Um Ambiente para Edição e Animação de Sistemas de G-Nets

Márcia Verônica Costa Miranda

Dissertação de Mestrado submetida à Coordenação do Curso de Pós-Gradução em Informática da Universidade Federal da Paraíba - Campus II como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Engenharia de Software

Jorge César Abrantes de Figueredo, D.Sc.
Orientador
Angelo Perkusich, D.Sc.
Orientador

Campina Grande, Paraíba, Brasil

Um Ambiente para Edição e Animação de Sistemas de G-Nets

Márcia Verônica Costa Miranda

Dissertação de Mestrado apresentada em Dezembro de 1996

Jorge César Abrantes de Figueredo, D.Sc.
Orientador
Angelo Perkusich, D.Sc.
Orientador

Giovanni Cordeiro Barroso, D.Sc.
Componente da Banca
Arturo Hernandez Domínguez, Doutor.
Componente da Banca

Campina Grande, Paraíba, Brasil, Dezembro de 1996

M672a Miranda, Marcia Veronica Costa
Um ambiente para edicao e animacao de sistemas de G-Nets / Marcia Veronica Costa Miranda. - Campina Grande, 1996.
104 f. : il.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. NetGraph - 2. Modelagem de Sistemas de Software 3. Redes de Petri 4. Edicao Grafica 5. Dissertacao I. Figueiredo, Jorge Cesar Abrantes de, Dr. II. Perkusich, Angelo, Dr. III. Universidade Federal da Paraiba - Campina Grande (PB) IV. Título

CDU 004.92(043)

UM AMBIENTE PARA EDIÇÃO E ANIMAÇÃO DE SISTEMAS DE G-NETS

MÁRCIA VERÔNICA COSTA MIRANDA

DISSERTAÇÃO APROVADA EM 30.12.96

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Presidente

PROF. ANGELO PERKUSICH, D.Sc
Examinador

PROF. GIOVANNI CORDEIRO BARROSO, D.Sc
Examinador

PROF. ARTURO HERNÁNDES DOMÍNGUEZ, Dr.
Examinador

CAMPINA GRANDE - PB

Dedicatória

Dedico este trabalho à Érico, companheiro de todos os momentos, Caio e André, filhos queridos e iluminação de meu caminho. Vocês são a razão de minha vida.

Agradecimentos

À Deus, acima de tudo, iluminação em todos os momentos de minha vida.

À meus orientadores, Jorge Cesar Abrantes de Figueiredo e Angelo Perkusich, pela confiança, dedicação e apoio dispensados na elaboração deste trabalho e por terem sido, não apenas meus professores, mas meus verdadeiros amigos, sempre dispostos a ajudar em horas difíceis. Agradeço-os por tudo.

À meus pais, José Francisco e D. Zefinha, por tudo que fizeram, pelos sacrifícios enfrentados para proporcionarem sempre o melhor a seus filhos. À meu irmão, Jefferson, pelo apoio e contribuições.

Aos professores, membros da banca examinadora, Giovanni Cordeiro Barroso e Arturo Hernandez Domínguez, pelos comentários e sugestões.

À César Augusto B. Santos, pela valiosa contribuição na implementação da ferramenta, pela amizade e incentivo em momentos difíceis encontrados durante o trabalho.

Aos colegas professores e funcionários do Departamento de Ciências Fundamentais e Sociais, do Centro de Ciências Agrárias desta Universidade, pela compreensão.

A todos professores do DSC e funcionários da COPIN (Aninha, Vera, Terezinha e Marcelo), pela presteza e atenção durante estes anos.

Às amigas Adriana e Lígia, pela paciência e compreensão.

Aos amigos alunos da Pós-graduação e graduação em Informática pela convivência sadia e valorosas discussões. Gostaria de citar, em especial, Evandro Costa, Dálmer B. de Azevedo Júnior, Adriano Sérgio Santos, José Antônio de Farias, Érica, Vera Lúcia Santos, Isaac Douglas, Robson, Gilson, Sonia Leila Silva, Marzina e Cláudia Procópio.

A todos, enfim, que contribuíram de alguma forma para a realização deste trabalho, desculpando-me e agradecendo aqueles que me ajudaram e não contam nesta lista.

À Érico, Caio e André, pela paciência e carinho, por me acompanharem e sempre estarem presentes a meu lado, enfrentando juntos as alegrias e tristezas em nossa caminhada.

Resumo

Esta dissertação discorre sobre o *NetGraph*, que é uma ferramenta gráfica e interativa para a especificação, modelagem e concepção de sistemas de software. Foi projetado para capacitar o profissional especialista a desenvolver modelos de sistemas baseados em uma classe de redes de Petri baseadas em objetos denominada *G-Nets*.

O *NetGraph* permite além da edição gráfica de *G-Nets*, a animação através de uma interface com um simulador distribuído para *G-Nets*.

Apresentamos a definição da arquitetura do ambiente de desenvolvimento para *G-Nets*, do qual o *NetGraph* é de fundamental importância. Detalhamos a concepção da ferramentas, bem como a estrutura de classes usada na sua implementação. Além disto detalhamos suas funcionalidades e exemplificamos sua utilização através de uma sessão de operação com o *NetGraph*.

Abstract

This work presents an interative graphical tool to the specification, modeling, and design of software systems, named *NetGraph*. The tool aims at aiding the designer to model systems based on a class of object based Petri Net, named *G-Nets*.

The tool *NetGraph*, developed as a graphical editor for *G-Nets*, permits the animation through an interface with a distributed simulator for *G-Nets*.

We present the architecture of an environemt for the design based on *G-Nets*, at wich the *NetGraph* tool plays an important role. We detail the design of the tool, as weel as the class structure used in the implementation. We also detail the functionalities and ilustrate its use by means of an example of the use of *NetGraph*.

Índice

Lista de Figuras

Capítulo 1

Introdução

Do ponto de vista da engenharia de software, vários problemas podem surgir durante o ciclo de vida do desenvolvimento de software, isto é, durante as fases de definição de requisitos, especificação, concepção, projeto, codificação e teste. Um dos desafios na concepção de sistemas de software é garantir a confiabilidade, principalmente no caso de sistemas críticos, como por exemplo sistemas médicos, controle de tráfego aéreo e aplicações espaciais. Por confiabilidade de software, entende-se a utilização de métodos que ofereçam um alto grau de segurança, onde os requisitos do sistema reflitam as necessidades críticas dos usuários e sua implementação em software seja a representação exata do projeto.

Embora confiabilidade de software seja o principal interesse da área da computação, não é o único [39]. Um outro aspecto importante a considerar é a preocupação na redução dos altos custos de desenvolvimento e manutenção de sistemas de software. Uma significativa porção neste custo é verificado pela ausência de práticas rigorosas que eliminem erros na concepção e especificação do sistema, causados, basicamente, por imprecisão, ambiguidade e erros de planejamento. Uma alternativa a ser utilizada na engenharia de software para solucionar estes problemas é a utilização de métodos formais para a especificação de sistemas [39].

A utilização de métodos formais provê uma notação para a especificação formal

de um sistema através da qual suas propriedades são descritas de forma que possam ser verificadas e bem representadas. Em geral, métodos formais apresentam uma base matemática sólida, podendo ser usados para derivar as propriedades de um sistema. Tais modelos, quando utilizados como recursos da engenharia de software, permitem aos projetistas verificar o comportamento e validar a exatidão de um sistema ao invés de confiar apenas em testes exaustivos.

Uma das vantagens mais evidentes na utilização de técnicas formais na concepção de sistemas é que estas possuem bases sólidas para a análise da validade da especificação de sistemas, de modo que possíveis erros possam ser detectados ainda durante a fase de concepção.

Devido as suas características, tais como representação gráfica e formalismo matemático, redes de Petri estão entre um dos formalismos mais aplicados para a especificação e concepção de sistemas de software [25, 34]. Além de sua base formal, as redes de Petri possuem uma interpretação gráfica que, potencialmente, as habilita como uma ferramenta a ser aplicada por projetista sem domínio completo dos aspectos formais nos quais elas se baseiam.

Várias extensões têm sido propostas ao modelo clássico de redes de Petri com a finalidade de capacitá-las a suportar diversos atributos relacionados tanto à capacidade de modelagem funcional quanto ao aspecto temporal. Quanto ao aspecto funcional, foram introduzidas características para facilitar a representação gráfica de sistemas complexos, reduzindo o tamanho da rede que representa o sistema modelado. Quanto ao aspecto temporal, provê meios para representar requisitos de tempo associados ao modelo do sistema.

Embora já existam extensões de redes de Petri que provêm eficiência na modelagem de sistemas quanto à capacidade funcional e ao aspecto temporal, em projetos de sistemas complexos, usuários necessitam de ferramentas para estruturação que permitam trabalhar com partes selecionadas do modelo, abstraindo os detalhes de baixo nível das outras partes [31]. Uma proposta, para solucionar este problema, foi a introdução

de um modelo que incorpora às características de redes de Petri uma metodologia orientada a objetos para concepção e especificação de sistemas, possibilitando o desenvolvimento evolucionário de sistemas. Este modelo provê uma forma que permite gerenciar a complexidade de modelagem e representação de sistemas, uma vez que introduz mecanismos de abstração valiosos como encapsulamento e classificação.

O conceito de *G-Nets* e sistemas de *G-Nets* foi introduzido em [9, 10, 47]. *G-Nets* são uma abordagem baseada em redes de Petri para especificação e concepção modular de sistemas de informação, a qual busca a integração da teoria de redes de Petri com a abordagem de engenharia de software baseada em objetos. A motivação para esta integração é criar uma ponte entre o tratamento formal de redes de Petri e uma abordagem modular, orientada a objetos para especificação, concepção e prototipagem de sistemas complexos de software.

A notação de *G-Nets* incorpora noções de módulo e estrutura de sistema às redes de Petri, promovendo abstração, encapsulamento e fraco acoplamento entre módulos.

Um sistema modelado através de *G-Nets* consiste, portanto, de um conjunto de módulos independentes e fracamente acoplados (*G-Nets*) que podem ser organizados em diferentes estruturas de sistemas, tais como estruturas interativas e estruturas recursivas. O encapsulamento de uma *G-Net* se dá de tal forma que um módulo só acessa um outro através de um mecanismo bem definido chamado *abstração de G-Net* e nenhuma *G-Net* afeta diretamente a estrutura interna de outro(s) módulo(s) que compõe o sistema.

Dificilmente um projetista concebe um sistema complexo corretamente na primeira tentativa, então a concepção de um sistema é um processo evolutivo, no qual repetidas alterações são necessárias [5]. Além do mais, os benefícios potenciais da prototipagem dependem da habilidade de modificar o comportamento do protótipo com esforço substancialmente menor que o requerido para modificar e produzir sistemas. Estas características de modularização e abstação inerentes à *G-Nets* são importantes na construção e prototipagem de sistemas complexos, uma vez que elas são um modelo

executável do sistema e provêem um suporte para a concepção incremental e facilidades de modificações ou atualizações sucessivas durante o projeto do sistema. A ferramenta desenvolvida facilitar a concepção e manutenção de sistemas.

Considerando que redes de Petri são um modelo matemático e o fato de ser uma ferramenta gráfica para modelagem de sistemas, é importante a existência de um ambiente que auxilie o usuário na concepção e desenvolvimento de sistemas, provendo, desta forma, uma visualização gráfica da estrutura do sistema modelado.

Com a finalidade de desenvolver um editor e animador gráfico para *G-Nets*, desenvolvemos o *NetGraph*. O *NetGraph* tem como objetivo prover aos usuários um ambiente que, através da utilização de uma metodologia orientada a objetos, possibilite a representação formal e executável de projetos de sistemas. Além disso, definiu-se um protocolo de forma que o animador possa ser acoplado com o simulador de *G-Nets* e ferramentas de análise de sistemas complexos [8, 10, 28], desenvolvidas por Chen [7].

O ambiente é implementado em estações de trabalho SUN e sistema operacional UNIX. As linguagens de programação utilizadas neste ambiente foram C++ e OSF/MOTIF que provê suporte para a construção de interfaces gráficas usando componentes como: ícones, botões, menus, barras de rolagem, etc. e outros mecanismos para criação de elementos gráficos.

Esta dissertação está organizada da seguinte forma: no capítulo 2, revisamos os conceitos básicos relacionados com a metodologia orientada a objetos da engenharia de software, redes de Petri e sistemas de *G-Nets*. No Capítulo 3 apresentamos, de forma sucinta, os critérios para desenvolvimento de interfaces gráficas com o usuário e a descrição de algumas ferramentas desenvolvidas dentro do contexto de redes de Petri. No Capítulo 4 descrevemos em detalhes o *NetGraph*, destacando a sua implementação, sua arquitetura e sua estrutura de classes. O Capítulo 5 contém a descrição dos comandos e das opções de operacionalização da ferramenta . No Capítulo 6, apresentamos a conclusão desta dissertação. O final do trabalho contém o Apêndice A onde apresentamos um exemplo de modelagem de uma *G-Net* utilizando o *NetGraph*.

Capítulo 2

Conceitos Básicos

O paradigma orientado a objetos na engenharia de software já é bastante popular atualmente. A aplicação deste paradigma pode ser identificada nos mais diversos domínios, como: construção de sistemas operacionais, sistemas de telefonia, banco de dados e aplicações multimídia, entre outros. Este paradigma modela sistemas [28], considerando-os como uma coleção de objetos cooperando entre si, operando com objetos individuais como instâncias de uma classe dentro de uma hierarquia de classes que compõem o sistema. Desta forma, torna-se um paradigma interessante para modelagem de sistemas complexos.

Uma das ferramentas mais poderosas para modelagem e análise de sistemas são as *redes de Petri*. Redes de Petri são um modelo matemático e gráfico aplicável na modelagem de muitos sistemas, principalmente sistemas de informações que são concorrentes, assíncronos e distribuídos. Além de seus recursos para visualização da estrutura do modelo, as redes de Petri possibilitam a análise do comportamento do sistema, seja de modo formal, por ser um modelo matemático, ou através de simulação, por ser um modelo executável.

Neste capítulo, apresentamos os conceitos básicos sobre o paradigma orientado a objetos, redes de Petri e definições relacionadas a sistemas de *G-Nets*. Apresentamos, finalmente, um exemplo de modelagem com *G-Nets* com a intenção de ilustrar os

conceitos apresentados.

2.1 Concepção Orientada a Objetos

Nesta seção abordamos as características do paradigma da engenharia de software que utiliza orientação a objetos para concepção de sistemas. Além de abordarmos seus conceitos básicos, fazemos um breve relato sobre suas características, vantagens e elementos principais que o compõem.

2.1.1 Introdução

Na década de 80, a utilização do *paradigma orientado a objetos* na engenharia de sofware começou a ser introduzido como uma abordagem poderosa no suporte ao processo de desenvolvimento de sistemas [41]. Hoje, este paradigma está se consolidando como método para desenvolvimento de sistemas e como complemento para outros métodos tradicionais. Ao invés de examinar o problema usando o modelo clássico de fluxo de informação (entrada-processamento-saída), o paradigma orientado a objetos introduz um novo conjunto de conceitos, como proposta para solucionar problemas inerentes aos métodos tradicionais, como: longo tempo de desenvolvimento de sistemas, dificuldades de manutenção e de rápida adequação às mudanças solicitadas durante sua utilização.

O paradigma orientado a objetos cria uma representação do domínio do problema do mundo real, mapeando-o para o domínio da solução [5]. Desta forma, aplicações são modeladas como classes de objetos inter-relacionadas e operações que modularizam informações e processos. Além disto, provê suporte para decomposição dos modelos do sistema, baseando-se nos conceitos de classes e abstrações de objetos para estruturar sistemas.

Esconder informação é uma estratégia de análise e projeto de sistemas, em que o maior número de informação possível é escondida dentro de seus componentes. Sua

premissa básica é enfatizar o controle lógico e a estrutura de dados do sistema, gerenciando a complexidade do sistema, definindo qual a sua funcionalidade e abstraindo detalhes dos níveis mais baixos da implementação.

Concepção orientada a objetos é baseado em *esconder informações* [35]. Difere da abordagem funcional para concepção de sistemas já que a visão do sistema de software é como um conjunto de objetos interagindo, com seus próprios estados privados, ao invés de um conjunto de funções.

As características do paradigma orientado a objetos são [35]:

- *Comunicação através de troca de mensagens.* Objetos se comunicam pelo envio de mensagens ao invés de compartilhar variáveis. Isto reduz o acoplamento global do sistema, já que não há possibilidade de modificações inesperadas de informações compartilhadas;
- *Objetos são entidades independentes* que podem ser modificados porque o estado e a representação da informação são internos aos objetos. Assim, não é possível acesso ou alteração acidental desta informação por outros objetos;
- *Objetos podem ser distribuídos* e executados sequencialmente ou em paralelo.

O paradigma orientado a objetos oferece um número de benefícios significativos que os outros modelos não provêem, entre os quais:

- Ajuda a explorar o poder expressivo de linguagens de programação orientada a objetos;
- O sistema desenvolvido é mais fácil de ser mantido, já que os objetos são independentes. Mudar a implementação de um objeto ou adicionar serviços não devem afetar outros objetos do sistema;
- A reutilização de componentes é mais simples, já que os componentes são encapsulados e os objetos são independentes. Novos sistemas podem ser desenvolvidos usando objetos existentes em sistemas criados anteriormente;

- Facilita o entendimento da concepção pois, para algumas classes de sistemas, existe um mapeamento entre as entidades do mundo real e os correspondentes objetos;

- O paradigma orientado a objetos se aproxima do trabalho de cognição humana, uma vez que, para diversas pessoas que não são profissionais de informática, identificam na idéia operacional deste paradigma uma forma natural de desenvolvimento de sistemas.

Uma das dificuldades observadas na utilização inicial deste paradigma é a identificação dos objetos do sistema. Esta dificuldade origina-se do fato de que é natural, do ponto de vista do projetista acostumado a desenvolver sistemas de forma procedimental, ter uma visão funcional do modelo, sendo difícil adaptar-se a conceituação da visão orientada a objetos.

A concepção de sistemas baseada no paradigma orientado a objetos não depende de linguagem de implementação específica para a codificação de sistemas por ela modelados. Linguagens de programação orientadas a objetos e pacotes com capacidade de encapsulamento de dados tornam projetos orientados a objetos mais simples de serem implementados. Porém, projetos orientados a objetos podem ser implementados em linguagens que não possuem estas características. O princípio de projetar um sistema como um conjunto de objetos interagindo é independente da linguagem utilizada, embora seja obviamente mais fácil implementá-lo se a linguagem tem disponível suporte para objetos.

2.1.2 Principais Características

Um objeto pode ser definido como uma entidade que [5]:

1. apresenta um conjunto de estados;

2. é caracterizado pelas ações que sofre e que pode requisitar de outros objetos;

3. é uma instância de alguma classe;

4. tem restrita visibilidade de outros objetos; e,

5. pode ser visualizado tanto por sua especificação como por sua implementação.

Os conceitos de classes e objetos estão fortemente relacionados, já que não podemos falar sobre objetos sem referenciá-los a sua classe. Porém, há diferenças importantes entre eles. Enquanto um objeto é uma entidade concreta que existe no tempo e espaço, a classe representa uma abstração, a essência do objeto. Uma *classe* é um grupo ou conjunto de objetos com atributos ou características comuns, baseados na qualidade, grau de competência ou condição. No contexto do paradigma orientado a objetos, o conjunto de instâncias de uma classe é um grupo de objetos que compartilham uma estrutura e comportamentos comuns [5]. O objeto é apenas uma instância de uma classe.

Os objetos se comunicam entre si através do recebimento e envio de mensagens, utilizando uma Interface, e especificam qual método do objeto é para ser executado. O objeto que recebe a mensagem determina como a operação solicitada é implementada.

Métodos representam a realização de funcionalidades de objetos, determinando como o objeto atuará quando receber uma mensagem. Os *métodos* operam em resposta às mensagens e manipulam os valores das variáveis de instância.

Uma mesma mensagem pode resultar em ações completamente diferentes quando recebidas por objetos diferentes. Esse fenômeno é denominado *polimorfismo*.

A noção de objeto não provê apenas uma integração dos dados e suas operações mas separa o comportamento intrínseco do objeto (visão externa) de sua definição (visão interna). Esse princípio de *encapsulamento* determina o ponto de vista do paradigma de orientação a objetos que enfatiza como o objeto aparece e não o que ele é.

Há quatro elementos principais do modelo orientado a objetos, são eles:

- Abstração;

- Encapsulamento;

- Modularidade;
- Hierarquia.

Apresentamos, a seguir, a descrição destes elementos.

Abstração

Shaw [40] define abstração como uma descrição simplificada, ou especificação, de um sistema que enfatiza alguns dos detalhes do sistema ou propriedades enquanto suprime outras. Uma boa abstração é aquela que enfatiza detalhes que são significativos para o usuário e suprime detalhes que são irrelevantes.

Uma abstração enfoca a visão externa do objeto e serve para separar o comportamento essencial do objeto de sua implementação. Além disto, uma abstração exprime as características relevantes de um objeto, diferenciando-o dos demais tipos de objetos que compõem o projeto, limitando-o, conceitualmente, em relação às perspectivas do usuário.

Encapsulamento

O princípio de encapsulamento define que nenhuma das partes componentes de um sistema, obtidas durante o seu desenvolvimento, deveriam depender de detalhes internos de outras partes.

Encapsulamento enfoca a implementação que executa o comportamento do objeto do modelo. É alcançada através do conceito de esconder informações, que consiste em esconder todas as estruturas de um objeto que não contribuem para suas características principais.

Enfim, encapsulamento é o processo de dividir os elementos de uma abstração [5], constituindo a estrutura e o comportamento do objeto; servindo para separar a interface contratual de uma abstração de sua implementação.

Modularidade

Modularização consiste em dividir o sistema em módulos, que são construídos separadamente, e têm conexões entre si. Estas conexões constituem a forma como os módulos executam operações ou serviços uns dos outros.

A finalidade da decomposição do sistema em módulos é reduzir o custo de desenvolvimento, permitindo que módulos sejam desenvolvidos e revisados independentemente.

O projetista deve construir módulos que são coesivos - agrupando-os, logicamente, através de abstrações comuns - e fracamente acoplados - os módulos devem ter dependência mínima entre si.

Assim, modularidade é a propriedade de um sistema que foi decomposto em um conjunto de módulos coesivos e fracamente acoplados [5].

Os princípios de abstração, encapsulamento e modularidade são sinergísticos. Um objeto provê uma cápsula ao redor de uma abstração e ambos, encapsulamento e modularidade, provêm barreiras ao redor da abstração.

Hierarquia

Abstração é um excelente conceito a ser utilizado em desenvolvimento de sistemas, mas em qualquer aplicação, mesmo a mais trivial, podem ser detectadas diferentes abstrações para um só entendimento do modelo. Encapsulamento ajuda a gerenciar esta complexidade escondendo a visão interna das abstrações. Modularidade provê um meio de agrupar logicamente as abstrações relatadas.

O desenvolvimento de sistemas complexos, frequentemente, gera alguma forma de hierarquia, já que sempre que este sistema é decomposto em módulos interrelacionados, estes módulos, também, podem ser divididos em seus próprios sub-módulos, e assim por diante, até atingir o nível mais baixo de componente elementares.

O conjunto de módulos compõem um conjunto de abstrações, formando uma hierarquia. Assim, hierarquia é o ordenamento ou posicionamento das abstrações identificadas no sistema.

Herança é a mais importante hierarquia e é um elemento essencial de sistemas orientados a objetos. É uma relação que permite a organização de objetos em termos de hierarquias de classes, e permite o compartilhamento de definições comuns por parte de uma classe.

2.2 Redes de Petri e Concepção Orientada a Objetos

Na concepção de sistemas complexos, os usuários precisam de uma ferramenta que os permitam trabalhar com partes selecionadas do modelo, abstraindo detalhes de outras partes. O casamento entre o formalismo de redes de Petri e a abordagem orientada a objetos da engenharia de software surge como uma solução com fundamentação formal para solucionar este problema. Entre vários modelos que utilizam esta direção podemos citar [2, 3, 4, 6, 11, 20, 44]. Neste trabalho, tratamos, mais especificamente, de sistemas de *G-Nets* que é o modelo utilizado para o desenvolvimento de nossa ferramenta.

No final desta seção, apresentamos uma solução para o problema produtor/consumidor utilizando *G-Nets* com o objetivo de ilustrar esta apresentação.

2.2.1 Redes de Petri

Antes de introduzirmos formalmente os conceitos de redes de Petri, faremos uma descrição geral, visando prover ao leitor uma intuição sobre seu conceitos básicos.

Redes de Petri são uma ferramenta de modelagem gráfica e matemática que pode ser aplicada em diversos tipos de sistemas. Aplicam-se apropriadamente a sistemas assíncronos e com alto índice de concorrência ou paralelismo. São, portanto, áreas privilegiadas as redes de computadores e protocolos de comunicações, sistemas operacionais, programação paralela, bancos de dados distribuídos e sistemas flexíveis de manufatura, entre muitas outras.

Uma Rede de Petri é um grafo direcionado bipartido, mais um estado inicial de-

nominado marcação inicial [26]. O grafo direcionado consiste de dois tipos de nós, denominados *lugares* e *transições*. Os nós em uma rede de Petri são relacionados (conectados) por arcos rotulados com pesos (números inteiros positivos). Um arco não pode relacionar componentes do mesmo tipo. Graficamente, lugares são representados por círculos e transições, por retângulos. Um lugar p é entrada para uma transição t se existe um arco direcionado conectando o lugar à transição, neste caso o lugar é um *lugar de entrada*. Um lugar p é saída para uma transição, se existe um arco direcionado conectando a transição ao lugar, neste caso o lugar é um *lugar de saída*. O grafo direcionado define a estrutura de um sistema representado por uma rede de Petri. A definição, informalmente, introduzida para redes de Petri é também denominada *grafo de suporte* ou *estrutura da rede*.

Uma marcação atribui a cada lugar p um número k, inteiro não-negativo, de elementos denominados *fichas*. Quando um número k, não negativo, de fichas é atribuído ao lugar, dizemos que o lugar está marcado com k fichas. Uma marcação é um vetor com o mesmo número de lugares que a estrutura da rede. O comportamento do sistema modelado pela estrutura da rede pode ser caracterizado pelo movimento de fichas através dos lugares quando a rede é *executada*. Este movimento de fichas caracteriza o comportamento dinâmico do sistema em termos de estados e suas mudanças. As fichas se movimentam através do disparo das transições, que devem estar *habilitadas* na marcação corrente para poderem disparar. Uma transição está habilitada quando todos os lugares de entrada são marcados, por pelo menos, o mesmo número de fichas definido pelo peso associado aos arcos conectando estes lugares à transição. Quando uma transição habilitada dispara, o número de fichas associados aos pesos dos arcos de entrada são removidos dos lugares de entrada e depositadas nos lugares de saída de acordo com os pesos associados aos arcos saindo da transição, conectando os lugares de saída. Este movimento de fichas é também conhecido como *jogo de fichas*.

2.2.2 Conceitos Básicos

Definição 2.1 Estrutura de Rede de Petri

Estrutura de Rede de Petri é uma 4-upla $N = (P, T, F, W)$, onde:

1. $P = \{p_1, p_2, ..., p_m\}$ é um conjunto finito de lugares;
2. $T = \{t_1, t_2, ..., t_n\}$ é um conjunto finito de transições;
3. $F \subseteq (P \times T) \cup (T \times P)$ é um conjunto finito de arcos;
4. $W : F \rightarrow I\!N^*$[1] é uma função peso; e,
5. $P \cap T = \phi$ e $P \cup T \neq \phi$.

Definição 2.2 Marcação de uma rede de Petri

Marcação de uma rede de Petri é uma função $M : P \rightarrow I\!N$.

$M(p)$ é a marcação do lugar p (número de **marcas** ou **fichas** contidas em p). A evolução da marcação simula o comportamento dinâmico de um sistema.

Definição 2.3 Rede de Petri

Rede de Petri é uma dupla $PN = (N, M_0)$, onde:

1. N é a estrutura da rede de Petri; e,
2. $M_0 : P \rightarrow I\!N$ é a marcação inicial.

Ou seja, uma rede de Petri é uma estrutura de rede mais uma marcação inicial.

Os elementos de P (lugares) são também denominados de *p-elementos.*

Os elementos de T (transições) são também denominados de *t-elementos.*

Na representação gráfica de uma rede de Petri, os lugares são representados por círculos e as transições, por retângulos ou barras. Os arcos, que representam uma

[1] $I\!N^*$ é o conjunto dos inteiros positivos: {1, 2, 3, 4, ...}, e $I\!N$ é o conjunto dos naturais: {0, 1, 2, 3, ...}

relação de fluxo e vão de um lugar para uma transição ou de uma transição para um lugar, são rotulados com seus pesos. O valor 1 é omitido e um arco com peso k pode ser interpretado como um conjunto de k arcos paralelos. As fichas são representadas por pontos no interior dos lugares. Podem-se usar também números indicando as quantidades de fichas nos lugares.

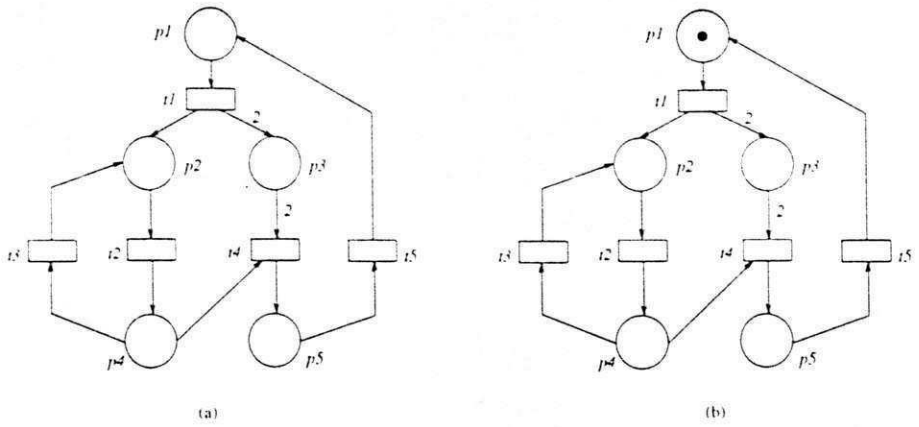

Figura 2.1: (a) Estrutura de uma rede e (b) rede de Petri

Informalmente, costuma-se referir à representação gráfica como se fosse a própria rede de Petri. Na Figura 2.1(a) temos a representação gráfica da estrutura (rede sem a marcação) de uma rede de Petri, enquanto que na Figura 2.1(b) temos a estrutura de uma rede de Petri com marcação.

Uma vez subentendida a ordem em que os lugares aparecem na notação das marcações, podemos indicar a marcação inicial da rede da Figura 2.1(b) simplesmente por $M_0 = (1, 0, 0, 0, 0)$.

Regra de Disparo

Para simular a dinâmica de um sistema, a marcação (ou estado) de uma rede de Petri evolui de acordo com a regra de disparo enunciada a seguir.

Definição 2.4 Regra de Disparo de uma rede de Petri

1. (Pré-condição): Uma transição t está *habilitada (ou é gatilhável)* se cada lugar de entrada p de t contém pelo menos $w(p,t)$ fichas, onde $w(p,t)$ é o peso do arco de p para t;

2. Uma transição habilitada pode ou não disparar (ou gatilhar); e,

3. (Pós-condição): Ao disparar t, $w(p,t)$ fichas são removidas de cada entrada p de t e $w(t,p)$ fichas são acrescentadas a cada saída p de t, onde $w(t,p)$ é o peso do arco de t para p.

Redes de Petri no Contexto de Sistemas Complexos

Usualmente, a concepção de sistemas complexos é baseada numa abordagem *top-down* ou *bottom-up*. A concepção *top-down* de um sistema inicia-se com uma descrição de alto nível do sistema. Esta descrição de alto nível é então redefinida através de sucessivos refinamentos, pela substituição de partes do modelo por modelos mais detalhados. Nesta abordagem, o projetista trata com sub-problemas relativamente pequenos, que podem ser distribuídos entre diversos projetistas. No caso da abordagem *bottom-up*, inicia-se com a concepção dos módulos de mais baixo nível do sistema, os quais são, então, abstraídos para obter a concepção de alto nível do sistema.

Entre outros formalismos adotados para a especificação e concepção de sistemas, redes de Petri [26] está entre os mais aplicados. Como anteriormente discutido, redes de Petri têm sido aplicadas para a modelagem de diversos sistemas que podem ser caracterizados como distribuídos, concorrentes e assíncronos.

Para modelar um sistema complexo, podem ser aplicadas técnicas de abstração e refinamento. No caso de redes de Petri, refinamento significa a substituição de uma transição ou um lugar por uma rede. Quando este tipo de substituição ocorre, as propriedades da rede refinada podem ou não ser preservadas.

Os maiores problemas, quando modelando grandes sistemas utilizando redes elementares ou redes lugar/transição, são: para um grande sistema as redes de Petri podem se tornar proibitivamente grandes; noções de estruturas de dados e dependência

de dados refinadas são perdidas, e, em muitas formulações de redes de Petri, somente *sistemas fechados* são descritos, de modo que a interação com algum tipo de ambiente não pode ser adequadamente expressada. Com a introdução do conceito de *fichas individuais*, representando elementos de dados fluindo na rede, foi possível modelar componentes comuns uma só vez, e atribuir fichas distinguíveis para cada componente idêntico. Este desenvolvimento levou a uma classe de redes de Petri denominda *redes de Petri de alto-nível*, incluindo *redes Predicado/Transição (redes PrT)* [13, 14], *redes de Petri Coloridas (redes CP)* [17], *redes Relação (Redes-Rel)* [38], e *redes de Petri com fichas individuais* [37]. Entretanto, mesmo com as redes de Petri de alto-nível, a modelagem de sistemas pode ainda ser difícil, pois não são providos meios, nestes modelos, para especificar a *estrutura* de um sistema. Por exemplo, não há conceito de hierarquia. Para remediar este problema, outras extensões às redes de Petri de alto-nível, incluindo o conceito de hierarquia, têm sido propostas. Os modelos de rede de Petri resultantes são denominados redes de Petri hierárquicas, incluindo *redes CP hierárquicas* [16, 17] e *redes PrT hierárquicas* [27]. Mesmo considerando que as construções hierárquicas introduzidas não extendem o poder de modelagem de redes de alto-nível, ferramentas de estruturação são introduzidas de modo a facilitar a construção de modelos de grandes sistemas na prática [16].

O problema de gerenciar grandes sistemas é principalmente uma questão de modularidade. Portanto, seria desejável compor grandes redes a partir de redes menores, de modo que a semântica da grande rede pudesse ser deduzida a partir das redes menores. Intuitivamente, isto pode ser caracterizado pela habilidade em *esconder* detalhes internos das redes e considerarem-se somente aqueles necessários.

Os sistemas de *G-Nets* surgiram neste contexto para suprir estas deficiências e possibilitar a utilização do paradigma orientado a objetos na modelagem de sistemas complexos.

2.2.3 G-Nets e Sistemas de G-Nets

Um princípio na engenharia de software amplamente aceito na concepção de sistemas, advoga que um sistema deve ser composto por um conjunto de módulos independentes. Cada módulo no sistema esconde detalhes internos de suas atividades de processamento. Além disto, módulos comunicam-se através de interfaces bem definidas [12]. A notação de *G-Nets* provê um forte suporte para este princípio. *G-Net* é uma estrutura baseada em redes de Petri para a especificação e concepção modular de sistemas distribuídos de informação. A estrutura é uma integração da teoria de redes de Petri com o paradigma de orientação a objetos para a concepção de sistemas. A motivação dessa integração é a ligação entre o tratamento formal de redes de Petri e a abordagem modular, orientada a objetos para a especificação e prototipagem de sistemas complexos de software. A notação de *G-Nets* incorpora às redes de Petri as noções de módulo e estrutura de sistemas e promove abstração, encapsulamento e fraco acoplamento entre módulos.

A especificação ou concepção baseada nas *G-Nets* consiste em um conjunto de módulos independentes e fracamente acoplados (*G-Nets*) organizados em termos de várias estruturas de sistemas. Uma *G-Net* é encapsulada de tal forma que um módulo só pode acessar um outro módulo através de um mecanismo bem definido chamado de *abstração de G-Net.*

Uma *G-Net*, G, é composta por duas partes: um lugar especial denominado *Lugar Genérico de Chaveamento* (GSP), e uma *Estrutura Interna* (IS). O GSP provê a abstração para um módulo, servindo como única interface entre a *G-Net* e outros módulos. A estrutura interna é uma rede de Petri modificada, e representa a realização interna detalhada da aplicação modelada. A notação de *G-Nets*, entre outras características, permite ao usuário indicar comunicação entre *G-Nets* e terminação. A notação para *G-Nets* é apresentada na Figura 2.2, e explicada como a seguir:

1. A estrutura interna da rede (IS) é definida por um retângulo com as bordas arredondadas, definindo o limite da estrutura interna.

2. O GSP é indicado por uma elipse no canto superior esquerdo do retângulo defi-

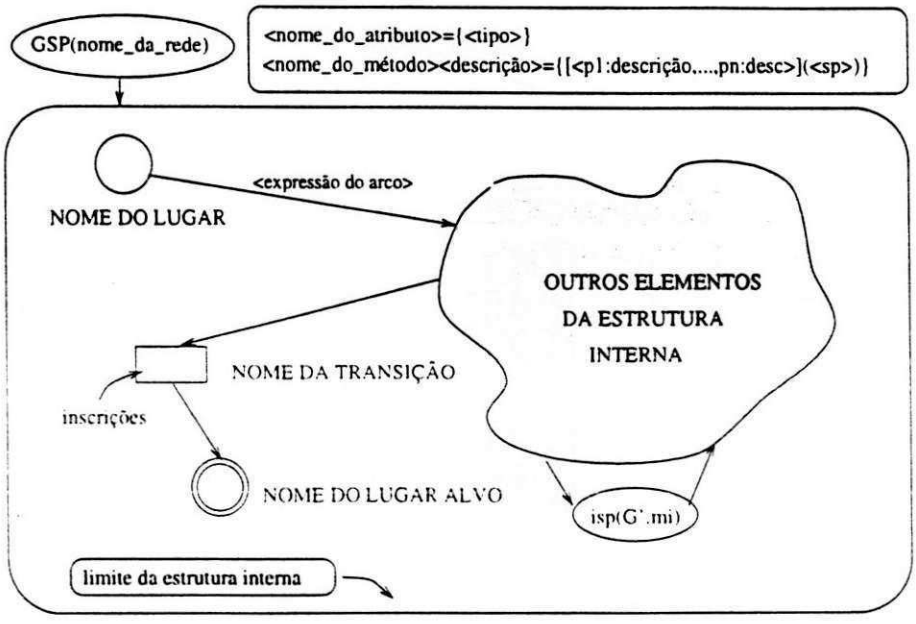

Figura 2.2: Notação utilizada para uma *G-Net*

nindo os limites da IS. A inscrição $GSP(nome\_da\_rede)$ define o nome da rede para ser referida por outras *G-Nets*.

3. O retângulo com as bordas arredondadas no canto superior direito do limite da IS é utilizado para identificar os métodos e atributos da rede, em que:

   - $\langle nome\_do\_atributo = \{\langle tipo \rangle\}$ define um atributo para rede, onde:

     $\langle nome\_do\_atributo \rangle$ é o nome do atributo, e

     $\langle tipo \rangle$ é o tipo para o atributo, o qual está restrito ao conjunto dos inteiros não negativos.

   - $\langle nome\_do\_método \rangle$ é o nome para um método, $\langle descrição \rangle$ é uma descrição para o método. $\langle$ *p1:descrição*, $\cdots$, *pn:descrição* $\rangle$ é uma lista de argumentos para o método. Por fim $\langle sp \rangle$ é o nome do *lugar inicial* para o método.

4. Um círculo representa um lugar normal.

5. Uma elipse na estrutura interna representa um *lugar de chaveamento para instanciação* (*instantiated switching place (isp)*). O *isp* é utilizado para prover

comunicação *inter-G-Nets*. A inscrição $isp(G'.mi)$ indica a invocação da rede G' com método mi.

6. Um retângulo representa uma transição, a qual pode ter uma inscrição associada. Esta inscrição pode tanto ser uma atribuição como uma restrição de disparo. Utilizaremos a notação padrão da Linguagem C tanto para as atribuições como para as restrições de disparo.

7. Um círculo duplo representa um lugar de terminação ou *lugar alvo*.

8. Lugares e transições são conectados por arcos direcionados que podem carregar uma expressão.

O *GSP* em uma *G-Net* G, denotado por $GSP(nome\_da\_rede)$ na elipse da Figura 2.2, unicamente identifica o módulo. O *GSP* contém a declaração para um ou mais métodos executáveis (descritos no retângulo com as bordas arredondadas na Figura 2.2), especificando as funções, serviços ou operações definidas para a rede, e um conjunto de atributos especificando propriedades passivas do módulo (quando definidas). A estrutura detalhada e o fluxo de informação de cada método são definidos por uma rede de alto-nível modificada na estrutura interna. Mais especificamente, um método define os parâmetros de entrada, a marcação inicial correspondente à rede de alto-nível interna (o estado inicial da execução). A coleção de métodos e os atributos (definidos) provêem a abstração ou visão externa do módulo.

Na estrutura interna, lugares representam *primitivas* e transições, juntamente com os arcos, representam conexões ou relações entre as primitivas. Um conjunto de lugares especiais, denominados *Lugares Alvo*, representa o estado final de uma execução, e os resultados (se algum) a serem retornados. Uma transição, juntamente com os arcos, define uma sincronização e coordena a transferência de informação entre seus lugares de entrada e saída.

Dada uma *G-Net* G, um *isp* em G é denotado por $isp(G_{nome}.mtd)$ (ou simplesmente $isp(G)$, se nenhuma ambigüidade ocorrer) onde G_{nome} é uma identificação única para

G, e *mtd* é um método definido para G. Um $isp(G_{nome}.mtd)$ denota uma instanciação da *G-Net* G, i.e., uma instância de invocação de G baseada no método *mtd*. Portanto, executando a primitiva *isp* implica na invocação de G (através do envio de fichas para G) utilizando um método especificado. As fichas contém os parâmetros necessários para a definição das fichas na marcação inicial da rede invocada. Esta interação entre *G-Nets* pode ser comparada ao método de chamada remota de procedimento [32].

O *isp* serve como mecanismo primário para a conexão ou relacionamento entre diferentes *G-Nets* (módulos). Embutindo um *isp* de uma *G-Net* de um nível inferior em uma *G-Net* em um nível superior, especifica uma configuração hierárquica. Ainda mais, embutindo um *isp* de uma *G-Net* especificando um servidor em uma *G-Net* especificando um cliente resulta em uma relação cliente-servidor [9].

Na Figura 2.3 apresentamos um exemplo de um sistema de *G-Nets* composto de duas *G-Nets*. Não estamos preocupados neste ponto com a funcionalidade das redes. A rede à esquerda é denominada *G1*, possuindo um método definido, o qual é *ma:processa a={[a:inteiro](P1)}*. Isto significa que somente um método está definido para *G1*, e que não há atributos definidos. O método *ma* recebe um inteiro a, e uma ficha com este valor é depositado no lugar inicial *P1*. Temos quatro lugares definidos, os quais são $\{P1,P2,P3,P4\}$, e duas transições, $\{t1,t2\}$. Os lugares *P1* e *P3* são lugares normais, e o lugar *P4* é uma lugar alvo. O lugar *P2* é um *isp*. A transição *t1* não tem inscrição alguma. Isto significa que a ficha depositada no lugar *P1* é removida sem nenhuma restrição, e é depositada nos lugares *P2* e *P3*, após o disparo da transição *t1*. A ficha depositada no lugar *P2* inicializa a invocação da rede $G2$. Após a rede $G2$ terminar sua execução, o resultado é retornado a $G1$ através do lugar alvo *P4* em $G2$. Finalmente, a transição *t2* de $G1$ dispara e a soma formal de a (a ficha em *P3*) e o valor retornado y (a ficha em *P2*) é calculada e depositado no lugar *P4*. Uma vez que *P4* é um lugar alvo, $G1$ termina sua execução e o valor resultante é retornado para a rede que invocou $G1$ (não representada na figura).

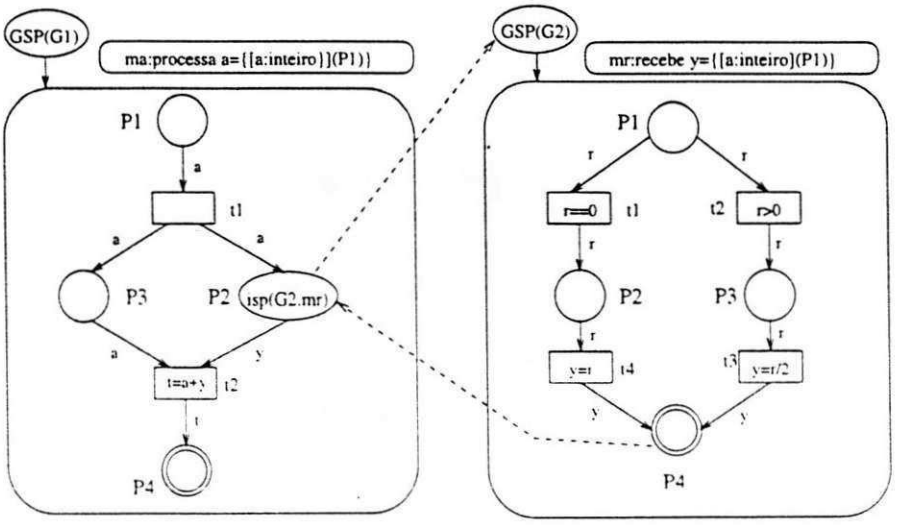

Figura 2.3: Duas *G-Nets* conectadas através de um *isp*

2.2.4 Formalização

Definição 2.5 Sistema de G-Nets

Um *sistema de G-Nets* (GNS) é uma tripla $GNS = (TS, GS, AS)$, onde

1. TS é uma coleção de fichas dinamicamente geradas durante a execução do sistema;
2. GS é um conjunto de *G-Nets*; e
3. AS é um conjunto, descentralizado, de agentes computacionais concorrentes, executando um sistema de *G-Nets*.

Definição 2.6 G-Net

Uma *G-Net* é a dupla $G = (GSP, IS)$, onde

1. GSP é um *Lugar Genérico de Chaveamento (GSP)* provendo uma abstração para a *G-Net*; e
2. IS é a *Estrutura Interna*, a qual é uma rede PrT modificada.

Definição 2.7 Lugar Genérico de Chaveamento (GSP)

Dado $GSP \in G$ ser um lugar genérico de chaveamento. Um GSP é definido por (NID, MS, AS), onde

1. NID é uma identificação única (nome) da *G-Net* G;

2. MS é um conjunto de métodos; e

   $\forall mtd_i \in MS$, $mtd_i = (\mathsf{m\_nome}_i, \mathsf{m\_argumentos}_i, \mathsf{m\_iniciador}_i)$, onde

   $\mathsf{m\_nome}_i$ é um nome para mtd_i.

   $\mathsf{m\_argumentos}_i$ é uma tupla de variáveis especificando a ficha de entrada para o método mtd_i.

   $\mathsf{m\_iniciador}_i$ é um mapeamento de $\mathsf{m\_argumentos}_i$ para a *marcação inicial* da rede pertencendo à estrutura interna associada ao método mtd_i.

3. AS é um conjunto de atributos, e $\forall as_i \in AS, as_i \in \mathbb{N}$.

A partir da Definição 2.7 dizemos que um GSP é unicamente identificado por um nome denotado por NID, abstraindo um conjunto de métodos denotado por MS. Os métodos definem como a estrutura interna pode ser executada. Na Definição 2.7 m_iniciador define como a informação em m_argumentos é transformada em fichas do tipo correspondente. Estas fichas serão depositadas, no caso de mais de um argumento ser definido, no lugar inicial para o método.

Definição 2.8 Estrutura Interna

A *estrutura interna* de uma *G-Net*, $G.IS$, é uma estrutura de rede, isto é, um grafo direcionado bipartido definido por $G.IS = (\Sigma, P, T, I, O)$, onde

1. Σ é uma estrutura consistindo de algum tipo de predicados, juntamente com o conjunto de relações e operações definidas para estes predicados.

2. P é um conjunto finito e não vazio de lugares, denotados por círculos.

3. T é um conjunto de transições, denotadas por retângulos.

4. $I : T \rightarrow P^{\infty}$ é denominada função de entrada, definindo inscrições para transições e arcos de entrada.

5. $O : T \rightarrow P^{\infty}$ é denominada função de saída, definindo inscrições para transições e arcos de saída.

Podemos dizer que a estrutura interna de uma *G-Net* é uma rede PrT modificada. A maior diferença é que agora podemos associar uma ação ou função aos lugares.

Definição 2.9 Conjunto de Lugares
Dado que $G.IS$ é a estrutura interna de uma *G-Net*, O conjunto de lugares $P \in G.IS$ é definido por $P = (\mathcal{ISP}, \mathcal{NP}, \mathcal{GP})$, onde

1. $\mathcal{ISP}$ é um subconjunto de *isp's*.

2. $\mathcal{NP}$ é um subconjunto de lugares normais.

3. $\mathcal{GP}$ é um subconjunto de lugares alvo.

A definição, acima apresentada para o conjunto de lugares $P \in G.IS$, é necessária pois cada subconjunto de lugares apresenta uma semântica diferente. Além disto, $\mathcal{ISP}$ e $\mathcal{GP}$ possuirão uma importância significativa para decomposição, análise e verificação de sistemas de *G-Nets*.

Um $isp \in \mathcal{ISP}$ provê mecanismos utilizados em sistemas de *G-Nets* para implementar interconexão entre *G-Nets*. Um *isp*, definido em uma rede G e invocando uma rede G', é denotado por $isp(G'm)$ e é definido pela quádrupla:

$$isp(G'm) = (NID, mtd, \text{ação\_antes}, \text{ação\_após})$$

NID é o identificador único de G', $mtd \in G'.GSP.MS$, ação_após e ação_antes são ações primitivas, cuja função é atualizar a *seqüência de propagação da ficha*, a qual introduziremos a seguir. Mais especificamente, um método $mtd \in G.MS$ define os parâmetros de entrada e a marcação inicial da rede de Petri interna (define o estado

inicial da execução). O conjunto de *Lugares Alvo* representa o estado final de execução para cada método e os resultados a serem retornados. A coleção de métodos e atributos (se algum estiver definido) provê a abstração ou visão externa do módulo. Impomos a restrição de que o conjunto de lugares alvo deve ser não vazio. Esta restrição é necessária para garantir que um método sempre alcance um estado final.

Definição 2.10 Ficha

Dado G ser uma *G-Net*, e tkn ser uma ficha, então

1. tkn é uma tupla de *itens* da forma $tkn = \langle scq, scor, d_1, \cdots, d_q \rangle$, onde $tkn.scq$ é a *seqüência de propagação* da ficha, $tkn.scor \in$ (antes, após) é a *cor de estado*, e $tkn.d_i$, $i \in \mathbb{N}$, é a mensagem associada à ficha. O item $tkn.scq$ é uma seqüência de $\langle NID, isp, PID \rangle$, onde NID é a identificação da *G-Net*, isp é o nome de um ISP e PID é um identificador único de processo. A ficha está disponível para habilitação e disparo de transições quando $scor =$ após.

2. Se $tkn = \omega$, ele pode ser casado com qualquer seqüência.

A seqüência de propagação da ficha somente é alterada em um ISP ou GSP. Quando uma *G-Net* G é invocada por um ISP (isp_i) em outra *G-Net* G' (quando isp_i recebe uma ficha), a tripla $\langle G', isp_i, Pid_{G'} \rangle$, onde $Pid_{G'}$ é o identificador do processo executando G', é associada à seqüência de propagação da ficha antes dela ser enviada à *G-Net* G. Esta tripla indica que quando a execução de G termina, a ficha resultante deve ser retornada ao lugar identificado por isp_i na *G-Net* G'. O identificador de processo é necessário para distinguir para qual instância de execução de G' a ficha sendo retornada pertence. Quando a ficha de entrada é recebida pelo GSP G, a tripla $\langle 0, 0, Pid_G \rangle$ é associada à seqüência de propagação da ficha, indicando que o agente responsável pela execução da invocação é identificado por Pid_G. Devido ao fato de Pid_G ser único, a seqüência de propagação é também única. A estrutura da ficha em *G-Nets* não somente garante que todas as fichas pertencendo à uma instância de execução de uma *G-Net* têm a mesma seqüência de propagação, mas também contém

a história completa de propagação das fichas, a qual governa as interações entre os processos executando um sistema de *G-Nets*.

Devido ao campo de seqüência de propagação associado à ficha, mais de uma invocação de uma *G-Net* pode ser executada simultaneamente. Isto é possível, pois diferentes seqüências de execução (invocações) podem ser unicamente identificadas.

Como mencionado anteriormente, a cor de estado da ficha pode assumir dois possíveis valores: antes ou após. Uma ficha é dita estar *disponível* se *scor* = após. Caso contrário, ela é dita estar *indisponível*. Quando uma ficha é depositada em um lugar, sua cor de estado é antes, e após a ação primitiva (se definida) ser executada, a cor da ficha é alterada para após, indicando que a ficha está pronta para ser utilizada para disparos de transições. O campo de mensagem de uma ficha é uma lista de indivíduos dependente da aplicação.

Um *lugar normal* $NP \in \mathcal{NP}$ possui as regras para manipular as fichas. Quando uma ficha é depositada em um NP, uma ação associada ao lugar é executada, baseada nos parâmetros especificados no campo msg. O resultado da ação é associado ao msg da ficha e a *cor de desvio* da ficha é definida. A cor de desvio da ficha serve para definir para qual transição de saída a ficha está disponível. Portanto, a mensagem associada aos lugares normais executa a mesma função das expressões associadas aos arcos e inscrições em transições para redes PrT. Finalmente o campo *scor* da ficha é atualizado para *scor* $\leftarrow$ após, e a ficha está disponível para habilitação e disparo de transições.

Um lugar alvo $GP_i \in \mathcal{GP}$ *deve pertencer* à marcação final de $G.IS$. Para cada método, *mtd*, os lugares alvo devem ser alcançáveis e devem ser únicos para cada método. Portanto, a informação resultante da execução pode ser retornada à *G-Net* que invocou.

O mecanismo de disparo de transição para *G-Nets* é definido pelas seguintes *regras de disparo*:

Definição 2.11 Regras de disparo para transições

Dado G ser uma *G-Net*, as regras de disparo de transição são definidas por:

1. Uma transição t é dita *habilitada* se e somente se cada $p \in I(t)$ possui pelo menos uma ficha, seu conteúdo satisfaz a condição definida por $I(p,t)$, e

   (a) todas as fichas envolvidas possuem a mesma seqüência de propagação;

   (b) todas as fichas envolvidas possuem suas *scor*= após;

   (c) o número de fichas definida em $O(t)$ é menor que a capacidade dos lugares.

2. Uma transição habilitada t pode *disparar* e quando dispara:

   (a) uma ficha satisfazendo $I(p,t)$ é removida de cada $p \in I(t)$;

   (b) uma ficha cuja seqüência de propagação é a mesma que a das fichas removidas de $I(t)$, cujas *scor* = após, e cuja campo de mensagem é definido por $O(t,p)$ é depositada em cada lugar $p \in O(t)$.

A invocação de uma *G-Net* define os mecanismos para iniciar a execução das estruturas internas, baseando-se na mensagem associada à ficha.

Definição 2.12 Invocação de uma G-Net
Dado G ser uma *G-Net*, a invocação da rede G baseada no método $mtd \in G.MS$ é conduzida da seguinte forma:

1. Determine a marcação inicial de G com base na definição para o método (o conteúdo das fichas depende da ficha de entrada);

2. Dispare as transições habilitadas, se alguma;

3. Invoque as primitivas habilitadas, se alguma;

4. Repita (2)-(3) até que um lugar alvo seja alcançado;

5. Envie o resultado da execução (se definida por mtd) para a rede que invocou.

2.2.5 Exemplificação da Aplicação de G-Nets

O problema produtor/consumidor é um problema bastante conhecido. Ele consiste da sincronização entre um ou mais produtores e um ou mais consumidores. Neste exemplo, assumimos que, inicialmente, um produtor é capaz de produzir n itens, e que o consumidor é capaz de consumir um item por vez.

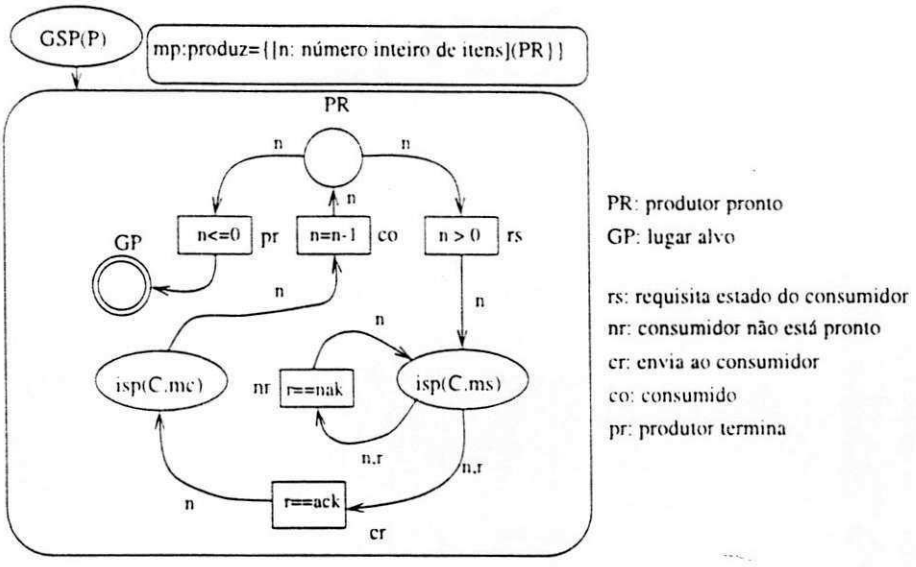

Figura 2.4: *G-Net G(P)* modelando o produtor

A Figura 2.4 mostra a *G-Net G(P)* para o produtor. Para esta rede, foi definido um método chamado *mp*, cuja função é produzir n itens para serem consumidos. Quando *G(P)* é invocada através de *GSP(P)*, uma ficha, juntamente com o campo n, expressando o número de ítens a produzir é depositada no lugar *PR*. A ação asociada ao lugar simplesmente decrementa o campo n. Se $n < 0$, a transição *pr* dispara e a invocação de *G(P)* termina quando o *lugar alvo GP* é alcançado. Se $n \geq 0$, a transição *rs* dispara, a ficha alcança o lugar *isp(C.ms)*, e a rede C é invocada utilizando o método *ms*. Esta invocação serve para consultar o estado de *G(C)*, de modo a garantir que se ela está pronta ou não para consumir um ítem. Se a rede *G(C)* não está pronta, a transição *nr* dispara, e a rede *G(C)* é consultada novamente. De outro modo, o consumidor está pronto e a transição *cr* dispara. Após o disparo de *cr*, uma ficha é depositada no

lugar *isp(C.mc)* e a rede *C* é invocada com o método *mc*, juntamente com o ítem a ser consumido. Quando a ficha é retornada pela rede *C*, a transição *co* dispara e uma ficha é novamente depositada no lugar *PR*.

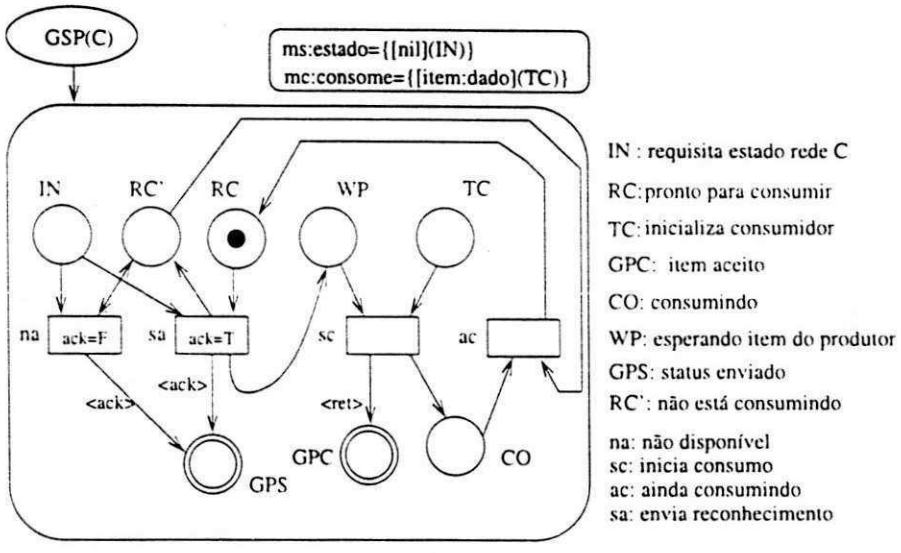

Figura 2.5: *G-Net G(C)* modelando o consumidor

A *G-Net* modelando o consumidor é mostrada na Figura 2.5. Dois métodos são definidos para *G(C)*: método estado (*ms*) e método consome (*mc*). Quando *G(C)* é invocada através de *GSP(C)*, se o método é *ms*, uma ficha é depositada no lugar *IN*. Dependendo do estado de *G(C)*, o qual pode ser tanto consumindo como pronta para consumir, a transição *na* ou *sa* disparará. A escolha entre *na* e *sa* baseia-se nos estados representados pelo lugar *RC* (pronto para consumir) e lugar *CO* (consumindo). Se *RC* está marcado, a transição *sa* dispara e uma ficha alcança o *lugar alvo GPS* com um campo *reconhecido (ack)* associado a ele. Contrariamente, um campo *não reconhecido (nak)* é associado à ficha. Após o disparo de *sa*, uma ficha é depositada no lugar de entrada *WP*, e o consumidor está pronto para consumir. Quando *G(C)* é invocada com o método consome, *mc*, uma ficha é depositada no lugar *TC*. Uma vez que *WP* foi previamente marcado, após uma execução com sucesso do método *ms*, a transição *sc* dispara e o lugar *CO* é marcado. Neste ponto, *G(C)* indica ao produtor a ação requerida que está sendo procesada ou ainda está em processamento, isto é, quando a

transição *sc* dispara, uma ficha é depositada no lugar *CO* e outra no lugar alvo *GPC*. Deve ser notado que a transição *ac* pode ou não disparar antes que o produtor receba a ficha de retorno de *GPC*. Após o disparo de *ac*, uma ficha é removida de *CO* e uma outra é removida de *RC'*, e uma ficha é depositada em *RC*. Logo *G(C)* está pronta para consumir um novo ítem.

Neste capítulo, apresentamos os conceitos básicos sobre metodologia orientada a objetos para desenvolvimentos de sistemas, além de apresentarmos uma visão geral sobre o formalismo gráfico e matemático para desenvolvimento de sistemas chamado redes de Petri e, finalmente, descrevemos uma extensão do modelo clássico de redes de Petri que utiliza metodologia orientada a objetos da engenharia de software para especificação, concepção e modelagem de sistemas complexos - sistemas de *G-Nets*. Apresentamos também um exemplo de modelagem com *G-Nets* para o problema produtor/consumidor. No próximo capítulo, apresentamos as características básicas de interfaces gráficas para usuário e a descrição sucinta de algumas ferramentas gráficas desenvolvidas para aplicações no contexto de redes de Petri.

Capítulo 3

Ferramentas Existentes

3.1 Introdução

Desenvolvemos um trabalho onde os critérios gráficos para a construção de interfaces com o usuário devem ser considerados. Apresentamos, nesta seção, uma síntese destes critérios, por uma questão de completude do documento ora apresentado. Posteriormente, fazemos uma descrição sucinta de ferramentas gráficas que são utilizadas dentro do contexto de redes de Petri semelhantes ao trabalho desenvolvido.

3.2 Princípios de Concepção de Interfaces

A concepção de interface com usuário leva em consideração as necessidades, experiência e capacidades do usuário do sistema [41].

Muitas ferramentas têm seu próprio estilo de interface, no entanto existem princípios gerais que podem ser aplicados a projetos de desenvolvimento de sistemas de interface homem-máquina. Os princípios mais importantes são:

1. A interface deve usar termos e conceitos familiares aos usuários.

2. A interface deve ser, apropriadamente, consistente.

3. O usuário não deve ser surpreendido com ações do sistema.

4. A interface deve incluir mecanismos que permitam aos usuários desfazer seus erros, quando usam o sistema. A interface pode minimizá-los, provendo mecanismos para facilitar a correção de erros cometidos.

5. A interface deve incorporar alguma forma de auxílio ao usuário.

O usuário não deve ser forçado a adaptar-se à interface por ser isto conveniente à implementação do sistema. A interface deve utilizar termos e objetos que tenham analogia direta com o ambiente do usuário.

A consistência da interface significa que comandos e menus do sistema devem ter o mesmo formato padrão de apresentação e a passagem de parâmetros deve ser semelhante para todos os comandos; isto facilita sua utilização, diminuindo o seu tempo de aprendizado.

Ao utilizar um sistema, o usuário constrói um modelo mental referente ao sistema que está utilizando e espera que o mesmo se comporte de acordo com aquele modelo. Se o sistema se comporta de maneira diferente para ações de contexto semelhante, o usuário fica surpreso e confuso. Assim, projetistas de interfaces devem elaborar sistemas onde ações semelhantes tenham efeitos semelhantes.

Enfim, interfaces devem ter facilidades de auxílio ao usuário, provendo, no sistema, diferentes níveis de ajuda e advertências. Devem conter informações básicas de como manipular o sistema até descrições detalhadas dos seus recursos.

Estes princípios enfatizam que o projeto de interface deve ser dirigido ao usário. Os projetistas devem ter em mente que o usuário, ao utilizar o sistema, tem uma tarefa a cumprir e a interface deve ser orientada em direção a esta tarefa.

3.2.1 Interfaces Gráficas do Usuário

Atualmente, as interfaces implementadas em computadores utilizam janelas, ícones, menus e dispositivos apontadores de modo a prover maiores facilidades e agilidade para a maioria dos usuários de computadores.

Estas interfaces são chamadas de *interfaces gráficas do usuário* que são caracterizadas por:

- Múltiplas janelas permitindo que informações diferentes sejam exibidas simultaneamente;
- Representação das informações através de ícones;
- Seleção de comandos via menus;
- Dispositivos apontadores, tais como mouse, para selecionar escolhas de um menu ou itens de interesse em uma determinada janela;
- Suporte à exibição de informações gráficas e visuais.

Dentre outras, podemos destacar as seguintes vantagens de interfaces gráficas:

- São fáceis de aprender e usar. Usuários iniciantes aprendem a usar a interface rapidamente, após poucas sessões de treinamento;
- Permitem ao usuário manipular múltiplas janelas para interação do sistema;
- Permitem interação rápida e *full-screen*, bem mais eficiente que interação orientada-a-linha como nas interfaces de comandos.

Até recentemente, a construção da interface gráfica dos sistemas era feita de acordo com os critérios de cada projetista ou das empresas, sem seguirem um padrão. Este tipo de tratamento dificultava o transporte destes sistemas para outros equipamentos, não aqueles sobre os quais estes sistemas tinham sido, especificamente, desenvolvidos.

Atualmente, tem-se procurado seguir uma padronização na construção de interfaces, principalmente, no caso do sistema operacional UNIX, com os padrões das interfaces X-Windows [19] e Motif [1]. No próximo capítulo, apresentamos um pouco mais de detalhes sobre estas interfaces.

Apresentamos, a seguir, as principais características que uma interface gráfica deve apresentar para facilitar e tornar mais agradável o trabalho do usuário.

Manipulação Direta

Permite ao usuário [15] manipular objetos que são uma representação visual do mundo real, capacitando-o a manipular componentes, alterar e mover objetos de forma semelhante como seria feito com o objeto real.

As vantagens de manipulação direta são:

- Os usuários sentem controle do computador e não são intimidados por ele;
- O tempo de aprendizado de um usuário típico é menor;
- Os usuários têm respostas imediatas de suas ações, possibilitando que erros sejam detectados e corrigidos mais rapidamente.

Modelos de Interface

Uma forma de alcançar consistência no projeto de interfaces é estabelecer um modelo consistente para interação do usuário com o computador. O modelo de interface do usuário é análogo a algum modelo do mundo real familiar ao usuário.

Existem vários modelos que podem ser utilizados para construir uma interface gráfica, dentre eles o modelo *desktop*, que representa em tela as entidades do usuário por formas sobre os *desktops*, mas é inadequado para serem utilizados em sistemas complexos. Outros modelos podem ser utilizados, que manipulam outros objetos de interface do usuário tais como botões, *sliders*, luzes, cores, e assim por diante.

Sistemas de Menus

Em uma interface com menus, usuários selecionam uma opção dentre um número de possibilidades e indicam suas escolhas à máquina. Menus provêem maneiras para que um número de comandos sejam rapidamente acessados e executados.

A utilização de menus tem várias vantagens:

- Usuários não precisam conhecer os nomes dos comandos;
- O esforço de digitação é mínimo;
- Erros do usuário são detectados pela interface;
- Provê mensagens de auxílio dependente-de-contexto.

O bom uso de menus inclui:

- Títulos de menus, juntamente com seus itens, consistentes e descritivos;
- Ordenação lógica de menus e itens gerenciáveis de cada menu;
- Uso de menus hierárquicos quando apropriados.

Exibição de Informações

É uma boa prática de projeto manter o software de apresentação da informação separado da própria informação, possibilitando à apresentação ser alterada sem mudar o sistema computacional.

Ao decidir a forma de apresentar as informações, o projetista deve levar em consideração os seguintes fatores:

- O interesse do usuário na informação a ser apresentada;
- A rapidez da mudança dos valores das informações;
- A interação do usuário com a informação exibida via manipulação direta da interface.

Ambientes gráficos bem projetados permitem ao usuário manipular facilmente os elementos da interface. Assim, é aconselhável a utilização de recursos como cores, sons e janelas para criar uma interface mais agradável para o usuário.

3.2.2 Interfaces Gráficas Específicas para Redes de Petri

Além dos critérios gerais apresentados na seção anterior, devemos considerar as particularidades relativas às interfaces para redes de Petri. Os elementos notacionais que constituem as redes de Petri, como lugares, transições, arcos, etc, têm particularidades e apresentam atributos que precisam ser trabalhados e executados por pacotes gráficos especiais, já que os existentes não são eficientes o bastante para dar seu tratamento adequado e específico a estes elementos.

Por serem, redes de Petri, uma ferramenta executável, possibilitando verificação e análise formal dos sistemas por ela modelados, é conveniente, também, uma interface que processe e exiba, graficamente, os modelos de redes de Petri.

Entre outras características, as interfaces gráficas que suportem redes de Petri devem ter:

- *Capacidade de Comentar*: É importante que redes de Petri sejam comentadas como linguagem de programação textual, por causa de sua natureza diagramática;
- *Instâncias de Módulos*: redes de Petri de alto nível suportam redes hierárquicas onde a instância de uma determinada rede está dentro de outras redes. Este tipo de pacote investiga como a interface manipula instâncias e suas informações inerentes;
- *Herança*: As características de herança de redes de Petri podem envolver informações necessárias para serem apresentadas de uma maneira lógica e medida;
- *Simulação*: Simulação é comum em pacotes de redes de Petri e a interface para esta característica deve ser analisada para que seja amigável ao usuário.

3.3 Ferramentas Gráficas Que Suportam Redes de Petri

Apresentaremos, nesta seção, uma breve descrição sobre as ferramentas gráficas para redes de Petri, porque dentro deste contexto, estas têm especial interesse, já que algumas apresentam características como tratamento de redes de Petri orientada a objetos, outras possuem tratamento de hierarquia, outras executam simulação de redes de Petri e, além disto, apresentam características gráficas relevantes para o contexto do ambiente que desenvolvemos. As ferramentas que discutimos são:

- GreatSPN - Editor, Simulador e Analisador de redes de Petri estocásticas;
- Design/CPN - Editor, Simulador e Analisador de redes de Petri Coloridas Hierárquicas;
- Xloopn - Simulador para redes de Petri Coloridas Orientadas a Objetos.
- CPN/AMI - Editor e Verificador para redes de Petri Coloridas Orientadas a Objetos - AMI-Nets.

3.3.1 GreatSPN - GRaphical Editor and Analyser for Timed and Stochastic Petri Nets

GreatSPN é um pacote que suporta Redes de Petri Temporizadas e Estocásticas, combinando editor gráfico com ferramentas de análise. O pacote consiste de pequenos programas integrados que podem estar distribuídos em máquinas diferentes conectadas por uma rede.

GreatSPN é executável em ambientes suportando X-Windows [36].

A ferramenta contém facilidades como o *Undo*, em múltiplos níveis, mas somente para o comando de remoção de objetos. Provê caixas de diálogos para o usuário desfazer uma ação antes que a mesma tenha continuidade provendo alto nível de interação com o usuário.

A interface do GreatSPN é, predominantemente, dirigida a *mouse*. A maioria dos comandos são acessados através de botões na área de menu principal da tela.

O *layout* de tela do GreatSPN consiste de três partes: o painel de controle, uma janela de trabalho para desenho onde as redes são criadas e uma janela *zoom* que permite a visão mais detalhada da rede desenhada na janela, implementado utilizando a interface *SunTools*. A ativação de menus, que, frequentemente, não aparecem na janela antes de serem ativados, é feita via botão direito do *mouse* na área de trabalho, assim, o usuário não tem como saber de sua existência.

A ferramenta possibilita o controle de algumas informações mostradas em tela como valor de tempo, nomes de lugares, etc. Suas alterações são feitas selecionando o objeto desejado. Ao selecionar um grupo de objetos, um conjunto de operações torna-se disponíveis na janela principal. GreatSPN provê, também, recursos para colocar e manipular objetos em quaisquer posições na janela, além de possibilitar rotacionar transições pelo botão do *mouse*.

GreatSPN não manipula com herança de redes, nem permite construir redes hierárquicas. Provê um grande espaço de trabalho virtual para modelar grandes redes, além do que é exibido na tela para o usuário. Este espaço é acessado através dos botões de *scroll bars*.

A ferramenta processa simulação de redes de Petri Estocásticas de três tipos: interativa, onde permite ao usuário selecionar transições a serem disparadas interativamente ou mudar a marcação dos lugares, não-interativa, que executa a simulação sem a interferência do usuário, no modo *batch* e jogo não-temporizado, ainda não disponível ao usuário.

3.3.2 Design/CPN

Design/CPN é uma ferramenta para edição, análise e simulação de Redes de Petri Coloridas [18]. Consiste de um editor gráfico, uma ferramenta de simulação que usa uma extensão de StandardML e provê facilidade de executar análises estáticas e de

alcançabilidade. Suporta múltiplas janelas e contém menu de barra. Tem versões que executam em ambientes X-Windows e Macintosh OS.

A ferramenta provê aos usuários facilidades para visualizar e manipular componentes da rede modelada, por exemplo: componentes podem ser movidos e mudados de tamanho facilmente, além de textos associados com lugares, arcos ou transições podem ser facilmente modificados.

Design/CPN tem uma maneira clara e organizada de apresentar seus comandos, usando menus de barras flutuantes, além disto faz uso de caixas de diálogos para manter o usuário informado sobre muitas operações que estão sendo executadas. Provê facilidades gráficas, como alterar atributos (estilo e espessura da linha, inversão de comportamento, etc.) de componentes individuais da rede ou diagramas inteiros. Possibilita ao usuário controlar a quantidade de informações dos componentes da rede mostrados na tela.

A ferramenta organiza as partes do diagrama de redes de Petri em *páginas*. Cada página tem sua própria janela e são ligadas com a página do nível mais alto chamada *página hierárquica*. Múltiplas páginas podem ser editadas e manipuladas ao mesmo tempo.

Possibilita a introdução de comentários aos diagramas da rede usando regiões de texto auxiliares. Estas regiões de texto não afetam o comportamento da rede de Petri durante a simulação e podem ser escondidos no editor, se desejado.

Design/CPN não manipula com herança. É possível executar segmentos de código durante a simulação, permitindo incorporar animação. O usuário pode mudar a marcação do lugar, parar a operação de simulação, executar um conjunto de etapas e verificar o resultado obtido.

3.3.3 Xloopn

Xloopn é um pacote de Redes de Petri desenvolvido na Universidade da Tasmânia. Suporta Redes de Petri Coloridas, incorporando herança e polimorfismo. Uma das

suas metas é atuar como interface gráfica pelo compilador simulador de Redes de Petri, LOOPN.

Foi implementado usando aplicações ET++ e provê implementações de elementos básicos de interface, obtendo uma interface mais consistente e amigável.

Xloopn provê facilidades para o usuário desfazer comandos como remoção, criação e movimento de objetos. Exibe caixas de diálogos para cancelar ou continuar alguma ação que esteja sendo executada e para notificar eventos que são importantes para o usuário. Utiliza atalhos para comandos de manipulação de arquivos e de edição e provê um bom *feedback* com o usuário ao manipular componentes da rede. Xloopn provê uma tela agradável e bem organizada, possibilitando o uso de barras de ferramentas para execução de comandos de edição gráfica. Permite ao usuário acessar às características dos objetos da rede e alterá-los, não sendo possível adicionar comentários arbitrários na rede ou em seus objetos.

Xloopn ainda não tem implementada a simulação de redes, no entanto, oferece fácil acesso às informações auxiliares sobre redes através de botões agrupados na tela principal. Cada botão é uma caixa de diálogo que contém informações específicas para a tarefa desejada.

3.3.4 CPN/AMI

CPN/AMI é uma ferramenta de redes de Petri desenvolvida na Universidade Paris VI. Suporta AMI-Nets [22], uma forma sintática das redes de Petri bem-formadas, incluindo redes Coloridas e Lugares/Transição. Foi implementada e executa em ambiente Macintosh mas também pode ser executada em ambiente Solaris 2 Sparc Station e em estações Hewlett-Packard HP-UX.

CPN/AMI provê um editor gráfico para modelagem de redes de Petri (MACAO), um simulador e verificador das propriedades gráficas e estruturais das redes modeladas.

MACAO é um editor gráfico homogêneo, organizado e amigável ao usuário. O editor gráfico utiliza serviços de janelas para informar o usuário de eventos (como

erros, etapas de execução) ocorridos durante sua utilização. Provê menus de barra para tratamento de arquivos, visualização da rede e desenho de seus componentes.

CPN/AMI possibilita inserir observações textuais inerentes aos componentes da rede, possibilitando sua alteração. Os textos não são considerados quando o simulador é executado.

Provê as seguintes funcionalidades:

- Um verificador que checa a corretude de modelos sintáticos e semânticos das redes modeladas.
- Um protótipo que gera uma rede de Petri modular para cada classe de rede definida e corrigida na ferramenta.

Para estas facilidades o ambiente tem ferramentas de verificação e análise para redes de Petri, além de possuir um animador que executa de quatro maneiras: *forward*, *backward*, automaticamente e por etapas.

Neste capítulo, apresentamos as características básicas de interfaces gráficas para o usuário e resumo de algumas ferramentas gráficas que suportam redes de Petri, importantes para o objetivo de nosso trabalho. No próximo capítulo, descreveremos o ambiente gráfico para edição e animação de *G-Nets*, o *NetGraph*. Além disto, especificamos a estrutura de classes utilizada na implementação do ambiente, descrevendo seus métodos principais e, finalmente, daremos uma visão geral da linguagem de programação C++ e da biblioteca Motif.

Capítulo 4

Especificação e Concepção do Ambiente NetGraph

4.1 Introdução

Neste capítulo, apresentamos o *NetGraph*, um ambiente gráfico e interativo para a especificação, modelagem e desenvolvimento de sistemas complexos, utilizando uma metodologia orientada a objetos. Este ambiente foi projetado para capacitar o profissional especialista a desenvolver modelos de sistemas baseados em *G-Nets* a partir da descrição de um sistema.

Além disto, apresentamos a estrutura de classes utilizada para implementação do ambiente e o diagrama das mensagens envolvidas durante sua execução.

O *NetGraph* foi implementado em estações de trabalho SUN e sistema operacional UNIX [43]. A linguagem de programação utilizada foi C++, com bibliotecas gráficas do OSF/MOTIF que fornecem suporte para à construção de interfaces gráficas usando componentes como: ícones, botões, menus, *scroll bars*, etc. e outros mecanismos para criar elementos gráficos.

4.2 Arquitetura do Ambiente

A Figura 4.1 apresenta a definição de arquitetura do ambiente, onde observam-se dois módulos distintos: *módulo de edição* e *módulo de execução*. A seguir descrevemos estes dois módulos.

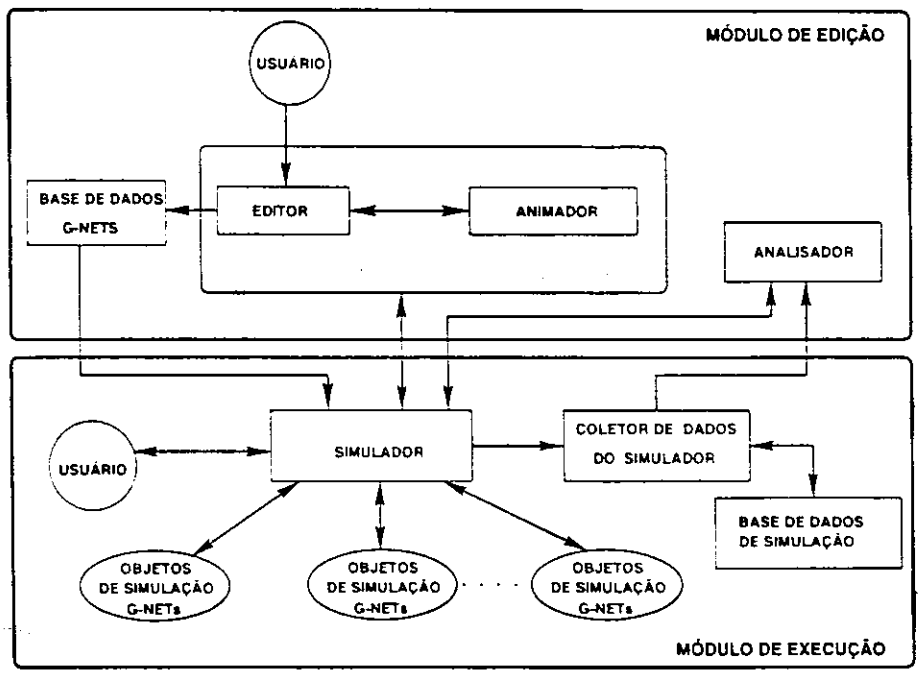

Figura 4.1: Arquitetura do ambiente

Módulo de Edição

O módulo de edição é constituído dos seguintes blocos: *base de dados de G-Nets*, *editor*, *animador* e *analisador*.

O animador acessa os modelos (*G-Nets*) construídos e armazenados pelo editor, exibindo graficamente o comportamento dinâmico da rede. O animador comunica-se com o simulador através de mensagens: o simulador, após executar os disparos das transições da rede, envia mensagens para o animador processar graficamente o resultado desta operação.

A base de dados de *G-Nets* contém especificações das redes modeladas pelo editor numa estrutura que possa ser executada pelo simulador.

O analisador é responsável pela organização, análise e interpretação dos dados coletados durante o processo de simulação e pode ser acionado em qualquer instante do processo de simulação para ajudar a interpretar os dados coletados até então e verificar as propriedades do sistema modelado.

O editor provê uma interface gráfica com a finalidade de facilitar a construção e edição de *G-Nets* [24]. O editor faz consistências estruturais da rede, interativamente, durante a sua modelagem, por exemplo: não permite a construção de arcos entre elementos do mesmo tipo, como, por exemplo, entre transições ou entre lugares; a construção de um arco inibidor [23] só será efetuada se o nó inicial for um lugar e o final, uma transição; etc. A manipulação e execução das opções do editor são feitas através de menus de barra e ícones relativos a cada função por ele desempenhada.

O editor contém um menu principal com as seguintes opções: *File*, *Export*, *Animation* e *Help*. A opção *File* possibilita a criação de novas *G-Nets*, salvar e editar redes já especificadas e armazenadas. Cada especificação de *G-Net* criada no editor é implementada e organizada internamente como uma definição de classe pré-definida para *G-Nets*; os componentes da rede modelada são armazenados como classes hierárquicas seguindo as técnicas de orientação a objetos e refletem a semântica das *G-Nets*.

A opção *Export* gera um arquivo texto, a partir de uma *G-Net* já modelada pelo usuário, cujo formato e estruturas são compatíveis com o simulador já existente. Cada arquivo gerado fica armazenado numa base de dados para ser utilizado pelo simulador.

Cada *G-Net* criada é armazenada em um arquivo com o seguinte formato:

```
[illegible]
[illegible] 0 caso contrario>
[illegible] como separador >
[illegible] de transitions>
&&
[illegible] do metodo >
[illegible]strategy1 < sempre goal_state>
[illegible]st>,<time>>
&& < Fim da definicao de GSP >
[illegible] lugar de entrada>
&num < numero de fichas levadas >
[illegible] Lugar >
[illegible] color do arco>
action_name1 < Acao associada >
[illegible]rario >
&& < Fim de Lugares Normais >
[illegible]da>
&num <numero de fichas depositadas>
[illegible] >
[illegible] do isp>
[illegible] do a invocar >
[illegible]ha>
[illegible] depois que ficha e' recebida>
```

O modelo do arquivo gerado tem a seguinte estrutura: inicialmente, especifica o *GSP*, separando a especificação de cada método por um &, depois vem a especificação dos *Lugares Normais*, cada um deles separados por &, após isto há a especificação de *ISP* e, finalmente, as *Transições*. Na especificação das transições, relaciona-se todos os lugaras de entrada, separados por um &, e todos os lugares de saída, separados por um &. Cada especificação de um objeto da *G-Net* é finalizado por &&. Ao término da definição de todos os objetos da rede, o arquivo é finalizado com &&&.

A opção *Animation* executa a animação gráfica da execução da rede especificada pelo usuário, a partir da saída do simulador de *G-Nets*.

A opção *Help* fornece informações gerais sobre a utilização deste ambiente ou sobre quaisquer comandos ou ícones nela existentes. Esta opção ainda não foi implementada nesta versão do ambiente.

O editor tem uma barra de ferramentas que é dividida em 2 partes: ícones com

elementos componentes da *G-Net* e ícones com os comandos de edição da rede. Os ícones relativos a barra de elementos são utilizados para construir cada componente da rede do usuário na área de trabalho (canvas). Assim, ao selecionar o ícone referente ao lugar, *GSP*, transição, *ISP*, arco, arco inibidor, lugar alvo e ficha, o usuário pode construí-lo em qualquer posição desejada da área de trabalho. Os ícones relativos ao *modo de edição* possibilitam alterações dos componentes desenhados, provendo ao usuário facilidades como: selecionar objetos, modificar, mover elementos da rede, ampliar ou reduzir o tamanho da rede, etc, ou de seus atributos (cor, posição, identificação, etc). É permitido ao usuário desfazer (*UNDO*) o último comando executado. Além disso, é feita, interativamente, uma verificação estrutural na rede, como a remoção consistente dos elementos que a compõem e manipulação consistente das alterações dos atributos dos métodos da GSP modelada.

O Editor contém outras facilidades como: barras de rolamento para possibilitar a visualização vertical e horizontal das partes da rede, que são maiores que a área visual do canvas. Para cada ação executada ou erro cometido pelo usuário, existe uma área de mensagem (última barra da tela) que o orienta sobre a ação ocorrida e a correção de um possível erro. Na versão atual, a exibição de mensagens foi implementada apenas para alguns casos.

Assim, com todas estas facilidades, o editor gráfico possibilita a construção e modelagem de *G-Nets* da forma mais amigável e consistente possível.

Módulo de Execução

O módulo de execução é distribuído sobre uma ou mais estações de trabalho em um ambiente de rede. Neste sistema, o simulador [21] é responsável pela criação e inicialização do ambiente de simulação sobre a máquina onde está localizado. Para cada *G-Net*, uma instância do simulador é criada para gerar seu comportamento e é responsável pela coordenação da comunicação entre objetos *G-Nets*.

O simulador é uma implementação das regras de execução *G-Net*. Uma instância

do simulador é criada sempre que uma *G-Net* é executada, gerando estes objetos, responsáveis pelo comportamento definido para a *G-Net*. Um objeto de simulação é uma instância de uma *G-Net*, que corresponde a um objeto dentro da especificação do projeto do sistema.

O coletor de dados do simulador contém uma base de dados armazenados durante o processo de simulação, gerados para fornecer informações atualizadas para o analisador.

A Figura 4.2 mostra uma cópia de tela do ambiente gráfico.

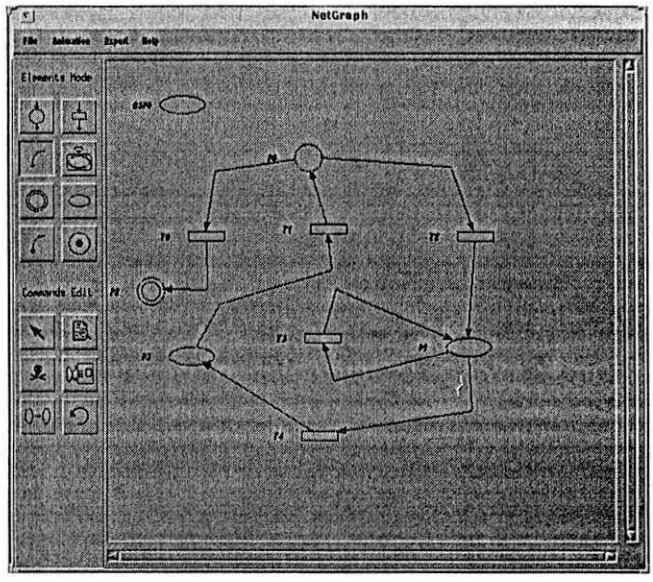

Figura 4.2: Ambiente Gráfico de G-Nets

4.2.1 Arquitetura de Simulação

Após a especificação e modelagem da *G-Net*, o ambiente provê ao usuário a possibilidade de simular a operação da rede e observar, graficamente, a animação do fluxo de fichas, etapa-por-etapa, dentro dos objetos desenhados no canvas.

Estes recursos possibilitam a análise do modelo por simulação. Propriedades, tais como verificação de conflitos e *deadlocks*, podem ser analisadas. A possibilidade de simulação em um ambiente baseado em rede de Petri é direto, uma vez que o modelo é executável. Por outro lado, o uso de simulação, como ferramenta de análise, permite

que o usuário se abstraia do formalismo envolvido na análise. Entretanto, o sistema prevê a aplicação de técnicas formais de análise consagradas para redes de Petri, como análise comportamental e estrutural [29, 30, 33].

Além destas vantagens, por terem, sistemas de *G-Nets*, suas especificações executáveis [9, 21], podem ser usadas como uma poderosa ferramenta de prototipagem de sistemas e sua execução, auxiliada pela animação gráfica com parâmetros selecionados, ajuda a encontrar erros de projeto e fazer correções em seus estágios iniciais, antes de sua implementação final.

Este recurso do ambiente é utilizado após o usuário ter executado a opção *export*, onde a rede modelada é inserida numa base de dados de redes cujas estruturas estão no formato a serem executadas pelo simulador *G-Net*.

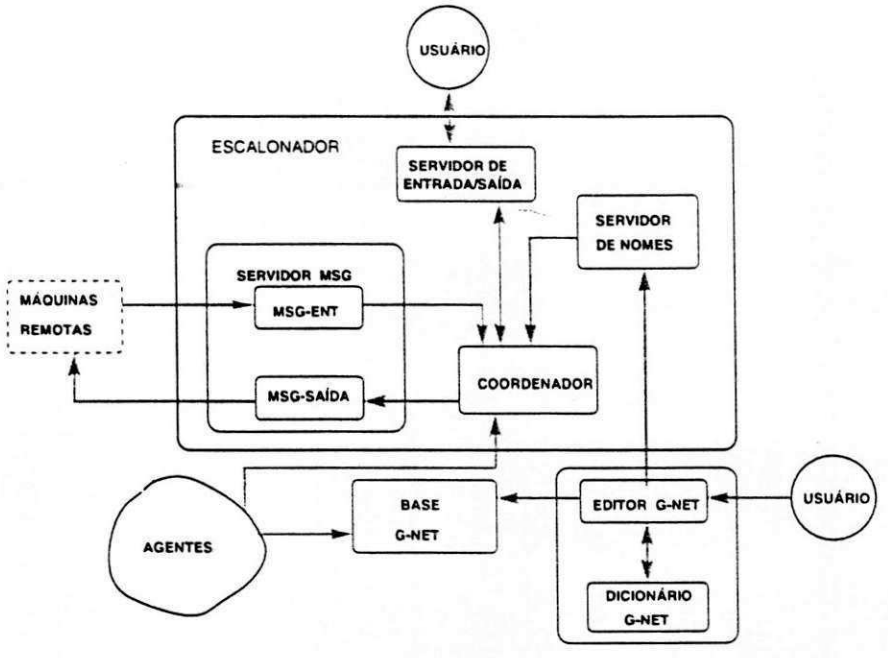

Figura 4.3: Diagrama do Simulador G-Net

A Figura 4.3 representa o diagrama de blocos para o simulador de sistemas de *G-Nets*. A estrutura do sistema e os detalhes da implementação podem ser encontrados em [9, 10]. Uma cópia do simulador reside em cada máquina do sistema distribuído para executar as *G-Nets*. O escalonador tem quatro módulos básicos. O coordenador é

a parte principal do escalonador e tem como funções: criação de agentes para executar uma especificação *G-Net* e prover interface entre os agentes e o servidor de mensagens (servidor msg). O módulo servidor msg manipula comunicações entre processos para agentes remotos e locais. O módulo servidor de nomes provê um mapeamento de um dado identificador lógico *G-Net* para um endereço físico do sistema. O dicionário *G-Net* é um conjunto pré-definido de *G-Nets* e a base *G-Net* contém arquivos textos especificando cada *G-Net* alocada à máquina.

4.3 Concepção do Ambiente

A concepção do ambiente baseou-se em técnicas de orientação a objetos, onde a representação interna de cada *G-Net* é implementada e organizada como hierarquia de classes orientada a objetos baseada na semântica formal de construções *G-Nets*. Cada modelo editado é armazenado de acordo com a definição de classe pré-definida de *G-Net* e tem mapeamento direto de cada elemento da rede construída.

Nesta seção, descreveremos a estrutura de classes do *NetGraph*, a forma como foram distribuídas, além dos atributos internos e funcionalidades de cada uma delas.

4.3.1 Estrutura de Classes e Funcionalidades

O ambiente foi especificado e projetado utilizando a metodologia orientada a objetos e está estruturado como mostrado na Figura 4.4.

O ambiente é gerenciado pela classe *Editor* que é composto por cinco classes: menu, canvas, lista de figuras, diálogos e botões de comandos. A classe menu define métodos para tratamento de todas as funções relativas à construção e manuseio de menus. A classe canvas provê métodos para tratamento de funções de alterações gráficas no domínio do canvas. A classe diálogos define todos os métodos inerentes às caixas de diálogos referentes aos atributos de todos os componentes da rede, mensagens de erros, opções de entrada de dados e diversas outras funções. A classe botões de comandos

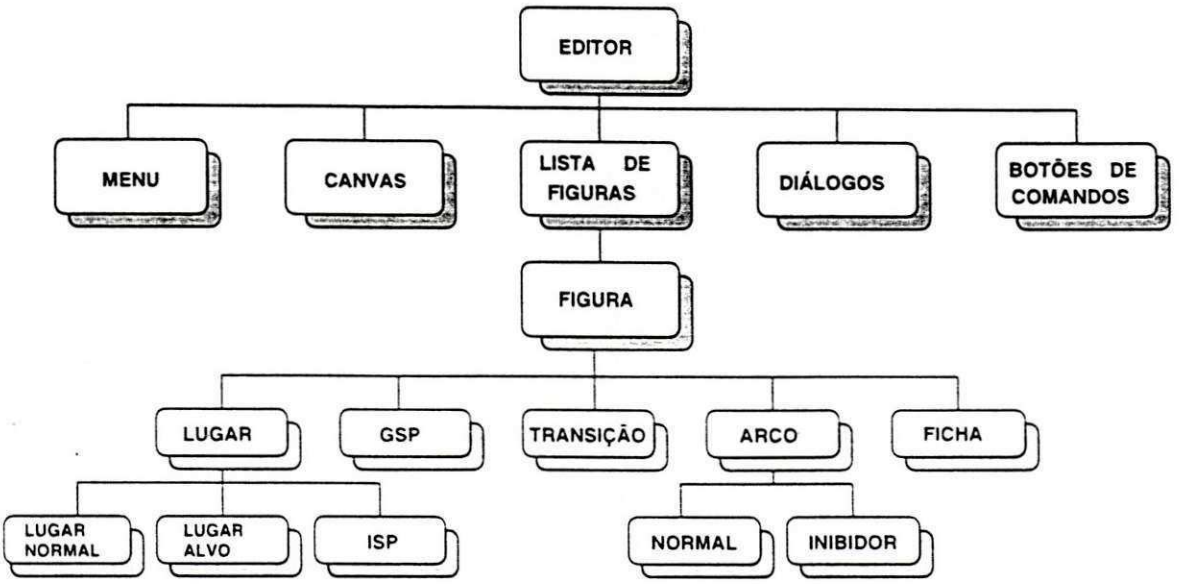

Figura 4.4: Estrutura de classes do editor

define todos os métodos que acionam os comandos da barra de ferramenta do Editor. Por fim, a classe lista de figuras corresponde a uma lista encadeada com todos os elementos da rede modelada. Para tanto, cada objeto que compõe a rede é considerado como uma figura. Por terem, sistemas de *G-Nets*, diversos tipos de componentes, a classe figura tem cinco tipos de herdeiros, relativos a cada um destes componentes. O elemento que é dividido em outros tipos, como lugar e arco, também tem herdeiros relativos a cada um dos tipos de seus componentes. *G-Net* tem três tipos de lugares, então seus herdeiros são lugar normal, lugar alvo e *isp*. Arco tem dois tipos de herdeiros: arco normal e arco inibidor. As classes do tipo figura definem todos os métodos que fazem tratamento de suas especificidades.

A classe *Editor* é a principal classe do ambiente, pois gerencia todas as ocorrências de eventos, manipulação dos objetos criados, solicitações do usuário, chamada de inicialização e finalização do sistema, enfim, coordena e gerencia todas as ocorrências do ambiente. O Editor foi assim dividido nestas classes depois da observação das necessidades e requisitos do usuário para utilizar o ambiente, vejamos:

- O ambiente deve prover uma área de trabalho para a modelagem do sistema do

usuário. Nesta área de trabalho, o *canvas*, o usuário poderá desenhar a rede, modificar as posições de seus componentes, seu tamanho e atributos, enfim, terá uma área gráfica amigável e agradável para trabalhar.

- O ambiente deve prover um sistema de menus para possibilitar a execução de comandos imprescindíveis - como manipulação de arquivos, animação, etc. - cujo conteúdo não é necessário ficar sendo exibido, permanentemente, na tela da ferramenta.

- O ambiente deve prover caixas de diálogos para avisar ao usuário sobre erros cometidos, como executar determinada tarefa ou, até mesmo, auxiliá-lo no caminho a ser seguido para atingir seus objetivos, além de prover avisos e alarmes necessários e convenientes.

- O usuário deve ter, a sua disposição, um conjunto de comandos para desenhar os elementos componentes da rede e manipulá-los, de forma fácil, agradável, eficiente e de rápida execução.

- O ambiente deve ter uma maneira eficaz de armazenar e gerenciar os elementos da rede modelada pelo usuário. A maneira encontrada foi armazená-la em forma de uma lista encadeada de objetos, cuja definição semântica define um modelo orientado a objetos da estrutura das *G-Nets*.

- Além disto, houve a necessidade de criar algumas classes para encapsular as funções do Motif, visto que o Motif não é orientado a objetos.

4.3.2 Definição das Classes

Nesta seção, apresentamos as classes que compõem o *NetGraph*, com a descrição de seus principais métodos.

Utilizamos, na implementação do *NetGraph*, algumas classes auxiliares que encapsulam funções gráficas do Motif e do Xt, como, por exemplo, a classe *Shell*.

Resumo do Motif

Motif é baseado na biliboteca *XtIntrinsics*, que é baseada num poderoso modelo de programação a seguido na construção das aplicações. Assim, muitas aplicações Motif executam as seguintes etapas básicas:

- Inicialização do Xt: Inicializa as estruturas de dados internas definidas pelo *Xt*, abre uma conexão com o *XServer* e cria uma estrutura de dados conhecida como *contexto da aplicação*.
- Criar um Shell: Todas aplicações devem criar um ou mais *widgets shell* para atuar como intermediários entre a aplicação e o gerenciador de janelas. O *widget shell* é criado no nível mais alto da janela de aplicação.
- Criar widgets: *widgets* são componentes da interface do usuário com os quais ele interage.
- Registrar funções *Callback*: *Callbacks* são funções que executam alguma ação específica da aplicação em resposta a alguma entrada do usuário, por exemplo: registrar quando o usuário pressiona um botão ou seleciona um item de um menu.
- Gerenciar todos os widgets: *widgets* formam uma hierarquia semelhante a árvore do sistema *X*. Assim, cada *widget* é responsável pelo tamanho e posição de seus *widgets* filhos.
- Manipular eventos: As aplicações devem ter um evento que permaneça executando em *loop* para receber eventos do *XServer*. Quando um evento ocorre, o evento *loop* remove o primeiro evento pendente e despacha-o para o *widget* apropriado manipulá-lo.

Widgets são componentes gráficos da interface do usuário. Existem dois tipos de *widgets*: um que apresenta a informação ao usuário ou recebe entradas do usuário, por exemplo: botões, rótulos e *scroll bars*. Outros que são usados para agrupar conjunto de *widgets* numa tela.

Shell

Toda aplicação Motif necessita de um inicializador do ambiente gráfico de trabalho. A classe *Shell* é responsável pela inicialização completa do ambiente.

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: Shell		
Métodos	*Descrição*	*Parâmetros*
Shell	Cria um contexto da aplicação, cria uma conexão com o servidor, inicializa a camada do *toolkit* e retorna um *widget*	Widget
GetWidget	Retorna a aplicação criada no ambiente gráfico	Widget
Run	Executa a aplicação criada, processando a ocorrência de eventos	void

Diálogo

Tem a finalidade de manipular as caixas de diálogos gerais que são exibidas durante a utilização do ambiente.

Contém atributos relativos a todas as funcionalidades das caixas de diálogos do ambiente.

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: Diálogo		
Métodos	*Descrição*	*Parâmetros*
CancelCallback	Cancela a entrada de dados da Caixa de Diálogos	Widget, XtPointer
GetFileSelection	Recebe os dados da Caixa de Diálogo	Widget, XtPointer, char
OpenDialog	Processa caixa de diálogo para abertura de arquivo	Widget, XtCallbackProc, XtPointer
NewDialog	Processa caixa de diálogo para limpar a área de trabalho	Widget, XtCallbackProc, XtPointer
HelpDialog	Processa caixa de diálogo para avisos de ajuda ao usuário	Widget
SaveDialog	Processa caixa de diálogo para gravar rede em arquivo já existente	Widget, XtCallbackProc, XtPointer
SaveAsDialog	Processa caixa de diálogo para criar arquivo, armazenando a rede modelada no canvas	Widget, XtCallbackProc, XtPointer
PrintDialog	Processa caixa de diálogo para impressão de redes	Widget
QuitDialog	Processa caixa de diálogo para encerrar os trabalhos no NetGraph	Widget, XtCallbackProc
ExportDialog	Processa caixa de diálogo para gerar arquivo a ser processado pelo Simulador	Widget, XtCallbackProc, XtPointer
AnimationDialog	Processa caixa de diálogo para processar Animação	Widget, XtCallbackProc, XtPointer

Canvas

Encapsula o tratamento da área gráfica do Motif. Refere-se a área de trabalho onde o usuário irá manipular os elementos do sistema de *G-Nets*, modelando sua aplicação.

Assim, o canvas tem atributos para fazer tratamento de informações referentes a mapeamento de cores, apontadores para região da memória onde está o objeto corrente manipulado (*display*), apontador para janela de trabalho, funções de desenho, posição do *mouse* na área de trabalho selecionada e uma área de memória (*pixmap*) para implementação do *redisplay* das figuras da rede.

Os métodos desta classe fazem tratamento de todos os eventos ocorridos e solicitados pelo usuário nesta área de trabalho, tais como: tratamento de *redisplay*, limpar área de trabalho e funções gráficas.

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: Canvas		
Métodos	*Descrição*	*Parâmetros*
RedisplayCallback	Redesenha as figuras no canvas, quando a janela é sobreposta ou minimizada	Widget, XtPointer, XtPointer
SetDisplayFunction	Especifica qual a função de desenho a ser executado no canvas, determinando como cada pixel da nova imagem é combinada com a área de desenho	int
ClearArea	Limpa a área de trabalho, apagando as figuras que estão desenhadas	void
GetWidget	Retorna o Widget de maior hierarquia da classe	void

Botões de Comandos

Esta classe tem a finalidade de organizar os botões de ícones criados pertencentes ao ambiente, relativos aos elementos componentes da rede e aos comandos para manipulá-los. A organização é através de barra de ferramentas, fazendo com que o projetista não se preocupe com suas posições, já que é feita automaticamente.

Além disto, implementa as funcionalidades dos ícones da barra de ferramenta.

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: Botões de Comandos		
Métodos	*Descrição*	*Parâmetros*
SetButtonPress	Simula o pressionamento do botão na barra de ferramentas	Widget
GetWidget	Retorna o Widget de maior hierarquia para a classe	void

Menu

É responsável pela implementação dos menus *pop-up* existentes no ambiente. Nestes menus, estão inseridos comandos para tratamento de arquivos, encerrar os trabalhos no ambiente, executar animação de redes modeladas e armazenadas na base de dados, gerar arquivo compatível com a entrada do simulador, mensagens de erros e advertências.

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: Menu		
Métodos	*Descrição*	*Parâmetros*
PostMenuHandler	Controla a ocorrência de eventos durante a utilização do menu criado	Widget, XtPointer, XEvent *, Boolean *
AttachPopup	Adiciona um manipulador de eventos ao menu quando o botão do mouse é pressionado	Widget, Widget
CreateMenuChildren	Cria sub-menus herdeiros do menu principal a partir da descrição da estrutura	Widget, MenuDescription *, XtPointer
CreateMenu	Cria menu através da descrição projetada e armazenada numa estrutura	MenuType, char *, Widget, MenuDescription *, XtPointer

Editor

Esta classe tem a finalidade de gerenciar a manipulação de figuras relativas à rede modelada pelo usuário, os eventos selecionados no *mouse* pelo usuário, a manipulação das caixas de diálogos, dos menus e a execução dos comandos implementados nos botões da barra de ferramenta.

Contém um apontador para uma lista encadeada contendo todos os objetos da rede; um apontador para a figura corrente da lista de figuras e estrutura de pilha para possibilitar a implementação do *Undo*, onde o usuário pode desfazer a última operação efetuada.

Esta classe, através de seus métodos, gerencia a ocorrência de todos os eventos relativos aos botões do *mouse*, tais como: pressionamento do botão esquerdo para ativar ou concluir comandos, movimentação do *mouse* com o botão esquerdo pressionado, liberação do botão esquerdo do *mouse* e pressionamento do botão direito do mouse para cancelar a construção de figuras.

Além disto, é responsável pelo acionamento e gerenciamento dos métodos de construção de figuras, das ações de animação e geração do arquivo que será executado pelo simulador, ações de criação de nova rede, salvamento da rede modelada no canvas e leitura de redes arquivadas na base de dados de *G-Nets*.

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: Editor		
Métodos	*Descrição*	*Parâmetros*
GetFieldsDialog	Obtém dados digitados de uma caixa de diálogo	Widget, MenuDescription *, XtPointer
OpenCallback	Invoca um método da classe diálogo para abertura de arquivos	MenuType, char *, Widget, MenuDescription *, XtPointer
HelpCallback	Exibe mensagens de ajuda sobre a ferramenta	Widget, XtPointer, XtPointer
NewCallback	Processa etapas para construção de novas redes: abre caixa de diálogo e executa criação de redes	Widget, XtPointer, XtPointer
SaveCallback	Processa etapas para salvar a rede editada em arquivo (já criado): abre caixa de diálogo e salva rede do canvas	Widget, XtPointer, XtPointer
SaveasCallback	Processa etapas para salvar em novo arquivo a rede editada no canvas: abre caixa de diálogo e cria novo arquivo com rede do canvas	Widget, XtPointer, XtPointer
QuitCallback	Processa caixa de diálogo para sair do editor	Widget, XtPointer, XtPointer
ExportCallback	Processa etapas para criar arquivo compatível com a entrada do simulador a partir da rede editada	Widget, XtPointer, XtPointer
AnimationCallback	Processa etapas para animação da rede carregada no canvas	Widget, XtPointer, XtPointer
SetFigureCallback	Seta o modo de trabalho, a partir do identificador do botão pressionado: Comandos de edição ou desenho de elementos da rede	Widget, XtPointer, XtPointer
LeftButtonPress	Processa ações ocorridas ao pressionar o botão esquerdo do mouse	Widget, XEvent *
ButtonPressMotion	Processa ações de movimento do mouse com o botão esquerdo pressionado	Widget, XEvent *
LeftButtonRelease	Processa ações ocorridas após liberação do botão esquerdo do mouse	Widget, XEvent *
RightButtonPress	Processa ação ao pressionar botão direito do mouse	Widget, XEvent *
Pointer Motion	Executa ação de construção de arcos	Widget, XEvent *
SetButtonCommand	Atualiza apontador da figura corrente e desenha-a no canvas	Widget, int
AnimMetodo	Processa a entrada de métodos a serem executados durante a animação da rede	Widget
DelFigura	Deleta figura selecionada	Figura *
Save	Grava rede modelada no canvas em arquivo	char *
Ler	Lê rede do arquivo e desenha-a no canvas	char *
Animação	Executa a animação da rede lida em arquivo	void
Novo	Limpa a área do canvas e libera o espaço da lista de figuras	void
Undo	Processa o cancelamento do último comando efetuado pelo usuário	void
InsideFig	Retorna um apontador para a figura sobre a qual o mouse está posicionado	int, int
ReturnFigure	Reinsere no canvas e na lista de figuras, figura deletada no canvas	Figura *
Export	Gera arquivo da rede modelada a ser executado pelo simulador	void

Figura

Esta classe contém a estrutura básica dos objetos que compõem a notação de uma *G-Net*. São atributos comuns a estes objetos, tais como as coordenadas da figura no canvas, identificação - única para cada objeto, cor de fundo e de borda.

Além disto, contém métodos virtuais que serão implementadas por cada um dos tipos de figuras existentes na lista. Estes métodos são explicados na tabela a seguir, porém são inerentes a todas as classes herdeiras de Figura.

Figura é a visão externa de um objeto da rede, não se tem a implementação semântica do mesmo.

Os métodos comuns a todas as figuras são: desenho, identificação, atualização de atributos, salvar, mover, selecionar, desfazer operações, mudar cores, atualizar figura na região de memória do canvas, etc.

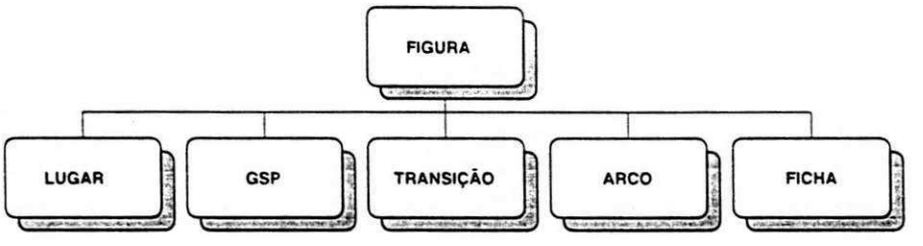

Figura 4.5: Herdeiros de Figura

Esta classe, como mostrado na Figura 4.5, tem como herdeiras as seguintes classes: Transição, Lugar, Arco, Ficha e GSP.

Os métodos desta classe implementam as funções virtuais comuns às suas herdeiras:

- Identificação: retorna o tipo da figura
- Desenho: contrói o desenho da figura no canvas
- Atualização de atributos: possibilita ao usuário alterar os valores padrões dos atributos da figura
- Salvar: gravar a figura no arquivo da base de dados
- Inserir e retirar arcos da lista de arcos de entrada ou saída

- Desfazer comandos
- Fazer consistências de inserção e remoção na rede
- Movimentação da figura no canvas
- Seleção da figura no canvas

Classe: Figura		
Métodos	*Descrição*	*Parâmetros*
WhoamI	Retorna o tipo de Figura corrente	void
Draw	Constrói o desenho da figura no canvas	Canvas *
DrawInPixmap	Desenha a figura corrente na região de memória do canvas	Canvas *
PertenceFigura	Verifica se o mouse está posicionado dentro da área da figura	int, int
UpDate	Processa a alteração dos atributos de um componente da rede	Widget
Selection	Executa a seleção de um componente da rede no canvas	Canvas *
ProcessMove	Processa movimentação de um componente da rede no canvas	Canvas *
MudaCorBorda	Altera a Cor de Borda de um componente da rede	Pixel
MudaCorFundo	Altera a Cor de Fundo de um componente da rede	Pixel

Gsp

Esta classe implementa o objeto Gsp componente de uma *G-Net*.

Uma Gsp contém os seguintes atributos:

- GSPid: Identificador da rede.
- attr: *string* de atributos da rede.
- ms: lista encadeada com os métodos relativos à rede construída.

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: GSP		
Métodos	*Descrição*	*Parâmetros*
MudaAtrib		char *
Gsp_Insmet	Insere os métodos, digitados pelo usuário, na lista de métodos do Gsp	MS_node *
Gsp_Remmet	Remove métodos da lista de métodos do Gsp	MS_node *
AddObjects	Insere um Gsp na lista de figuras e atualiza seus atributos	Canvas *, Figura *
ReturnAtrib	Tira cópia do Gsp para a Pilha do Undo	void
Consiste	Processa consistência para construção de Gsp	Figura *

Transição

Esta classe implementa o objeto Transição componente de uma *G-Net*.

Uma transição contém os seguintes atributos:

- Tid: Identificador da transição
- status: Indica se a transição está ou não habilitada.
- TimeMin: Tempo Mínimo para disparo
- TimeMax: Tempo Máximo para disparo
- In: Lista de arcos de entrada
- Out: Lista de arcos de saída

A tabela a seguir apresenta a descrição dos principais métodos desta classe.

Classe: Transition		
Métodos	*Descrição*	*Parâmetros*
GetFieldsDialog	Recebe os atributos, da transição corrente, a serem alterados	Widget, XtPointer, XtPointer
CancelDialog	Cancela a entrada de dados dos atributos da transição	Widget, XtPointer, XtPointer
OptionChangedColor	Muda Cor de Fundo e de Borda da transição	Widget, XtPointer, XtPointer
Trans_InsArcoIn	Insere um objeto Arco na lista de Arcos de Entrada da transição	Arco *
Trans_RemArcoIn	Remove um Arco da lista de Arcos de Entrada da transição	Arco *
Trans_InsArcoOut	Insere um Arco na lista de Arcos de saída da transição	Arco *
Trans_RemArcoOut	Remove um Arco na lista de Arcos de saída da transição	Arco *
Consiste	Processa consistência para construção de uma Transição	Figura *

Lugar

Esta classe implementa o objeto Lugar componente de uma *G-Net*. Como *G-Net* tem três tipos de lugares, esta classe tem três classes herdeiras, como mostrado na Figura 4.6: Lugar Normal, Isp e Lugar Alvo.

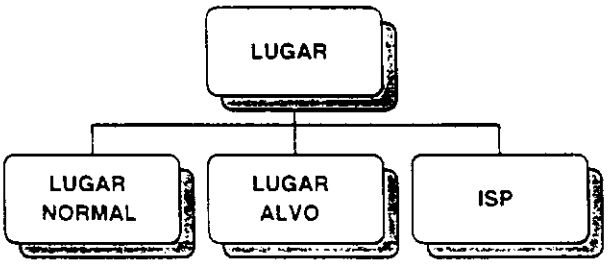

Figura 4.6: Classes herdeiras de Lugar

Os atributos básicos e comuns às classes herdeiras de Lugar são:

- Pid: Identificador do Lugar
- capacity: Capacidade do Lugar (número máximo de tokens suportado)
- In: Lista de arcos de entrada
- Out: Lista de arcos de saída

- Tkptr: Lista de fichas pertencentes ao lugar
- Msg: Mensagem

Classe: Place		
Métodos	*Descrição*	*Parâmetros*
Consiste	Processa a consistência para construção de um Lugar	Widget, XtPointer, XtPointer
Place_InsArcoIn	Insere um objeto Arco na lista de Arco de Entrada do Lugar	Widget, XtPointer, XtPointer
Place_RemArcoIn	Remove um Arco da lista de Arcos de Entrada do Lugar	Widget, XtPointer, XtPointer
Place_InsArcoOut	Insere um objeto Arco na lista de Arco de Saída do Lugar	Arco *
Place_RemArcoOut	Remove um Arco da lista de Arcos de Saída do Lugar	Arco *
InsToken	Insere uma ficha na lista de Fichas do Lugar	Arco *
RemToken	Remove uma ficha na lista de Fichas do Lugar	Arco *
ApagaToken	Apaga a figura da ficha no canvas, após sua remoção	Figura *

Lugar Normal

Contém como atributo a função primitiva relativa ao Lugar Normal.

Contém a implementação dos métodos virtuais da classe hierarquicamente superior.

A tabela abaixo apresenta a descrição dos principais métodos desta classe.

Classe: Normal_Place		
Métodos	*Descrição*	*Parâmetros*
GetFieldsDialog	Recebe os atributos, do Lugar Normal corrente, a serem alterados	Widget, XtPointer, XtPointer
TokenSelection	Seleciona um, dentre o conjunto de fichas do lugar, para alterar seus atributos	Widget, XtPointer, XtPointer
AddObject	Insere um Lugar Normal na lista de figuras e atualiza seus atributos	Canvas *, Figura *

Isp

Contém como atributos:

- Nid: Nome que identifica a rede chamada
- method: Nome do método invocado

- Arg: Lista de atributos para o método
- action_before: Nome da primitiva executável para a ação antes da invocação da rede
- action_after: Nome da primitiva executável para ação após recebimento da ficha da rede invocada

Contém a implementação dos métodos virtuais da classe hierarquicamente superior. A tabela abaixo apresenta a descrição dos principais métodos desta classe.

Classe: Isp		
Métodos	*Descrição*	*Parâmetros*
GetFieldsDialog	Recebe os atributos, do Isp corrente, a serem alterados	Widget, XtPointer, XtPointer
TokenSelection	Seleciona um, dentre o conjunto de fichas do Isp, para alterar seus atributos	Widget, XtPointer, XtPointer
Isp_InsAttr	Insere atributos nos métodos do Isp	Attr_list *
Isp_RemAttr	Remove atributos dos métodos do Isp	Attr_list *
AddObject	Insere um Isp na lista de figuras e atualiza seus atributos	Canvas *, Figura *

Lugar Alvo

Não tem atributos adicionais e implementa os métodos virtuais da sua classe hierarquicamente superior.

A tabela abaixo apresenta a descrição dos principais métodos desta classe.

Classe: Goal_Place		
Métodos	*Descrição*	*Parâmetros*
GetFieldsDialog	Recebe os atributos, do Lugar Alvo corrente, a serem alterados	Widget, XtPointer, XtPointer
TokenSelection	Seleciona um, dentre o conjunto de fichas do Lugar Alvo, para alterar seus atributos	Widget, XtPointer, XtPointer
AddObject	Insere um Lugar Alvo na lista de figuras e atualiza seus atributos	Canvas *, Figura *

Arco

Esta classe implementa o objeto Arco componente de uma *G-Net*. Como *G-Net* tem dois tipos de arcos, a classe *Arco* tem duas classes herdeiras, como mostrado na Figura 4.7: Arco Normal e Arco Inibidor.

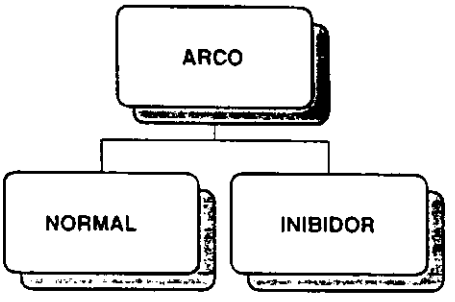

Figura 4.7: Classes herdeiras de Arco

A classe Arco contém atributos básicos e comuns a suas herdeiras, são eles:

- Arcid: Identificador do Arco
- Aid: Nome do arco
- Peso: Peso do Arco (número de fichas a serem inseridas - se arco de saída - ou removidas dos lugares - se arco de entrada)
- bb: Branch color das fichas a serem levadas aos lugares
- Figstart: Apontador para a figura inicial
- Figend: Apontador para a figura final
- BreakArc: Coordenadas das quebras do Arco
- NumArc: Número de quebras

Os métodos desta classe implementam as funções virtuais comuns às suas herdeiras:

- Construção
- Atualização de atributos: possibilita ao usuário alterar os valores padrões dos atributos da figura
- Salvar: gravar a figura no arquivo da base de dados
- Inserir e retirar arcos da lista de arcos de entrada ou saída
- Cálculos do Ponto de intersecção nas figuras inicial e final

Classe: Arco		
Métodos	*Descrição*	*Parâmetros*
GetFieldsDialog	Recebe os atributos, do Arco corrente, a serem alterados	Widget, XtPointer, XtPointer
OptionChangedColor	Muda Cor de Fundo e de Borda do Arco	Widget, XtPointer, XtPointer
ConsisteMoveArc	Faz consistências na construção do Arco, durante sua movimentação	Canvas *
Consiste	Faz consistências na construção do Arco, durante sua criação	Figura *
BuildArc	Processa construção do Arco no Canvas	int, int
MoveDraw	Processa desenho do arco no canvas durante sua movimentação	Canvas *
InvalidaArco	Processa destruição do arco, caso tenha sua construção invalidada	Canvas *

Arco Normal

Contém a implementação dos métodos virtuais da sua classe hierarquicamente superior.

A tabela abaixo apresenta a descrição dos principais métodos desta classe.

Classe: Arco Normal		
Métodos	*Descrição*	*Parâmetros*
DrawArrow	Desenha seta na extremidade do Arco	Canvas *, int, int, int, int, int
AddObject	Insere um Arco Normal na lista de figuras e atualiza seus atributos	Canvas *, Figura *

Arco Inibidor

Contém a implementação dos métodos virtuais da sua classe hierarquicamente superior.

A tabela abaixo apresenta a descrição dos principais métodos desta classe.

Classe: Arco Inibidor		
Métodos	*Descrição*	*Parâmetros*
DrawCircleArcInib	Desenha círculo na extremidade do Arco	Canvas *, int, int, int, int, int
AddObject	Insere um Arco Inibidor na lista de figuras e atualiza seus atributos	Canvas *, Figura *

Fichas

Esta classe implementa o objeto Ficha componente de uma *G-Net*.

Uma Ficha contém os seguintes atributos:

- Tokid: Identificador da ficha
- Seq: sequência da ficha: Nid:ISP:Pid...
- s_color: Sequência de cores da ficha
- b_color: *Branch color* indicando qual a transição que está habilitada para esta ficha
- Msg: Mensagem da ficha. Tem o seguinte formato: nome_campo:valor_campo....

A tabela abaixo apresenta a descrição dos principais métodos desta classe.

Classe: Token		
Métodos	*Descrição*	*Parâmetros*
GetFieldsDialog	Recebe os atributos, da ficha corrente, a serem alterados	Widget, XtPointer, XtPointer
CancelDialog	Cancela a entrada de dados dos atributos da ficha	Widget, XtPointer, XtPointer
OptionChangedColor	Muda Cor de Fundo e de Borda da ficha	Widget, XtPointer, XtPointer
DrawCounter	Desenha, no canvas, o número de fichas contidas no lugar	Canvas *, Figura *
Consiste	Faz consistências durante a criação da ficha	Figura *
Get_Tkid	Retorna o identificador da ficha	void

4.3.3 Diagrama de Mensagens

A Figura 4.8 apresenta um diagrama simplificado do relacionamento entre as classes do ambiente.

A classe Editor relaciona-se com todas as classes através do envio e recebimento de mensagens durante a utilização do *NetGraph* pelo usuário.

A comunicação com a classe Canvas se dá através da solicitação de eventos ocorridos com a manipulação dos botões do *mouse*: pressão do botão esquerdo do mouse (LMBP - Left Mouse Button Press), indica a ativação de um comando; liberação do botão esquerdo do mouse (LMBR - Left Mouse Button Release), indica a conclusão de um comando; pressão do botão direito do mouse (RMBP - Right Mouse Button Press) e liberação do botão direito do mouse (RMBR - Right Mouse Button Release), indicam o processo de cancelamento da construção de elementos componentes da rede;

e movimentação do mouse (PM - Pointer Motion) na área de trabalho com o botão esquerdo do mouse pressionado.

Além disto, o Editor recebe mensagens de solicitação de eventos para atualização e exposição da área gráfica (*redisplay*); envia mensagens para alterar a configuração do canvas através de mensagens para limpar a tela, aumentar o tamanho dos componentes da rede (*zoom in*) ou diminuir o tamanho da rede (*zoom out*).

A comunicação com a classe Figura se dá através do envio de mensagens para seleção, movimentação, deleção e construção dos elementos da rede no canvas.

Com as demais classes, o relacionamento se dá através do recebimento e emissão de eventos tais como: mostrar diálogos (para classe Diálogo), escolha de opções de Menu e comando selecionado (da classe Botões de Comandos).

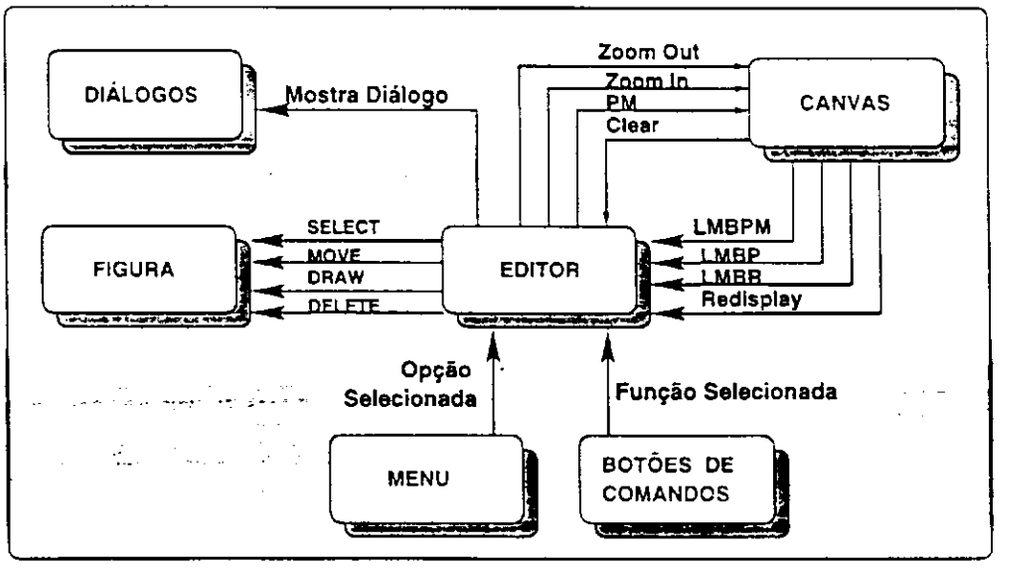

Figura 4.8: Diagrama de Mensagens entre Classes

4.3.4 Ambiente de Programação

X/Motif

O sistema XWindow [45], também conhecido simplesmente como "X", é um sistema de janelas padrão que provê uma base portátil para aplicações com interfaces gráficas ao usuário. Motif é um *toolkit* de alto nível que provê componentes de interface comum

ao usuário necessários para aplicações baseadas em X. Motif está escrito em C, baseado em arquitetura orientada a objetos da *Xt Intrinsics*.

Aplicações Motif [46] são construídas em três níveis distintos como mostrado na Figura 4.9. *Xlib* provê uma interface de baixo nível para os serviços básicos do sistema de janela *underlying*, que consiste em prover operações primitivas que permitem aplicações criarem janelas, textos, gráficos, etc., a biblioteca *Xt Intrinsics* provê uma arquitetura *toolkit* de alto nível cujas facilidades são necessárias para construir os componentes definidos pelo Motif e define a arquitetura usada por todos os *widgets*, e Motif provê uma coleção de componentes para criar interfaces ao usuário, tais como: botões, *scroll bars* e menus.

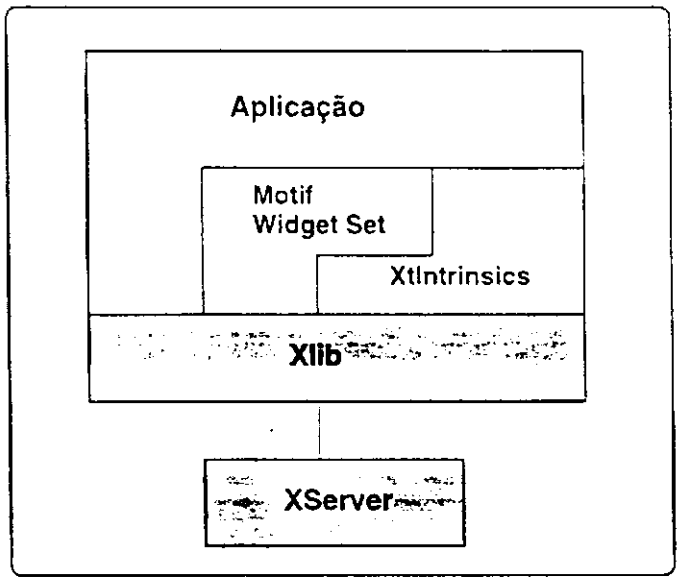

Figura 4.9: Arquitetura do Sistemas de Janelas XWindows

O *XServer* é um processo que normalmente executa sobre uma máquina local e se comunica com clientes através de um protocolo de rede. O *Xlib* manipula o tráfego da rede do lado do cliente e apresenta a interface da linguagem de baixo nível C para as facilidades do *XServer*. A biblioteca *XtIntrinsics* é construída sobre a *Xlib* e esconde muitos detalhes de baixo nível do *Xlib*. Motif é construído sobre a biblioteca *Xt* mas, ocasionalmente, chama funções *Xlib* diretamente. Assim, aplicações Motif usam funções destas três bilbiotecas.

Linguagem de Programação C++

C++ [42] é uma linguagem de programação de propósito geral projetada para tornar a programação mais agradável para programadores profissionais. C++ é um superconjunto da linguagem de programação C. Além das facilidades providas por C, C++ provê outros benefícios eficientes como a definição de novos tipos. Provê técnica para elaboração de programas chamada de *abstração de dados*, que consiste em dividir a aplicação em partes gerenciáveis através da definição de novos tipos, fortemente ligados aos conceitos da aplicação. Objetos de tipos definidos pelo usuário contém informações inerentes a aplicação. Programas usando objetos de tais tipos são chamados *baseados em objetos*. Sendo bem mais compreensíveis e fáceis de manter.

O conceito chave em C++ é *classe*. Uma classe é um tipo definido pelo usuário. Classes permitem esconder dados, inicialização de dados, conversão implícita de tipo para o tipo definido pelo usuário, gerenciamento de memória controlado pelo usuário, tipos dinâmicos e mecanismos para operadores *overloading*. Além disso, C++ provê muito mais facilidades de modularidade, manipulação de objetos do hardware (bits, bytes, endereços, etc.), checagem de tipos de dados, etc. permitindo tipos definidos pelo usuário serem implementados com alto grau de eficiência.

Neste capítulo, apresentamos o *NetGraph* - ambiente para edição e animação de *G-Nets*, descrevemos sua estrutura de classes com seus principais métodos e, finalmente, apresentamos uma visão geral da linguagem de programação utilizada na implementação do ambiente, C++, e da biblioteca gráfica utilizada, Motif. No próximo capítulo, apresentaremos uma sessão típica do ambiente, ao modelarmos uma rede simples.

Capítulo 5

Editor e Animador para G-Nets

A modelagem gráfica de sistemas de *G-Nets* necessita de ferramentas computacionais adequadas para auxiliar o usuário a manipular todos os detalhes de um sistema complexo. Assim, é necessário disponibilizar um editor que possibilite a construção, checar a sintaxe e modificação de *G-Nets*.

Esta seção descreve as funcionalidades da ferramenta *NetGraph*, detalhando as opções e recursos nela disponíveis.

Uma das principais vantagens de usar o editor é a obtenção de melhores resultados durante a modelagem de sistemas, evitando as dificuldades encontradas quando feitas manualmente.

Outra vantagem que podemos citar é a obtenção de resultados mais rápidos. O editor facilita e provê rapidez nas modificações a serem feitas na rede, sem ter que redesenhar o diagrama inteiro. Possibilita a construção de novas partes da rede, copiando ou modificando partes já existentes.

A terceira vantagem é a possibilidade de tornar interativa a apresentação dos resultados de simulação da rede. O animador exibe a sequência de diferentes ocorrências dos disparos das transições de uma *G-Net*. Possibilita a execução contínua (sem a interrupção do usuário) ou permite a interrupção do usuário para ver as transições habilitadas e escolher quais serão disparadas.

5.1 Interface Gráfica

A interface gráfica do *NetGraph* é composta por três partes:

- Canvas: Área onde o usuário pode criar e modificar *G-Nets* usando as facilidades providas pelo editor gráfico.

  O canvas é maior que a área de trabalho exibida para o usuário. Desta forma, o usuário não se restringe a modelar suas redes apenas na área visível, deve usar o *scroll bars*, que está localizado do lado direito e abaixo do canvas, para movimentar para uma parte sua que não é visível.

- Barra de Ferramenta: Compreende um conjunto de botões do lado esquerdo da janela do *NetGraph*, dividida em duas partes:

  1. Elementos da rede: Contém os botões referentes a cada um dos componentes de uma *G-Net*, possibilitando desenhá-los no canvas.
  2. Comandos de Edição: Contém os botões referentes aos comandos para manipulação da rede, bem como modificação e atualização dos dados de seus componentes.

- Menu Principal: Menu de barra localizado no topo da janela do *NetGraph*, exibe as seguintes opções: File, Animation, Export, Help.

  Quando o usuário pressiona uma das opções no Menu de barra, uma janela *popup* é exibida e os comandos deste menu são escolhidos. Cada menu contém um conjunto de comandos relacionados uns com os outros. Por exemplo, o menu *File* contém os comandos para operações de arquivos (salvar, abrir, etc.).

5.2 Menu de Barras

5.2.1 File

No *NetGraph*, as redes modeladas são armazenadas em arquivos. Os arquivos podem ser criados, armazenados e recuperados. Estas operações são executadas nas opções do menu *File* do Menu Principal.

New

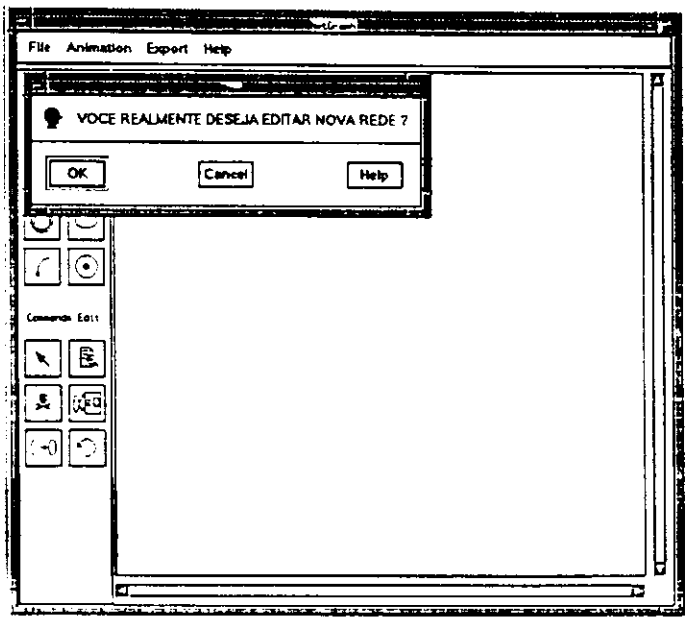

Figura 5.1: Janela da Opção *New*

Ao iniciar os trabalhos no *NetGraph*, o usuário que desejar criar uma nova rede não precisa usar esta opção. Porém se já existir, no canvas, alguma rede modelada e o usuário desejar criar uma nova, esta opção tem que ser executada para apagar todos os elementos existentes na lista de figuras e liberar espaço para a criação de uma nova rede.

A opção *New* abre uma janela como mostrado na Figura 5.1.

Open

As redes modeladas e armazenadas em arquivos são recuperadas usando a opção *Open* do menu File, como mostrado na Figura 5.2. Após sua execução, a rede é desenhada no canvas e armazenada na lista de Figuras.

Quando o usuário seleciona esta opção, uma janela *popup* aparece com a seguinte composição:

- Filter: Caixa de texto onde o usuário entra com o caminho e o padrão do nome do arquivo que especifica o tipo dos arquivos a serem exibidos na lista de arquivos.

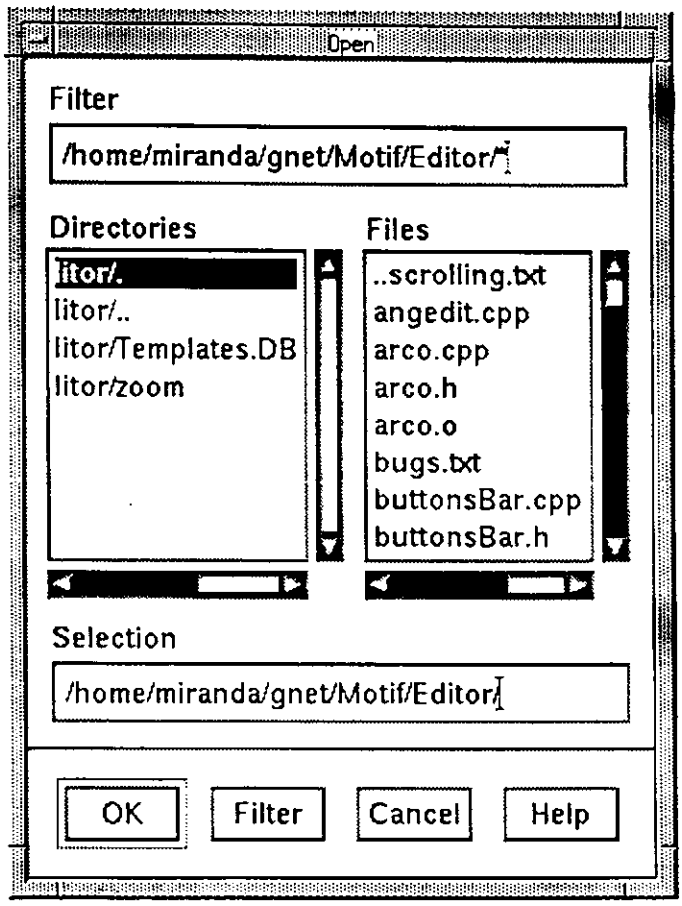

Figura 5.2: Janela da Opção *Open*

O *default* é o diretório corrente do usuário e o tipo é *.* significando todos os arquivos com qualquer extensão.

Quando *Filter* é alterado, *Directories* da lista de arquivos também são atualizados.

- Directories: Lista com os nomes dos diretórios no caminho corrente. Uma dupla seleção com o botão esquerdo do *mouse* sobre um dos elementos desta lista torna o elemento selecionado o diretório corrente. Este efeito é semelhante quando executado no botão *Filter* quando um dos elementos desta lista é selecionado. Quando o diretório é mudado, o caminho no *Filter* é modificado e a lista do *Files* é atualizada.

- Files: Lista de nomes dos arquivos, cujo tipo foi especificado no *Filter*, no diretório

corrente são exibidos.

Uma dupla seleção com o botão esquerdo do *mouse* sobre o nome do arquivo ou selecionar o botão OK, o arquivo selecionado é carregado no canvas e na lista de Figuras.

- Selection: Caixa de textos onde o caminho e o nome do arquivo correntemente selecionado são exibidos. O usuário pode entrar com o nome do arquivo e pressionar o botão OK para carregar o arquivo.
- Cancel: Botão que cancela o processo de abertura de arquivos sem efetuar nenhuma mudança.
- Ok: Botão que executa e finaliza o processo de abertura de arquivos.

Save e Save as

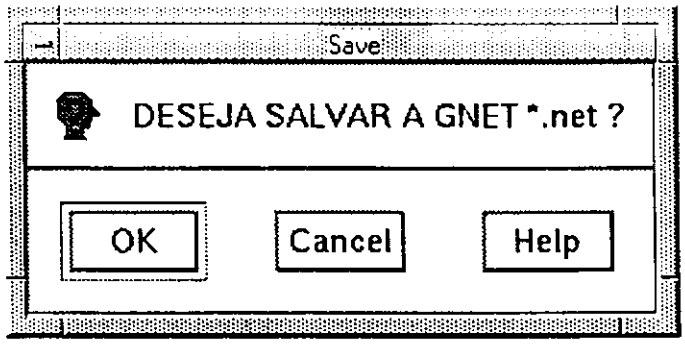

Figura 5.3: Janela da Opção *Save*

As redes modeladas podem ser armazenadas em arquivos usando duas opções diferentes do menu File: *Save* e *Save as*, como mostrado nas Figuras 5.3 e 5.4.

A opção *Save* salva a rede corrente do canvas no diretório de onde foi carregada, com seu nome original. Se esta opção for usada, pela primeira vez que a rede é salva, será gravada com o nome *default* **noname.net** no diretório corrente.

Quando o usuário seleciona a opção *Save as*, o usuário atribui um nome para o arquivo onde será armazenada a rede corrente do canvas. Exibe uma janela *popup* que aparece com a seguinte composição:

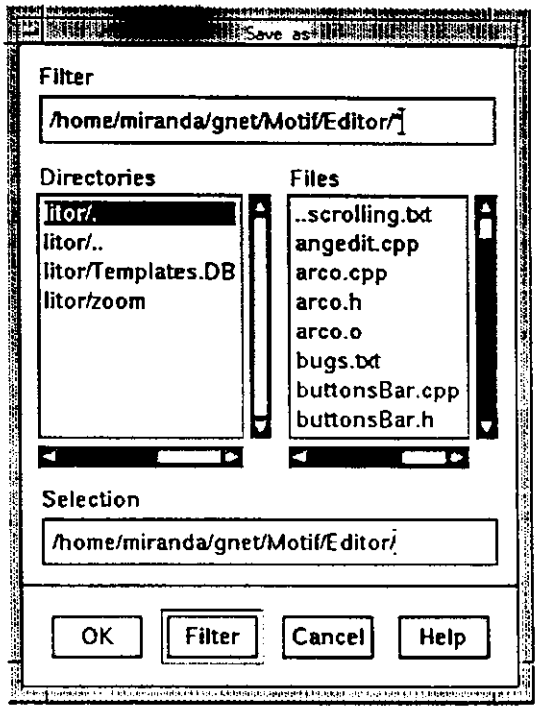

Figura 5.4: Janela da Opção *Saveas..*

- Filter: Caixa de texto onde o usuário entra com o caminho e o padrão do nome do arquivo que especifica o tipo dos arquivos a serem exibidos na lista de arquivos. O *default* é o diretório corrente do usuário e o tipo é *.* significando todos os arquivos com qualquer extensão.

  Quando *Filter* é alterado, *Directories* da lista de arquivos também são atualizados.

- Directories: Lista com os nomes dos diretórios no caminho corrente. Uma dupla seleção com o botão esquerdo do *mouse* sobre um dos elementos desta lista torna o elemento selecionado o diretório corrente. Este efeito é semelhante quando executado no botão *Filter* quando um dos elementos desta lista é selecionado. Quando o diretório é mudado, o caminho no *Filter* é modificado e a lista do *Files* é atualizado.

- Files: Lista de nomes dos arquivos, cujo tipo foi especificado no *Filter*, no diretório corrente são exibidos.

- Selection: Caixa de textos que conterá o caminho e o nome do arquivo onde

armazenará a rede corrente do canvas. O usuário pode entrar com o nome do arquivo e pressionar o botão OK para salvar o arquivo.

- Cancel: Botão que cancela o processo de abertura de arquivos sem efetuar nenhuma mudança.
- Ok: Botão que executa e finaliza o processo de abertura de arquivos.

Se já existir um arquivo com o mesmo nome do que está sendo salvo, então o arquivo anterior será destruído e o atual conterá a rede modelada no canvas.

Quit

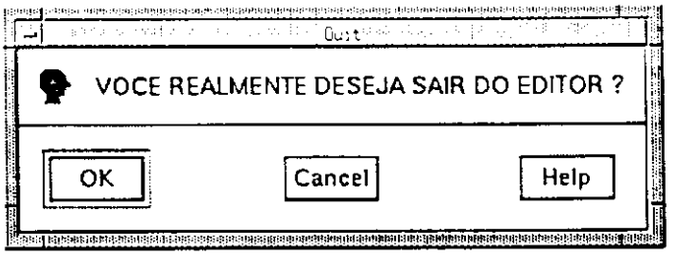

Figura 5.5: Janela da Opção *Quit*

A opção *Quit* encerra os trabalhos no *NetGraph*. Caso exista alguma rede modelada no canvas, o usuário deverá providenciar seu salvamento antes de executar esta opção, caso contrário, perderá as alterações efetuadas e a lista de Figuras terá seu espaço liberado.

A Figura 5.5 exibe a janela da opção *Quit*.

5.2.2 Export

Esta opção processa a geração de um arquivo, relativo à rede corrente do canvas, cujo formato é compatível com a entrada do Simulador *G-Net*. Assim, antes de ser selecionada, o usuário deverá carregar, anteriormente, a rede que deverá ser processada, caso contrário, o editor emitirá uma mensagem de erro.

5.2.3 Animation

Esta opção inicia o processo de animação da rede carregada pelo usuário. Para que este processo seja executado com sucesso, é necessário que o usuário tenha gerado o arquivo compatível com o simulador (através da opção *Export*), carregue a rede desejada no canvas (através da opção *Open*) e selecione a opção *Animation* no Menu Principal.

Será aberta uma caixa de diálogos, solicitando o método que deverá se executado nesta rede. Após isto, a animação será iniciada.

O processo consiste numa troca de mensagens com o Simulador de *G-Nets*. Estas mensagens têm a seguinte definição sintática:

$Mensagem :=< arg1 >< arg2 > ... < argn > \backslash$ n

Onde:

- $< argi >:=$ cadeia de caracteres com terminação #
- Cadeia de caracteres $:= Pn_1Tn_2$, onde:
  - n_1 é o identificador do Lugar
  - T é o tipo de operação (P ou R): (P: colocar fichas e R significa remover fichas)
  - n_2 é o número de fichas a serem processadas.

5.3 Barra de Ferramentas

5.3.1 Elementos da Rede

O editor gráfico *NetGraph* permite desenhar e editar os elementos componentes de uma *G-Net*. Eles podem ser desenhados no canvas usando os ícones da barra de ferramenta como na Figura 5.6.

Lugar Normal

Para criar um Lugar Normal, o usuário deve ativar o botão relativo na barra de ferramentas, seu ícone é um círculo.

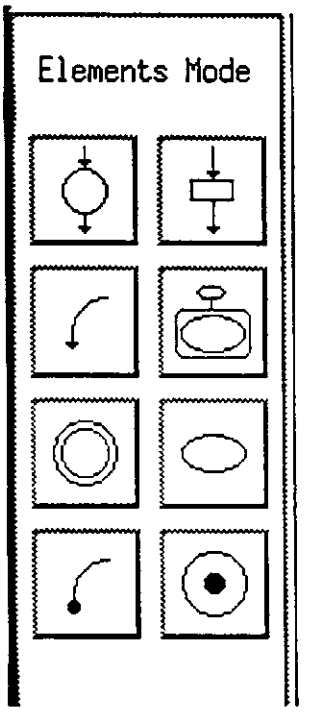

Figura 5.6: Barra de Ferramentas - Elementos da Rede

Após ativar o ícone de Lugar Normal, o apontador do *mouse* entra no canvas indicando que é possível desenhá-lo.

Quando um Lugar Normal é criado, seus atributos são, inicialmente, um conjunto de valores *default*, como a seguir:

- Identificador do Lugar: valor composto pela letra P e um número inteiro consecutivo ao último lugar criado, isto é, os lugares são identificados como P0, para o primeiro lugar criado, P1, para o segundo lugar criado, e assim por diante. Os nomes aparecem do lado esquerdo do lugar.
- Coordenadas no canvas: valor correspondente às posições x e y no canvas onde o lugar foi desenhado.
- Largura: tem valor *default* 30.
- Altura: tem valor *default* 30.
- CorFundo: tem valor *default* 0.

- CorBorda: tem valor *default* 0.
- Capacidade: valor *default* igual a 1, ou seja, comporta uma ficha, inicialmente.
- Mensagem: inicialmente, é vazia.
- Primitiva: refere-se à função criada pelo usuário para o Lugar Normal.

Para alterar os valores destes atributos, tanto para Lugar Normal quanto para as outras herdeiras da classe Lugar, o usuário deve seguir os procedimentos descritos no botão *Update* da barra de ferramentas (ver Seção 5.3.2).

Após sua criação, o lugar não tem nenhuma ficha a ele associada. O usuário pode inserir fichas utilizando os procedimentos descritos no botão *Ficha* da barra de ferramentas (ver posteriormente nesta Seção).

Lugar Alvo

Para criar um Lugar Alvo, o usuário deve ativar o botão relativo na barra de ferramentas, seu ícone é um círculo duplo.

Após ativar o ícone de Lugar Alvo, o apontador do *mouse* entra no canvas indicando que é possível desenhá-lo.

Quando um Lugar Alvo é criado, seus atributos são, inicialmente, um conjunto de valores *default*, como a seguir:

- Identificador do Lugar: valor composto pela letra P e um número inteiro consecutivo ao último lugar criado, isto é, os lugares são identificados como P0, para o primeiro lugar criado, P1, para o segundo lugar criado, e assim por diante. Os nomes aparecem do lado esquerdo do lugar.
- Coordenadas no canvas: valor correspondente às posições x e y no canvas onde o lugar foi desenhado.
- Largura: tem valor *default* 30.
- Altura: tem valor *default* 30.
- CorFundo: tem valor *default* 0.

- CorBorda: tem valor *default* 0.
- Capacidade: valor *default* igual a 1, ou seja, comporta uma ficha, inicialmente.
- Mensagem: inicialmente, é vazia.

Isp

Para criar um Isp, o usuário deve ativar o botão relativo na barra de ferramentas, seu ícone é uma elipse.

Após ativar o ícone de Isp, o apontador do *mouse* entra no canvas indicando que é possível desenhá-lo.

Quando um Isp é criado, seus atributos são, inicialmente, um conjunto de valores *default*, como a seguir:

- Identificador do Lugar: valor composto pela letra P e um número inteiro consecutivo ao último lugar criado. Os nomes aparecem do lado esquerdo do lugar.
- Coordenadas no canvas: valor correspondente às posições x e y no canvas onde o lugar foi desenhado.
- Largura: tem valor *default* 50.
- Altura: tem valor *default* 20.
- CorFundo: tem valor *default* 0.
- CorBorda: tem valor *default* 0.
- Capacidade: valor *default* igual a 1, ou seja, comporta uma ficha, inicialmente.
- Mensagem: inicialmente, é vazia.
- Identificador da Rede: refere-se à rede a ser invocada. É, inicialmente, vazio.
- Nome do método: inicialmente, é vazio.
- Atributos: inicialmente, é vazio.

Transição

Para criar uma Transição, o usuário deve ativar o botão relativo na barra de ferramentas, seu ícone é um retângulo.

Após ativar o ícone de Transição, o apontador do *mouse* entra no canvas indicando que é possível desenhá-lo.

Quando uma Transição é criada, seus atributos são, inicialmente, um conjunto de valores *default*, como a seguir:

- Identificador da Transição: valor composto pela letra T e um número consecutivo, isto é, as transições são identificados como T0, para a primeira transição criada, T1, para a segunda transição criada, e assim por diante. Os nomes aparecem do lado esquerdo da transição.
- Coordenadas no canvas: valor correspondente às posições x e y no canvas onde a transição foi desenhada.
- Largura: tem valor *default* 40.
- Altura: tem valor *default* 10.
- CorFundo: tem valor *default* 0.
- CorBorda: tem valor *default* 0.
- Condição de habilitação: inicialmente, é considerada como passiva, sem condições de disparo.
- Tempo Mínimo: refere-se ao tempo mínimo de habilitação. É, inicialmente, 0.
- Tempo Máximo: refere-se ao tempo máximo de habilitação. É, inicialmente, 0.

Para alterar os valores destes atributos, o usuário deve seguir os procedimentos descritos no botão *Update* da barra de ferramentas (ver Seção 5.3.2).

Arco Normal

Para criar um Arco Normal, o usuário deve ativar o botão relativo na barra de ferramentas, seu ícone é um arco com uma séta na extremidade.

Para criar um arco, o usuário deve primeiro selecionar o nodo inicial (Lugar ou transição) e, depois, selecionar o nodo final. Os arcos normais só são construídos de transições para lugares ou de lugares para transições.

Uma vez que o usuário selecionou um nodo, nenhum arco é desenhado até que um outro nodo, válido, seja selecionado. Por exemplo, se o usuário selecionou a transição T0, posteriormente selecionou a transição T1 e, finalmente, selecionou o lugar P2, um arco normal é desenhado de T0 até P2. Neste caso, a seleção da transição T1 não tem efeito.

Seguindo a descrição anterior, arcos são construídos como linhas retas. No entanto, o usuário pode desenhar arcos com quebras de linhas. Por exemplo, o usuário seleciona um nodo e seleciona uma posição vazia do canvas para ser o ponto de quebra. O usuário pode executar até 50 pontos de quebra, até selecionar o nodo final.

Quando um arco normal é criado, seus atributos são, inicialmente, um conjunto de valores *default*, como a seguir:

- Identificador do Arco Normal: valor composto pela letra A e um número consecutivo, isto é, os arcos são identificados como A0, para o primeiro arco criado, A1, para o segundo arco criado, e assim por diante.
- Coordenadas no canvas: vetor de valores correspondente às posições x e y no canvas onde ocorreu quebra de arcos.
- CorFundo: tem valor *default* 0.
- CorBorda: tem valor *default* 0.
- Peso do arco: inicialmente, é atribuído 1, ou seja, apenas uma ficha é retirada ou inserida dos lugares de entrada ou saída.
- Branch Color: refere-se ao *b_color* da ficha a ser processada pelo arco.

Para alterar os valores dos atributos de arco normal e arco inibidor, o usuário deve seguir os procedimentos descritos no botão *Update* da barra de ferramentas (ver Seção 5.3.2).

Arco Inibidor

Para criar um Arco Inibidor, o usuário deve ativar o botão relativo na barra de ferramentas, seu ícone é um arco com um círculo na extremidade.

Para criar um arco inibidor, o usuário deve primeiro selecionar o nodo inicial (lugar) e, depois, selecionar o nodo final(transição). Os arcos inibidores só são construídos de lugares para transições.

Uma vez que o usuário selecionou um lugar, nenhum arco inibidor é desenhado até que uma transição seja selecionada.

Os arcos inibidores têm procedimento de construção semelhante ao arco normal, descrito anteriormente.

Quando um arco inibidor é criado, seus atributos são, inicialmente, um conjunto de valores *default*, como a seguir:

- Identificador do Arco Inibidor: valor composto pela letra A e um número consecutivo, isto é, os arcos são identificados como A0, para o primeiro arco criado, A1, para o segundo arco criado, e assim por diante.
- Coordenadas no canvas: vetor de valores correspondente às posições x e y no canvas onde ocorreu quebra de arcos.
- CorFundo: tem valor *default* 0.
- CorBorda: tem valor *default* 0.
- Peso do arco: inicialmente, é atribuído 1, ou seja, apenas uma ficha é retirada ou inserida dos lugares de entrada ou saída.
- Branch Color: refere-se ao *b_color* da ficha a ser processada pelo arco.

Ficha

Para criar Fichas, o usuário deve ativar o botão relativo na barra de ferramentas, seu ícone é um círculo com uma esfera preenchida no seu interior.

Fichas são criadas dentro de Lugares. Assim, para adicioná-las, o usuário deve primeiro selecionar o Lugar. Quando uma ficha é inserida pela primeira vez em um lugar, seu símbolo representativo aparece dentro daquele lugar. Após a primeira, sempre que uma ficha é criada, aparece um número sobre a ficha, indicando sua quantidade naquele lugar.

Quando uma ficha é criada, seus atributos são, inicialmente, um conjunto de valores *default*, como a seguir:

- Identificador da Ficha: valor composto pelas letras Tk e um número consecutivo, isto é, as fichas são identificados como Tk0, para a primeira ficha criada, Tk1, para a segunda ficha criada, e assim por diante.
- Coordenadas no canvas: valor correspondente às posições x e y no canvas do lugar contendo a ficha.
- CorFundo: tem valor *default* 0.
- CorBorda: tem valor *default* 0.
- Sequencia de Propagação da Ficha: inicialmente, é vazia.
- Branch Color: refere-se ao *b_color* indicando que transição habilita a ficha.
- Mensagem: inicialmente, é vazia.

Para alterar os valores destes atributos, o usuário deve seguir os procedimentos descritos no botão *Alteração de Atributos* da barra de ferramentas (ver Seção seguinte).

5.3.2 Comandos de Edição

O editor gráfico *NetGraph* permite alterar e manipular os elementos componentes de uma *G-Net*. Eles podem ser manipulados usando os ícones da barra de ferramenta como na Figura 5.7.

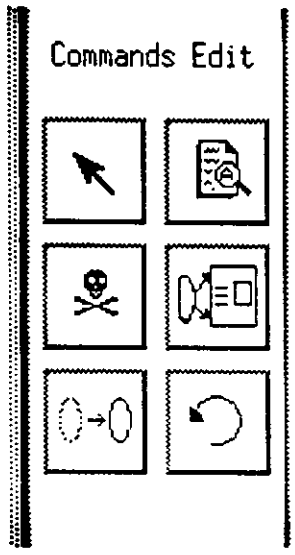

Figura 5.7: Barra de Ferramentas - Comandos de Edição

Seleção

O ícone relativo a este comando é um botão com uma seta na barra de ferramentas. Para selecionar objetos é necessário clicar o botão esquerdo do *mouse* dentro da figura desejada. A figura selecionada fica circundada com um retângulo com bordas destacadas.

Para selecionar várias figuras, o usuário deve selecionar o botão esquerdo do *mouse* sobre as diversas figuras desejadas. Para cancelar a seleção de uma figura, é necessário apenas um novo clique sobre a figura desejada.

Move

O usuário pode mover objetos dentro do canvas para a posição que desejar.

Quando o botão é ativado, o usuário posiciona o *mouse* no objeto que deseja movimentar, pressiona o botão esquerdo do *mouse* e arrasta-o até a nova posição, mantendo o botão do *mouse* pressionado. Ao liberar o botão do *mouse*, o objeto aparece na posição indicada.

Se os objetos movimentados forem lugares ou transições, os seus respectivos arcos de entrada e de saída são ajustados para a nova posição dos objetos por eles conectados.

A movimentação de arcos, só pode ser efetuada posicionando o *mouse* nas suas

quebras. O usuário posiciona o *mouse* na quebra do arco desejado e arrasta-o para a nova posição. Ao liberar o *mouse*, o arco será redesenhado para a nova posição do movimento executado.

Remoção

O ícone relativo a este comando é uma caveira situado na barra de ferramentas.

Para apagar um objeto do canvas, ativa-se o botão de *Remoção*, posiciona-se o *mouse* sobre o objeto desejado e pressiona o botão esquerdo do *mouse*.

Se o objeto deletado for lugar ou transição, também serão apagados todos seus respectivos arcos de entrada e de saída. Estes objetos também serão removidos da lista de Figuras, gerenciada pelo Editor.

Zoom

Através dos dois ícones de Zoom, o usuário pode aumentar ou reduzir a rede, correntemente, mostrada no canvas. Caso o usuário queira verificar o diagrama inteiro da rede, ele executa o zoom para diminuir o tamanho da rede e caso o usuário queira olhar detalhes de parte da rede modelada no canvas, o usuário executa o zoom para aumentar o tamanho da rede.

A quantidade de *zoom's* a ser executado é limitado pela capacidade de memória da região do canvas. Assim, este processo pode ser feito em média 4 a 5 vezes para aumentar ou diminuir o tamanho da rede.

Undo

Este comando é executado quando o usuário deseja desfazer algum comando processado anteriormente.

O *Undo* empilha a última operação executada e ao ativá-lo este comando é imediatamente desfeito. A lista de Figuras e a rede, correntemente, modelada no canvas voltam ao estado que estavam antes do comando ser executado.

O *Undo* é executado sobre criação, movimentação, deleção, seleção e atualização de atributos de objetos da rede corrente do canvas.

Na versão atual, só existe um nível de *Undo*.

Alteração de Atributos

Este comando é executado quando o usuário deseja alterar os atributos dos objetos da rede.

Para alterar os atributos de um objeto, o usuário deve seguir os seguintes passos:

- Ativar o botão *UpDate* da barra de ferramentas.
- Selecionar o objeto que será alterado.
- Preencher os campos da caixa de diálogo correspondente ao objeto selecionado.
- Confirmar as alterações efetuadas.

Cada tipo de objeto tem diferentes propriedades e caixas de diálogos para modificá-los. Os atributos de cada um dos objetos foram listados na seção anterior.

Capítulo 6

Conclusão

Nos capítulos apresentados nesta dissertação, mostramos o desenvolvimento de um ambiente, denominado *NetGraph*, para edição e animação de uma classe de redes de Petri de alto nível denominada *G-Nets*.

Com o objetivo de desenvolvermos esta ferramenta gráfica, discutimos os critérios principais para a elaboração de uma interface gráfica para redes de Petri. Para tanto, estudamos algumas ferramentas que apresentam características similares àquela que desenvolvemos.

O *NetGraph* é uma ferramenta gráfica cujo objetivo é prover aos usuários um ambiente que, através da utilização de uma metodologia orientada a objetos, possibilita a representação formal e executável de projeto de sistemas. Além disto, provê a animação de modelos de sistemas complexos, facilitando a visualização gráfica da execução do modelo, possibilitando a eliminação de erros de projetos e inconsistências. Recursos gráficos, como botões, barras de rolamento e menus, facilitam o trabalho do usuário, por permitir correções de seus erros interativamente e manipular os objetos da rede de maneira fácil e eficiente.

Além disto, o *NetGraph* possui recursos que favorecem a construção de modelos, onde são feitas consistências estruturais da rede, interativamente, durante a modelagem.

O animador, parte constituinte do *NetGraph*, é responsável pela implementação da regra de disparo para *G-Nets*. Para tanto, definimos e implementamos um protocolo de comunicação entre o animador e simulador de forma que possibilite o simulador

enviar mensagens para o animador para representar, graficamente, o movimento de fichas na rede. Isto facilitou o trabalho do usuário, uma vez que pode ser observado o comportamento dinâmico da rede.

Definimos a arquitetura para o *NetGraph* para integrá-la ao ambiente de modelagem e verificação de *G-Nets*. Especificamos uma hierarquia de classes representando a semântica formal dos objetos componentes de uma rede *G-Net*. A abordagem utilizada na implementação do *NetGraph* facilita a manutenção do código e adição de novos recursos, o que torna possível a criação de novos trabalhos no sentido de aperfeiçoá-lo e ampliá-lo.

Um problema tratado na implementação desta ferramenta foi a necessidade de encapsular diversas funções do Motif, representando-as através de classes, já que Motif e C++ possuem características diferentes de programação. Toda a inicialização do ambiente e seu tratamento gráfico foram encapsulados em classes que compõem a especificação final do ambiente. Esta decisão facilitou o gerenciamento dos objetos gráficos que compõem a ferramenta.

O *NetGraph* em sua implementação atual tem 21.000 linhas de código fonte.

6.1 Trabalhos Futuros

- Integrar o *NetGraph* a um ambiente cooperativo, para possibilitar a integração de equipes de desenvolvimento, provendo um ambiente de desenvolvimento concorrente com trabalhos cooperativos.

- Algumas facilidades, de cunho basicamente gráfico, podem ser introduzidas, como por exemplo: alteração de expessura de linhas, inclusão de *grid*, etc.

Bibliografia

[1] *OSF/Motif Programmer's Guide*, 1990.

[2] M. Baldassari, G. Bruno, V. Russi, and R. Zompi. PROTOB a hierarchical object-oriented CASE tool for distributed systems. In C. Ghezzi and J.A. McDermid, editors, *ESEC'89*, volume 387 of *Lecture Notes in Computer Science*, pages 425–445. Springer-Verlag, 1989.

[3] E. Battiston, F. De Cindio, and G. Mauri. OBJSA nets: a class of high-level nets having objects as domains. In G. Rozenberg, editor, *Advances on Petri Nets 1988*, volume 340 of *Lecture Notes in Computer Science*, pages 20–43. Springer-Verlag, 1988.

[4] E. Battiston and F. de Condio. Class orientation and inheritance in modular algebraic nets. In *Proc. of IEEE International Conference on Systems, Man and Cybernetics*, pages 717–723, Le Touquet, France, 1993. IEEE.

[5] G. Booch. *Object Oriented Design: With Applications.* Menlo Park, CA: Benjamin/Cummings, 1994.

[6] D. Buchs and N. Guelfi. CO-OPN: A concurrent object oriented petri net approach. In *Peoceedings of the 12th International Conference on the Application and Theory of Petri Nets*, Lecture Notes in Computer Science. Springer Verlag, Gjern, Denmark, 1991.

[7] T.C. Chen, Y. Deng, and S.K. Chang. A simulator for distributed systems using g-nets. In *Proceedings of 1992 Pittsburgh Simulation Conference*, Pittsburgh, PA, USA, May 1992.

[8] J.C.A. de Figueiredo. *Redes de Petri com Temporização Fuzzy.* PhD thesis, Universidade Federal da Paraíba - Campus II, Campina Grande, Paraíba, 1994.

[9] Y. Deng. *A Unified Framework for the Modeling, Prototyping and Design of Distributed Information Systems.* PhD thesis, Department of Computer Science, University of Pittsburgh, 1992.

[10] Y. Deng, S.K. Chang, J.C.A. de Figueiredo, and A. Perkusich. Integrating software engineering methods and petri nets for the specification and prototyping of complex software systems. In M. Ajmone Marsan, editor. *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 206 - 223. Springer-Verlag, Chicago, USA, June 1993.

[11] J. Engelfriet, G. Leih, D. Rozenberg, G.Janssens, and G. Rozenberg. Net-based description of parallel object-based systems, or POTs and POPs. In J.W. de Bakker, W.P. de Roever, J.W. Rozenberg, G.de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science, pages 229–244229–244. Springer-Verlag.

[12] R. Fairley. *Software Engineering Concepts.* MacGraw-Hill, New York, NJ, 1985.

[13] H.J. Genrich. Predicate/Transition nets. In W. Brauer, W. Reisig, and G. Rozemberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[14] H.J. Genrich and K. Lautenbach. System modeling with high level-petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[15] Philip Gavin Hodgson. *Backend to the Xloopn screen Editor.* Master thesis, Universidade da Tasmania, November 1994.

[16] P. Huber, K. Jensen, and R.M. Shapiro. Hierarchies in coloured petri nets. In Jensen. K. and G. Rozenberg, editors, *High-Level Petri Nets: Theory and Application*, pages 313–341. Springer-Verlag, 1991.

[17] K. Jensen. Coloured petri nets: A high level language for system design and analysis. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[18] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use.*, volume 1. Springer-Verlag, 1992. to appear in EATCS Monograph on Theoretical Computer Science.

[19] Oliver Jones. *Introduction to the XWindow Systems*. Prentice-Hall, 1989.

[20] C.A. Lakos and C.D. Keen. Modelling layered prorocols in LOOPN. In *Proceeding of Fourth International Workshop on Petri Nets and Performance Models*, Melbourne, Australia, 1991.

[21] S. Lu and Y. Deng. An environment for specification, simulation and analysis of distribuited object-oriented systems. *7th International Conference on Software Engineering and Knowladge Engineering*, pages 402–410, Junho 1995.

[22] MARS-Team. Cpn-ami 2.0 (beta release) - the mars method. Technical report, Laboratoire MASI, Institut Blaise Pascal, Universite P. and M. Curie - 4 place Jussieu, 75252 Paris, April 1996.

[23] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Francheschinis. *Modelling with Generalized Stochastic Petri Nets*. Jonh Wiley and Sons, 1995.

[24] M.V.C. Miranda, J.C.A de Figueiredo, and A. Perkusich. Um ambiente para edição e animação de g-nets. In *Anais do Congresso Brasileiro de Automática*, September 1996.

[25] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[26] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.

[27] H. Oswald, R. Esser, and R. Mattmann. An environment for specifying and executing hierarchical petri nets. In *Proceedings of the 12th International Conference on Software Engineering*, pages 164–172, 1990.

[28] A. Perkusich. *Análise de Sistemas Complexos Baseada na Decomposição de G-Nets.* PhD thesis, Curso de Pòs Graduação em Engenharia Elétrica, Universidade Federal da Paraíba, Campina Grande, PB, August 1994. Também disponível em inglês com título: Analysis of G-Net Systems Based Upon Decomposition.

[29] A. Perkusich and J.C.A. de Figueiredo. A g-net based environment for logical and timing analysis of software system. In *Anais do SBES'95, Simpósio Brasileiro de Engenharia de Software*, pages 56–75, Recife, PE, October 1995.

[30] A. Perkusich and J.C.A. de Figueiredo. G-nets: A petri net based approach for logical and timing analysis of complex software systems. *To Apper in Journal of Systems and Software*, 1997.

[31] A. Perkusich, J.C.A. de Figueiredo, and S.K Chang. Embedding fault-tolerant properties in the design of complex systems. *Journal of Systems and Software*, 2(25):23–37, 1994.

[32] A. Perkusich and J.C.A. Figueiredo. Concepção de sistemas orientados a objetos: Abordagem por redes de petri. *XXI Conferência Latino-Americana de Informática. Canela-Brasil*, Julho / 1995.

[33] A. Perkusich, M.L.B. Perkusich, and S.K Chang. G-nets: A petri net based approach for logical and timing analysis of complex software systems. *International Journal of Software Engineering and Knowledge Engineering*, 6(3):447–476, 1996.

[34] J. L. Peterson. *Petri Nets Theory and Modeling of Systems.* Prentice-Hall, 1981.

[35] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*, volume 1 of *Computer Science Series.* McGraw-Hill International Editions, second edition, 1988.

[36] V. Quercia and T. O'Reilly. *X Windows Systems User's Guide - Vol. 3*, volume 3. O'Reilly and Associates, 1993.

[37] W. Reisig. *Petri Nets.* Springer-Verlag, 1985.

[38] W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[39] Hossein Saiedian. An invitation to formal methods. *IEEE - Computer*, pages 16–17, April 1996.

[40] M. Shaw. *The Impact of Modeling and Abstration Concerns on Modern Programming Languages*, volume 1. M. Brodie, J. Mylopoulos and J. Schmidt, New York, NY, 1984.

[41] Ian Sommervill. *Software Engineering*, volume 1. Addison-Wesley Publishers Ltd, fourth edition, 1992.

[42] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, second edition, 1994.

[43] UNIX System Laboratories, Inc., editor. *Open Look, Graphical User Interface: Programmer's Guide*. Unix Press, 1992.

[44] P.A.C. Verkoulen. *Integrated Information Systems Design: An Approach Based on Object-Oriented Concepts and Petri Nets*. PhD thesis, Technical University of Eindhoven, The Neetherlands, 1993.

[45] D.A. Young. *The X Window System Programming and Applications with Xt*. Prentice Hall, New Jersey, 1990.

[46] Douglas A. Young. *Object-Oriented Programming with C++ and OSF/Motif*. Prentice-Hall PTR Ed, 2 edition, December 1995.

[47] T. Znati, Y. Deng, B. Field, and S.K. Chang. Multi-level specification and protocol design for distributed multimedia communication. In *Procceedings of Conference on Organizational Computing Systems*, pages 255–268, Atlanta, GA, USA, November 1991.

Apêndice A

Operação do NetGraph

Neste apêndice, apresentamos um exemplo de uma sessão de modelagem e edição de uma rede, utilizando o *NetGraph*. Exibimos uma sucessão de telas e passos a serem seguidos para editar uma rede. Por uma questão de simplicidade, apresentamos, como exemplo, a rede do produtor/consumidor, mais especificamente o módulo do produtor.

A.1 Exemplo de Sessão com o NetGraph

Para executar o *NetGraph* é necessário o usuário digitar o seguinte comando na linha de comando do UNIX:

% netgraph

A modelagem começa a partir do objeto *GSP*. Ao inserir um GSP no canvas, aparece uma caixa de diálogos para que o usuário digite dados sobre a rede que será modelada, como: identificador da rede, atributos, métodos e argumentos.

A Figura A.1 mostra a entrada do GSP no canvas.

Durante o processo de edição, o usuário pode executar comandos da barra de ferramentas, necessários para o processo de modelagem da rede.

A construção de uma rede pode ser feita livremente pelo usuário. Os objetos componentes da rede podem ser desenhados na sequência que o usuário desejar. A Figura A.2 e A.3 mostram as etapas parciais e a figura A.4 exibe o resultado final da modelagem.

Para movimentar elementos da rede para outras posições no canvas, o usuário deve

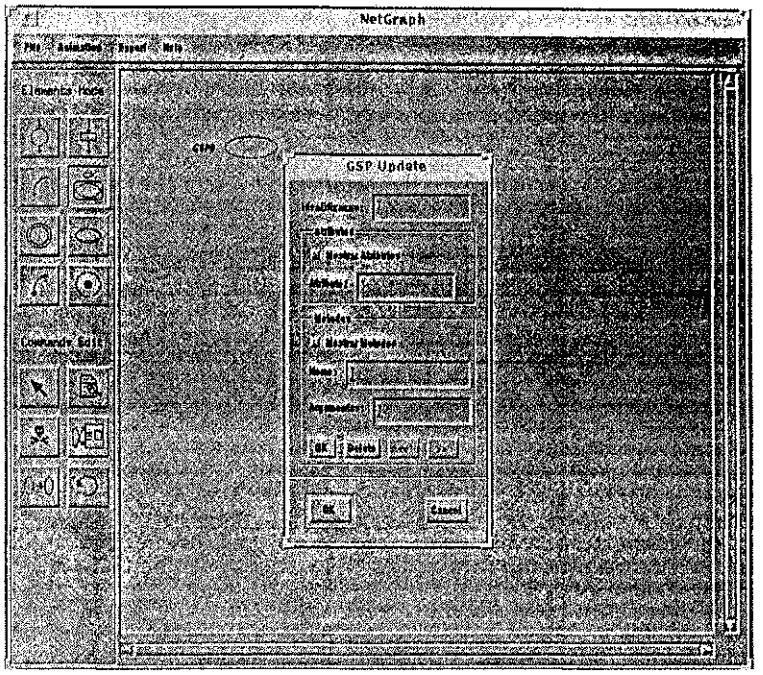

Figura A.1: Criação do objeto GSP

se posicionar nos elementos desejados e arrastá-los para suas posições finais. A Figura A.5 mostra que os arcos ligados aos elementos P0 e T4 foram também deslocados.

Para executar a operação de *Seleção*, o usuário deve selecionar o ícone com o desenho de uma seta e pressionar o mouse sobre o elemento que deverá ser selecionado. A Figura A.6 mostra os elementos P1 e T2 com contornos escuros, indicando sua seleção.

Para remover um determinado elemento da rede, o usuário deve selecionar, na barra de ferramentas, o botão com a figura de uma caveira e posicionar o mouse no elemento desejado, efetuando a remoção deste elemento do canvas e da lista de figuras. Após a remoção, todos os arcos de entrada e saída deste elemento serão também removidos. A Figura A.7 mostra a remoção do elemento P1, onde seus arcos também foram suprimidos.

Para alterar atributos dos componentes da rede, o usuário deve utilizar a opção *UpDate*.

Esta operação pode ser executada em qualquer momento da edição. Para alterar os atributos de um lugar normal, mostrado na Figura A.8, o usuário deve selecionar o lugar desejado. Aparecerá uma caixa de diálogos com os atributos possíveis de serem

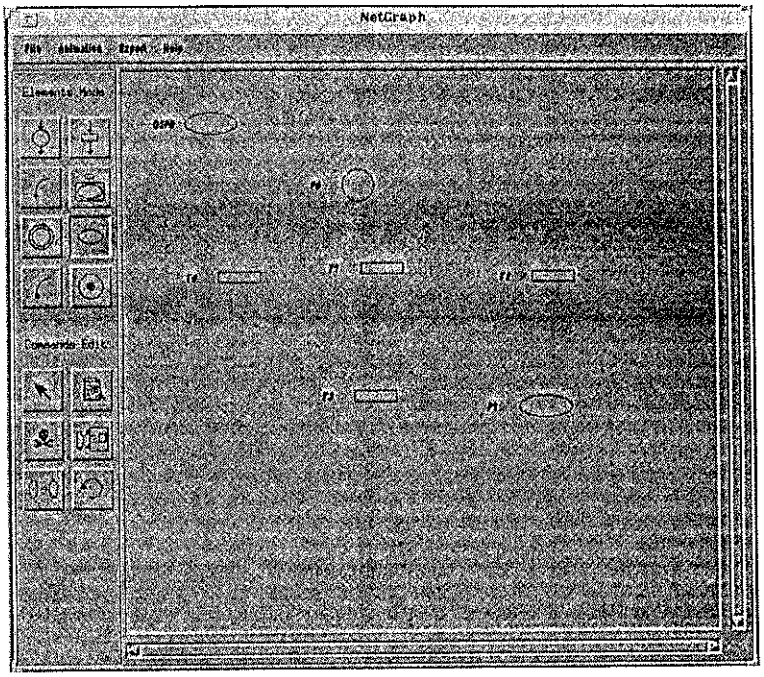

Figura A.2: Rede modelada pelo usuário - exibição Parcial

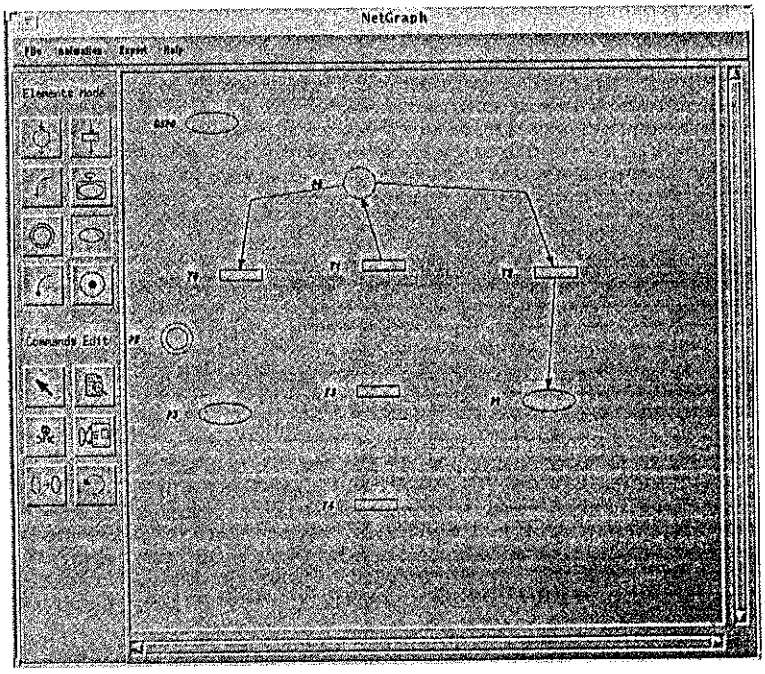

Figura A.3: Rede modelada pelo usuário - exibição Parcial

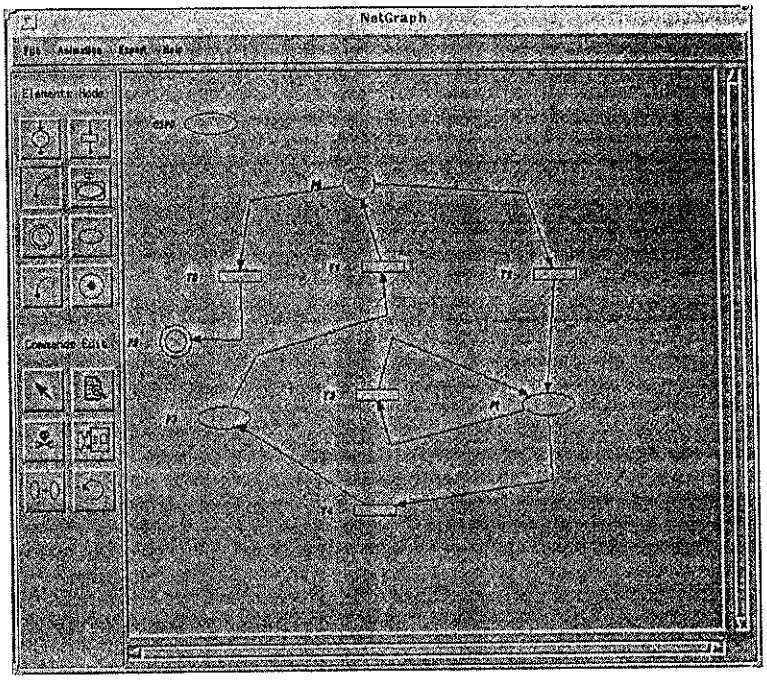

Figura A.4: Rede modelada pelo usuário - Resultado Final

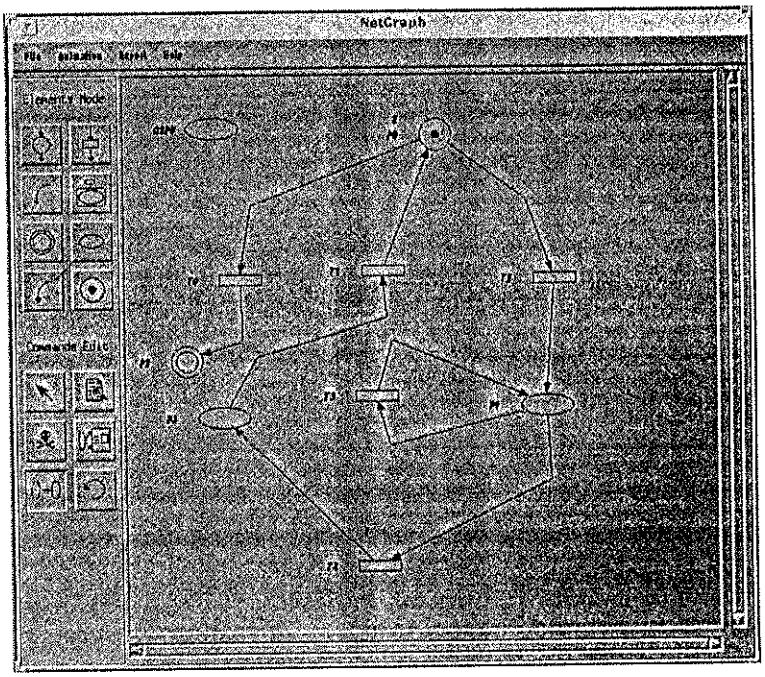

Figura A.5: Rede após o comando *Move*

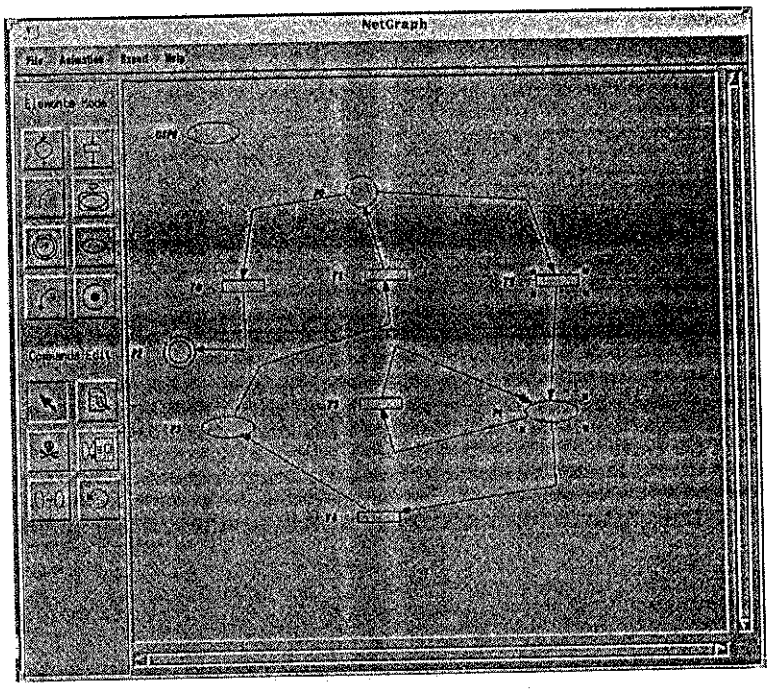

Figura A.6: Rede exibindo a seleção de seus componentes

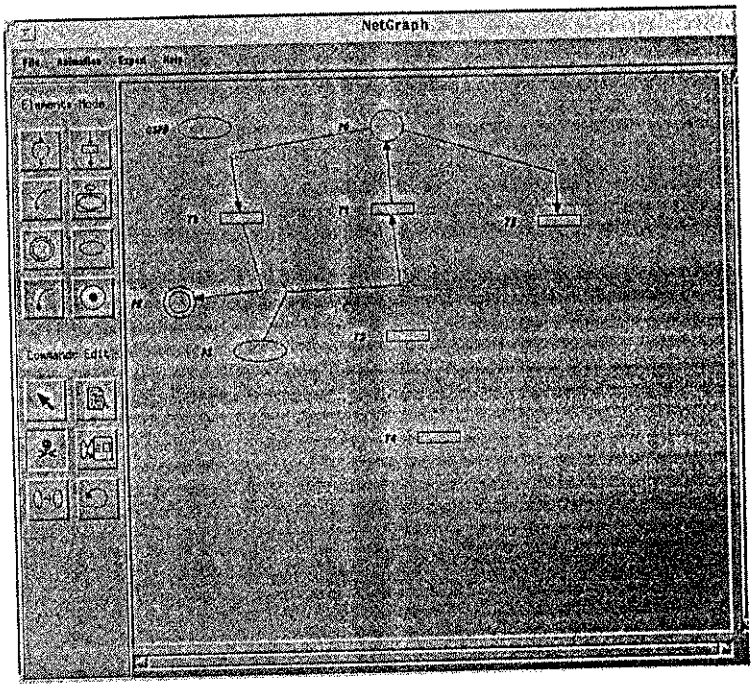

Figura A.7: Rede após a remoção de um de seus componentes

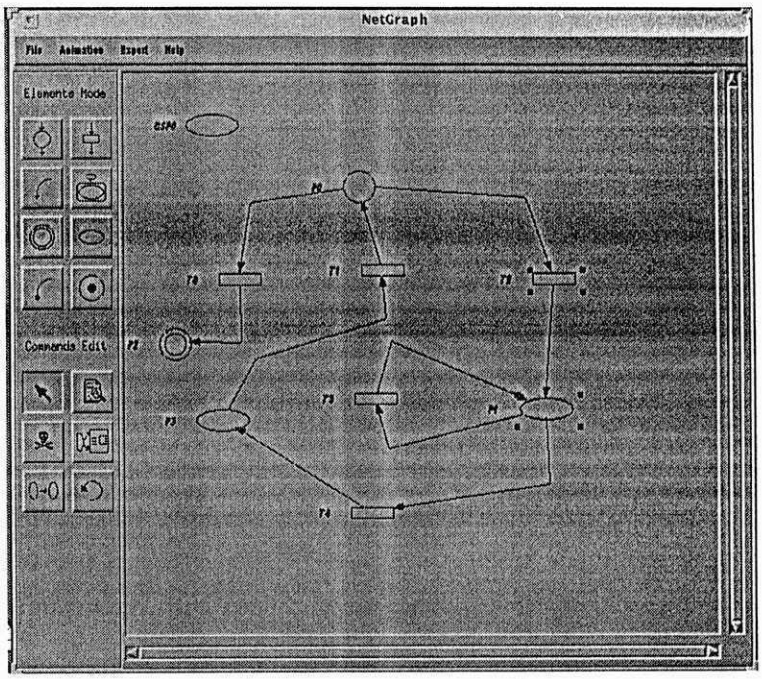

Figura A.6: Rede exibindo a seleção de seus componentes

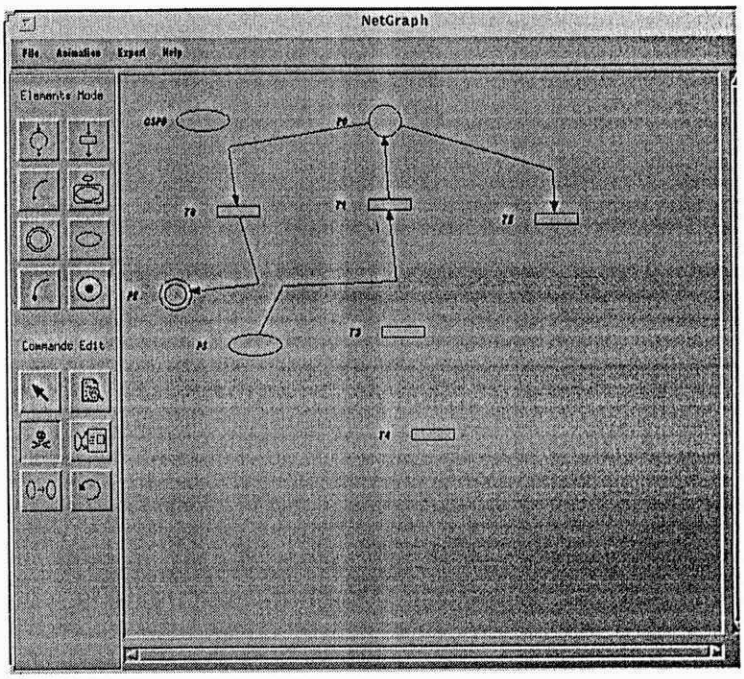

Figura A.7: Rede após a remoção de um de seus componentes

alterados, como: capacidade, mensagem e função primitiva que ele deseja alterar. Se este lugar contiver fichas e o usuário desejar alterar os atributos de alguma(s) dela(s), deve selecionar a ficha desejada pressionando o mouse sobre o identificador da ficha. Aparecerá uma caixa de diálogos com os atributos referentes à ficha selecionada pelo usuário.

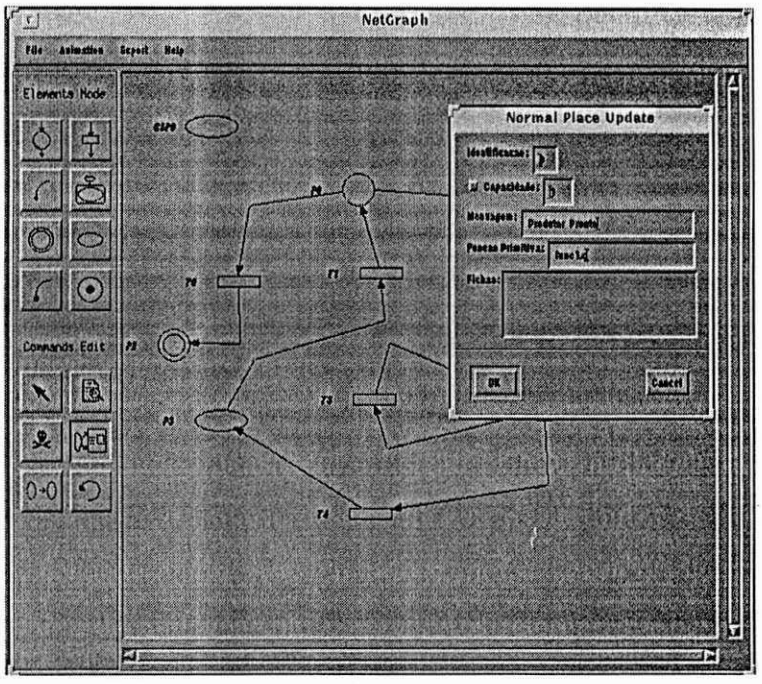

Figura A.8: Janela para alteração de atributos de Lugar Normal

Por questão de simplicidade, não detalharemos todas as janelas de atualização de atributos dos demais componentes da rede. Nas Figuras A.9, A.10, A.11, A.12 e A.13 exibem as caixas de diálogos para atualização de atributos de lugar alvo, transição, ficha, arco e isp, respectivamente.

No término dos trabalhos, o usuário deve salvar a rede editada através da opção SAVE (ou SAVE AS...) do menu File. A Figura A.14 mostra a tela para salvar a rede modelada com o nome *prod.net*.

Para encerrar os trabalhos no editor, o usuário deve selecionar a opção *Quit* do menu File, como mostrada na Figura A.15.

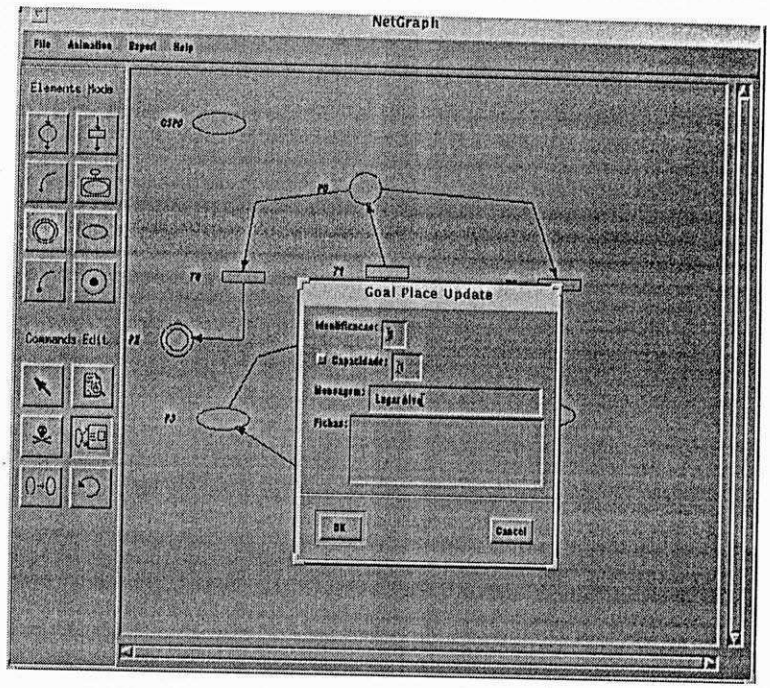

Figura A.9: Alteração dos atributos de Lugar Alvo

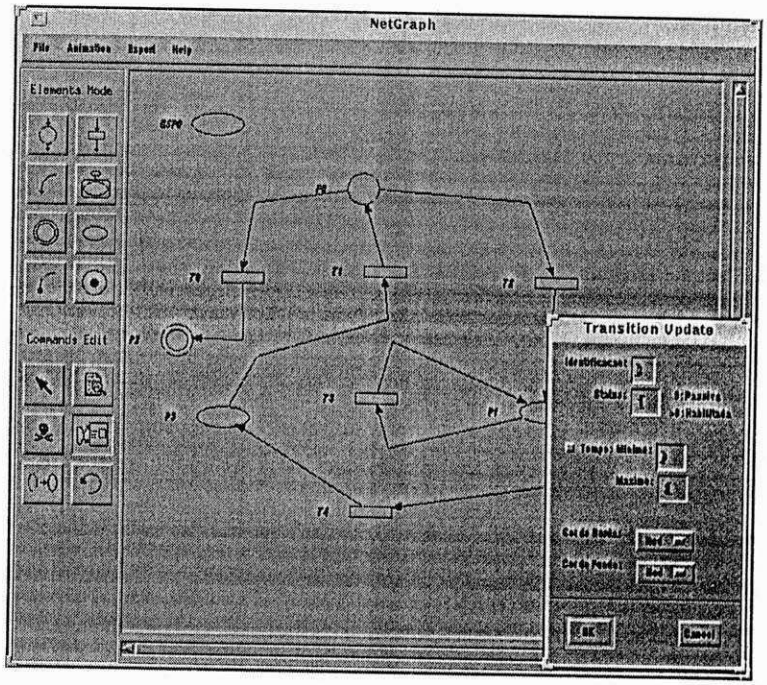

Figura A.10: Alteração dos atributos de Transição

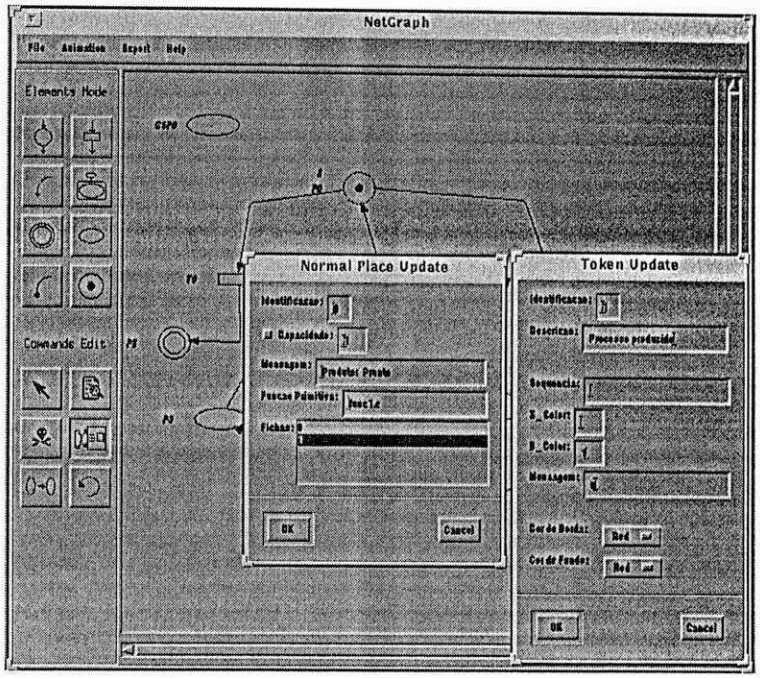

Figura A.11: Alteração dos atributos de Fichas

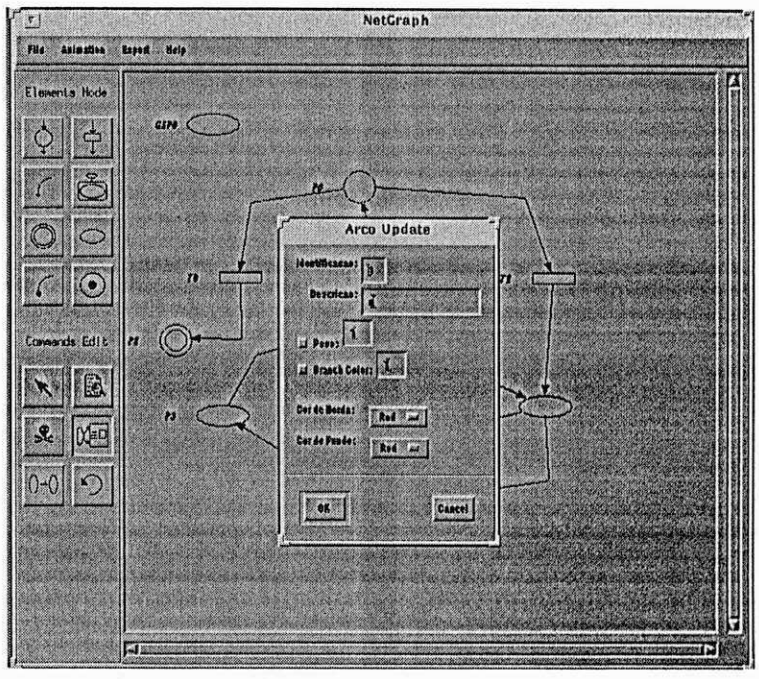

Figura A.12: Alteração dos atributos de Arcos

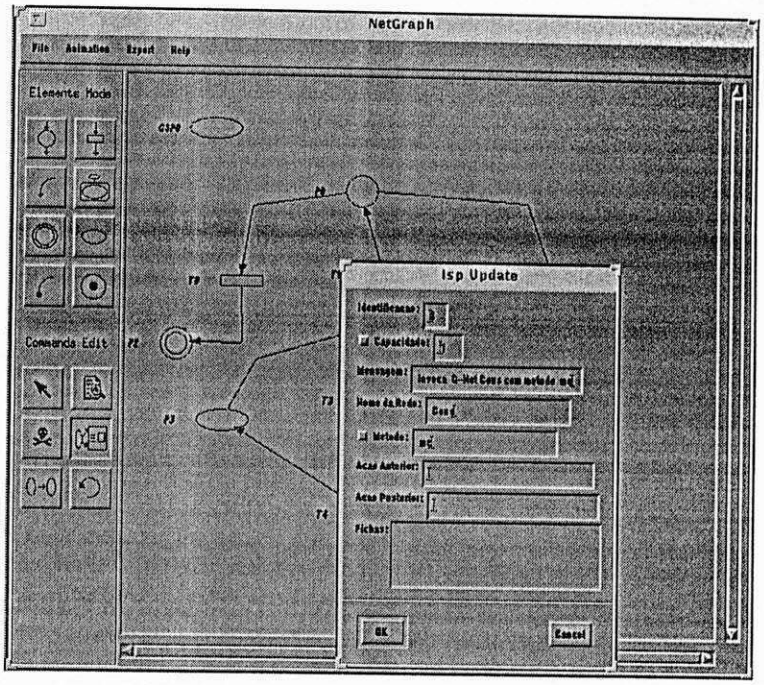

Figura A.13: Alteração dos atributos de Isp

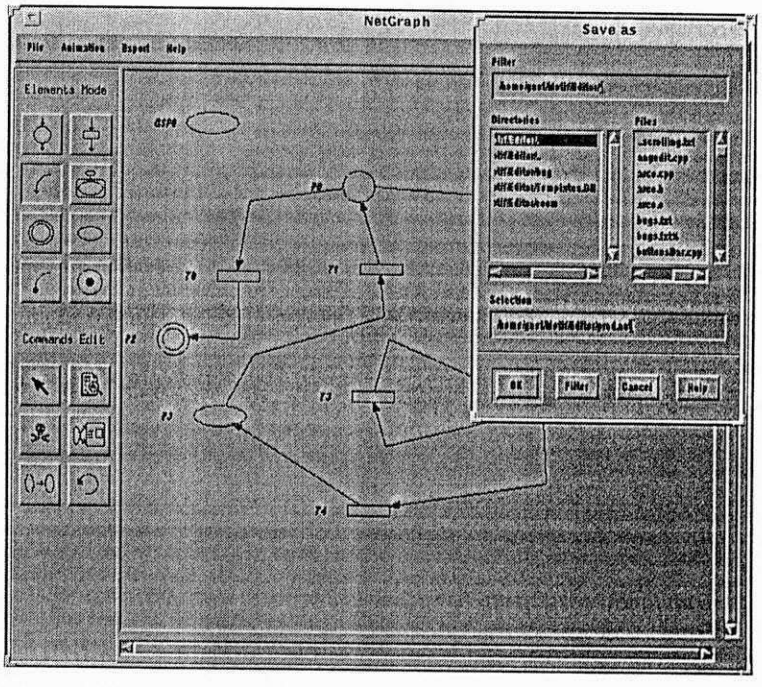

Figura A.14: Tela que exibe o salvamento da rede modelada

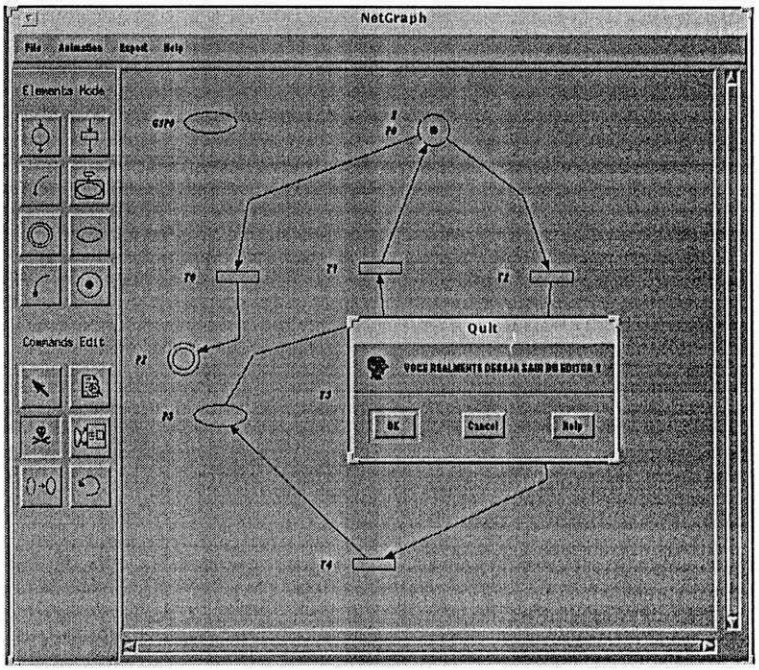

Figura A.15: Encerra trabalhos no *NetGraph*