo

Entre os temas teóricos com os quais pensávamos ter que confrontá-nos estavam os seguintes perguntas:

- Qual seria a interface com o usuário mais adequada para a LegoShell, já que lidamos com objetos de estrutura topológica?
- Qual seria o formalismo mais adequado para representar o diálogo?
- Qual seria o formalismo mais adequado para expressar as restrições?
- Qual dos níveis de abstração do X Windows seria mais adequado para desenvolver a interface entre a interface do usuário e o editor?
- Qual seria o tipo mais adequado de interface entre ferramentas?

Podemos, olhando restrospectivamente, dizer que algumas das perguntas eram muito ambiciosas, mas não pretendíamos dar uma resposta total e definitiva. Ao longo deste três anos de trabalho, tivemos que analizar estas e muitas outras questões, para tomar decisões de projeto, que seria longo aqui enumerar.

A primeira conclusão importante à qual chegamos durante a definição detalhada da LegoShell foi que a linguagem só poderia expressar fluxo de dados. Não encontramos uma forma simples de expressar gráficamente o fluxo de controle. Consideramos que este é ainda um campo em aberto. A solução proposta pelo A_HAND é a linguagem de comandos $CO^2$ Shell. Ela permitirá expressar o fluxo de controle textualmente, incorporará todas as características das programação orientada para objetos, e os programas

escritos em esta linguagem poderão ser encapsulados e considerados objetos básicos da LegoShell. A linguagem $CO^2$ também poderá acionar a execução de computações completas de LegoShell e mesmo programas Cm. Ela será interpretada suportando ligação dinâmica de tipos, o que a torna adequada como linguagem de prototipagem rápida. Temos assim todos os níveis necessários de abstração e linguagens apropriadas para um amplo espectro de aplicações.

Outro dos campos que mereceram nossa atenção foi o da catalogação e recuperação de informação, que é um dos aspectos considerados cruciais para uma real reusabilidade de módulos ou classes. Sendo este um tema ainda em aberto, concluímos que deveria ser incluído na Representação Essencial, como um atributo da classe, um descritor da sua funcionalidade, que permitisse a recuperação. Há varias linhas de pesquisa nesta área, que o projeto A_HAND deverá abordar no futuro: linguagens formais de especificação, técnicas de engenharia reversa e recuperação de projeto, técnicas de catalogação e recuperação usando sistemas especialistas etc.

Abordamos a definição da arquitetura do protótipo usando uma técnica de generalização sobre um tipo abstrato de dados cada vez mais completo, quando definimos a representação interna, e usamos uma técnica de refinamento quando projetamos a interface do ambiente. Após ter definido as funções do ambiente estávamos em condições de definir as funções que o editor deveria prover. Muitas das decisões de projeto discutidas e adotadas durante este tempo restringiram o universo de funcionamento do editor. Mas isto foi logo detetado e um projeto modular, que preserva a independência entre as distintas camadas, foi obtido. Consideramos que nossa implementação é totalmente compatível com a realizada por Carmem Satie Hara sobre o Gerenciador de Base de Dados Orientada a Objetos DAMOKLES, o que permitiria facilmente substituir a representação interna e o núcleo topológico por um BDOO como o definido em [Har90].

Mesmo assim e levando em consideração que o DAMOKLES é um protótipo cujo desenvolvimento foi abandonado pela Karlsruhe Universität, o projeto decidiu pelas estruturas "ad-hoc" para a representação interna. Ao implementar o protótipo deixou-se para uma etapa posterior, as considerações de eficiência, tanto no armazenamento quanto no desempenho. No futuro deverá ser estudada um SGBDOO que mais se adeqüe às necessidades do A_HAND em base à definição feita em [Har90]. Um estudo comparativo entre o desempenho de vários gerenciadores manipulando as estruturas do A_HAND (Cm, LegoShell e $CO^2$Shell) seria o mais adequado.

Outro aspecto que deve ser estudado em maior detalhe é a Representação Essencial parametrisável. A idéia fundamental é que toda linguagem cuja estrutura modular pudesse ser descrita nesta representação poderia ser utilizada no ambiente. Consideramos que a representação interna que definimos pode ser o embrião para esse desenvolvimento.

A escolha do formalismo para modelar o diálogo e a semântica do editor recaiu inicialmente no "statecharts" propostos por Harel [Ha87]. Esta escolha foi influenciada por nossa experiência anterior no uso de máquinas de estado, e nas dificuldades que nelas encontramos. Outro aspecto importante que levamos em consideração no momento da escolha, foi a capacidade de representação gráfica dos estados para permitir uma comunicação fluente entre varios integrantes do grupo.

Pensávamos inicialmente que irfamos usar os "statecharts" na especificação do diálogo com o usuário mas terminamos usando-os para especificar a semântica do editor e para implementar a memória dos estados que ele atravessa. Como usamos os "widgets" para implementar a interface com o usuário, foi muito mais fácil mapear ações do usuário sobre eventos e depois enviar estes eventos ao cliente X (o editor) usando as facilidades do X-Windows. Já no editor estes eventos são os que provocam as mudanças de estado. A grande facilidade que encontramos na implementação da máquina de "statecharts" desenvolvida por Figueiredo Filho e que ela nos permitiu rápidamente passar da especificação ao protótipo, e que qualquer alteração nos estados ou nos eventos é resolvido simplesmente alterando uma tabela. Isto oferece grandes vantagens pela modularidade e a claridade

com que conseguimos definir as interfaces, demonstrando então que a escolha foi adequada para nossas necessidades.

Um dos temas que não conseguimos abordar adequadamente por falta de tempo, foi o das restrições. Nossa idéia inicial era que junto com a definição da representação interna dos objetos fosse definido um conjunto de regras de restrição formando uma base de conhecimento dos objetos. Esta base de conhecimentos permitiria verificar as regras de consistência, as regras de construção etc. No nosso protótipo, implementamos a verificação de consistência de modo procedural e com regras fixas. Qualquer modificação das regras implica em modificação no código.

A escolha do X Windows como sistema de janelas implicou também numa mudança radical na nossa forma de ver a interface com o usuário. No modelo cliente servidor, o controle está nas maos do servidor, e não da aplicação. Isto implica numa mudança na forma em que se programa, na direção de programação orientada a objetos. Decidimos usar widgets porque era o nível mais alto de abstração que o X provê. Usamos o "Athena Toolkit" porque é de dominio público e contávamos com a documentação que acompanha o X11R3 e X11R4. Com a chegada das estações de trabalho Sparc Station 1 + da Sun ao DCC, começamos a ter contacto com interfaces com o ususário mais elaboradas que a nossa, e com outros "toolkits". Uma extensão necessária ao nosso editor é o transporte para outros "toolkits", especialmente o OpenLook e o OSF Motif.

Também deveria ser estudada uma outra interface com o usuário, especialmente projetada para usuários experientes, tomando como ponto de partida a proposta e implementada neste trabalho. Após ter pensado durante muito tempo qual seria a melhor interface com o usuário e implementado uma que achávamos "boa" chegamos a conclusão que tanto os aspectos de psicologia aplicada quanto os artísticos, que não dominamos, são muito importantes e devem ser levados em conta na confexão da interface com o usuário quando se deseja algo além de um protótipo acadêmico.

## 8.3. Reflexões

Este trabalho tinha como objetivo inicial mostrar que as idéias da LegoShell eram factíveis de ser concretizadas. Acreditamos que um passo muito importante foi dado. A definição completa do Ambiente A_HAND e da LegoShell foi finalizada e contamos com um protótipo do Editor que permite manipular os objetos complexos da LegoShell. Uma vez terminada a implementação do protótipo e da máquina de LegoShell que está sendo desenvolvida no DCC da UNICAMP, teremos a possibilidade de mostrar toda a potencialidade deste novo paradigma de especificação de computações.

Um outro aspecto muito importante foi a interação com os membros do Projeto A_HAND, seja na colaboração direta ou através da reutilização de código, ou no simples contrapor das idéias, o que religiosamente fazíamos todas as terças feiras das 18:30 em diante nas reuniões do Projeto. A experiência adquirida na aplicação de novas técnicas de Engenharia de Software e de metodologias experimentais no desenvolvimento de trabalho em grupo nos será de grande proveito.

O fato de ter trabalhado com os "widgets" nos introduziu no mundo da programação orientada para objetos, o que consideramos outro ponto muito positivo do trabalho.

A última reflexão é sobre as dificuldades que um aprendiz de feiticeiro enfrenta em países como os nossos (Brasil e Argentina): tudo é difícil, tudo é complicado. Esperamos dois anos pela estação de trabalho e quando chegou tivemos que esperar ainda seis meses para ter uma versão do servidor compatível com as bibliotecas do X-Windows das quais dispunhamos de documentação. Mas, mesmo assim, nos sentíamos privilegiados ao olhar ao redor.

# BIBLIOGRAFIA


[ Abr88 ] Abramowicz, K., et al, DAMOKLES – Database Management System for Design Applications – Reference Manual Release 2.0, Forschungszentrum Informatik an der Universität, Karlsruhe, Mar 1988

[ Ba90 ] Baran, N., "IBM in the Nineties", Byte, IBM Special Edition, Fall 1990, pp 63-70

[ Bal86 ] Balzer, R.M., "Living in the Next Generation Operating System", Proc. of the IFIP 10th World Computer Congress, Sep 1986, pp. 283-291

[ Bar86 ] Barth, P.S., "An Object Oriented Approach to Graphical interfaces", ACM Transactions on Graphics, Vol. 5, No.2, Apr 1986

[ BS84 ] Barstow, D.R., Shrobe, H.E., Sandwall, E., "Interactive Programming Environments", McGraw Hill, 1984

[ Car89 ] Cardelli, L., "Typeful Programming", IFIP State of the Art Seminar on Formal Description of Programming Concepts, 18-28 Abril 1989, Petropolis, Rio de Janeiro

[ Con86 ] Conradi, R., Didriksen, T.M., Warwik, D.H., "Advanced Programming Environments", Proc. Int. WShop, Trondheim, Norway, June 16-18,1986, Lecture Notes in Computer Science No244, Springer-Verlag

[ DA79 ] Davis, P.J., Anderson, J.A., "Nonanalytic Aspects of Mathematics and their Implications for Research and Education", SIAM Revue, Vol 21, No. 1, Jan 1979, pp. 112-127

[ Dar87 ] Dart, S.A., et al, "Software Development Environments", Computer , Nov 1987, pp. 18-28

[ DES90 ] DESQview/X, "Extending DOS into the 21st Century", Byte, IBM Special Edition, fall 1990

[ DLi87a ] Drummond, R. e Liesenberg, H., "A_HAND: Ambiente de Desenvolvimento de Software Baseado em Hierarquias de Abstração em Níveis Diferenciados", IV Encontro do Projeto ETHOS , Abr 1987, pp. 313-322

[ DLi87b ] Drummond, R. e Liesenberg, H., "Requisitos para um Ambiente de Desenvolvimento de PROGRAMAS", I Encontro IBM de Ciência e Tecnologia em Informática , Nov 1987

[ Do78 ] Doherty, W.J.,"Commercial Significance of Man-Computer Interaction", RC7279 (#31412), 9/11/78, IBM Research Report

[ DN85 ] Drapper, D., Norman, H., "Software Engineering for User interfaces", IEEE Transactions on Software Engineering, Vol. 11, No. 3, Mar 1985

[ Dru89 ] Drummond, R., "LegoShell: Linguagem de Computações", Anais do III Simp. Bras. de Engenharia de Software , 1989

[ Dru90 ] Drummond, R., Correspondência Particular, Campinas-SP-Brasil, nov 90

[ DSi88 ] Drummond, R. e da Silva, F.Q.B., "Manual de Referência – Linguagem Cm", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, Mai 1988

[ Fig90 ] Figueiredo Filho, A.G., "Estadogramas", Tese de Mestrado em andamento, DCC-IMECC-UNICAMP, Campinas-SP-Brasil

[ FLi90 ] Figueiredo Filho, A. G. , Liesenberg, H. K. E., "Geração de Gerenciadores de Sistemas Reativos", Relatório Técnico N'20/90, IMECC-UNICAMP, Campinas-SP-Brasil, mai 90

[ Ful88 ] Fulton, J.,"X-Window System – Release 3 – Release Notes", MIT 1988

[ Fur90a ] Furuti, C.A., "Cm – Linguagem e Implicações", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, 1990

[ Fur90b ] Furuti, C.A., "Implementação do Tradutor Cm → C", tese de mestrado em andamento, Departamento de Ciência da Computação, UNICAMP

[ GC87 ] Gardiner, M.M., Christie,B.,"Applying Cognitive Psychology to User-Interface Design", John Willey & Sons, 1987

[ GeSN89 ] Gettys, J., Scheiffer, R. W., Newman, R., "Xlib – C Language X Interface", MIT X Consortium Standard – X V11 R4 – MIT 1989

[ Gr86 ] Green, M., "A Survey of Three Dialogue Models", ACM Transactions on Graphics, Vol. 5, No. 3, Jul 1986, pp. 244-275

[ Ha87 ] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, North-Holland, Vol. 8, 1987, pp. 231-274

[ Ha88 ] Harel, D., "On Visual Formalisms", Communications of the ACM, Vol. 31, No. 5, May 1988, pp514-530

[ Har90 ] Hara, Carmem Satie, "Utilização de um Banco de Dados Orientado a Objetos em um Ambiente de Desenvolvimento de Software", Tese de Mestrado, DCC-IMECC-UNICAMP, Campinas -SP-Brasil, set 90

[ HBe85 ] Hoffnagle, G.F., Beregi, W.E., "Automating the software development process", IBM SYSTEMS JOURNAL, Vol. 24, No. 2, 1985, pp. 102-120

[ HFW86 ] Hopgood, F.R.A., Duce, D.A., Fielding, E.V.C., Robinson, K., Williams, A.S., "Methodology of Window Management", Springer-Verlag, 1986

[ Hi86 ] Hill, H., "Human-Computer Interaction – The SASSAFRAS UIMS", ACM Transactions on Graphics, Vol5, No. 3, Jul 1986

[ Hix89 ] Hix, D., "User Interfaces: Openning a Window on the Computer", IEEE Software, jun 89

[ HK86 ] Henderson, K., Kard, L., "ROOMS", ACM Transactions on Graphics, Vol. 5, No.2 Apr 1986

[ HN87 ] Henderson, P.B., Notkin, D., "Integrated Design and Programming Environments", IEEE Computer, nov 87

[ Ja86 ] Jacob, R.J.K., "A Specification Language for Direct-Manipulation User interfaces", ACM Transactions on Graphics, Vol. 5, No. 4, Oct 1986, pp. 283-317

[ KRi78 ] Kernighan, B.W. e Ritchie, D.M, "The C Programming Language" , Prentice-Hall, 1978

[ Le89 ] Ledgard, H.P., "The Case Against User Interface Consistency", CACM, Vol. 32 No.10, oct 89, pp 1164-1173

[ LSi87 ] Larkin, J.H., Simon, H., "Why a Diagram is (Sometimes) Worth Ten Thousand Words", COGNITIVE SCIENCES, Vol 11, 1987, pp. 65-99

[ MAS88 ] Mc Cormack, J., Asente, P., Swick, R.R., "X toolkit Intrinsics – C Language Interface", X Window System – X V11 R3 – MIT 1988

[ Me87 ] Melo, R., "interfaces de usuário e Banco de Dados", Tutorial em Ambientes de Desenvolvimento de Software, Florianópolis, SC, Set 1987

[ Mey87 ] Meyer, B., "Reusability: The Case for Object-Oriented Design", IEEE Software , Mar 1987, pp. 50-64

[ Moo90 ] Moore, D., "The migration of X Window System", Byte, IBM Special Edition, fall 1990

[ Mu88 ] Munsey, G.J., "Moving X Windows to your Environment", UnixWorld, May 88, pp. 81-89

[ NOR90 ] Nye, A., O'Reilly, T., "The definitive Guides to the X Window System", vol 0 to 7, O'Reilly & Associates, Inc. Sebastopol CA, 1990

[ OR89 ] O'Reilly, T., "The Toolkits (and Politics) of X Windows", Unix World, feb89, pp 66-72

[ Pe89 ] Peterson, C., "Athena Widget Set – C Language Interface", X Window System – X V11 R4 – MIT 1989

[ Pen88 ] Penedo, M.H., Riddle, W.E., "Software Engineering Environment Architectures", IEEE ToSWEng, Vol. 14 No. 6, jun 88, pp 689-696

[ Pol90 ] Polanczyk, C.A., "Uma Ferramenta Baseada em Hipertexto para Desenvolvimento de Software", Tese de Mestrado, DCC-IMECC-UNICAMP, Campinas-SP-Brasil, dez 90

[ Pot87 ] Potosnak, K., "Creating Software that People Can and Will Use", IEEE Software, Sep 87

[ Pot88 ] Potosnak, K., "What's Wrong with Standar User Interfaces", IEEE Software, Sep 88, pp 91-92

[ Pn86 ] Pnuelli, A., "Application of Temporal Logic to the Specification and Verification of Reactive Systems: A survey of Current Trends", In Current Trends in Concurrency, Bakker, J.W. et alts., Lecture Notes in Computer Sciences, vol. 224, Springer-Verlag, New York, 1986, pp. 510-584

[ Ra82 ] Rajaraman, M.K., "A Caracterization of Software Design Tools", SW Eng Notes, Vol. 7 No.4, oct 82, pp 14-17

[ Re86 ] Reisner, P., "Human Computer Interaction: What is it and what is needed", IBM Technical Report, No. RJ5308, Sep 1986

[ Sch86 ] Schmucker, K.J., "MACAPP: an Application Framework", Byte, aug 86, pp 189-193

[ SDr88 ] de Sarno, A. e Drummond, R., "LegoShell: Linguagem de Configuração de Programas", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, Dez 1988

[ SGe86 ] Scheifler, R., Gettys, J., "The X Window System", ACM Transactions on Graphics, Vol. 5, No. 2, Apr 1986, pp. 79-109

[ Sh87 ] Schriver, B.D., "Integrated Environments", IEEE Software, Nov 87

[ SHB86 ] Sibert, J.L., Hurley, W.D., Bleser, T.W., "An Object-Oriented User interface Management System", SIGGRAPH' 86, ACM, Dallas, Vol. 20, No. 4, Aug 1986, pp. 259-268

[ SIKV82 ]Smith, D.C., Irby, C., Kimbal, R., Verplank, B., "Designing the Star User interface", BYTE, Apr 1982, pp.242-282

[ SLD88 ] da Silva, F.Q.B, Liesenberg, H. e Drummond, R., "Programação em Cm", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, Mar 1988

[ SMa84 ]Silva, M.V., Magalhães, L.P., "Processadores Universais de Diálogo: Formas de Representação do Diálogo", Informe Técnico, UNICAMP - FEE - DEE,1984

[ Spr ] Sproul et al. "Device Independent Graphics",Chapter 9

[ St87 ] Stallmann, R., "GNU Emacs Manual", Free Software Foundation, Cambridge, Mar 87

[ Sun90a ] Sun Microsystems, Inc.,"OpenLook – Graphical User Interface–Functional Specification", Addison –Wesley Publishing Company Inc. 1990

[ Sun90b ]Sun Microsystems, Inc.,"OpenLook – Graphical User Interface–Application Style Guidelines", Addison –Wesley Publishing Company Inc. 1990

[ SW88 ] Swick, R., Weissman, T., "X toolkit Athena Widgets – C Language Interface", X Window System, MIT 1988

[ Tei81 ] Teitelbaum, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Communications of the ACM , vol. 24, no. 9, Set 1981, pp. 563-573

[ Tre88 ] Treadway, R., "X Window Consortium Creates Open Standard", UnixWorld, Jun 1988, pp. 129-130

[ Tro90 ] Trombetta, O., "Editor Sensível a Sintaxe para a Linguagem Cm", tese de mestrado em andamento, Departamento de Ciência da Computação, UNICAMP

[ UIG84 ] "User Interface Guideline", Apple Computer Inc., 1984 (draft), pp 5 -7

[ VMD89 ] Victorelli, E.Z., Magalhães, G.C. e Drummond, R., "Mecanismo de Gerenciamento de Versões e Configurações do A_HAND", Revista Brasileira de Computação , vol. 5, no. 2, Dez 1989, pp. 3-9

# APÊNDICE A

# MÓDULOS QUE COMPÕEM O EDITOR GRÁFICO DA LEGOSHELL

| lins9 | word2 | chars. | names |
|---|---|---|---|
| 76 | 116 | 20628 | actibarra.c |
| 31 | 67 | 614 | acticabe.c |
| 1450 | 3131 | 30492 | actiicones.c |
| 75 | 148 | 1317 | activistot.c |
| 90 | 186 | 1568 | actiwidget.c |
| 79 | 181 | 1527 | allocate.c |
| 208 | 643 | 5460 | avistot.c |
| 232 | 637 | 5893 | buildf.c |
| 159 | 477 | 4435 | cabecalho.c |
| 399 | 1326 | 10807 | color.c |
| 1087 | 2893 | 28670 | copy.c |
| 152 | 492 | 3547 | crazy.c |
| 529 | 1074 | 11491 | descreve.c |
| 4 | 2 | 20 | dir.c |
| 153 | 337 | 3946 | edgraf.c |
| 380 | 728 | 8347 | edtop.c |
| 958 | 4114 | 34354 | engine.c |
| 209 | 792 | 5481 | error.c |
| 2041 | 6205 | 69565 | exemplo.c |
| 215 | 923 | 5461 | expfn.c |
| 26 | 47 | 736 | geo.c |
| 260 | 731 | 5992 | hashtabl.c |
| 231 | 424 | 6913 | hierarc2.c |
| 215 | 649 | 4799 | lista.c |
| 35 | 139 | 1195 | lixo.c |
| 718 | 1947 | 22300 | main.c |
| 718 | 1946 | 22332 | mainbw.c |
| 1499 | 5209 | 41729 | mbarra.c |
| 1872 | 6866 | 55012 | micones.c |
| 117 | 352 | 2760 | mncimplem.c |
| 345 | 693 | 5309 | objeto.c |
| 392 | 441 | 4146 | onentry.c |
| 34 | 57 | 692 | pilha.c |
| 581 | 2012 | 17794 | pintel.c |
| 1644 | 5803 | 50510 | pinte2.c |
| 492 | 1151 | 11165 | rapaga.c |
| 30 | 58 | 1164 | repre.c |
| 1206 | 3007 | 30635 | rprint.c |
| 978 | 2580 | 24179 | rrender.c |
| 1000 | 2633 | 24411 | rrendpar.c |
| 44 | 121 | 1053 | save.c |
| 119 | 294 | 2511 | savemain.c |
| 1025 | 4021 | 29655 | savetype.c |
| 130 | 354 | 2975 | stack.c |
| 796 | 2133 | 22116 | top.c |
| 166 | 550 | 3910 | tree.c |
| 29 | 132 | 1501 | ttt.c |
| 51 | 90 | 833 | verif.c |
| 80 | 323 | 2804 | Load.h |
| 76 | 366 | 2605 | Loadp.h |

| | | | |
|---|---|---|---|
| 251 | 307 | 3166 | act.h |
| 254 | 381 | 3922 | activate.h |
| 15 | 37 | 338 | ahand.h |
| 12 | 18 | 134 | buildf.h |
| 22 | 65 | 542 | color.h |
| 27 | 45 | 697 | crazy.h |
| 61 | 145 | 991 | def1.h |
| 25 | 61 | 482 | def2.h |
| 54 | 167 | 1675 | eng.h |
| 199 | 397 | 2773 | entrando.h |
| 30 | 171 | 1112 | error.h |
| 56 | 194 | 1413 | flist.h |
| 78 | 278 | 1935 | global.h |
| 33 | 117 | 1116 | hashtabl.h |
| 231 | 424 | 7426 | hierarc2.h |
| 144 | 524 | 4246 | libtree.h |
| 64 | 192 | 2486 | listeve.h |
| 99 | 297 | 3851 | listhier.h |
| 358 | 479 | 4660 | main.h |
| 48 | 41 | 404 | menus.h |
| 7 | 14 | 105 | mydebug.h |
| 475 | 1648 | 14751 | objeto.h |
| 368 | 461 | 5016 | onentry.h |
| 92 | 207 | 2433 | option.h |
| 19 | 49 | 340 | perm.h |
| 27 | 77 | 701 | stack.h |
| 17 | 52 | 423 | status.h |
| 222 | 1652 | 7678 | symtab.h |
| 14 | 32 | 202 | trace.h |
| 26 | 76 | 676 | tree.h |
| 14 | 28 | 309 | actibarra.u |
| 16 | 29 | 308 | actiicones.u |
| 15 | 44 | 482 | allocate.u |
| 29 | 53 | 491 | buildf.u |
| 18 | 52 | 560 | copy.u |
| 2 | 4 | 43 | dir.u |
| 14 | 36 | 396 | edgraf.u |
| 13 | 34 | 375 | edtop.u |
| 48 | 181 | 1385 | engine.u |
| 20 | 52 | 552 | error.u |
| 17 | 49 | 537 | exemplo.u |
| 12 | 32 | 353 | geo.u |
| 14 | 42 | 462 | lista.u |
| 56 | 114 | 1350 | main.u |
| 159 | 189 | 2065 | micones.u |
| 34 | 55 | 644 | mnoimplem.u |
| 77 | 212 | 2142 | onentry.u |
| 11 | 32 | 355 | pilha.u |
| 20 | 40 | 417 | pinte.u |
| 24 | 62 | 645 | rapaga.u |
| 12 | 32 | 356 | repre.u |
| 16 | 54 | 580 | rprint.u |
| 19 | 59 | 627 | rrender.u |
| 28 | 100 | 1004 | savemain.u |
| 63 | 140 | 1099 | savetype.u |
| 17 | 54 | 567 | top.u |
| 8 | 12 | 130 | ttt.u |
| 9 | 28 | 333 | allocate.e |
| 9 | 28 | 332 | copy.e |
| 9 | 23 | 277 | edgraf.e |
| 8 | 29 | 268 | edtop.e |
| 15 | 48 | 531 | error.e |
| 7 | 21 | 258 | exemplo.e |
| 8 | 29 | 262 | geo.e |
| 19 | 73 | 831 | lista.e |
| 78 | 236 | 11396 | out.e |
| 9 | 29 | 269 | pilha.e |
| 17 | 78 | 836 | pinte.e |
| 9 | 28 | 345 | rapaga.e |
| 9 | 29 | 269 | repre.e |
| 9 | 28 | 338 | rprint.e |
| 9 | 28 | 332 | rrender.e |
| 8 | 29 | 259 | savemain.e |
| 11 | 30 | 353 | top.e |
| 1 | 5 | 37 | listupd1.awk |
| 1 | 9 | 64 | listupd2.awk |
| 38 | 402 | 2527 | icon/cabs.bit |
| 38 | 402 | 2527 | icon/cadm.bit |
| 52 | 577 | 3624 | icon/cblac.bit |
| 52 | 577 | 3624 | icon/ccmpr.bit |
| 50 | 571 | 3578 | icon/ccomp.bit |
| 36 | 396 | 2483 | icon/ccon.bit |
| 52 | 577 | 3624 | icon/cdevi.bit |
| 8 | 47 | 312 | icon/cdown.bit |
| 38 | 402 | 2527 | icon/cedi.bit |
| 38 | 402 | 2527 | icon/cexc.bit |
| 38 | 402 | 2527 | icon/cexi.bit |
| 38 | 402 | 2527 | icon/cexp.bit |
| 52 | 577 | 3624 | icon/cfile.bit |

```
   18    186    1166 icon/cin.bit
    9     81     515 icon/cleft.bit
   38    402    2527 icon/cloa.bit
   52    577    3624 icon/cmail.bit
   20    192    1213 icon/cout.bit
   52    577    3624 icon/cpipe.bit
   38    402    2527 icon/cqui.bit
   38    402    2527 icon/crep.bit
   11     87     566 icon/cright.bit
   38    402    2527 icon/csel.bit
   52    577    3624 icon/cstar.bit
   38    402    2527 icon/csto.bit
    8     47     302 icon/cup.bit
   18    172    1092 icon/lcmpr.bit
   18    172    1092 icon/lcomp.bit
   26    269    1698 icon/ldevi.bit
   26    262    1656 icon/lfile.bit
   18    172    1092 icon/lmail.bit
   18    172    1092 icon/lstar.bit
29751  94095  839899 total
```