Instituto de Computação
Universidade Estadual de Campinas

Smart: um Editor Gráfico de Diagramas em Xchart

Osvaldo Severino Junior

Dezembro 1996

Banca Examinadora:

- Cecília Mary Fischer Rubira
  Instituto de Computação - UNICAMP
- Eliane Martins (Suplente)
  Instituto de Computação - UNICAMP
- Hans Kurt Edmund Liesenberg (Orientador)
  Instituto de Computação - UNICAMP
- Luiz Eduardo Buzato (Co-orientador)
  Instituto de Computação - UNICAMP
- Maria da Graça Campos Pimentel
  Instituto de Computação - USP São Carlos

Dissertação submetida ao Instituto de Computação da
Universidade Estadual de Campinas, como requisito parcial para
a obtenção do título de Mestre em Ciência da Computação

UNICAMP
BIBLIOTECA CENTRAL

UNIDADE BC
Nº CHAMADA
T/UNICAMP
S673s
V. Ex.
TOMBO BC/ 30174
PROC. 281/97
C [ ] D [X]
PREÇO R$ 11,00
DATA 15/05/97
Nº CPD

CM-00098153-0

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

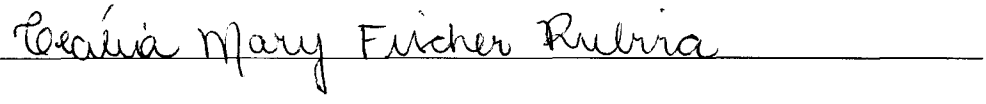
Severino Junior,Osvaldo
S83s Smart: um editor gráfico de diagramas em Xchart/ Osvaldo Severino Junior. -- Campinas, [SP : s.n.], 1996.

Orientador : Hans Kurt Edmund Liesenberg, Luiz Eduardo Buzato
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

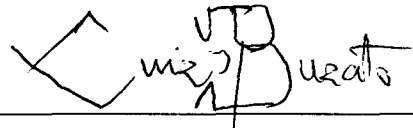
1.C++(Linguagem de programaçãode computador). 2. Interface (Computador). 3. Software - Desenvolvimento. 4. Grafico por computador . I. Liesenberg, Hans Kurt Edmund II. II. Universidade Estadual de Campanas. III. Titulo.

Tese de Mestrado defendida e aprovada em 18 de dezembro de 1996 pela Banca Examinadora composta pelos Professores Doutores

Profª. Drª. Maria da Graça Campos Pimentel



Profª. Drª. Cecília Mary Fischer Rubira

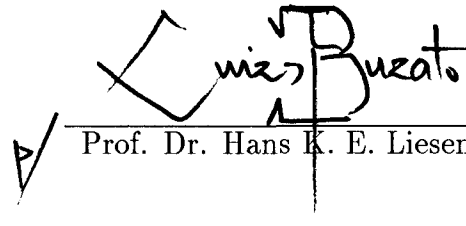


Profº. Drº. Luiz Eduardo Buzato

Smart: um Editor Gráfico de Diagramas em Xchart

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. Osvaldo Severino Junior e aprovada pela Comissão Julgadora

Campinas, 20 de dezembro de 1996



Prof. Dr. Hans K. E. Liesenberg

Dissertação apresentada ao Instituto de Computação - UNICAMP, como requisito parcial para obtenção do Título de **Mestre** em **Ciência da Computação**

Resumo

Smart é um editor gráfico desenvolvido para o ambiente **Xchart**[8]. **Xchart** é um ambietne de programação que provê ferramentas para o projeto e implementação de sistemas interativos. Sistemas Interativos [12] são compostos por três subsistemas: (i) apresentação, (ii) diálogo e (iii) aplicação. A apresentação é responsável pela aparência do sistema interativo, organizando os aspectos visuais e físicos de uma interaface homem-computador(ícones, janelas, etc.). A aplicação é o componente responsável pelo controle da comunicação entre a apresentação e a aplicação de um sistema interativo. *Xchart* focaliza ferramentas para o projeto e implementação do subsistema diálogo. *Smart* é o resultado de um projeto e implementação de uma interface homem-computador que adere aos métodos desenvolvidos no projeto Xchart.*Smart* permite a manipulação (captura, edição, leiaute automático e geração de código) de programas escritos na linguagem visual *Xchart*. Esta linguagem permite a descrição do comportamento do diálogo de um sistema interativo. O comportamento do diálogo define o mapeamento de eventos de entrada em ações derivadas por estes eventos.

O editor utiliza uma árvore de estados e um grafo de transições sobreposto à árvore para capturar e editar um *Xchart*. A árvore é o resultado da adaptação e implementação do algoritmo de leiaute automático para traçado de árvores. Após a captura e edição de um *Xchart*, o *Smart* permite a geração da forma usual de um *Xchart*, um grafo. A visualização da forma usual de um *Xchart* é o resultado da adaptação e implementação do algoritmo de leiaute automático para traçado de diagramas. A edição de um *Xchart*, também, permite ao *Smart* invocar o compilador de *Xchart* que gerará o respectivo código do *Xchart*.

Abstract

Xchart is a programming environment that provides tools for the design and implementation of interactive systems[8]. Interactive systems[12] have three main components: (i) presentation, (ii) dialog, and (iii) application. The *presentation* is responsible for the look and feel of an interactive system, it organizes the visual and physical aspects of the human-computer interface (icons, windows, device drivers, etc). The *application* is the component that implements the funcionality of the interactive system. The *dialog* component is responsible for the mediation of the communication between presentation and application. Xchart's focus is on tools for the design and implementation of *dialog* components. Smart is Xchart's tool for the editing (capturing, edition, automatic layout, and code generation) dialog specifications written in the visual language Xchart. The visual language Xchart is based on states and transitions.

Smart has an object-oriented design and implementation. One of its main characteristics is the use of algorithms for the automatic layout of diagrams. A dialog specification written in the language Xchart can be captured and automatically drawn as a tree and as a graph. Smart also provides a Xchart compiler, that is, Xchart dialog specifications are captured as diagrams and then translated into a textual language for use by other tools of the Xchart environment. This characteristic of Smart eases significantly the design and implementation of dialogs.

Agradecimentos

Especialmente ao Prof. Dr. Luiz Buzato que me auxiliou para a conclusão da dissertação.

À minha esposa Ciomara que me apoiou e soube compreender-me nos momentos mais difíceis da dissertação.

Aos meus pais, Osvaldo e Aparecida, pelo apoio que me prestaram e aos meus irmãos, especialmente ao Carlos que sempre me ajudou nos momentos de dificuldade.

Ao Instituto de Computação da Unicamp e ao Prof. Dr. Hans que me aceitou como orientando.

A colaboração e a ajuda do meu amigo Fábio Lucena e a todos que direta ou indiretamente contribuíram de certa forma para a realização desta.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), processo n^o 134160/94 − 0, e à Fundação de Amparo à Pesquisa do Estado de São Paulo (Fapesp), processo n^o 94/4176 − 3, pelo apoio financeiro.

Conteúdo

Lista de Figuras

Introdução

Esta dissertação descreve o projeto e implementação de um editor gráfico para o ambiente **Xchart**[8] denominado *Smart*, que permite a manipulação (captura, edição, leiaute automático e geração de código) de programas escritos na linguagem visual *Xchart*. Esta linguagem permite a descrição do comportamento de um sistema interativo.

Por quê Desenvolver Smart?

Smart foi desenvolvido devido:

1. À inexistência de editores gráficos que pudessem ser adaptados às necessidades de *Xchart*;

2. Às necessidades de construção de uma interface de programação amigável e de alto nível para o ambiente de programação **Xchart**.

Editores gráficos convencionais poderiam ter sido utilizados para a captura de *Xcharts*, se fossem facilmente adaptáveis para auxiliar o projetista na representação de construções intrínsecas ao *Xchart*. Em geral, os editores de diagramas de linguagens visuais existentes são de difícil adaptação porque possuem um acoplamento muito forte entre a sua interface (representação do diagrama) e as suas funções de edição. Assim, um editor gráfico de cunho geral dificilmente poderia ser modificado para representar automaticamente os diagramas hierárquicos existentes em *Xchart*.

Smart foi implementado de forma a permitir adaptações, para isto tem a sua arquitetura de software baseada no modelo de Seeheim [12]. Esta arquitetura torna mais fraco o acoplamento entre a interface e o editor.

A disponibilização de um editor gráfico para *Xchart* traz benefícios para o usuário projetar e implementar uma interface, pois as atividades de manipulação de especificações em *Xchart* tornam-se muito mais eficientes e rápidas. O esforço de geração e alteração das especificações é reduzido e muitas tarefas suscetíveis a erros, tais como a verificação da consistência de uma especificação, são automatizadas.

Contribuições

Esta dissertação contribui com:

1. Desenvolvimento de uma interface de programação amigável e de alto nível para o ambiente **Xchart**. A interface de programação é materializada pelo editor gráfico *Smart*, isto é, a especificação do controle do *diálogo* homem-computador é realizada através do *Smart*;

2. Adaptação e implementação do algoritmo de leiaute automático para traçado de árvores, proposto por Moen [17] e do algoritmo de leiaute automático para traçado de diagramas, proposto por Sugiyama [7];

3. Projeto e implemantação de uma interface homem-computador para *Smart* que adere aos métodos desenvolvidos no projeto Xchart. *Smart* é um exemplo de uso da metodologia de desenvolvimento de interfaces proposta pelo projeto Xchart. Neste sentido, o projeto e implementação serviram como um experimento na área de construção de interfaces homem-computador.

Organização da Dissertação

Esta dissertação está estruturada em seis capítulos. O Capítulo 1 define sistema interativo e termos associados a um sistema interativo, procurando facilitar o entendimento da arquitetura de software do *Smart*. O Capítulo 2 introduz o ambiente **Xchart** e descreve a arquitetura do *Smart*. O Capítulo 3 descreve a interface do *Smart*. O Capítulo 4 descreve *Tree*, o componente de *Smart* responsável pelo traçado de *Xcharts* como árvores. O Capítulo 5 descreve *Xgraph*, o componente de *Smart* responsável pelo traçado de *Xcharts* como grafos. Finalmente o Capítulo 6 traz as conclusões do trabalho, críticas, contribuições e sugestões para pesquisas futuras.

Capítulo 1

Sistemas Interativos

Este Capítulo introduz alguns fundamentos sobre sistemas interativos, com o objetivo de permitir que a arquitetura de *software* adotada por *Smart* seja compreendida com maior clareza.

1.1 Definições Preliminares

Uma *interação* implica um relacionamento binário, isto é, uma entidade age sobre outra e vice-versa. Ao interagir, estas entidades podem estabelecer um padrão de comunicação, de forma a tornar mais efetiva a interação. É possível estender o conceito de interação para um relacionamento entre um grupo de entidades, mas neste texto, restringiremos as definições a conceitos associados a interações binárias.

Cada um dos agentes de uma interação é um *sistema interativo.* Quando um dos componentes do sistema interativo é um humano e o outro é um computador, temos uma *interação homem-computador.* Assim, uma interação homem-computador é definida pela composição de ações físicas, conceituais e lingüísticas entre um usuário humano e um computador, com o objetivo de executar uma tarefa. Toda interação precisa ser apoiada por componentes que traduzem as ações entre os sistemas interativos, o papel de tradução (interpretação) de ações fica a cabo de uma interface.

Uma *interface* é um plano, ou meio, entre dois sistemas, constituindo a sua fronteira comum. De outra forma, uma interface é uma conexão entre dois ou mais objetos, agentes ou sistemas. Esta conexão, ou fronteira comum, atua como mediador entre os sistemas conectados.

Finalmente, uma interação homem-computador é a superfície física e as facilidades que provêem o meio através do qual o homem e o computador podem se conectar e interagir. A interação homem-computador provê os meios físicos (visuais, sonoros e táteis) e métodos que geram a interação homem-computador, por simplicidade interface.

O projeto de interfaces engloba muitas áreas do conhecimento humano (ergonomia, fatores humanos, psicologia, etc.) que estão fora do escopo desta dissertação. Apesar da interface do *Smart* procurar atender a vários requisitos levantados por especialistas em interfaces, limitaremo-nos a discutir os aspectos relacionados à sua arquitetura.

Uma arquitetura de *software* especifica o arranjo e as interfaces dos sistemas que a compõem. A arquitetura a ser adotada no projeto do *Smart* foi proposta por Seeheim [12]. A seção seguinte apresenta o modelo de Seeheim em detalhe, frisando a sua importância para o projeto de sistemas interativos e, em particular, para o projeto do *Smart*.

1.2 Modelo de Seeheim

O modelo de Seeheim[12] define um sistema interativo composto por três subsistemas: apresentação, diálogo e aplicação (Figura 1.1). A *apresentação* é responsável pela aparência externa do sistema, definindo o arranjo de elementos visuais, sonoros e táteis para capturar e transmitir o comportamento dos sistemas reativos. O *diálogo* permite uma comunicação controlada entre a apresentação e a aplicação. O diálogo dispara ações da aplicação baseadas na geração de eventos pela apresentação. As ações disparadas pelo diálogo dependem do evento ocorrido e do estado corrente dos três subsistemas. Desta forma, o diálogo define qual a seqüência de eventos, gerados pela apresentação, que deverão ou não ser considerados pela aplicação. A *aplicação* implementa a funcionalidade do sistema, determinando todas as ações a serem disparadas pelo sistema; em comparação com um compilador, a apresentação é o analisador léxico, o diálogo é o analisador sintático e a aplicação é o tradutor/gerador de código.

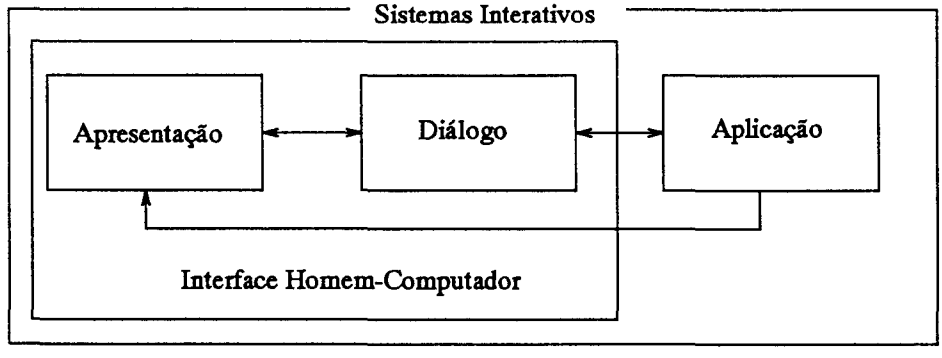


Figura 1.1: O Modelo de Seeheim[12]

1.2.1 Apresentação

A apresentação é o componente responsável pela aparência da aplicação. Este componente inclui rotinas de entrada/saída específicas, o arranjo do "display" (windows, menus, ícones, os padrões de cores, etc.) e algumas tarefas básicas de gerência de periféricos de entrada/saída. A apresentação traduz eventos gerados pela interação do usuário com o computador em *tokens* que serão processados pelo componente de diálogo e em eventos gerados pelo componente de diálogo para o usuário.

A importância da apresentação deve-se ao fato da eficiência do usuário, no uso do sistema interativo, ser fortemente influenciada pela sua funcionalidade. Uma apresentação mal construída pode restringir inadequadamente a funcionalidade do sistema interativo e, conseqüentemente, empobrecer a interação entre o usuário e o sistema computacional, causando um impacto direto na capacidade do usuário em executar tarefas previstas.

Uma das dificuldades no projeto de uma apresentação reside no fato de não haver normas que transcrevam os passos para a obtenção de uma aparência mais "agradável" e funcional. Hohl[9] identificou alguns aspectos a serem considerados durante o projeto do componente de apresentação:

- **Estrutura.** A estrutura estabelece a organização dos objetos de interação na apresentação. Um objeto de interação pode ser composto por outros objetos de interação, formando uma hierarquia. Por exemplo, um barra de menu, que é um objeto de interação, por sua vez pode ser constituída por outros objetos de interação, representados pelas opções dispostas na barra de menu;

- **Geometria e Leiaute.** A geometria de um objeto de interação determina o seu tamanho e forma. O leiaute de um objeto de interação determina a sua posição. O leiaute depende da geometria do objeto e da geometria dos seus componentes. Por exemplo, os componentes de uma barra de menu podem estar dispostos verticalmente ou horizontalmente na tela, dependendo das regras de leiaute definidas para a apresentação;

- **Aparência.** A aparência de um objeto de interação é determinada por atributos como a sua cor, fonte de texto, etc.

Para a definição da apresentação de um sistema interativo é comum a utilização de ferramentas denominadas *toolkits* que permitem a definição da estrutura, geometria e leiaute dos objetos de interação e estabelecem a aparência destes objetos. Exemplos de *toolkits* são *WxWindows*[1], *Tcl/tk*[2], etc.

[1] *Toolkit* de domínio público obtido em ftp www.aiai.ed.ac.uk/~jacs/vxwin.html
[2] *Toolkit* de domínio obtido em ftp cciwww.unige.ch/eao/www/Tcltk.html

1.2.2 Diálogo

Este componente é responsável pela interação entre os componentes de apresentação e a aplicação e vice-versa. O diálogo é responsável pela implementação dos estilos de interação (conversacional, icônico, baseado em templates, etc). Vários modelos foram desenvolvidos para facilitar a especificação do comportamento de um diálogo. As 3 principais classes de modelos são: baseados em sintaxe (lingüístico), baseados em eventos e baseados em diagramas de estados. Após estudos comparativos, o projeto Xchart havia optado pela utilização de *Statecharts* [14]; uma linguagem de especificação baseada no modelo de diagramas de estados. Experiências com *Statecharts* [5] motivaram a sua extensão, adequando-a à especificação de diálogos de sistemas interativos. Como resultado, uma nova linguagem foi criada e denominada *Xchart*[3][8]. Harel [15] afirma que o uso de linguagens como *Xchart* têm produzido resultados positivos na construção de sistemas interativos.

O componente diálogo define a reatividade [14] que descreve os objetos de interação definido pelo componente de apresentação do sistema. O comportamento é descrito por eventos que disparam ações. Os eventos podem ser representados por entradas pelo teclado, pelo mouse e por outros periféricos. As ações disparadas podem afetar a todos ou a somente alguns dos três subsistemas que compõem a arquitetura de um sistema interativo. Por exemplo, cada opção de uma barra de menu pode ser acionada, por uma tecla ou um clique no botão do mouse, e disparar uma ação. Esta ação pode invocar uma função do diálogo ou ser interpretada pelo próprio componente de apresentação.

1.3 O Exemplo: um Cronômetro

Para ilustrar o modelo de Seeheim descrevemos um sistema interativo definido por um cronômetro. Este exemplo foi escolhido pelo projeto Xchart porque é simples o bastante para ser tratado em um texto como este e complexo o bastante para ilustrar o uso de *Xchart* e de *Smart*. Ao longo da dissertação, utilizaremos este exemplo para descrever o funcionamento do editor gráfico.

1.3.1 Aplicação

O cronômetro apresenta cinco botões e um visor que determinam a funcionalidade do cronômetro. Os botões são:

1. **On**. A função do botão **On** é ligar o cronômetro (*Power On*);
2. **Off**. A função do botão **Off** é desligar o cronômetro (*Power Off*);

[3]Por curiosidade: a 22ª letra do alfabeto grego (X é χ) é identificada em inglês pela palavra CHI, que coincide com o acrônimo de *Computer-Human Interface*.

3. **Timer**. A função do botão **Timer** é disparar e parar o tempo de cronometragem;

4. **Reset**. A função do botão **Reset** é zerar o tempo de cronometragem. A função do visor é mostrar o tempo de cronometragem;

5. **Display**. A função do botão **Display** é permutar entre os modos de saída do visor: normal ou em segundos.

Os botões definem estados e os estados podem serem compostos por subestados. O botão **Timer** define o estado TIMER. O estado TIMER é composto pelos subestados ON e OFF que disparam e param o tempo de cronometragem. O botão **Display** define o estado DISPLAY. O estado DISPLAY é composto pelos subestados NORMAL e INSEC que mostram o tempo normal e em segundos respectivamente. O botão **Reset** define o estado RESET que zera o tempo de cronometragem.

1.3.2 Apresentação

A apresentação do cronômetro é composta por um quadrado, que define o visor, inscrito em um outro quadrado com cantos arredondados que representa os botões. Os botões **Timer**, **Display**, **On** e **Off** são representados por semicírculos traçados nos cantos arredondados do cronômetro, enquanto o botão **Reset** é traçado ao lado esquerdo do cronômetro. A Figura 1.2 ilustra a apresentação do cronômetro.

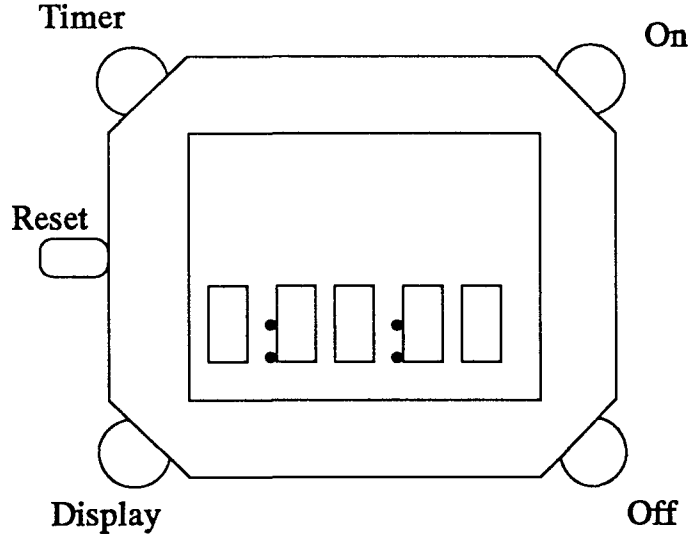


Figura 1.2: Apresentação do cronômetro

1.4 Diálogo

Na Figura 1.2 o ato de pressionar um botão do cronômetro caracteriza a ocorrência de um evento. Portanto, os eventos atribuídos à apresentação são:

- $e1$ - pressionar botão **Off**;
- $e2$ - pressionar botão **Timer**;
- $e3$ - pressionar botão **Display**;
- $e4$ - pressionar botão **Reset**;
- $e5$ - pressionar botão **On**.

De acordo com a funcionalidade dos botões que são apresentados pelos estados $e1$ a $e5$ pode-se estabelecer o comportamento do cronômetro que descreve as transições entre os seus estados. A transição de um estado para o outro dependerá do evento ocorrido e do contexto em que se encontra o cronômetro. O contexto define os estados ativos do sistema quando da ocorrência de um evento. Na transição de um estado para outro são disparadas ações, que constituem funções associadas a cada botão. A seguir descreveremos o comportamento do cronômetro de forma textual, após apresentamos a descrição do cronômetro utilizando *Xchart* definindo os seus estados *default* e a relação de exclusão e concorrência entre os subestados de um estado.

Na Figura 1.2, o subestado *default* do estado TIMER é o estado ON, pois inicialmente o cronômetro deve ser disparado para poder cronometrar o tempo. Entretanto pressionando o botão **Timer** ocorre um evento que causa a parada do tempo de cronometragem, caracterizando a transição do subestado ON para o subestado OFF e vice-versa. O cronômetro pode apenas estar acionado ou parado em relação ao tempo de cronometragem, portanto o estado TIMER será um estado exclusivo, apenas um dos seus subestados ON ou OFF estará ativo.

O subestado *default* do estado DISPLAY é o subestado Normal, isto significa que inicialmente o cronômetro mostra o tempo normal de cronometragem. Entretanto, pressionando o botão **Display** ocorre um evento que causa a mudança do visor para o modo InSec, caracterizando a transição do subestado NORMAL para o subestado INSEC e, vice-versa. O cronômetro pode apenas estar mostrando o tempo normal ou em segundos, portanto DISPLAY é um estado exclusivo, apenas um dos seus subestados NORMAL ou INSEC estará ativo. Tanto o estado DISPLAY como o estado TIMER são responsáveis pelo modo de operação do cronômetro, portanto os estados DISPLAY e TIMER compõe um outro estado denominado OPERATION. Após o disparo do tempo de cronometragem é possível alterar o modo normal para o modo em segundos para verificação do tempo de cronometragem,

portanto tanto o estado DISPLAY como o estado TIMER podem agir concorrentemente no cronômetro, caracterizando o estado OPERATION como um estado concorrente. Deste modo, quando o estado OPERATION for ativado tanto o estado DISPLAY como o estado TIMER serão ativados concorrentemente.

Caso o cronômetro esteja no modo de operação e o tempo de cronometragem já tenha sido disparado, pressionando o botão **Reset** o tempo de cronometragem será zerado, caracterizando uma transição do estado OPERATION para o estado RESET. Após o tempo de cronometragem ter sido zerado, para reinicializá-lo faz-se necessário pressionar o botão **Timer**, caracterizando uma transição do estado RESET para o estado OPERATION. Se, após ter sido zerado o tempo de cronometragem, o botão **Reset** for pressionado, o tempo de cronometragem será zerado novamente, caracterizando uma transição do estado RESET para ele mesmo. Tanto o estado RESET como o estado OPERATION efetuam atividades no cronômetro, portanto estes estados fazem parte do estado ativo do cronômetro denominado ALIVE, porém apenas um dos estados estará ativo, caracterizando o estado ALIVE como um estado exclusivo. Quando o cronômetro é ativado pela primeira vez, este estará zerado, caracterizando o estado RESET como o estado *default* do estado ALIVE, porém uma nova ativação do estado ALIVE deve ser efetuada a partir da última configuração do sistema; portanto o estado ALIVE deve permitir um mecanismo especial de ativação, o mecanismo de histórico composto.

DEAD é um estado caracterizado por não possuir atividade. Se o tempo de cronometragem estiver parado e o botão **Reset** for pressionado, então o cronômetro permanecerá sem atividade, caracterizando uma transição do estado ALIVE para o estado DEAD. Se o cronômetro não estiver efetuando nenhuma atividade e o botão **Timer** for pressionado, o tempo de cronometragem será disparado, caracterizando uma transição do estado DEAD para o estado ALIVE. Se o cronômetro não estiver em atividade e o botão **Off** for pressionado, então o cronômetro será desligado, caracterizando uma transição do o estado DEAD para o estado mais externo do *Xchart*. Se o cronômetro for ligado pressionando o botão **On**, o cronômetro permanecerá ligado e aguardando para executar uma atividade, caracterizando uma transição do estado mais externo do *Xchart* para o estado DEAD. Portanto o *Xchart* mais externo que descreve o cronômetro será denominado STOPWATCH e será composto pelos estados DEAD e ALIVE. O estado ALIVE será o estado *default* para o STOPWATCH. O cronômetro pode estar em atividade ou parado, caracterizando STOPWATCH como um estado exclusivo.

Utilizando-se da notação de *Xchart*, é possível descrever o comportamento do cronômetro definido como *Stopwatch*. A Figura 1.3 ilustra a descrição do *Xchart Stopwatch*.

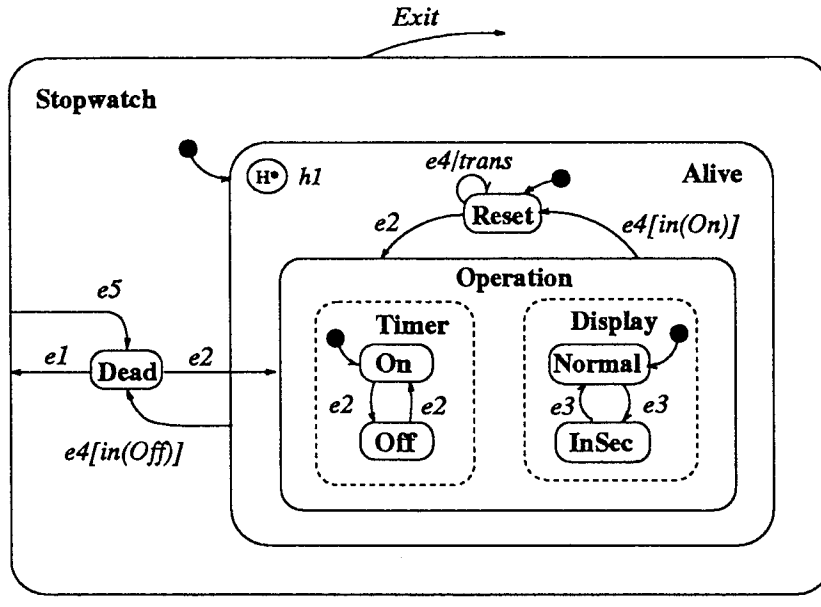


Figura 1.3: Exemplo de um Xchart

1.5 Resumo

Este Capítulo definiu o que é um sistema interativo e introduziu o exemplo do cronômetro para ilustrar a linguagem *Xchart*. Um sistema interativo é composto por três subsistemas: apresentação, diálogo e aplicação. A apresentação é responsável pela apresentação externa do sistema. O diálogo define qual a seqüência de eventos, gerados pela apresentação, que deverão ou não ser considerados pela aplicação. A aplicação determina todas as ações a serem disparadas pelo sistema. No próximo capítulo serão analisadas as arquiteturas de *software* do ambiente **Xchart** e do *Smart*.

Capítulo 2

Smart: um Editor Gráfico

Este Capítulo descreve as arquiteturas de software do ambiente **Xchart** e do editor gráfico *Smart*. A descrição das arquiteturas procede da descrição do ambiente para a descrição do editor com o objetivo de fornecer: (i) uma visão geral do ambiente *Xchart* e (ii) uma visão detalhada do editor gráfico *Smart*.

2.1 Arquitetura do Ambiente Xchart

O ambiente **Xchart** é constituído por uma série de ferramentas que auxiliam o projeto e desenvolvimento de sistemas interativos, com ênfase no desenvolvimento do componente de diálogo.

A construção de uma interface no ambiente **Xchart** implica o projeto e implementação dos componentes de apresentação e diálogo de uma interface. A Figura 2.1 mostra uma representação da arquitetura de *software* do ambiente **Xchart**, seus componentes são explicados a seguir.

O componente de apresentação é criado com o auxílio de uma ferramenta chamada *Grace*. *Grace* é uma ferramenta sendo desenvolvida no projeto Xchart que possuirá um editor gráfico e terá a sua implementação baseada em um *toolkit* de geração de apresentações. *Grace* permitirá a definição da aparência da interface e gerará o código C++ que a implemente.

O componente de diálogo é produzido com o auxílio de *Smart*. *Smart* é utilizado para manipular representações de *Xcharts* que capturam o comportamento do diálogo da interface. *Smart* traduz as representações diagramáticas capturadas durante o seu uso para uma linguagem arquiteturalmente neutra denominada *TeXchart* (Textual *Xchart*). Finalmente o compilador de *TeXchart* traduz esta representação intermediária do *Xchart* para C++.

Os códigos gerados pelas ferramentas *Grace* e *Smart* são acoplados ao código das

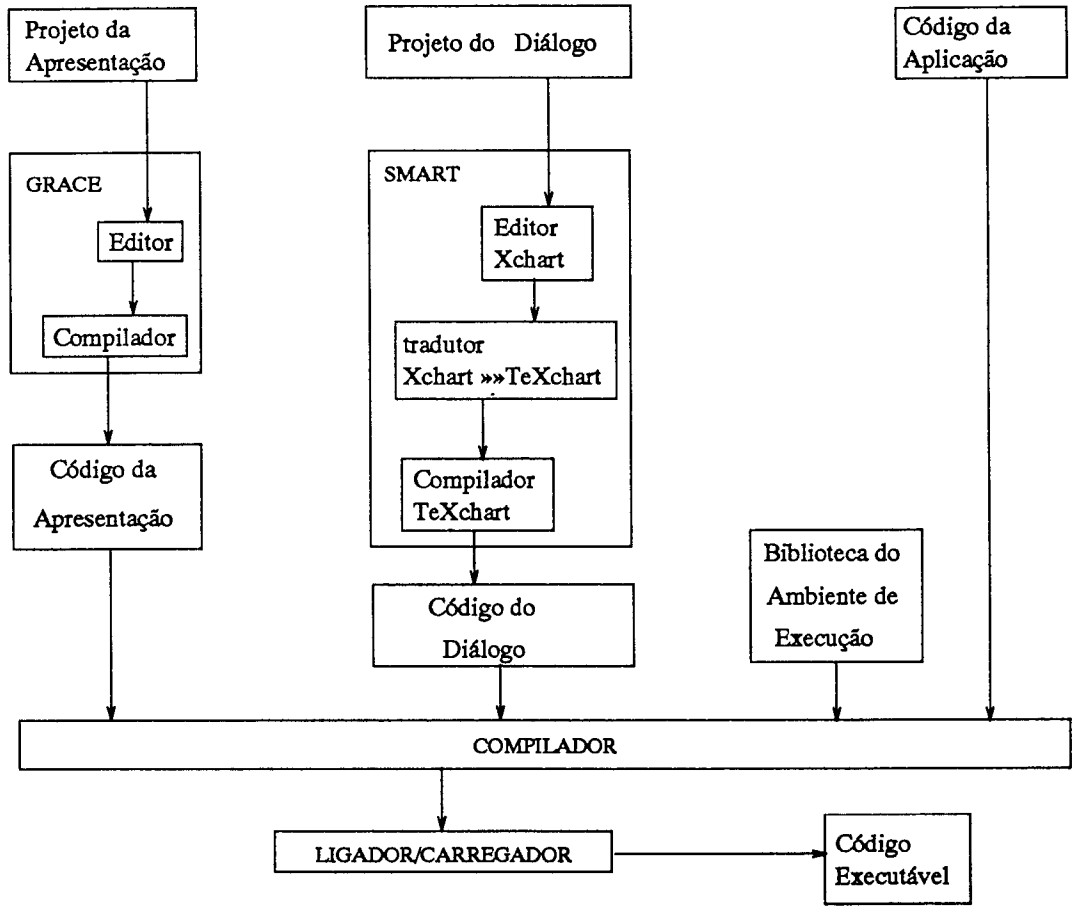


Figura 2.1: Arquitetura do ambiente Xchart

bibliotecas do ambiente de execução de *Xchart* e ao código da aplicação, gerando o código do sistema interativo desejado. A biblioteca de objetos do ambiente *Xchart* contém o código do sistema de controle de comunicação, distribuição e ações atômicas distribuídas necessárias para o código executável do sistema interativo funcionar como projetado.

2.1.1 O Ambiente de Execução de Xcharts

O ambiente, ou sistema, de execução de um *Xchart* é o componente responsável pela coordenação da interação entre os objetos que compõem o diálogo de uma interface. A apresentação, diálogo e aplicação podem ser distribuídos e, neste caso, o ambiente de execução provê os mecanismos necessários ao controle da execução distribuída (resolução de nomes, comunicação, ativações remota, etc). A Figura 2.2 ilustra o ambiente de execução de um *Xchart*, e seus componentes são descritos a seguir.

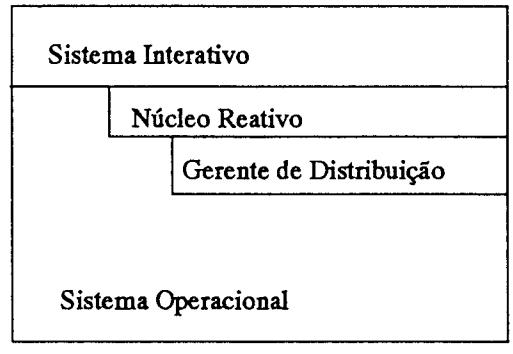


Figura 2.2: O Ambiente de execução de um Xchart

Núcleo Reativo

O núcleo reativo é fundamentalmente um interpretador de *Xcharts*. Durante a execução do sistema interativo(aplicação), o núcleo reativo mantém e executa o conjunto de *Xcharts* responsável pelo comportamento do componente de diálogo da interface do sistema interativo.

Eventos são gerados pelo usuário do sistema interativo e capturados pela apresentação do sistema interativo e vice-versa, isto é, eventos gerados pela execução de ações da aplicação/diálogo geram ações que são executadas na apresentação.

Gerente de Distribuição

O gerente de distribuição, permite, quando for o caso, a execução concorrente e distribuída das tarefas realizadas pelo núcleo reativo, utilizando serviços fornecidos por um sistema de ações atômicas distribuídas [3] e, por um sistema de comunicação em grupo [2].

Sistema Operacional

O ambiente de programação de interfaces *Xchart* está sendo constituído sobre *Windows NT* 3.5.1.

2.2 Arquitetura do Smart

Uma especificação em *Xchart* é descrita na forma de um grafo de *Statecharts* [13]. Porém, analogamente à forma de grafo, uma especificação em *Xchart* pode ser descrita através de uma estrutura hierárquica e um grafo de transições sobreposto à estrutura [18]. A estrutura hierárquica é representada por uma estrutura do tipo árvore que descreve a hierarquia dos estados e subestados do *Xchart*. O grafo de transições sobreposto à árvore representa as transições de estados. O grafo de transições complementa a árvore, em termos de captura de uma descrição de comportamento, e ambos definem o mesmo tipo de informação.

Em termos topológicos, as duas formas alternativas representam o mesmo tipo de informação. Em termos de aspecto visual, contudo, a notação de grafo é muito mais ilustrativa, enquanto que em termos de operações, como por exemplo inserção e remoção de estados e transições, a árvore e o grafo de transição são mais simples de serem manipulados e gerenciados e, também, possibilitam a verificação da profundidade e da largura da hierarquia de estados.

O *Smart* utiliza as duas formas de representação de um *Xchart*. Da forma de grafo, o *Smart* beneficia-se do aspecto visual. Da forma de árvore e grafo de transições, o *Smart* beneficia-se da simplicidade de manipulação e gerenciamento dos componentes de um *Xchart*. O usuário manipulará uma árvore e definirá um grafo de transições, sobreposto à árvore, para editar um *Xchart*. Após a edição do *Xchart*, o usuário, também poderá gerar a representação em forma de grafo do *Xchart* editado.

O *Smart* é um sistema interativo, portanto possui uma interface e uma aplicação. O *Smart* é composto pelos componentes: interface, estrutura de dados árvore, estrutura de dados grafo, estrutura de dados neutra, tradutor de *TeXchart* e compilador de *TeXchart*. A Figura 2.3 ilustra os componentes que compõe a arquitetura do *Smart*.

A interface permite editar e visualizar o *Xchart* representado pela estrutura de dados árvore e apenas visualizar o *Xchart* representado pela estrutura de dados grafo que representa a forma de estados "encaixados" de *Xchart*.

A estrutura neutra gerada pelo *Smart* e, é composta pela hierarquia dos estados e as suas respectivas transições. Ela comporta todos os atributos de um estado descrito na linguagem *Xchart*. Estes atributos são: identificador do estado, o estado pai, o estado filho, tipo de estado(seqüencial ou concorrente) e especificação de mecanismo de histórico.

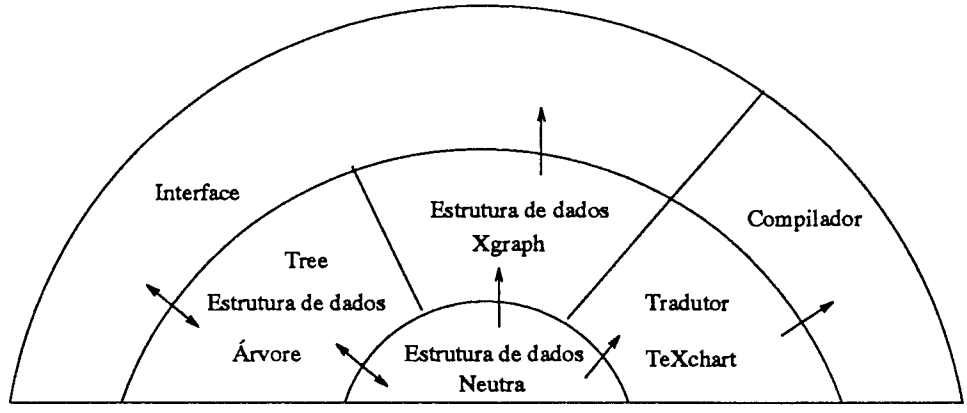


Figura 2.3: Arquitetura do editor gráfico Smart

A estrutura de dados árvore, denominada *Tree*, é a representação de um *Xchart* na forma de árvore e um grafo de transições sobreposto à árvore. A estrutura de dados grafo, denominada *Xgraph*, é a representação da estrutura neutra em forma grafo. O compilador de *Xchart* é responsável pela geração de código a partir da estrutura neutra e possui, como entrada, uma linguagem textual denominada *TeXchart*. O *Smart* possui um tradutor que mapeia a representação gráfica do *Xchart* em código *TeXchart*.

2.2.1 Tree

Tree é a representação da estrutura neutra com o objetivo de alocar os estados, representados pelos nós na árvore, tão próximos quanto possível. Para alcançar este objetivo, a árvore possui mais informações que a estrutura neutra de forma, que ao percorrer a árvore, em profundidade ou em largura, gera-se a estrutura neutra. A árvore também permite a criação, remoção, manipulação e gerenciamento dos estados que constituem o *Xchart* a ser especificado. A solução, proposta para construir a árvore de estados, é derivada de Moen [17] que apresentou um algoritmo para desenhar dinamicamente árvores que apresentem nós com tamanho e contorno variados e que sejam tão compactos quanto possível. O grafo de transições, sobreposto à árvore, consiste de uma lista de adjacências. O Capítulo 4 descreve a solução proposta para *Tree*.

2.2.2 Xgraph

Xgraph permite, apenas, a visualização da estrutura neutra na forma de estados "encaixados". As alterações e manipulação de estados são efetuadas apenas por *Tree*. A solução proposta para traçar os estados nesta forma é derivada de Sugiyama [7], que efetua o traçado de uma árvore de estados e um grafo de transições, análogo a um *Xchart* descrito

na forma de estados "encaixados". O Capítulo 5 descreve a solução proposta para *Xgraph*.

2.2.3 Geração de Código: TeXchart

O tradutor de *TeXchart* é um componente do *Smart* que gera uma descrição textual do *Xchart* descrito pelo *Smart*. Para gerar a descrição textual, o tradutor necessita da estrutura neutra gerada pelo *Smart* e de declarações dos componentes da linguagem *Xchart*. Estas declarações são capturadas pelo *Smart* através de um módulo de edição textual, denominado *TextSmart*.

O *TextSmart* é definido por uma janela que permite a edição de texto. A política de geração de um arquivo texto foi adotada, pelo grupo responsável pelo projeto Xchart, pelo fato dos componentes *Smart* e o compilador *Xchart* estarem sendo desenvolvidos simultaneamente, assim como outros componentes responsáveis pelas primitivas de comunicação de mais baixo nível.

Após a integração de todos os componentes do ambiente **Xchart**, a ser realizada pelo grupo de integração do projeto, o acesso ao compilador pelo *Smart* será efetuado via memória, eliminando este arquivo texto.

2.3 Resumo

Este Capítulo apresentou a arquitetura do ambiente **Xchart** e do *Smart*. O ambiente **Xchart** constitui-se de uma série de ferramentas que auxiliam o projeto e desenvolvimento de sistemas interativos, com ênfase no desenvolvimento do componente de diálogo. A construção de uma interface no ambiente **Xchart** implica o projeto e implementação dos componentes de apresentação e diálogo de uma interface. A apresentação é criada com o auxílio de uma ferramenta chamada *Grace*. *Grace* permitirá a definição da aparência da interface e gerará o código C++ que a implemente. O diálogo é produzido com o auxílio de *Smart*. O *Smart* utiliza as duas formas de representação de um *Xchart*: a árvore ou o grafo. Da forma de grafo, o *Smart* beneficia-se do aspecto visual. Da forma de árvore, o *Smart* beneficia-se da simplicidade de manipulação e gerenciamento dos componentes de um *Xchart*. *Smart* traduz as representações diagramáticas capturadas durante o seu uso para uma linguagem arquiteturalmente neutra denominada *TeXchart* (Textual *Xchart*). Finalmente, o compilador de *TeXchart* traduz esta representação intermediária do *Xchart* para C++. O *Smart* é composto pelos componentes: interface, estrutura de dados árvore, estrutura de dados grafo, estrutura de dados neutra, tradutor de *TeXchart* e compilador de *TeXchart*. A estrutura neutra do *Smart* é composta pela hierarquia dos estados e as respectivas transições entre os mesmos.

Capítulo 3

A Apresentação de Smart

Este Capítulo descreve o componente de apresentação do *Smart*, que define a aparênica visual do *Smart*. A aparência de uma interface homem-computador é definida em função de tarefas representativas de uso da aplicação para a qual ela está sendo desenvolvida. Assim, iniciaremos a descrição da *apresentação* do *Smart* analisando tarefas que usuários do ambiente *Xchart* realizariam para editar um *Xchart*. O primeiro passo para o desenvolvimento da apresentação de um sistema interativo é a identificação dos objetos de interação que compõe a interface do sistema. Para identificar estes objetos, analisa-se as tarefas normalmente executadas pelo sistema. Após a identificação destes objetos, são estabelecidas as suas características de acordo com a sua estrutura, geometria, leiaute e aparência e é definida a sua reatividade de acordo com o seu comportamento.

3.1 Análise de Tarefas

Para editar um *Xchart*, o usuário interage com o *Smart*, através de solicitações de tarefas. Numa fase preliminar, definiu-se que o *Smart* implementaria tarefas que permitiriam a edição, impressão, compilação e simulação de um *Xchart*. A simulação permite a análise de um *Xchart* em que seja necessária ligá-lo à aplicação ao qual se destina. A compilação gera um arquivo texto que descreve o *Xchart* capturado usando o *Smart*. As tarefas de edição permitem a captura e edição, com auxílio de funções de leiaute de diagramas, de um *Xchart* que pode ser visualizado como uma árvore e como um grafo. Neste contexto, o leiaute da árvore é efetuado automaticamente, pelas tarefas responsáveis pela manipulação da árvore, enquanto o leiaute do grafo é efetuado explicitamente, devido ao alto custo computacional desta tarefa. O usuário utilizará o *TextSmart*, em uma janela de edição textual, para definir características textuais de *Xchart* que serão acopladas ao *TeXchart*. A edição é a tarefa básica da interface do *Smart* e, devido a sua funcionalidade, ela é composta por subtarefas. A seguir enumera-se as subtarefas que permite a edição de um

Xchart no *Smart*:

1. **Novo Xchart.** Permite a especificação de um novo *Xchart*;
2. **Abra um Xchart.** Permite carregar, no editor, um *Xchart* salvo anteriormente;
3. **Feche um Xchart.** Finaliza uma descrição de um *Xchart* salvando-a num dispositivo determinado pelo usuário;
4. **Salve um Xchart.** Armazena um *Xchart* num dispositivo determinado pelo usuário;
5. **Salve como um Xchart.** Permite armazenar um *Xchart*, atribuindo um novo nome (identificador de arquivo), em um dispositivo determinado pelo usuário;
6. **Desfaça a operação.** Desfaz a última operação especificada pelo usuário;
7. **Selecione o modo nó.** Permite selecionar entre o modo nó, estabelecendo que as tarefas a serem efetuados referem-se a um nó e, o modo árvore, estabelecendo que as tarefas a serem efetuadas referem-se a uma subárvore;
8. **Recorte(nó/subárvore/transição).** Remove e copia, para a área de trabalho, um nó ou uma subárvore da árvore de estados ou uma transição do grafo de transições, anteriormente selecionados;
9. **Copie (nó/subárvore/transição).** Copia um nó ou uma subárvore da árvore de estados ou uma transição do grafo de transições, anteriormente selecionado, para a área de trabalho;
10. **Cole (nó/subárvore/transição).** Copia, a partir da área de trabalho, um nó ou uma subárvore da árvore de estados ou uma transição do grafo de transições para uma coordenada x, y específica;
11. **Destrua (nó/subárvore/transição).** Apenas remove um nó ou uma subárvore da árvore de estados ou uma transição do grafo de transições, anteriormente selecionado;
12. **Zoom-in.** Diminui duas vezes um *Xchart* descrito no editor;
13. **Zoom-out.** Amplia duas vezes um *Xchart* descrito no editor;
14. **Insira nó.** Insere um nó na árvore de estados;
15. **Insira transição.** Insere uma transição no grafo de transições;

16. **Procure (nó/evento/variável).** Permite procurar um nó, na árvore de estados, ou um evento ou variável, no grafo de transições;

17. **Substitua identificador de (nó/evento/variável).** Substitui um identificador de um nó, na árvore de estados, ou de um evento ou variável, no grafo de transições, por um novo identificador determinado pelo usuário;

18. **Repita a última busca.** Repete a última busca solicitada;

19. **Selecione um (nó/subárvore).** Seleciona um nó ou uma subárvore na árvore de estados;

20. **Defina atributos de (nó/transição).** Define ou altera os atributos de um nó, na árvore de estados, ou de uma transição, no grafo de transições

21. **Defina uma fonte de letra.** Define uma fonte de letra para ser utilizada pelo editor.

22. **Gere representação em forma de estados "encaixados".** Gera, a partir da árvore de estados e do grafo de transições, a representação em forma de estados "encaixados" de um *Xchart*;

A próxima seção identifica os objetos de interação a partir da definição destas tarefas.

3.2 Objetos de Interação

O *Smart* possui três objetos de interação responsáveis pela edição de um *Xchart*. O primeiro destina-se à edição da árvore de estados e é denominado *Tree*. O segundo destina-se à edição de um *Xchart* na forma de grafo e é denominado *Xgraph*. O terceiro destina-se à definição de características textuais de um *Xchart* e é denominado *TextSmart*. O *Smart* possui, também, mais três objetos de interação que permitem ao usuário solicitar as tarefas responsáveis pelo gerenciamento e manipulação de *Xcharts* denominados barra de menu, barra de ícones e a barra de status. A Figura 3.1 ilustra os objetos de interação do *Smart*. O objeto *Tree* será analisado no capítulo 4 e o objeto *Xgraph* será analisado no capítulo 5.

A barra de menu é composta por todas as tarefas do *Smart* organizadas em subitens nos itens arquivo, edição, *zoom*, inserção, procura, ferramentas e auxílio. A Figura 3.2 ilustra a estrutura dos objetos de interação da barra de menu do *Smart*.

A barra de ícones é responsável pelo acesso às tarefas mais referenciadas, segundo a sua funcionalidade. A Figura 3.3 ilustra a barra de ícones do *Smart*. A barra de *status* permite a comunicação do *Smart* com o usuário.

UNICAMP
BIBLIOTECA CENTRAL

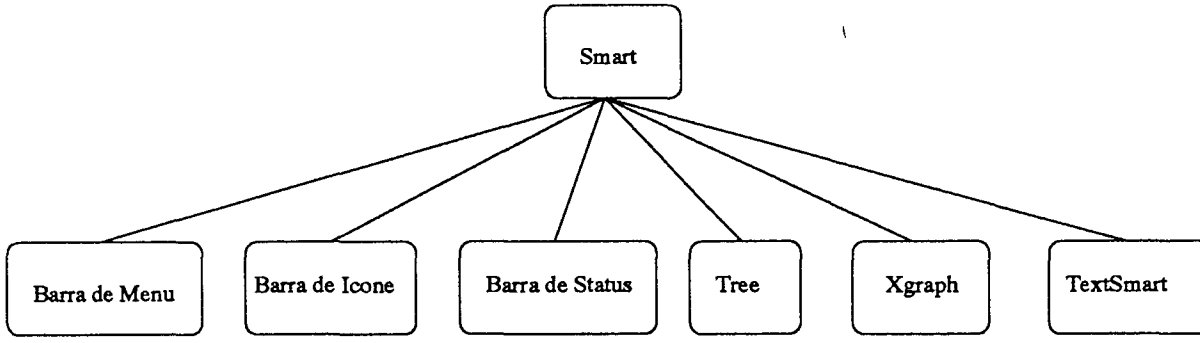


Figura 3.1: Objetos de interação do Smart

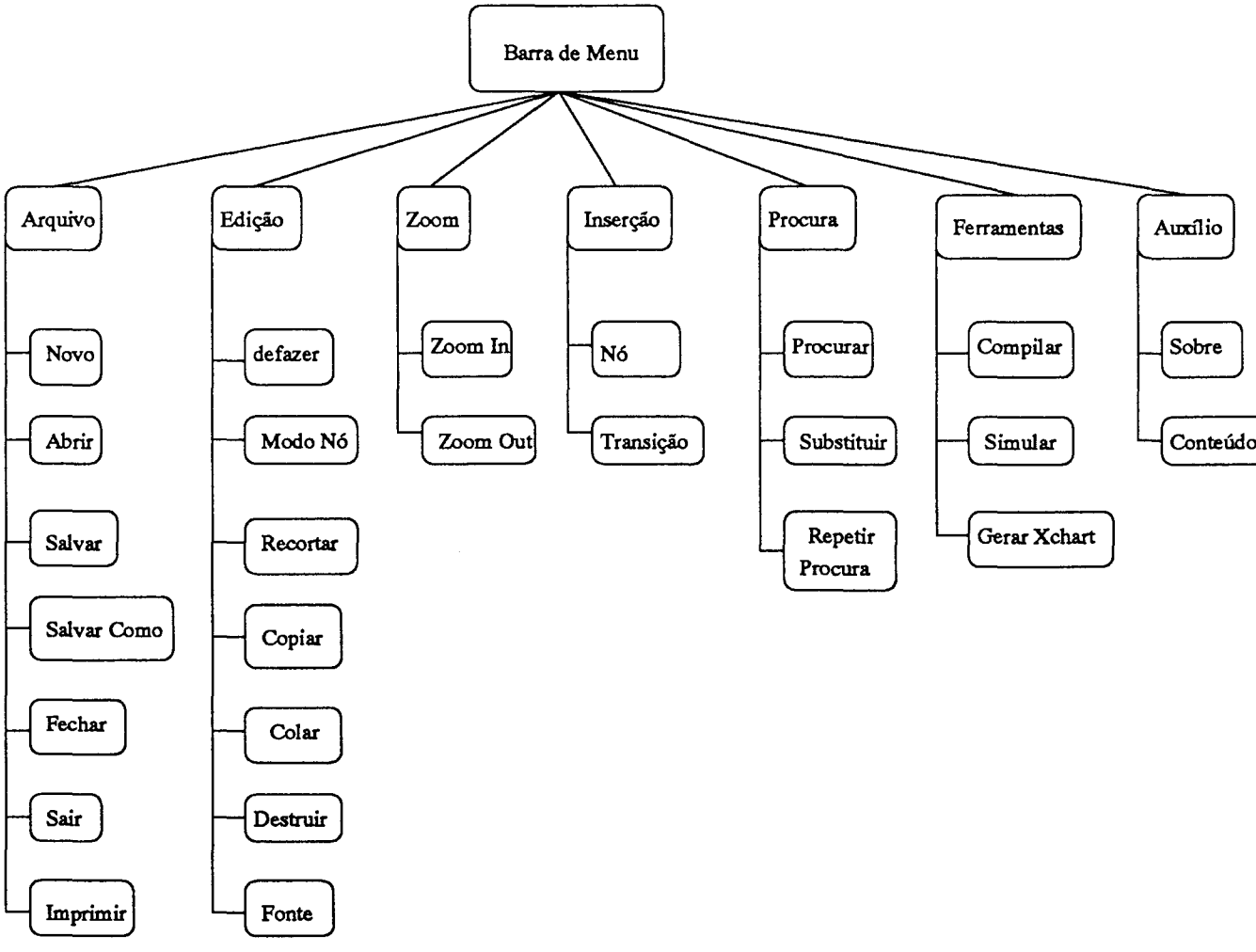


Figura 3.2: Estrutura dos objetos de interação da barra de menu

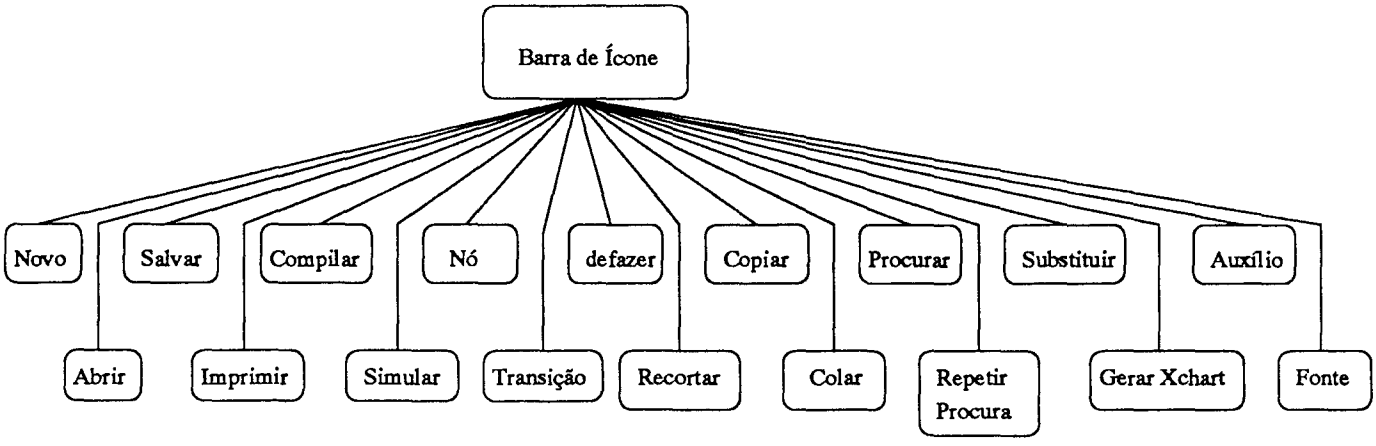


Figura 3.3: Estrutura dos objetos de interação da barra de ícone

A Figura 3.4 ilustra a geometria, leiaute e apresentação dos objetos de interação do *Smart*. A parte central da figura, à esquerda, representa a janela para edição da árvore de estados e abaixo desta, a janela de edição do *TextSmart* e, à direita, a janela para edição do *Xchart* na forma de grafo. A primeira "faixa" da Figura representa a barra de menu, a segunda, representa a barra de ícones e a última, a barra de status.

A notação *User Action Notation* (*UAN*)[4], própria para a descrição da reatividade dos objetos de interação, foi utilizada para descrever, verificar e validar a construção de protótipos da reatividade dos objetos de interação do *Smart*.

3.3 Arquitetura e Implementação

A Figura ilustra a arquitetura da apresentação do *Smart*. A cada objeto de interação está associada uma classe que descreve as operações pertinentes àquele objeto. Desta forma, a classe barra de menu descreve o objeto barra de menu, a classe barra de ícones descreve o objeto barra de ícones, a classe barra de status descreve o objeto barra de status, a classe *Tree* descreve o objeto árvore de estados, a classe *Xgraph* descreve o objeto estados "encaixados" e a classe *TextSmart* descreve o objeto *TextSmart*. A classe interface possui um ponteiro para cada uma das classes, e é responsável pela estrutura, leiaute, geometria e apresentação dos objetos de interação da interface. A reatividade dos objetos da interface é definida pela chamada de métodos das classes associadas, de acordo com o evento gerado.

Para gerar o código da interface, foi utilizado o *toolkit wxWindows*[1], que permite gerar código para plataformas distintas sem a necessidade de alterar o código fonte, permi-

[1] *Toolkit* de domínio público obtido em ftp www.aiai.ed.ac.uk/~jacs/vxwin.html

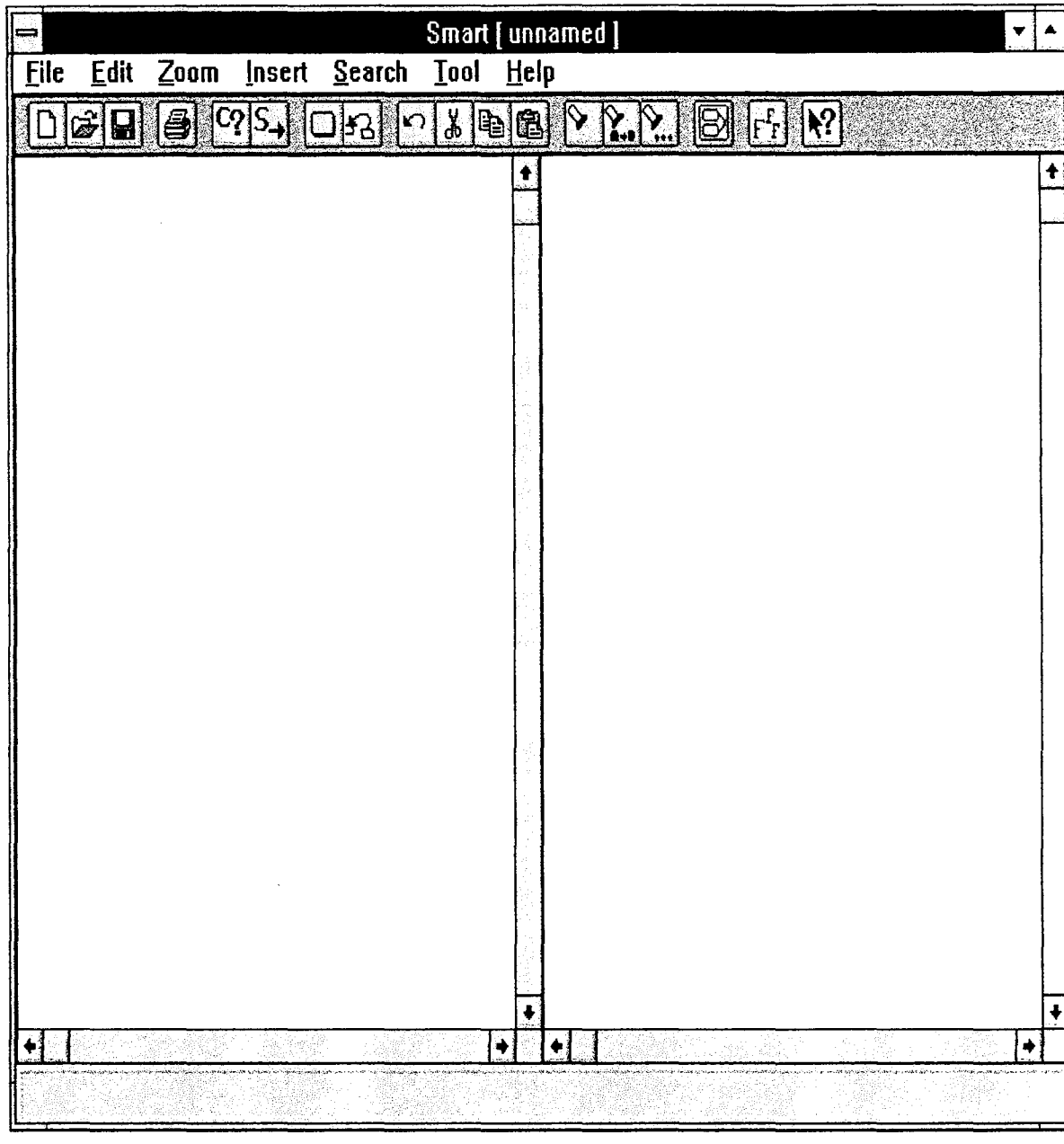


Figura 3.4: Geometria, leiaute e apresentação dos objetos de interação

tindo uma maior flexibilidade em relação às plataformas de execução. A próxima seção apresenta um exemplo da descrição de um *Xchart* efetuada pelo *Smart*.

3.4 O Cronômetro é Revisto

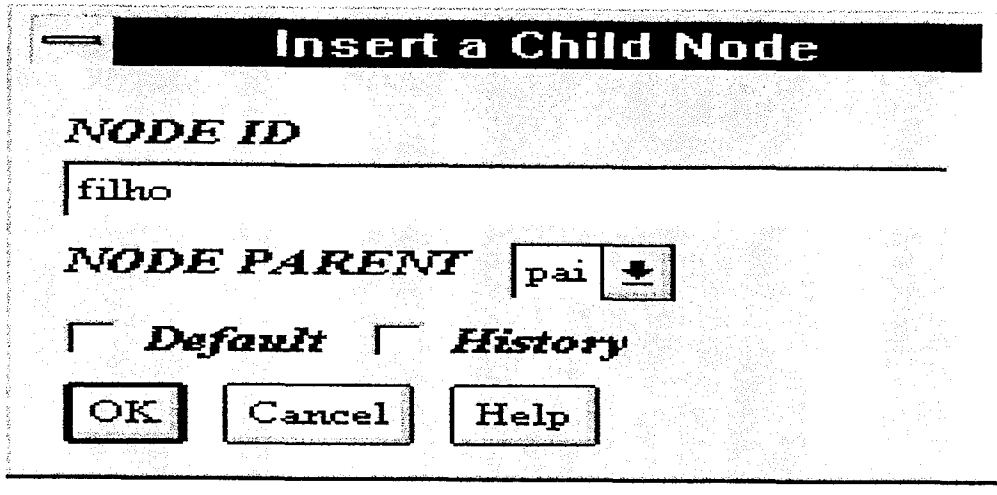


Figura 3.5: Inserção de um nó na árvore de estados

Nesta seção o exemplo do cronômetro ilustra como se comporta a interface homem-computador do *Smart* quando utilizada para a edição de uma especificação *Xchart*. Por exemplo, para capturar parte da especificação do *diálogo* do cronômetro, um usuário executaria os seguintes passos:

1. Inserir todos os vértices da árvore de estados. A inserção é feita através do *mouse* clicando no ícone "Nó" na barra de ícones ou através do teclado selecionando a opção "Inserção" e após, a opção "Nó" na barra de menu. Para efetivar a inserção é fornecido o identificador do nó, o respectivo nó pai, se houver, e o atributo *default* e *history*, se apresentar. A Figura 3.5 ilustra a inserção de um nó filho a o no pai;

2. Inserir todas as transições. A inserção é feita através do *mouse* clicando no ícone "Transição" na barra de ícones ou através do teclado selecionando a opção "Inserção" e após, a opção "Transição" na barra de menu. Para efetivara a inserção é fornecido o identificador do nó origem e do nó destino e o rótulo da transição. A Figura 3.6 ilustra a definição entre o nó pai e o nó filho definida pelo rótulo *evento [condição]/ação*, onde *evento* define o evento que define a transição, *condição* especifica uma condição que restringe a execução de uma transição, e *ação* define a ação a ser disparada na ocorrência da transição;

3. Gerar o *Xchart* na forma de grafo. A geração é feita através do *mouse* clicando no ícone "Gerar Xchart" na barra de ícones ou através do teclado selecionaria a opção "ferramentas" e, após, a opção "Gerar Xchart" na barra de menu.

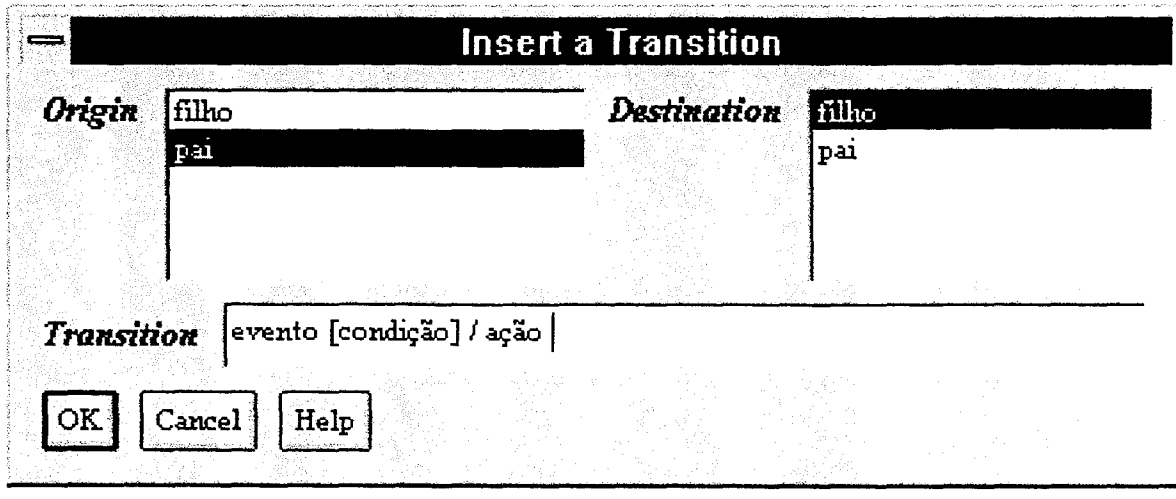


Figura 3.6: Inserção de uma transição no Smart

3.4.1 Xchart: Formato de um Arquivo

Para gravar um *Xchart*, o *Smart* gera uma descrição em forma de cláusulas em *prolog*. Estas cláusulas descrevem a árvore (cláusulas *node*, *default* e *history*) de estados, o grafo de transições (cláusula *arc*), onde:

- *node*("id","father").
- *default*("id").
- *history*("id").
- *arc*("origin","destination","transition").

Esta solução foi adotada no *Smart* pelo fato do *toolkit wxWindows*, utilizado para sua implementação, suportar um banco de dados definido por cláusulas *prolog*. Esta solução permite, também, que a descrição de *Xcharts* estensos seja realizada de forma a minimizar o espaço para a sua armazenagem em disco.

3.5 Resumo

Neste Capítulo foi descrito o componente de apresentação do *Smart*. Nesta descrição da apresentação foram definidos os objetos de interação, derivadas das tarefas pertinentes à *Smart*, a estrutura, geometria, leiaute, apresentação e reatividade de cada objeto. No próximo capítulo é descrito o componente *Tree*, responsável pela geração da árvore de estados de um *Xchart*.

Capítulo 4

Tree: O Componente de Leiaute de Árvores

Neste Capítulo descreve-se o componente do *Smart* responsável pela representação de *Xcharts* como árvores. Este componente, denominado **Tree**, foi projetado e implementado para ser utilizado pelo *Smart* e também de forma autônoma, como uma biblioteca de auxílio à vizualização de árvores. Em geral, a visualização de um *Xchart* como uma árvore permite que o projetista avalie a hierarquia de estados do *Xchart*, a sua "profundidade" e a sua "largura" facilitando o seu entendimento.

4.1 A Arquitetura de Tree

Tree é o componente do *Smart* que permite o leiaute de árvores. **Tree** trata a representação de árvores que podem apresentar nós com um número arbitrário de filhos, possuindo formatos e tamanhos diferentes. Os métodos de **Tree** efetuam: cálculo de leiaute, espelhamento, orientação, ajustamento, posicionamento, inserção e remoção de nós e de subárvores. **Tree** apenas determina a estrutura, geometria e leiaute de uma árvore, não gera, de fato, o traçado final do diagrama da árvore, pois o traçado final é, em geral, dependente do dispositivo gráfico utilizado para efetuá-lo. Os algoritmos para o controle de periféricos podem ser ligados a **Tree** através do uso de herança.

Tree foi implementada em C++ e, os métodos referentes ao traçado de uma árvore, foram definidas como virtuais. Estes métodos virtuais deverão ser definidas em subclasses, cujo código será dependente da plataforma de execução. **Tree** define a estrutura neutra do *Smart* e, é composta pelas classes:

- classe *Node*
- classe *Layout*

- classe *PrimitiveTree*

- classe *Tree*

A Figura 4.1 ilustra a organização das classes em **Tree**, de acordo com a notação *Object Modeling Technique* (OMT) [10].

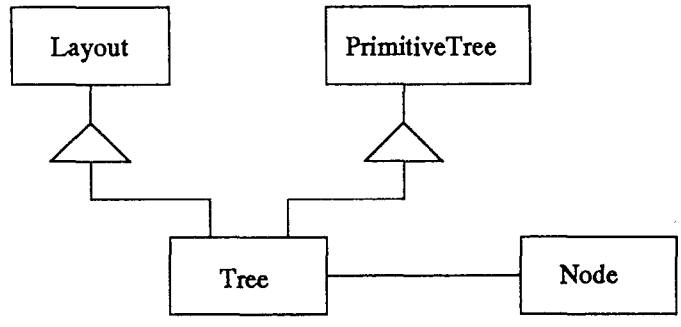


Figura 4.1: Organização das classes em Tree

A classe *Node* define o nó que é manipulado pela árvore. A classe *Layout* implementa o algoritmo de leiaute utilizado por **Tree**. A classe *PrimitiveTree* implementa os métodos primitivos da classe *Tree*. A classe *Tree* implementa todos os métodos necessários para o leiaute de árvores. A classe *Tree* herda as características das classes *Layout* e *PrimitiveTree*. Uma instância de *Tree* contém um ponteiro para um nó, definido pela classe *Node*.

O algoritmo de leiaute utilizado por **Tree** é uma adaptação do algoritmo de Moen [17]. O algoritmo de Moen realiza o leiaute de uma árvore referencialmente no sentido horizontal, traçando os nós da árvore da esquerda para a direita. Para qualquer subárvore da árvore, temos o nó raiz posicionado à esquerda e os seus nós filhos à direita. Todo nó da árvore é composto por um contorno. O contorno de um nó é representado, explicitamente, por polígonos. O objetivo, do algoritmo, é posicionar os nós da árvore tão próximos quanto possível. Para posicionar estes nós, o algoritmo, baseia-se nas dimensões do contorno de cada nó. O algoritmo também gera as posições relativas ao nó pai ou nó irmão para cada nó da árvore. O algoritmo de traçado utiliza coordenadas relativas durante o leiaute, minimizando o número de mapeamentos para as coordenadas absolutas. As extensões, efetuadas no algoritmo de Moen, permite **Tree** expressar o leiaute de uma árvore tanto no sentido horizontal como no vertical, bem como, a execução do espelhamento da árvore segundo o eixo x e o eixo y.

4.2 A Classe PrimitiveTree

A classe *PrimitiveTree* possui todos os métodos que efetuam as trocas de coordenadas e a correção para alterar a orientação da árvore, efetuar as trocas de coordenadas no espelhamento da árvore e a inserção ou remoção de um nó da árvore, atualizando-a. Esta classe permite manter o código da classe **Tree** mais limpo, pois o código necessário para concretizar os métodos em *Tree* estão todos definidos na classe *PrimitiveTree*.

4.3 A Classe Layout

A classe *Layout* determina o contorno e a posição de todos os nós na árvore, após definir os valores dos atributos comprimento, largura e borda de cada nó. A posição atribuída a cada nó é, de fato, a posição relativa do nó em relação ao seu predecessor.

A posição absoluta, no sistema de coordenadas utilizado para o traçado, é recalculada sempre que a árvore é alterada. Para efetuar o cálculo da posição absoluta de um nó, é necessário fixar a posição absoluta do nó raiz e, então, percorrer em profundidade a árvore adicionando à posição relativa de cada nó o deslocamento absoluto do nó predecessor para se obter a posição absoluta do nó corrente.

A classe *Layout* percorre a árvore em profundidade e estabelece o contorno de cada subárvore para efetuar o leiaute da árvore. O contorno de uma subárvore acopla o contorno do nó pai ao contorno do conjunto formado pelos respectivos nós filhos da subárvore, formando apenas um único contorno. O contorno de uma subárvore é dividido em três etapas:

1. A formação do contorno de cada nó folha (método *LayoutLeaf*);
2. A união dos contornos dos nós filhos (método *join*) é armazenada no atributo de contorno do nó pai. Esta união acopla os contornos dos nós filhos tão próximos quanto possível, e registra as posições relativas entre os nós filhos;
3. O cálculo da posição relativa entre o nó pai e os respectivos nós filhos (método *AttachParent*).

A seguir são analisadas cada um dos respectivo métodos que efetuam o contorno de uma ramificação.

4.3.1 Método LayoutLeaf

O método *LayoutLeaf* constrói o contorno de um nó folha. O contorno de um nó folha é um quadrilátero que coincide inicialmente com a borda do nó. No contorno do nó é

omitido o segmento esquerdo, retardando, desta forma, o fechamento do contorno. Esta política de fechamento deve-se ao fato da tentativa de minimizar o custo para acoplar novos nós a este para a formação de uma subárvore.

4.3.2 Método Join

O método *Join* inicialmente atribui o contorno do nó pai igual ao contorno do seu nó filho. A partir do ponteiro *sibling* do nó filho, os demais nós filhos são percorridos e o contorno do nó pai é estendido ao contorno de todos os seus nós filhos. A Figura 4.2 ilustra a formação do contorno de uma subárvore. O método *Join* também atualiza a posição relativa de cada nó filho percorrido. Os contornos dos nós filhos não são destruídos, pois podem ser reaproveitados futuramente.

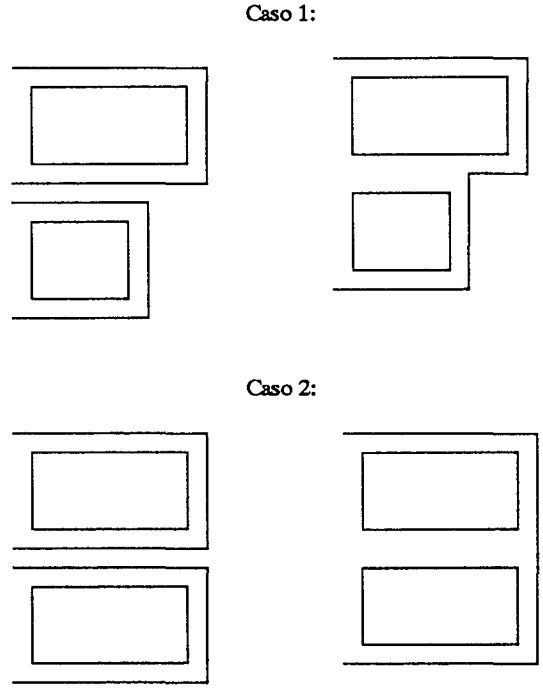


Figura 4.2: Contorno de dois nós irmãos após a execução do método Join

4.3.3 Método AttachParent

O método *AttachParent* calcula a posição relativa do nó pai em relação ao seus respectivos nós filhos. A posição é armazenada no nó filho acessado pelo ponteiro *child* do nó pai. O método *AttachParent* acrescenta segmentos de retas para envolver o nó pai no contorno formado pela união dos respectivos nós filhos. A Figura 4.3 ilustra o acoplamento do

contorno do nó pai ao contorno formado pela união dos contornos dos nós filhos. O segmento esquerdo do contorno do nó pai é, também, omitido, como na formação do contorno do nós folhas da árvore.

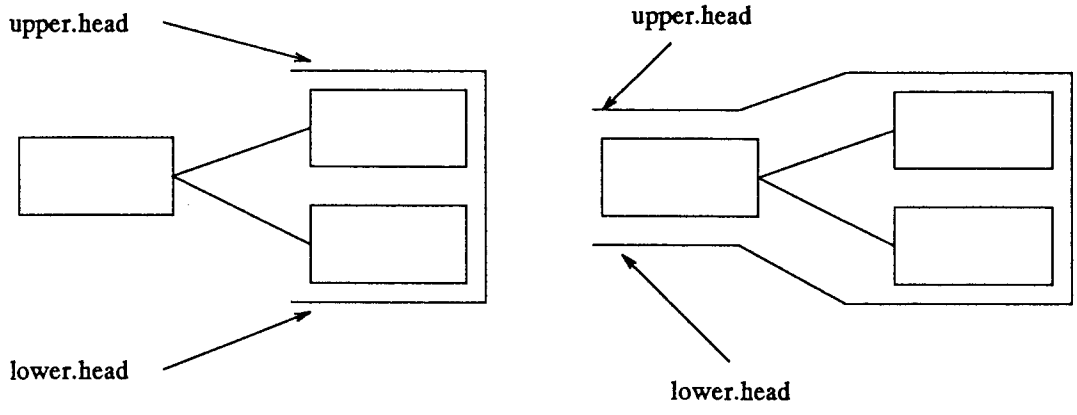


Figura 4.3: Representação da ação da método AttachParent

4.4 A Classe Tree

A classe *Tree* permite a inserção e remoção de um nó na árvore. Para inserir um nó é necessário fornecer os identificadores do nó, do nó pai e/ou do nó irmão. Se for fornecido apenas os identificadores do nó e do nó pai, este nó será inserido como nó filho do nó pai, se o nó pai não possuir nenhum nó filho, ou, será inserido como um nó irmão do nó mais abaixo dos nós filhos do nó pai, se o nó pai já possuir um nó filho. Se for fornecido os identificadores do nó, do nó pai e do nó irmão; o nó será inserido como irmão do nó especificado e terá, como nó pai, o nó especificado.

Para remover um nó da árvore é preciso fornecer apenas o identificador do nó e informar se a opção implica remover apenas o nó ou a subárvore formada pelo nó a ser removido. Após a inserção ou remoção de um nó, o leiaute da árvore é efetuado, de fato, no sentido horizontal, porém, a classe *Tree* permite efetuar um espelhamento segundo o eixo x e y e a mudança de orientação na árvore.

Para efetuar tanto o espelhamento independente do eixo, bem como a mudança de orientação, a classe *Tree* manipula geometricamente os valores dos atributos de cada nó da árvore. Como resultado, obtêm-se o efeito visual desejado sem alterar, de fato, a estrutura da árvore gerada pelo algoritmo de leiaute. A classe *Tree* utiliza os atributos *orientation* e *view* para controlar o que é realmente visto e, o que é realmente representado pela estrutura interna da árvore.

O atributo *orientation* (*NORTH, SOUTH, EAST* e *WEST*) indica o sentido em que a árvore está sendo espelhada. O atributo *view* (*AS_SAME_AS* e *INVERTED*) indica se a

estrutura interna da árvore está sendo editada no sentido horizontal, quando possui o valor *AS_SAME_AS*, ou no sentido vertical, quando possui o valor *INVERTED*. Internamente, em termos de estrutura, a árvore é sempre mantida no sentido horizontal.

O método *DoMirrorX* permite efetuar um espelhamento da árvore segundo o eixo x, independente do sentido no qual a árvore tenha sido editada anteriormente: horizontal, Figuras 4.4.a e 4.4.b, ou vertical, Figuras 4.4.c e 4.4.d.

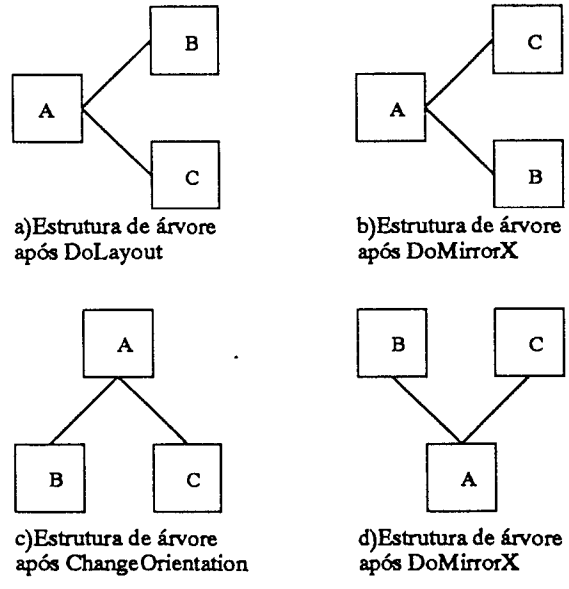


Figura 4.4: Representação do método de espelhamento segundo o eixo x

O método *DoMirrorY* permite efetuar um espelhamento da árvore segundo o eixo y, independente do sentido no qual a árvore tenha sido editada anteriormente: horizontal, Figuras 4.5.a e 4.5.b, ou vertical, Figuras 4.5.c e 4.5.d.

O método *ChangeOrientation* permite editar uma árvore alterando o atual sentido de edição, isto é, se a árvore estiver editada horizontalmente, o método edita-la-á verticalmente, e vice-versa. Para efetuar a alteração na orientação, a classe *Tree* inverte a posição das coordenadas x pela y do contorno do nó raiz e todas as coordenadas absolutas de todos os nós da árvore. Porém, a troca de coordenadas não é suficiente para completar a mudança de orientação. É necessário efetuar uma correção. Esta correção consiste em efetuar um espelhamento segundo o eixo x ou y, dependendo do sentido em que a árvore estava editada. A Figura 4.6 mostra um exemplo de *ChangeOrientation* aplicada a uma árvore editada horizontalmente.

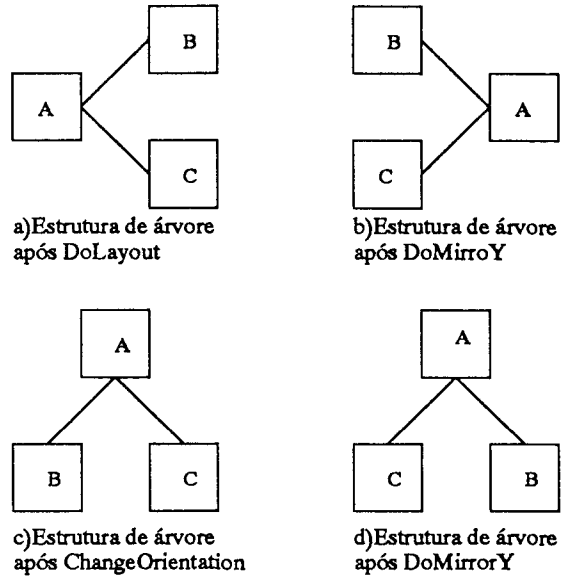


Figura 4.5: Representação do método de espelhamento segundo o eixo y

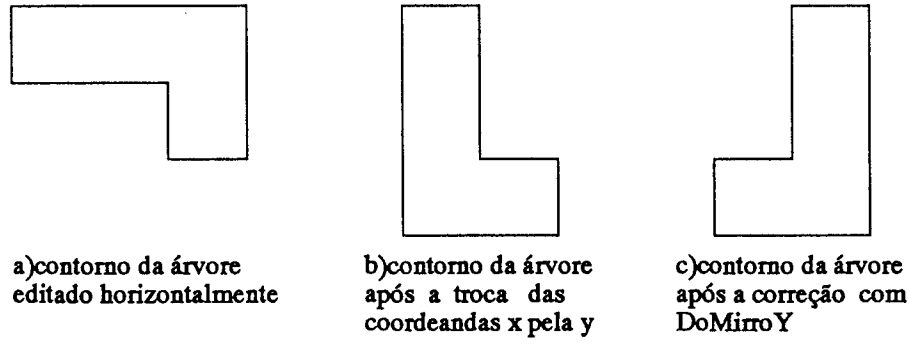


Figura 4.6: ChangeOrientation aplicada a uma estrutura de árvore editada horizontalmente

4.5 A Classe Node

Para **Tree**, o nó é representado por um quadrilátero e, é envolvido por outro quadrilátero, denominado contorno. O nó também possui os atributos: comprimento, altura e borda. Estes atributos conferem uma maior flexibilidade ao algoritmo, permitindo que cada nó possua seu próprio comprimento, largura e borda. Um nó possui, ainda, um rótulo. Este rótulo é definido por uma cadeia de caracteres, que permite a identificação do nó. O nó possui ponteiros para os nós pai, filho e irmão. O ponteiro para o nó filho conduz ao nó mais à direita (supondo que a árvore é representada no sentido horizontal), enquanto o ponteiro para o nó irmão conduz ao primeiro nó imediatamente abaixo do respectivo nó. A Figura 4.7 ilustra um nó derivado da classe *Node*.

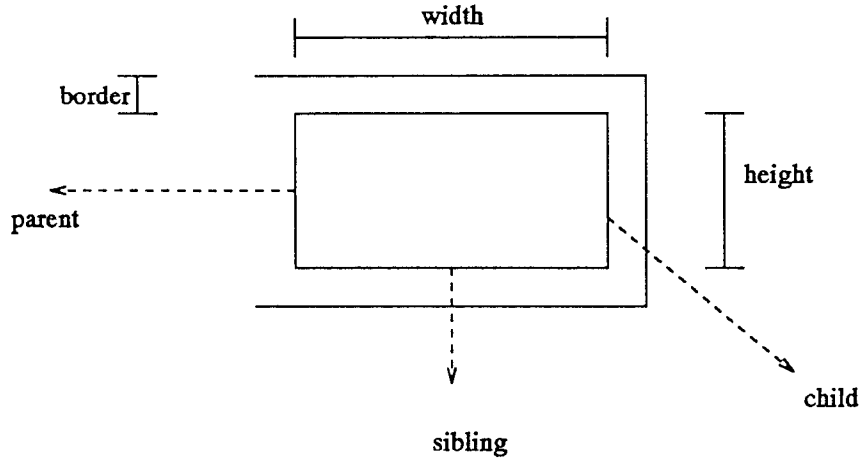


Figura 4.7: Representação de um nó definida pela classe Node

O contorno do nó é representado por duas poli-linhas: *upper* e *lower*. Cada poli-linha é formada por uma seqüência de um ou mais segmentos de reta. O nó possui um ponteiro para o primeiro e o último segmento de reta, *head* e *tail*, para cada uma das duas polilinhas. A Figura 4.8 representa as polilinhas do nó ilustrado na Figura 4.7.

A posição de um nó é definida pelas coordenadas *pos_offset* e *offset*. A coordenada *pos_offset*, gerada pelo algoritmo de leiaute, define a posição do nó em relação ao seu predecessor. O predecessor de um nó filho é o nó pai, enquanto o predecessor dos nós não filhos é o nó irmão acima dele na árvore. Na Figura 4.9 é possível determinar os nós predecessores de cada nó da árvore, obtendo como resultado:

- $predecessor(root) = nulo$;
- $predecessor(node1) = root$;
- $predecessor(node2) = node1$;

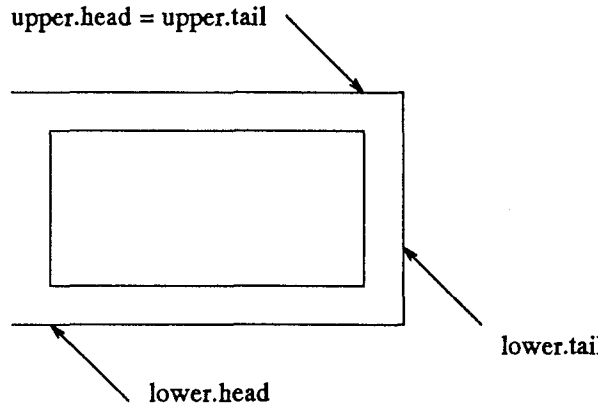


Figura 4.8: Representação da poli-linha que define o contorno de um nó

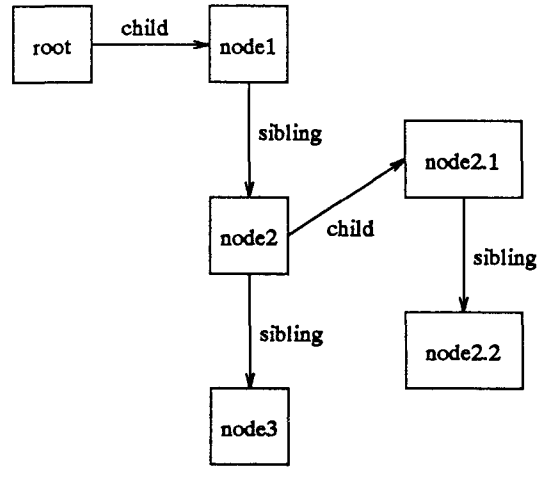


Figura 4.9: Representação de uma árvore gerada pelo algoritmo de leiaute

- $predecessor(node3) = node2$;
- $predecessor(node2.1) = node2$;
- $predecessor(node2.2) = node2.1$;

A coordenada *offset*, gerada pelo algoritmo de traçado, define a posição absoluta do nó em relação à raiz da árvore no sistema de coordenadas utilizado. No sistema de coordenadas utilizado para o traçado, os valores no eixo x variam em ordem crescente da esquerda para direita e os valores do eixo y variam em ordem crescente de cima para baixo.

4.6 A Biblioteca Tree

Para o algoritmo de leiaute de **Tree**, um nó é representado por um quadrilátero. Para uma aplicação do usuário, porêm, um nó pode ser representado por qualquer figura geométrica. A Figura 4.10 representa um nó definido por uma aplicação particular.

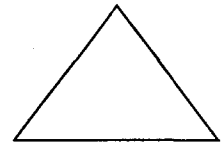


Figura 4.10: Representação de um nó para uma aplicação particular

O usuário, então, definirá uma nova classe que constituirá uma herança da classe *Node*. Esta nova classe será responsável pelo mapeamento do nó da aplicação para o nó definido pelo algoritmo de leiaute. A Figura 4.11 demonstra o mapeamento do nó da aplicação da Figura 4.10 para o nó definido pelo algoritmo de leiaute.

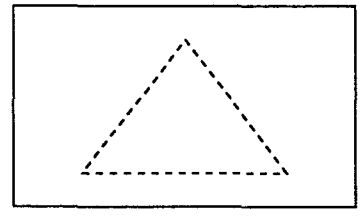


Figura 4.11: Nó de uma aplicação visto pelo algoritmo de leiaute

A nova classe deverá definir, também, os pontos de ligação (*Branch*) do nó da aplicação em relação ao nó gerado pelo algoritmo de traçado. Os pontos de ligação são responsáveis pela conexão dos nós na árvore, de acordo com o sentido expresso pelo algoritmo de leiaute.

Os pontos de ligação são formados por quatro coordenadas: *UpperBranch*, *LowerBranch*, *LeftBranch* e *RightBranch*. A Figura 4.12 mostra os pontos de ligação definidos pelo nó da aplicação da Figura 4.10.

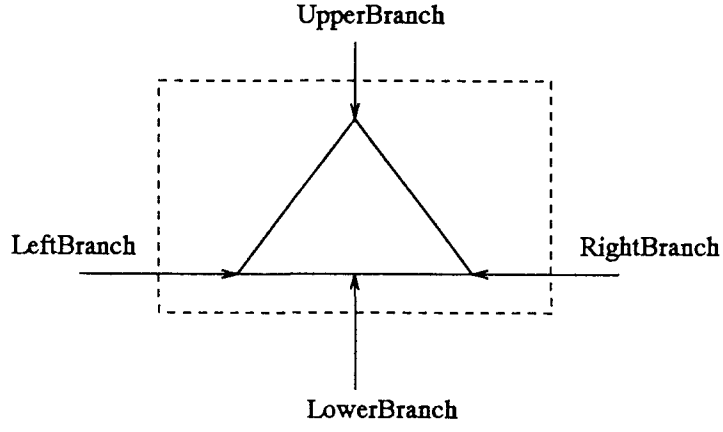


Figura 4.12: Representação dos pontos de ligação pela aplicação do usuário

A nova classe, também, deverá definir os métodos responsáveis pelo traçado do nó, do seu respectivo contorno e pela impressão do seu identificador. Os métodos de traçado e impressão são dependentes do dispositivo gráfico e da plataforma de execução utilizados pela aplicação. Portanto, **Tree** pode ser gerada como uma biblioteca autônoma para o traçado de subárvores específicas.

4.7 Tree e o Cronômetro

Esta seção traz a representação do componente de diálogo do cronômetro gerada pelo componente **Tree** do *Smart*. O componete de diálogo do cronômetro é ilustrado na Figura 4.13. Utilizando as tarefas de edição do *Smart*, pode-se capturar e editar a especificação do cronômetro. A Figura 4.14 ilustra a árvore de estados gerada por *Smart* a partir dos dados capturados. As transições serão visualizadas no leiaute de grafo da especificação do cronômetro.

4.8 Resumo

Este Capítulo apresentou o componente **Tree**, ilustrado na Figura 2.3, responsável pelo leiaute automático de árvores no *Smart*. O componente **Tree** é reponsável pela captura e edição de *Xcharts* representados por uma árvore de estados e um grafo de transições sobreposto à árvore que definem a estrutura neutra do *Smart*. **Tree** possui métodos que manipula uma estrutura de árvore. Estes métodos efetuam: cálculo de leiaute, espelhamento,

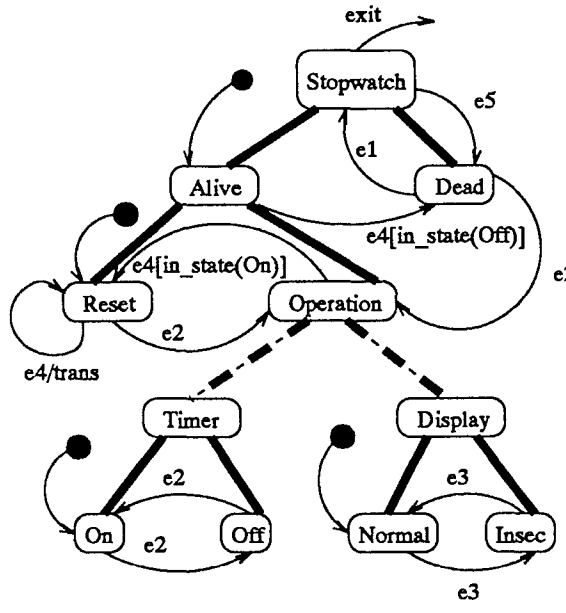


Figura 4.13: Representação de transições em uma hierarquia

orientação, ajustamento, posicionamento, inserção e remoção de nós e de subárvores. **Tree** apenas determina a estrutura, geometria e leiaute de uma árvore, não gerando, de fato, o traçado da árvore, isto se deve ao fato de que parte dos recursos necessários para tal são dependentes do dispositivo gráfico utilizado para efetuar o traçado. Estes dispositivos devem ser providos por subclasses específicas, através de herança. O *Smart* herda estas classes e define os dispositivos gráficos necessários para efetuar o traçado. **Tree** é composta pelas classes: *Node*, *Layout*, *PrimitiveTree* e *Tree*. A classe *Node* define o nó que é manipulado pela árvore. A classe *Layout* implementa o algoritmo de leiaute utilizado por **Tree**. A classe *PrimitiveTree* implementa os métodos primitivos necessárias para a classe *Tree* definir e manipular uma árvore. A classe *Tree* herda as características das classes *Layout* e *PrimitiveTree*. Uma instância de *Tree* contém um ponteiro para um nó definido pela classe *Node*. O algoritmo de leiaute percorre a árvore que compõe a hierarquia de estados de um *Xchart*, em profundidade, acoplando o contorno de cada subárvore, obtendo o contorno final da árvore. Este contorno possui a característica de alocar os nós tão próximos quanto possível. **Tree** define a estrutura neutra do *Smart*. O algoritmo implementado apresenta-se como uma adaptação do algoritmo de traçados de árvores proposto por Moen[17]. No próximo capítulo é apresentada a componente *Xgraph* responsável pelo leiaute de *Xchart* na sua forma usual, um grafo de *Statecharts*[5].

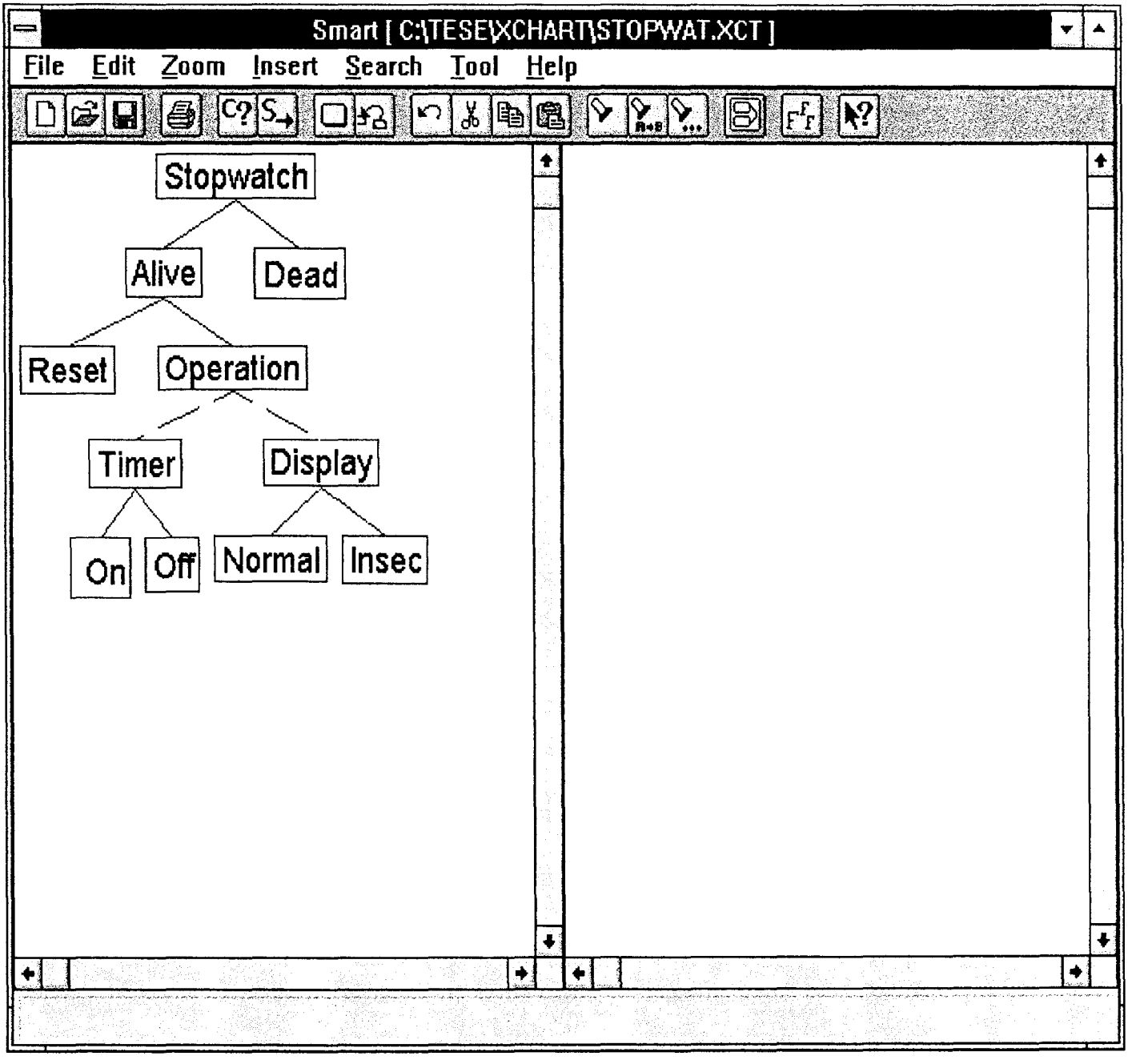


Figura 4.14: Captura da árvore de estados do cronômetro

Capítulo 5

Xgraph: O Componente de Leiaute de Grafos

No Capítulo anterior foi detalhado o componente **Tree** do *Smart* que é responsável pelo desenho de um *Xchart* como uma árvore. Neste capítulo será apresentado o componente *Xgraph* que é responsável pelo desenho de um *Xchart* como um grafo, isto é, como um *dígrafo composto*. A implementação deste módulo baseia-se em um algoritmo para leiaute de dígrafos compostos proposto por Sugiyama [7]. O algoritmo é composto por quatro fases autônomas que induziram a organização da arquitetura de *software* de *Xgraph* em quatro classes principais.

O Capítulo está organizado da seguinte maneira. Inicialmente, apresenta-se uma visão geral da arquitetura de *software* de *Xgraph*; a apresentação é baseada em um modelo de objetos. Em seguida, apresentam-se os requisitos que um traçado de um dígrafo composto deve satisfazer para ser facilmente visualizado por um ser humano. O algoritmo implementado atende à maioria desses requisitos. Finalmente, detalha-se os métodos de cada uma das classes da arquitetura de *software* de *Xgraph*. A descrição do comportamento das classes é equivalente à descrição de cada uma das fases do algoritmo de leiaute de dígrafos compostos.

5.1 Arquitetura do Xgraph

O algoritmo de Sugiyama é composto por quatro fases: *hierarquização*, *normalização*, *ordenação* e *métrica*. As fases do algoritmo executam operações distintas sobre um *dígrafo composto*: o grafo orientado composto que representa o *Xgraph* a ser desenhado.

A Figura 5.1 ilustra a arquitetura do *Xgraph*. A classe *Xgraph* implementa o protocolo que garante o leiaute de um grafo. Este protocolo utiliza métodos herdados de quatro classes base, uma para cada fase do algoritmo de leiaute. A classe *Xgraph* implementa os

métodos utilizados por *Smart* para coordenar o traçado de um *Xchart* como uma árvore e como um grafo hierárquico. A arquitetura de software do componente *Xgraph* reflete diretamente as fases do algoritmo de leiaute de grafos.

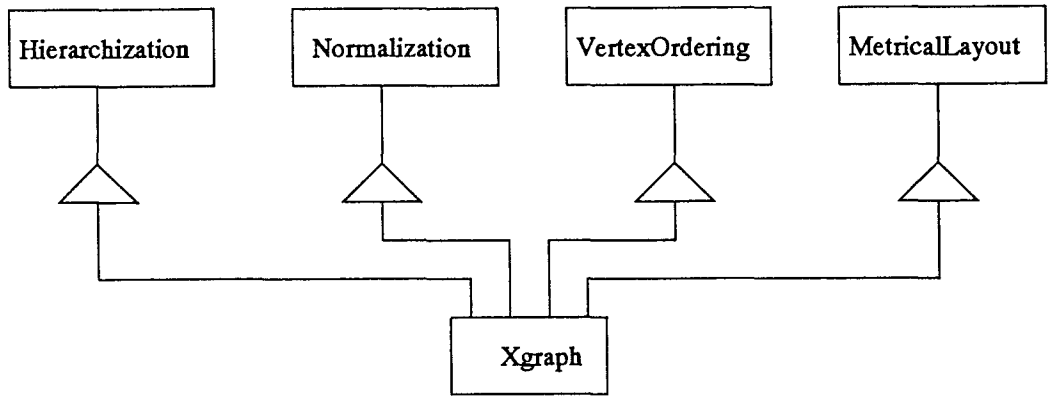


Figura 5.1: A arquitetura do Xgraph

Antes de realizarmos a descrição detalhada das funções de cada uma das classes representadas na arquitetura de *software* do *Xgraph* (Figura 5.1), é necessário definir a classe de grafos a qual o algoritmo se aplica.

5.2 Dígrafos Compostos

O algoritmo implementado pela arquitetura de *software* de *Xgraph* efetua o traçado de dígrafos compostos. Dígrafos compostos são obtidos pela composição de dois grafos: (i) uma árvore orientada de inclusão e (ii) um dígrafo de adjacência. A árvore orientada de inclusão representa a hierarquia de estados de um *Xchart*, um estado pode estar *incluído* em outro estado e assim sucessivamente. O dígrafo de adjacência representa o relacionamento de vizinhança entre estados de um *Xchart*.

5.2.1 Árvore de Inclusão

A *árvore orientada de inclusão*, por simplicidade *árvore de inclusão*, é definida pela dupla: $D_\iota = (V, E)$. A árvore D_ι é enraizada, isto é, contém um vértice especial que é a sua *raiz*. Uma seqüência de vértices $v_1, v_2, \ldots, v_k$ tal que $(v_j, v_{j+1}) \in E$, $1 \leq j \leq \mid k-1 \mid$, é denominado *caminho* de v_1 a v_k. Sejam v e w dois vértices de D_ι. Suponha que v pertença ao caminho da raiz de D_ι a w. Então v é *ancestral* de w, sendo w descendente de v. Adicionalmente, se (v, w) é uma aresta de D_ι, então v é *pai* de w, sendo w *filho* de v. A raiz de uma árvore não possui pai, enquanto todo vértice $v \neq w$ possui um único. Uma *folha* é um vértice que não possui filhos.

As relações pai, filho, ancestral e descendente de um vértice $v \in V$ são denominadas $Pai(v)$, $Filho(v)$, $An(v)$ e $De(v)$, respectivamente. A profundidade de qualquer vértice $v \in V$, denominada $dep(v)$, é o número de vértices entre no caminho que vai de v até a raiz, onde $dep(\text{raiz}) = 1$. $An(v)$ e $De(v)$ incluem o vértice v.

Relação de Inclusão

Seja $(u, v) \in E$, então u *incluí* v se, e somente se, $Pai(v) = u$ em D_ι, portanto: $D_\iota = \{(V, E) | (u, v) \in E \longleftrightarrow u \text{ inclui } v\}$ onde E é um conjunto finito de arestas de inclusão.

O *Xchart* representado na Figura 5.2 pode ter os relacionamentos de inclusão entre seus estados representado pela àrvore de inclusão da Figura 5.3.

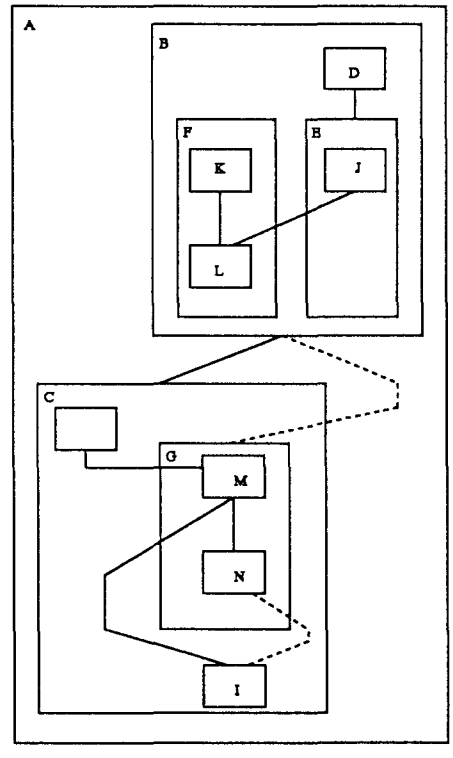


Figura 5.2: Traçado de um Xchart.

5.2.2 Dígrafo de Adjacência

Um *dígrafo de adjacência* é definido pela dupla: $D_\alpha = (V, F)$. Seja $(u, v) \in F$, então u é *adjacente* a v se $\{v \in An(u) \cup De(u)\} = \emptyset$. Seja v um vértice de uma árvore de inclusão, então nenhum dos descendentes ou ancestrais de v é adjacente a v. Logo,

$$D_\alpha = \{(V, E) | (u, v) \in F \longleftrightarrow u \textit{ adjacente a } v\}$$

onde F é um conjunto finito de arestas de adjacência.

O *Xchart* representado na Figura 5.2 pode ter os relacionamentos de adjacência entre seus estados representado pelo dígrafo de adjacência da Figura 5.4.

Podemos agora definir um *dígrafo composto* como a ênupla $D = (V, E, F)$ obtida pela composição da árvores de inclusão com o dígrafo de adjacência. A Figura 5.5 traz uma representação do dígrafo composto obtido pela composição dos grafos representados em 5.3 e 5.4.

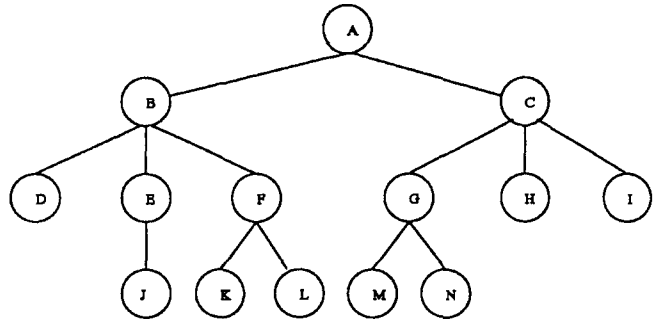


Figura 5.3: Árvore de inclusão.

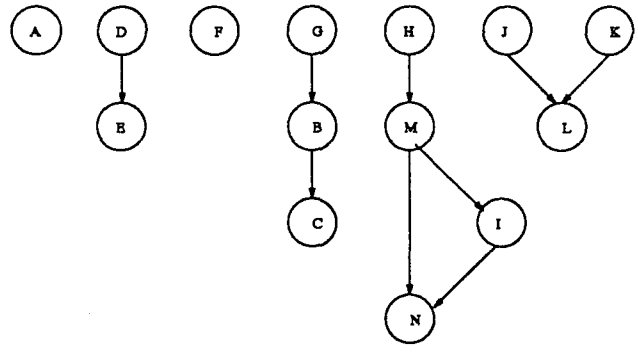


Figura 5.4: Dígrafo de adjacência.

5.3 Requisitos para o Traçado de Dígrafos Compostos

Uma característica do algoritmo de leiaute de dígrafos utilizado por *Xgraph* é a preocupação com os aspectos de cognição visual humana, então, o algoritmo define propriedades que permitem melhorar a legibilidade do traçado de um dígrafo. Por simplicidade, o

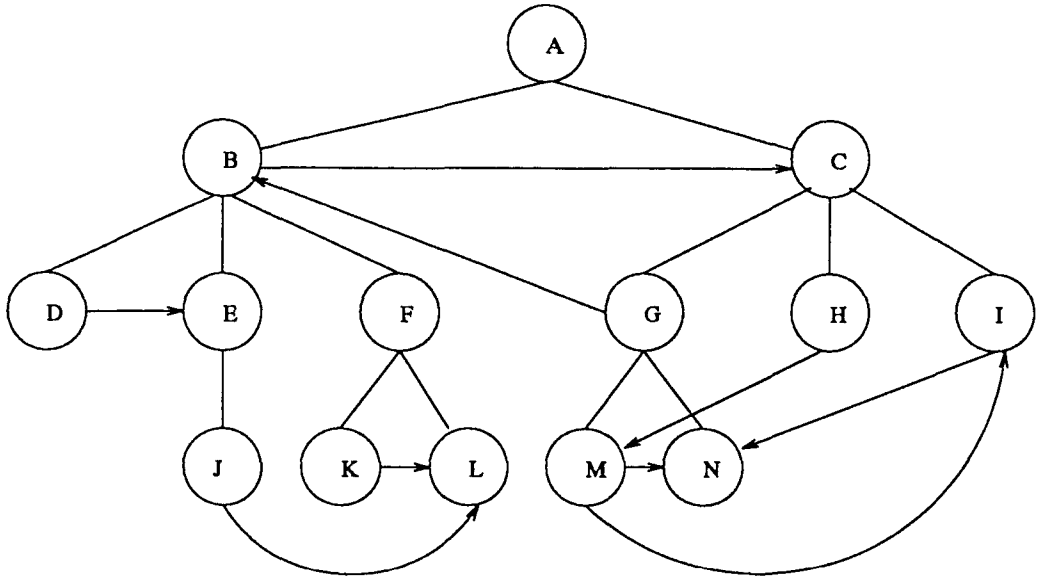


Figura 5.5: Dígrafo composto.

termo *mapa* será utilizado no lugar da frase *traçado de um dígrafo composto.* As propriedades para melhorar a legibilidade de um mapa são classificadas em *convenções* e *regras* de traçado. Convenções representam propriedades fundamentais, isto é, se um mapa não as tiver, então ele provavelmente será menos legível que o mapa que as tem. Regras são propriedades opcionais de mapas, isto é, não necessariamente um mapa que não as tem será menos legível que um mapa que as tem.

5.3.1 Convenções

A seguir são enumeradas as convenções[7] relevantes para o traçado de um mapa:

1. **Retângulo.** Um vértice $v \in V$ é traçado na forma de um retângulo;

2. **Inclusão.** Uma aresta de inclusão $(u, v) \in E$ é traçada de modo que o retângulo u inclua o retângulo v. Retângulos que não apresentam a relação de inclusão devem estar dispostos de forma disjunta;

3. **Hierarquia.** O leiaute dos vértices é efetuado de forma hierárquica entre níveis como no grafo de inclusão;

4. **Ligação de Vértices.**

   Uma aresta de adjacência $(u, v) \in F$ é traçada como um segmento orientado de reta com origem na linha horizontal inferior do retângulo correspondente a u e fim na linha horizontal superior do retângulo correspondente a v.

5.3.2 Regras

As regras de traçado são classificadas quanto à semântica e à estrutura de um mapa. A semântica define o significado dos vértices e arestas para um ser humano, enquanto a estrutura define o arranjo geométrico a topologia do mapa. A seguir enumeram-se as regras de traçado:

1. **Proximidade.** Os vértices são traçados tão próximos quanto possível;
2. **Cruzamento entre Linhas.** O número de cruzamentos entre linhas devido as arestas de adjacência é reduzido tanto quanto possível;
3. **Cruzamento entre Linhas e Retângulos.** O número de cruzamento entre arestas de adjacência e retângulos é reduzido tanto quanto possível;
4. **Minimização de Segmentos de Retas.** As arestas de adjacência que estão distanciadas de apenas um nível no mapa (isto é, arestas entre retângulos que se encontram no mesmo nível no grafo de inclusão) são traçadas como um único segmento de reta, enquanto aquelas que estão distantes de mais de um nível são traçadas como linhas constituídas por um conjunto de segmentos de retas. O objetivo final é minimizar o número de segmentos e aumentar o tamanho de segmentos de retas no sentido vertical tanto quanto possível;
5. **Balanceamento.** Os segmentos de retas ligados a um vértice $v \in V$ são traçados de forma a apresentar uma estética balanceada.

A cada regra de traçado é associada uma prioridade. Neste trabalho adota-se a seguinte priorização para as regras de traçado: $R_1 > R_2 > \ldots > R_5$ onde $R_i > R_j$ significa que R_i possui uma prioridade maior que R_j. Esta priorização pode ser justificada empiricamente da seguinte maneira. As três primeiras regras estão relacionadas primariamente com características topologicas de mapas, enquanto R_4 e R_5 estão associadas a métricas de traçado. Experimentos em leiaute de grafos têm demonstrado que a priorização acima produz mapas legíveis[7]. A Figura 5.6 ilustra um exemplo de um mapa traçado segundo as convenções e as regras de traçados citadas.

5.3.3 Adaptação para Xcharts

A definição da relação de adjacência para um dígrafo composto não permite que um vértice v tenha vértices adjacentes entre os seus ancestrais e descendentes. Entretanto, se representarmos um *Xchart* como um dígrafo composto, verificaremos que é possível a existência de vértices adjacentes entre vértices pertencentes a um mesmo caminho, isto

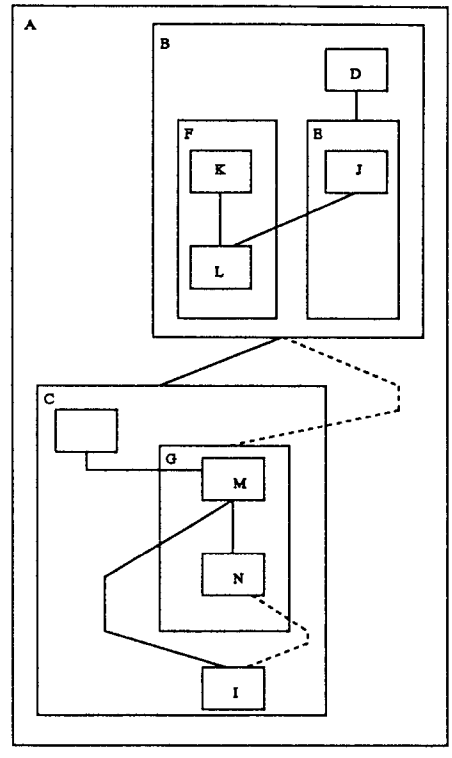


Figura 5.6: Traçado de um mapa.

é, entre ancestrais e descendentes de um vértice dado. A solução proposta para este problema exige uma adaptação do algoritmo proposto por Sugiyama [7]. A adaptação do algoritmo de Sugiyama efetuada no *Smart*, ocorre após o mapeamento de um *Xchart* para um dígrafo. As arestas que violam a *condição de adjacência* do dígrafo são removidas no início da fase de hierarquização e re-introduzidas no grafo durante a fase de leiaute métrico. A fase de leiaute métrico foi alterada para incluir o traçado de elementos gráficos particulares de *Xchart*. Um estado de um *Xchart* é marcado como estado inicial através do desenho de uma seta cuja ponta encosta no estado por fora. Além disso, estados em *Xchart* podem conter outras construções gráficas que exigem a adaptação do algoritmo de leiaute de dígrafos. O traçado destas construções provocou alterações no algoritmo da fase de leiaute métrico. Estas alterações permitem que a área reservada para o traçado de um estado seja aumentada ou diminuída em função do número de construções gráficas que ele deve acomodar.

5.4 A Classe Hierarchization

A classe *Hierarchization* é responsável pela implementação de um algoritmo de ordenação lexográfica que permite encontrar conjuntos de vértices (estados) irmãos, isto é, vértices de

mesmo nível. Estes conjuntos de vértices irmãos devem ser desenhados horizontalmente, de acordo com as convenções expostas na Seção 5.3.1.

5.4.1 A Hierarquização

Seja $D = (V, E, F)$ um dígrafo. Uma *Hierarquização* em D atribui um nível a cada $v \in V$. As definições seguintes permitem que rótulos (inteiros) sejam atribuídos a cada vértice do dígrafo e que, em seguida, estes rótulos sejam ordenados, definindo assim o nível, ou hierarquia, de traçado para cada vértice.

5.4.2 Definição de Nível

Sejam $\Sigma = (1, 2, 3, \ldots)$, $\Sigma^i = \{\text{seqüência de } i \text{ elementos de } \Sigma\}$, $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$. O comprimento de uma seqüência $s \in \Sigma^+$ é denotada por $compr(s)$. Definimos uma ordem lexográfica entre pares de elementos de Σ^+, como por exemplo: $(1,1,3) < (1,2,1)$, $(1,1,2) < (1,2)$, $(1,2) < (1,2,1)$, $(1,2,1) = (1,2,1)$ (Figura refclev). Reduzimos o problema de encontrar vértices irmãos ao problema de encontrar o mapeamento $ncp : V \rightarrow \Sigma^+$ que satisfaça as convenções de *inclusão* e *ligação de vértices* (Seção 5.3.1).

A convenção de *inclusão* pode ser expressa como:

$$\forall v \in V, ncp(v) \in \Sigma^{dep(v)} \tag{5.1}$$

$$\forall e = (v, w) \in E, ncp(w) = concat(ncp(v), s), s \in \Sigma. \tag{5.2}$$

Concat é a função que concatena um componente a uma seqüência.

A formulação matemática da convenção de *ligação de vértices* é um pouco mais complexa. Assim, ela pede algumas explicações adicionais. Para toda aresta de adjacência $e = (v, w) \in F$, podemos encontrar na árvore de inclusão D_ι um caminho único de v a w tal que:

$$p_m(= v), p_{m-1}, \ldots, p_1, t, q_1, \ldots, q_{n-1}, q_n(= w) \tag{5.3}$$

t é a raiz de topo ou vértice de profundidade minimal do caminho entre v e w. Isto significa que a aresta de adjacência e tem origem no retângulo de v, atravessa os vértices $p_{m-1}, \ldots, p_1$, passa por t, por $q_1, \ldots, q_{n-1}$ e termina em w. A convenção de *ligação de vértices* é formulada pela especificação de uma ordem para todo par (p_i, q_i) de vértices irmãos (mesma profundidade), para qualquer aresta $(v, w) \in F$ como:

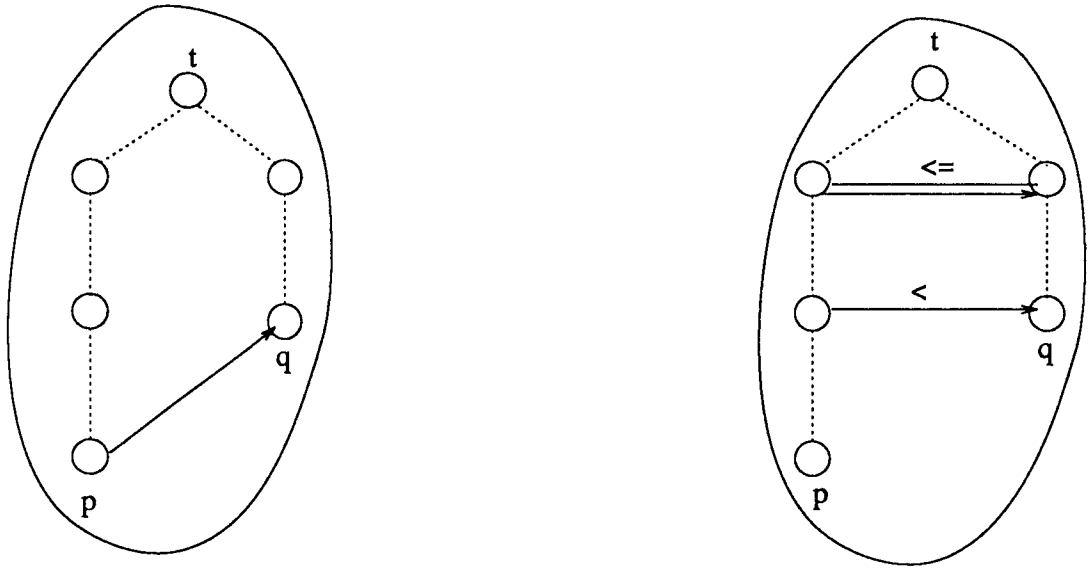


Figura 5.7: $ncp(p_n) < ncp(q_n), m > n$

$$\text{se} dep(v) > dep(w)(\text{ou } m > n), ncp(p_i) \leq ncp(q_i), i = 1, \ldots, n-1, \tag{5.4}$$

$$ncp(p_n) < ncp(w)(Figura\ 5.7); \tag{5.5}$$

$$\text{se} dep(v) \leq dep(w)(\text{ou } m \leq n), ncp(p_i) \leq ncp(q_i), i = 1, \ldots, m-1, \tag{5.6}$$

$$ncp(v) < ncp(q_m)(Figura\ 5.8). \tag{5.7}$$

Existe um mapeamento de vértices para níveis compostos para um dígrafo composto D se, e somente se, existe um mapeamento $ncp : V \rightarrow \Sigma^+$ que satisfaz as expressões 5.1, 5.2 e 5.4–5.7.

5.4.3 Algoritmo de Hierarquização

Há dígrafos compostos para os quais não existe um mapemento para níveis compostos, devido a existência de ciclos no dígrafo e às convenções de leiaute (Seção 5.3.1). Conseqüentemente, é necessário o desenvolvimento de um algoritmo que obtenha um traçado em níveis mesmo quando o mapeamento para níveis compostos não é possível. O algoritmo desenvolvido contém três fases: (i) substituição de arestas de adjacência, (ii) composição de níveis para os vértices e (iii) reversão da orientação de arestas adjacentes.

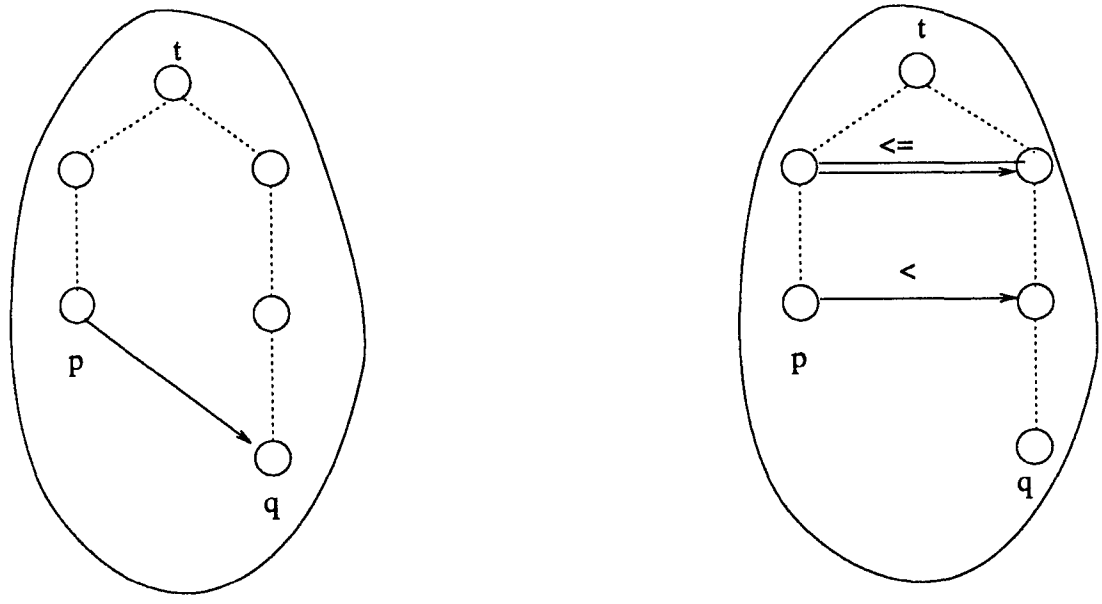


Figura 5.8: $ncp(p_m) < ncp(q_m), m \leq n$

5.4.4 Substituição de Arestas de Adjacência

Para descrever o código que implementa um método em *Xgraph* utilizaremos a sintaxe:

Nome do método(parâmetros de entrada ; parâmetros de saída)
início
  código
fim

Seja $D = (V, E, F)$ um dígrafo composto, então todas as arestas de adjacência são substituídas por dois tipos de arestas: $\rightarrow$ e $\Rightarrow$ que indicam as relações expressas por 5.5 e 5.7. O código do método que implementa a substituição de arestas pode ser expresso como:

ReplaceEdge $(D = (V, E, F)\ ; \mathcal{D} = (V, E, \mathcal{F}, \varphi)$ onde $\varphi : \mathcal{F} \rightarrow \{\rightarrow, \Rightarrow\})$
início
1. $\mathcal{F} = \emptyset$
2. **para** $\forall\ (u, w) \in F$
   2.1 **se** $m > n$ **então**
      2.1.1 ComposeEdge$(p_n, q_n, \rightarrow)$
      2.1.2 **para** $i = 1, \ldots, n-1$
         2.1.2.1 **ComposeEdge**$(p_i, q_i, \Rightarrow)$
   2.2 **caso contrário**
      2.2.1 ComposeEdge$(p_m, q_m, \rightarrow)$
      2.2.2 **para** $i = 1, \ldots, m-1$
         2.2.2.1 ComposeEdge$(p_i, q_i, \Rightarrow)$

fim

ReplaceEdge percorre todas as arestas de adjacência substituindo-as por dois tipos de arestas: $\rightarrow$ e $\Rightarrow$ que indicam as relações expressas por 5.5 e 5.7.

ComposeEdge $(p, q, type\ ; \mathcal{F} = \mathcal{F}\ \cup\ (p, q))$
início

1. **se** $(p, q) \notin F$ **então**
   1.1 Create(p, q)
   1.2 $\varphi(p, q) = type$
2 **caso contrário**
   2.1 $\varphi(p, q) = \&(p, q)$
3. $\mathcal{F} = \mathcal{F}\ \cup\ (p, q))$

fim

ComposeEdge compõe o tipo da aresta de adjacência segundo a regra de composição(&), caso a aresta de adjacência especificada exista ou cria uma nova aresta de adjacência com o tipo especificado, gerando um novo dígrafo de adjacência.

Regra de Composição (&)

&	$\rightarrow$	$\Rightarrow$
$\rightarrow$	$\rightarrow$	$\rightarrow$
$\Rightarrow$	$\rightarrow$	$\Rightarrow$

Comentários

Ao final do processo de substituição de arestas adjacentes, gera-se um dígrafo $\mathcal{D} = (V, E, \mathcal{F}, \varphi)$ onde $\varphi : \mathcal{F} \rightarrow \{\rightarrow, \Rightarrow\}$. É importante observar que:

- cada aresta em $\mathcal{F}$ liga dois vértices de mesma profundidade;
- uma aresta $(u, w) \in \mathcal{F}$ é denominada de aresta original, se $(u, w) \in F$, e de aresta derivada, se $(u, w) \notin F$.

Exemplo

A Figura 5.9 ilustra o processo de substituição de arestas de adjacência do dígrafo representado na Figura 5.5. A aresta de adjacência (b, c) é substituída por $b \rightarrow c$, (d, e) por $d \rightarrow e$), (g, b) por $c \rightarrow b$, (h, m) por $h \rightarrow g$, (i, n) por $i \rightarrow g$, (j, l) por $j \rightarrow l$ e $e \Rightarrow f$, (k, l) por $k \rightarrow l$, (m, i) por $g \rightarrow i$ e (m, n) por $m \rightarrow n$ onde as arestas de adjacências geradas $b \rightarrow c, d \rightarrow e, j \rightarrow l, k \rightarrow l$ e $m \rightarrow n$ são arestas originais, enquanto as arestas $e \Rightarrow f, c \rightarrow b, h \rightarrow g, g \rightarrow i$ e $i \rightarrow g$ são arestas derivadas. Como resultado todas as arestas de adjacência interligam vértices no mesmo nível na árvore de inclusão.

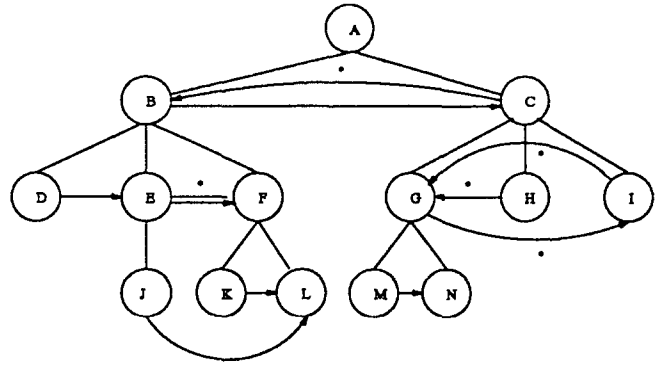


Figura 5.9: Dígrafo obtido após a substituição das arestas de adjacência

5.4.5 A Operação de Composição de Níveis para os Vértices de um Dígrafo

Seja $\mathcal{D} = (V, E, \mathcal{F}, \varphi)$, então pode-se assumir que o $ncp(raiz) = 1$, sem perda de generalidade e então pode-se atribuir níveis para todos os vértices de V, recursivamente do nó raiz para os nós folhas em $\mathcal{D}$.

AttributeLevel $(\mathcal{D} = (V, E, \mathcal{F}, \varphi)\ ; ncp : V \rightarrow \Sigma^+)$
início
1. $ncp(raiz) = 1$
2. **para** $\forall\, v \in V$
   2.1 $w \cup \forall\, v \equiv ncp$
   2.2 $U = Ch(w)$
   2.3 **se** $U \neq \emptyset$ **então**
      2.3.1 $H = \{(u, w) \in \mathcal{F} \mid u, w \in U\}$
      2.3.2 $\eta : \{\rightarrow, \Rightarrow\}$
      2.3.3 ComposeLevel$(\iota = (U, H, \eta))$

fim

AttributeLevel atribui o $ncp(raiz) = 1$ e percorre todos os vértices de V, recursivamente do nó raiz para os nós folhas em $\mathcal{D}$ chamando *ComposeLevel* para atribuir um nível para os nós filhos de todos os nós que possuem o mesmo *ncp*.

ComposeLevel $(\iota = (U, H, \eta)\ ; ncp : U \rightarrow \Sigma^+)$
início
1. W = componentes fortemente conexas [1] de ι
2. **se** $W \neq \emptyset$ **então**
   2.1 RemoveCycle(W)
3. $m = 1$
4. **enquanto** $U \neq \emptyset$
   4.1 $L_m = \{\ u \in U \mid \{\ w \in U \mid (w,u) \in H\ \} = \emptyset$
   4.2 $U = U - L_i$
   4.3 $m = m + 1$
5. **para** $\forall\ u \in L_1$
   5.1 $ncp(u) = append(ncp(Pai(u)), 1)$
6. **para** $l = 2 \ldots m$
   6.1 **para** $\forall\ u \in L_l$
      6.1.1 $l_u = max(l_v \mid (v,u) \in H,\ v \in L_1 \cup L_{m-1})$
         $l_v = \text{dep}(v) + 1$ if $\varphi(v,u) = \rightarrow$
         ou $l_v = \text{dep}(v)$ if $\varphi(v,u) = \Rightarrow$
      6.1.2 $ncp(u) = append(ncp(Pai(u)), l_u)$
7. **se** $p_V \neq \emptyset$ **então**
   7.1 **para** $\forall u \in p_v$
      7.1.1 $ncp(u) = ncp(p_v)$
fim

ComposeLevel verifica a presença de componentes fortemente conexas de ι. Caso existam, então *RemoveCycle* será responsável pela eliminação dos ciclos existentes nestas componentes. Após a eliminação das componentes fortemente conexas, *ComposeLevel* subdivide todos os vétrice pertencentes a U em blocos denominados nível (L_i), tal que, todos os vértices pertencentes a L_1 possuirão o *ncp* do nó pai concatenado a "1", enquanto os vértices dos demais blocos L_i dependerão dos vértices que constituem a origem destes vértices nas arestas de adjacências de F. *ComposeLevel* verifica a existências de vértices empacotados em um único vértice, gerados por *RemoveCycle*, atribuindo o *ncp* deste vértice único aos demais vértices pertencentes ao empacotamento.

RemoveCycle $(C\ ;H)$
início
1. **enquanto** $C \neq \emptyset$
   1.1 **se** $\forall\, \varphi(u,v) \in W \Rightarrow$ **então**
      1.1.1 $p_v = W$
      1.1.2 $W = \emptyset$
   1.2 **caso contrário se** $\exists\, \varphi(u,v) \in W \Rightarrow$ **então**
      1.2.1 $H = H - (u,v)$
   1.3 **caso contrário se** $\exists\, \varphi(u,v) \in W \rightarrow e\ (u,v) \in \mathcal{F}$ **então**
      1.3.1 $H = H - (u,v)$
   1.4 **caso contrário se** $\exists\, \varphi(u,v) \in W \rightarrow e\ (u,v) \in F$ **então**
      1.4.1 $H = H - (u,v)$
   1.5 $C =$ componentes fortemente conexas de ι
fim

RemoveCycle remove a retro-adjacência mínima das componentes fortemente conexas, segundo os tipos das arestas que compõe estas componentes.

Comentários

A remoção da retro-adjacência mínima de uma componente fortemente conexa é condição necessária e suficiente para transformar esta componente em uma árvore; este problema é NP-Completo [6]. Portanto, o método *RemoveCycle* utiliza a seguinte heurística para efetuar a remoção de ciclos: arestas do tipo *Rightarrow* são eliminadas antes de arestas do tipo *rightarrow* não pertencentes ao conjunto original de arestas F. Estas últimas são eliminadas antes das arestas que pertencem ao conjunto de arestas originais F do dígrafo composto.

Exemplo

A Figura 5.10 ilustra o processo de atribuição de uma composição de níveis para todos os vértices do dígrafo ilustrado na Figura 5.5. Atribuindo $ncp(a) = 1$ constitui-se o subgrafo $\Gamma = (U, H, \eta)$ onde $U = \{b,c\}, H = \{(b,c),(c,b)\}$ e $\eta((b,c)) = \eta((c,b)) = \rightarrow$. Em Γ há apenas um única componente conexa $\{b,c\}$ possuindo arestas do tipo $\rightarrow$ que diferem do tipo $\Rightarrow$, portanto pela regra de eliminação elimina-se a aresta (c,b). Repetindo a busca por componentes conexas encontra-se as componentes $\{b\}$ e $\{c\}$, enumeradas como 1 e 2, respectivamente. A composição de nível para o vértice b será $ncp(b) = (1,1)$ e do vértice

c será $ncp(c) = (1, 2)$. Na Figura 5.10 pode-se verificar a execução do algoritmo para composição de nível para todos os vértices do dígrafo ilustrado em 5.5.

5.4.6 A Operação de Reversão da Orientação de Arestas de Adjacência

Cada aresta de adjacência (v, w) de uma composição de dígrafos $\mathcal{D} = (V, E, \mathcal{F}, ncp)$ é verificada para testar se $ncp(v) < ncp(w)$. Caso esta condição não seja satisfeita, a orientação da aresta é invertida. O resultado final é um dígrafo com atribuição de níveis $\mathcal{D} = (V, E, \mathcal{F}', ncp)$.

RevertOrientation $(\mathcal{D} = (V, E, F, ncp)\ ; \mathcal{D} = (V, E, F', ncp))$
início
1. $F' = \emptyset$
2. **para** $\forall\ (u, v)\ \in\ F$
   2.1 **se** $ncp(u) > ncp(v)$ **então**
      2.1.1 $F' = F'\ \cup\ (v, u)$
   2.2 $F' = F'\ \cup\ (u, v)$

fim

RevertOrientation altera o sentido das arestas de adjacência, tal que, qualquer aresta de adjacência pertencente a F apresentará o vértice origem com menor ncp em relação a vértice destino.

Exemplo

No dígrafo ilustrado na Figura 5.5, a orientação das arestas de adjacência (g, b) e (i, n) são invertidas para (b, g) e (n, i), pelo fato de que $ncp(g) = (1, 2, 2) > ncp(b) = (1, 1, 1)$ e $ncp(i) = (1, 2, 3) > ncp(1, 2, 2, 2)$ respectivamente. A Figura 5.11 mostra como o dígrafo da Figura 5.5 após a aplicação do algoritmo de hierarquização.

5.5 A Classe Normalization

A Normalização converte uma composição hierárquica em uma composição hierárquica própria. Esta converção é efetuada reposicionando todas as arestas de adjacência que não sejam próprias por vértices fictícios, arestas de inclusão fictícias e arestas de adjacências fictícias.

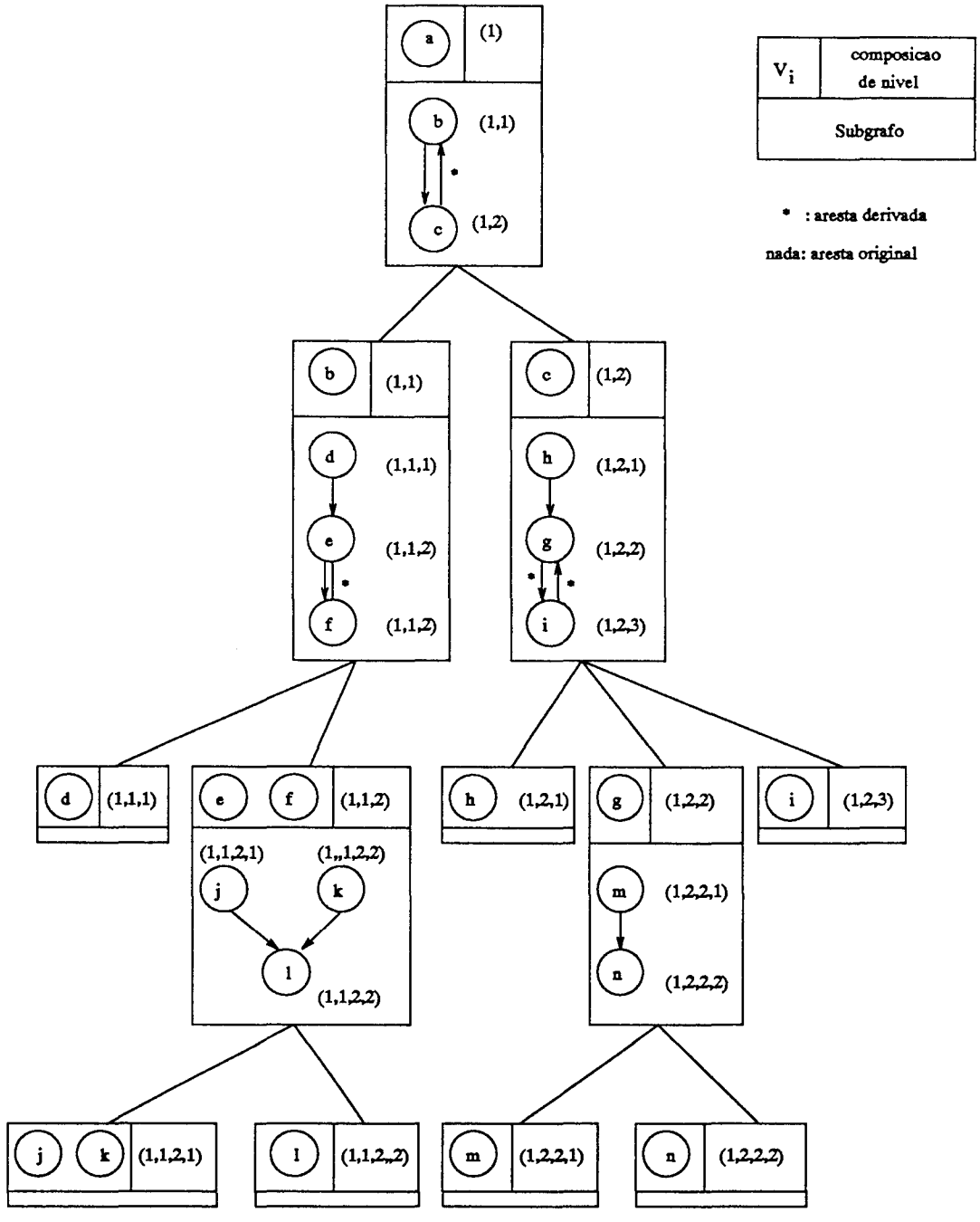


Figura 5.10: Atribuição da composição de nível para um dígrafo

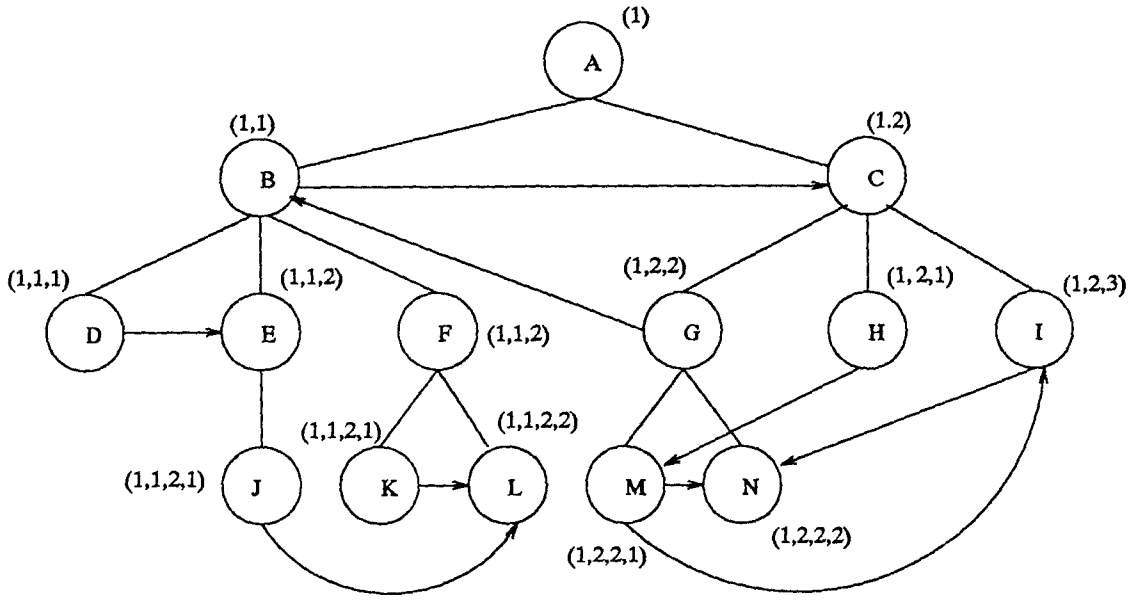


Figura 5.11: Dígrafo após a aplicação do algoritmo de hierarquização

5.5.1 Definição de Arestas Próprias

Seja $D = (V, E, F, ncp)$ uma composição de dígrafo com uma atribuição de níveis para os seus vértices, então $ncp(v) = (c_1, \ldots, c_d)$ para todo vértice $v \in V$, onde d é a profundidade do vértice v. Seja a função *last* responsável pelo retorno do último inteiro c_d da seqüência $(c_1, \ldots, c_d)$, $last(ncp(v)) = c_d$. Então uma aresta de adjacência $(v, w) \in F$ é denominada própria se:

$$ncp(Pai(v)) = ncp(Pai(w))$$
$$last(ncp(v)) = last(ncp(w)) - 1$$

Portanto, uma composição de dígrafo com atribuição de níveis $D = (V, E, F, ncp)$ é denominada uma composição própria de dígrafos se e somente se, todas as arestas de adjacência em D são próprias.

Normalize $(D = (V, E, F, ncp)\ ; D = (V, E', F', ncp))$
início
1. $E' = \emptyset$
2. $F' = \emptyset$
3. **para** $\forall\,(u, v)\ \in\ F$
   - 3.1 **se** $dep(u) < dep(v)$ **então**
     - 3.1.1 NormalizeSmallerGreater(u, v) em D
   - 3.2 **caso contrário se** $dep(u) > dep(v)$ **então**
     - 3.2.1 NormalizeGreaterSmaller(u, v) em D
   - 3.3 **caso contrário se** $ncp(Pai(u)) = ncp(Pai(v))$
     - 3.3.1 NormalizeSame(u, v)
   - 3.4 $E' = E'\ \cup\ u\ \cup\ v$

fim

Normalize efetua a normalização de todas as arestas de adjacência pertencentes a F segundo a sua profunidade.

NormalizeSmallerGreater $(u, v\ E', F')$
início
1. Create a dummy n
2. $ncp(n) = ncp(v) - 1$
3. $E' = E' \cup n$
4. Create a dummy (n, v)
5. $F' = F' \cup (n, v)$
6. $v = n$
7. **enquanto** $dep(u) \neq dep(v)$
   - 7.1 Create a dummy n
   - 7.2 $ncp(n) =$ remove level $dencp(v)$
   - 7.3 $E' = E' \cup n$
   - 7.4 $Ch(n) = v$
   - 7.5 $v = n$
8. $v = Filho(Pai(u))$
9. **enquanto** $last(ncp(u)) \neq last(ncp(v)) - 1$
   - 9.1 Create a dummy n
   - 9.2 $ncp(n) = ncp(v) - 1$
   - 9.3 $E' = E' \cup n$
   - 9.4 Create a dummy (n, v)
   - 9.5 $F' = F' \cup (n, v)$
   - 9.6 $v = n$
10. Create a dummy (u, n)
11. $F' = F' \cup (u, n)$

fim

NormalizeSmallerGreater efetua a normalização das arestas de adjacência que possuam o vértice origem com menor profunidade que o vértice destino, acrescentando vértices e arestas de adjacência fictícias.

NormalizeGreaterSmaller $(u, v\ E', F')$
início
1. Create a dummy n
2. $ncp(n) = ncp(u) + 1$
3. $E' = E' \cup n$
4. Create a dummy (u, n)
5. $F' = F' \cup (u, n)$
6. $u = n$
7. **enquanto** $dep(u) \neq dep(v)$
   - 7.1 Create a dummy n
   - 7.2 $ncp(n) =$ remove level $dencp(u)$
   - 7.3 $E' = E' \cup n$
   - 7.4 $Ch(n) = u$
   - 7.5 $u = n$
8. $u = Filho(Pai(v))$
9. **enquanto** $last(ncp(u)) \neq last(ncp(v)) - 1$
   - 9.1 Create a dummy n
   - 9.2 $ncp(n) = ncp(u) + 1$
   - 9.3 $E' = E' \cup n$
   - 9.4 Create a dummy (u, n)
   - 9.5 $F' = F' \cup (u, n)$
   - 9.6 $u = n$
10. Create a dummy (n, v)
11. $F' = F' \cup (n, v)$

fim

NormalizeGreaterSmaller efetua a normalização das arestas de adjacência que possuam o vértice origem com maior profunidade que o vértice destino, acrescentando vértices e arestas fictícias.

NormalizeSame $(u, v\ E', F')$
início
1. Se $ncp(u) > ncp(v)$ então
   1.1 Change(u, v)
2. **enquanto** $last(ncp(u)) \neq last(ncp(v)) - 1$
   2.1 Create a dummy n
   2.2 $ncp(n) = ncp(u) + 1$
   2.3 $E' = E' \cup n$
   2.4 Create a dummy (u, n)
   2.5 $F' = F' \cup (u, n)$
   2.6 $n = Filho(Pai(u))$
   2.7 $u = n$

fim

NormalizeSame efetua a normalização de arestas das adjacência que possuam o vértice origem com igual profunidade que o vértice destino podendo ser acrescentados vértices e arestas de adjacências fictícias.

Comentários

Nesta fase todas as arestas de adjacência conectam vértices que se encontram na mesma profundidade e obtêm-se um dígrafo próprio $D = (V, E', F', ncp)$. O tempo de complexidade desta fase é linear ao número de arestas de adjacência. O objetivo desta etapa é minimizar o número de cruzamentos entre linhas e retâgulos.

Exemplo

O dígrafo $D = (V, E, F, ncp)$ ilustrado na Figura 5.11 apresenta arestas de adjacência $(b, g), (h, m), (m, i)$ e (n, i) que não são próprias, portanto estas arestas deverão ser substituídas por:

$$b(1.1) \rightarrow o(1.2) \supset p(1.2.1) \rightarrow g(1.2.2)$$

$$h(1.2.1) \rightarrow q(1.2.2) \supset r(1.2.2.0) \rightarrow m(1.2.2.1)$$

$$m(1.2.2.1) \rightarrow s(1.2.2.2) \subset t(1.2.2) \rightarrow i(1.2.3)$$

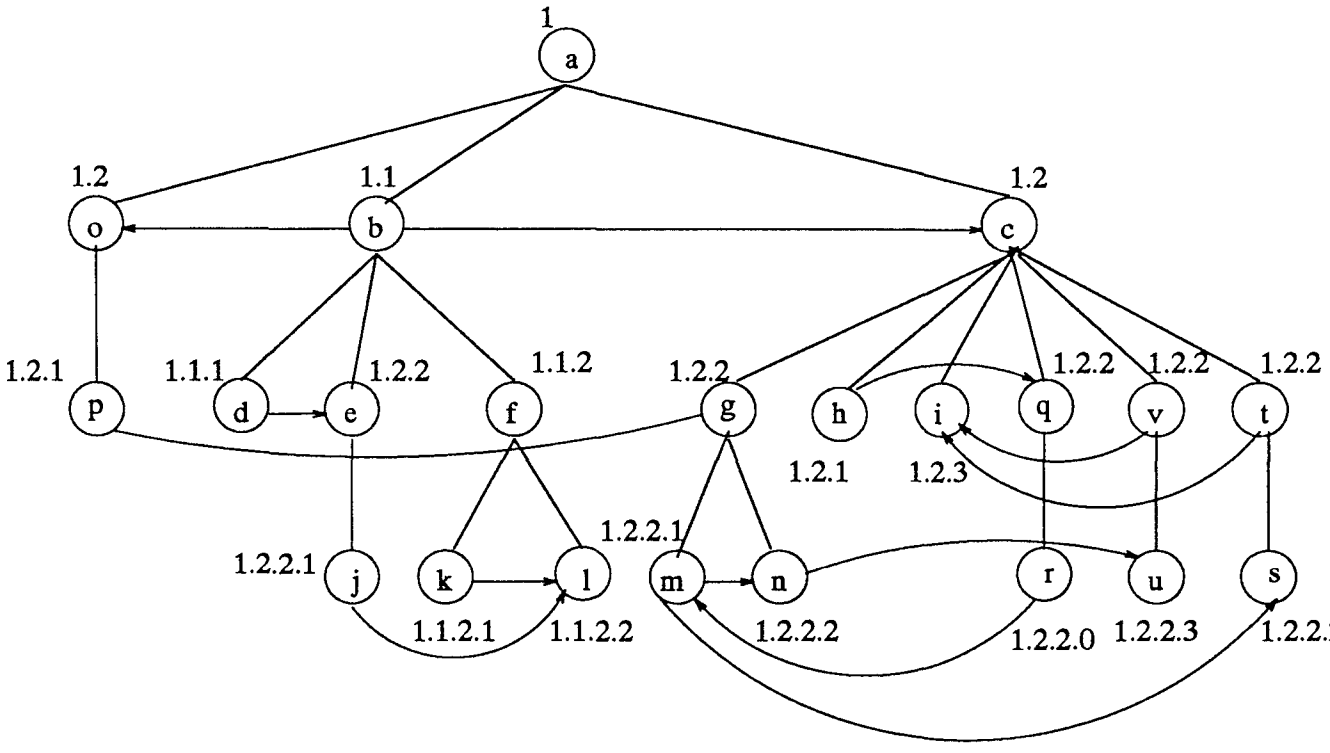


Figura 5.12: Normalização do dígrafo em 5.11.

$$n(1.2.2.2) \rightarrow u(1.2.2.3) \subset v(1.2.2) \rightarrow i(1.2.3)$$

onde o, p, q, r, s, t e v são vértices fictícios e os símbolos $\subset, supset$ indicam arestas de inclusão fictícias. A Figura 5.12 ilustra o dígrafo D após a normalização. A Figura 5.13(a) ilustra o traçado da aresta (h, m) após o processo de normalização, enquanto a Figura 5.13(b) ilustra o traçado final no mapa da respectiva aresta.

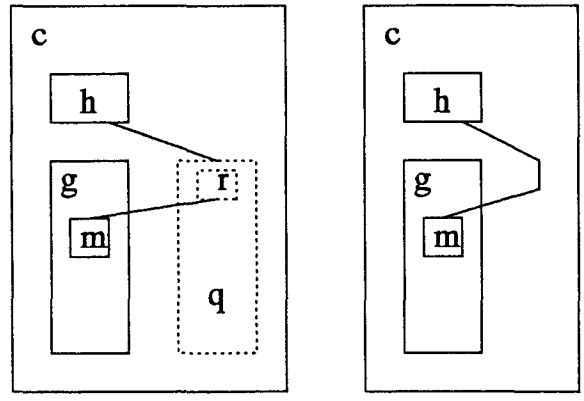


Figura 5.13: Comparação da normalização e o mapa final de um dígrafo

5.6 A Classe VertexOrder

No processo de hierarquização alguns vértices podem possuir a mesma posição no nível de composição. O método *Hierarchization* não trata deste casos, apenas identifica todos os vértices de cada nível que possuem as mesmas posições. Após o processo de normalização, podem ocorrer também vértices que possuem posições coincidentes, resultantes do método *Normalization*. Para posicionar de forma correta estes vértices aplica-se uma reordenação hierárquica local no grafo de inclusão, com o objetivo de reposicionar os vértices que apresentam as mesmas posições hierárquicas. Em cada hierarquia local, é efetuada uma ordenação dos vértices. Esta ordenação é resultado da permutação dos vértices que possuem uma mesma posição. A ordenação, também, verifica as regras de proximidade, cruzamento de linha bem como cruzamento de linha e retângulos entre os vértices da permutação. O tempo de complexidade do problema para minimizar o cruzamento de linhas é NP-Completo, mesmo quando o número de níveis da hierarquia local considerado é apenas dois [16]. O problema de minimizar o cruzamento de linhas e retângulos é equivalente ao problema de arranjo linear [11], cujo tempo de complexidade também é NP-Completo. Por este motivo o algoritmo de Sugiyama adota métodos heurísticos para resolvê-los.

5.6.1 Definição da Ordenação de um Vértice

Seja $D = (V, E, F, ncp)$ um dígrafo próprio e $W = V - \{folhas\} = \{v_1, \ldots, v_N\}$. Suponha que os filhos de $Ch(v)$ $\forall v \in W$ possam ser particionados em $n(v)$ subconjuntos de acordo com seus níveis, isto é:

$$Ch(v) = V_1(v) \cup \ldots \cup V_i(v) \cup \ldots \cup V_{n(v)}(v)$$

onde $V_i(v) = \{u \in Ch(v) \mid tail(ncp(v)) = i\}$ e $V_i(v)$ é denominado *ith* nível. Um dígrafo ordenado é definido por

$$D(\iota) = (V, E, F, ncp, \iota)$$

onde $\iota = (\iota(v_1), \ldots, \iota(v_N))$, $tau(v_j) = (\iota_1(v_j), \ldots, \iota_{n(v_j)}(v_j))$ e $\iota_i(v_j)$ estabelece a ordem de todos os vértices de cada $V_i(v_j)$.

Seja $S = (S(v_1), \ldots, S(v_N))$, $S(v_j) = S_1(v_J) \times \ldots \times S_{n(v_j)}(V_j)$ e $S_i(v_j)$ ser um conjunto de todas as possíveis ordens de $\iota_i(v_j)$. Então o problema de verificar as regras de proximidade, cruzamento de linhas e cruzamento de linhas e retângulos é definido como

$$P: \; min \; \{c_1 C(D(\iota)) + c_2 K(D(\iota)) + c_3 Q(D(\iota)) \mid \iota \in S\}$$

onde:

1. $C(D(\iota))$,$K(D(\iota))$ e $Q(D(\iota))$ são medidas quantitativas para verificar as regras proximidade, cruzamento de linhas e cruzamento de linhas e retângulos, respectivamente;

2. c_1,c_2 e c_3 são pesos constantes que satisfazem a condição $c_1 + c_2 + c_3 = 1$

Para resolver o problema P o algoritmo de Sugiyama decompõe P em subproblemas de hierarquia local de $D(\iota)$

$$P(v_j): \; min \; \{c_1 C(H(\iota(v_j))) + c_2 K(H(\iota(v_j))) + c_3 Q(H(\iota(v_j))) \mid \iota(v_j) \in S(v_j)\},$$

$$j = 1, \ldots, N.$$

Hierarquia Local de um Vértice

Em um dígrafo ordenado $D(\iota)$, seja $A(v)$ um conjunto de vértices (exceto v) cujo nível é $ncp(v)$. Então $A(v)$ pode ser particionado em $A^L(v)$ e $A^R(v)$ onde todo vértice em $A^L(v)$ e $A^R(v)$ está a esquerda e a direita de v em ι respectivamente. Então uma hierarquia local de um vértice $v \in V - \{folhas\}$

$$H(\iota(v)) = (Ch(v), F(v), n(v), \iota(v), \lambda, \rho, \omega)$$

onde:

1. $F(v)$ constitui de dois conjuntos:
   (a) F^d é o conjunto de arestas entre níveis diferentes, portanto $F^d = \{(u,w) \in F \mid u,w \in Ch(v)\}$.
   (b) F^s é o conjunto de arestas no mesmo nível, portanto $F^s = \{(u,w) \mid (x,y) \in F,\ x \in De(u) - \{u\}, y \in De(w) - \{w\}, u,w \in Ch(v)\}$.
2. $\lambda(w), \rho(w)$ representa o número de arestas entre os descendentes de $Ch(v)$ e descendentes de $A^L(v)$ ou $A^R(v)\ \forall w \in Ch(v)$ respectivamente.
3. $\omega(u,w)$ é a multiplicidade de arestas $(u,w) \in F^s$.

Aspectos Teóricos e Heurísticos de P

Os subproblemas de hierarquia local de P relacionam as minimizações de:

1. $C(H(\iota))$ que indica proximidade. Esta otimização é realizada calculando para cada vértice $v \in V$, $t = \lambda(v) - \rho(v)$ e ordenando os vértice em $H(\iota)$ em ordem crescente de t e, é denominada método *Split*;
2. $K(H(\iota))$ que indica cruzamento de linhas. Esta otimização é equivalente ao problema de encontrar o conjunto mínimo de arestas de retro-adjacência que é NP-Completo [16]. Conseqüentemente, o algoritmo de Sugiyama desenvolveu um método heurístico denominado método *Barycentric* (BC). Este método aplica a ordeção baricêntrica dos vértices no primeiro e segundo níveis alternadamente. Seja $Ft(w) \subset F^d$ o conjunto de arestas com destino em w e $Fo(w) \subset F^d$ o conjunto de arestas com origem em w, tem-se que:

$$B_u(w) = \begin{cases} \frac{\sum_{(u,w) \in Ft(w)} ordem(u)}{|Ft(w)|} & \text{Se } |\ Ft(w)\ | > 0 \\ 0 & \text{caso contrário} \end{cases}$$

$$B_l(w) = \begin{cases} \frac{\sum_{(w,u) \in Fo(w)} ordem(u)}{|Fo(w)|} & \text{Se } |\ Fo(w)\ | > 0 \\ 0 & \text{caso contrário} \end{cases}$$

Onde $B_u(w)$ significa baricentro superior enquanto $B_l(w)$ significa baricentro inferior de w. A Figura 5.14 ilustra o método. Na Figura 5.14 o baricentro superior B_u do vértice f, por exemplo, é calculado com $2.0 = (1+2+3)/3$ porque o vértice f está conectado ao primeiro, segundo e terceiro vértice (a,b,c) do primeiro nível. Como os baricentros superiores dos vértices no segundo nível 5.14.a não apresentam ordenados em ordem crescente, é aplicada a ordenação baricêntrica e obtêm-se

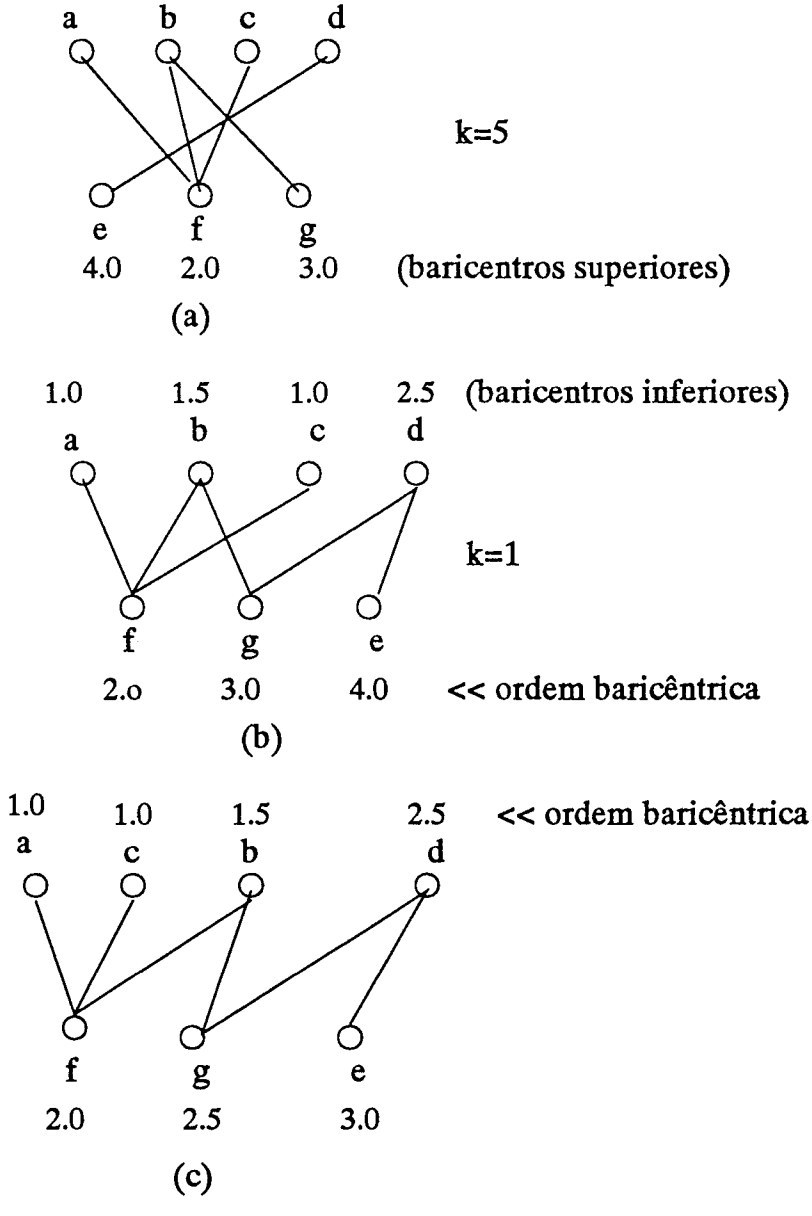


Figura 5.14: Aplicação do método Barycentric

5.14.b. Da mesma forma, a ordenação baricêntrica é aplicada aos vértices do primeiro nível em 5.14.b utilizando os valores dos seus baricentos inferiores e obtêm-se 5.14.c. Como resultado, o número de cruzamentos k é reduzido de 5 para 0.

3. $Q(H(\iota))$ que indica cruzamento entre linhas e retângulos. Por simplicidade, a multiplicidade de arestas w. Esta otimização refere-se apenas a um nível e é equivalente ao problema de arranjo linear [11], cuja complexidade é NP-Complete desde que cada aresta em $H(\iota)$ o número de cruzamentos entre arestas e retângulos é igual ao tamanho da aresta menos 1 [11]. Conseqüentemente o algoritmo de Sugiyama desenvolveu um método heurístico denominado método *InsertBarycentric* (IBC). A Figura 5.15 ilustra o método. Na Figura 5.15 $H(\iota)$ onde $\iota = (g, h, i, j, k)$ é ilustrado em 5.15.a. Em 5.15.b os vértices w, x, y e z são inseridos e ordenados de acordo com seus baricentros em (w, x, y, z). então a ordenação baricêntrica é aplicada aos vértices do segundo nível e temos $\iota = (i, g, j, k, h)$ em 5.15.c, enquanto 5.15.d ilustra a solução ótima.

5.6.2 Algoritmo

VertexOrder $(D = (V, E, F, ncp), v \in E\ ; D(\iota) = (V, E, F, ncp, \iota))$
início
1. **se** $Ch(v) \neq 0$ **então**
   - 1.1 Create $H(\iota(v))$
   - 1.2 Split $H(\iota(v))$
   - 1.3 LocalOrder $H(\iota(v))$
   - 1.4 Replace $(Ch(v))$
   - 1.5 **para** $\forall\, w \in Ch(v)$
     - 1.5.1 VertexOrder$(D, w\ ; D(\iota))$

fim

VertexOrder percorre de forma recursiva todos os vértices da raiz até as folhas em D criando a hierarquia de todos os vértices filhos do vértice visitado. *VertexOrder* aplica a orderanção desta hierarquia e reposiciona os os vértices em E, de acordo com a ordenação realizada.

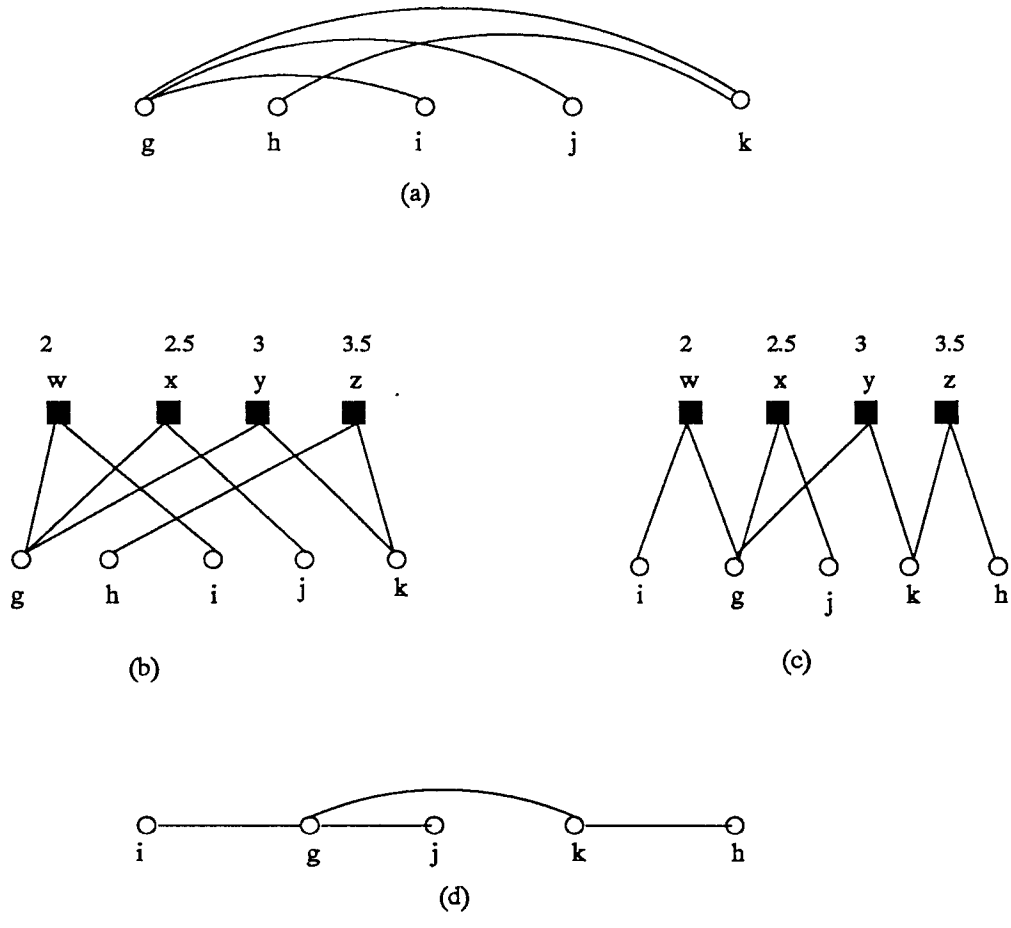


Figura 5.15: Aplicação do método InsertBarycentric

Split $(H(\iota(v)) = (Ch(v), F(v), n(v), \iota(v), \lambda, \rho, \omega)$;
$H(\iota(v)) = (Ch(v), F'(v), n(v), \iota'(v), \lambda, \rho, \omega'))$
início
1. **para** $\forall\, w \in Ch(v)$
   1.1 $t(w) = \lambda(v) - \rho(v)$
2. Order$(\iota(v), t(w), w \in Ch(v))$
3. Eliminate $\forall\, w$ with $t(w) \neq 0$ and $\forall\, (u, w)$ and $(w, u) \in F^d$

fim

Split fixa as posições dos vértices de uma hierarquia que possuam os valores de λ e ρ diferentes de zero.

LocalOrder $(H(\iota_1, ldots, \iota_n)\,; H(\iota_1', ldots, \iota_n'))$
início
1. $I = 1$
2. **enquanto** $I \neq 0$
   2.1 $I = 0$
   2.2 InsertBarycenterUpper(1)
   2.3 $B_u(1)$
   2.4 **para** $j = 2 \ldots n$
      2.4.1 $B_u(j)$
      2.4.2 InsertBarycenterUpper(j)
      2.4.3 $B_u(j)$
   2.5 InsertBarycenterLower(n)
   2.6 $B_l(n)$
   2.7 **para** $j = n - 1 \ldots 1$
      2.7.1 $B_l(j)$
      2.7.2 InsertBarycenterLower(j)
      2.7.3 $B_l(j)$

fim

LocalOrder aplica o método *Barycentric* e *InsertBarycentric* para eliminação de cruzamento de linhas e o cruzamento de linhas e retângulos entre os vértices de uma hierarquia.

5.6.3 Exemplo

A Figura 5.16 ilustra a ordenação de vértices do dígrafo $D = (V, E, F, ncp)$ da Figura 5.12. Nota-se que os vértices o e g com os respectivos vértices de inclusão e as respectivas arestas de adjacência.

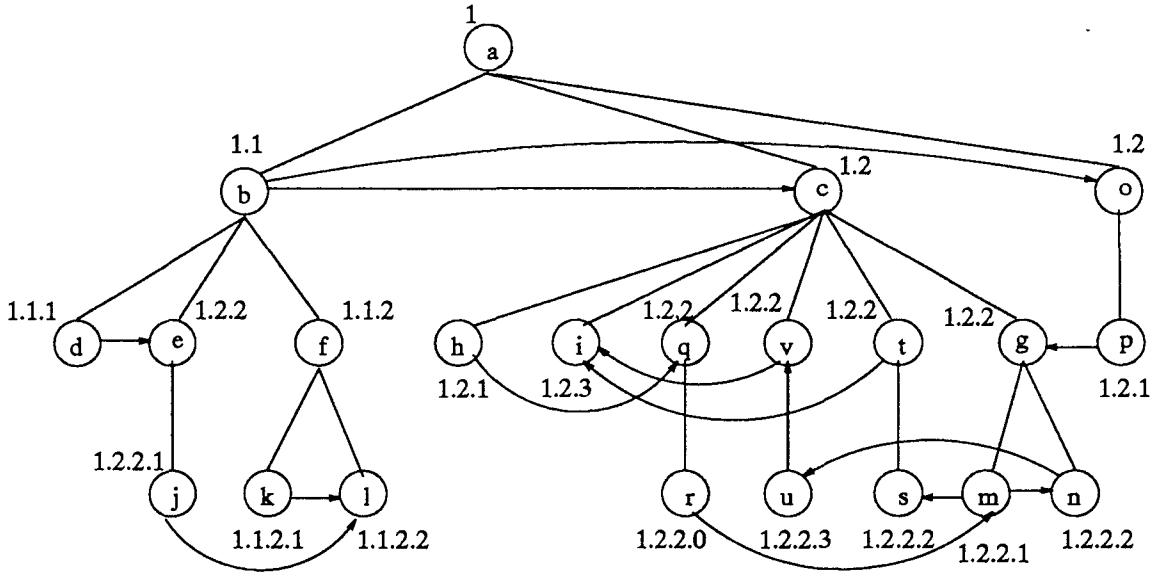


Figura 5.16: Ordenação dos vértices do dígrafo 5.12.

5.7 A Classe MetricalLayout

As métricas de leiaute determinam a posição horizontal, vertical, comprimento e altura de cada vértice. O algoritmo de leiaute métrico cuida do posicionamento de cada vértice e, em seguida, do traçado das arestas $(u, v) \in F$: (i) como linhas retas, se o vértices u e v forem vértices originais do grafo ou como segmentos de retas, se o vértices u e v forem vértices fictícios do grafo, pois o comprimento e altura destes vértices são atribuídos a zero.

As métricas de leiaute de um retângulo (posições horizontais, verticais, comprimento e altura) são subdividida em duas: (i) **métrica horizontal** que calcula a posição horizontal e o comprimento de retânguos e (ii) **métrica vertical** que calcula a posição vertical e a altura dos retângulos.

Seja $D(\iota) = (V, E, F, clev, \iota)$ então as métricas de leiaute do dígrafo são definidas por:

$$MD = (V.E, F, ncp, \iota, \chi, \Psi, \delta, \eta, d_1, d_2, d_3, d_4)$$

onde:

1. $\chi, \Psi : V \rightarrow R$ e para cada $v \in V$ $\chi(v)$ e $\Psi(v)$ correspondem as coordenadas (x, y) do centro do retângulo correspondente a v;

2. $\delta, \eta :\rightarrow R^+$ e para cada $v \in V$ $\delta(v)$ e $\eta(v)$ correspondem ao comprimento e largura do retângulo correspondente a v;

3. d_1, d_2, d_3, d_4 são parâmetros que especificam as distância básicas em um mapa.

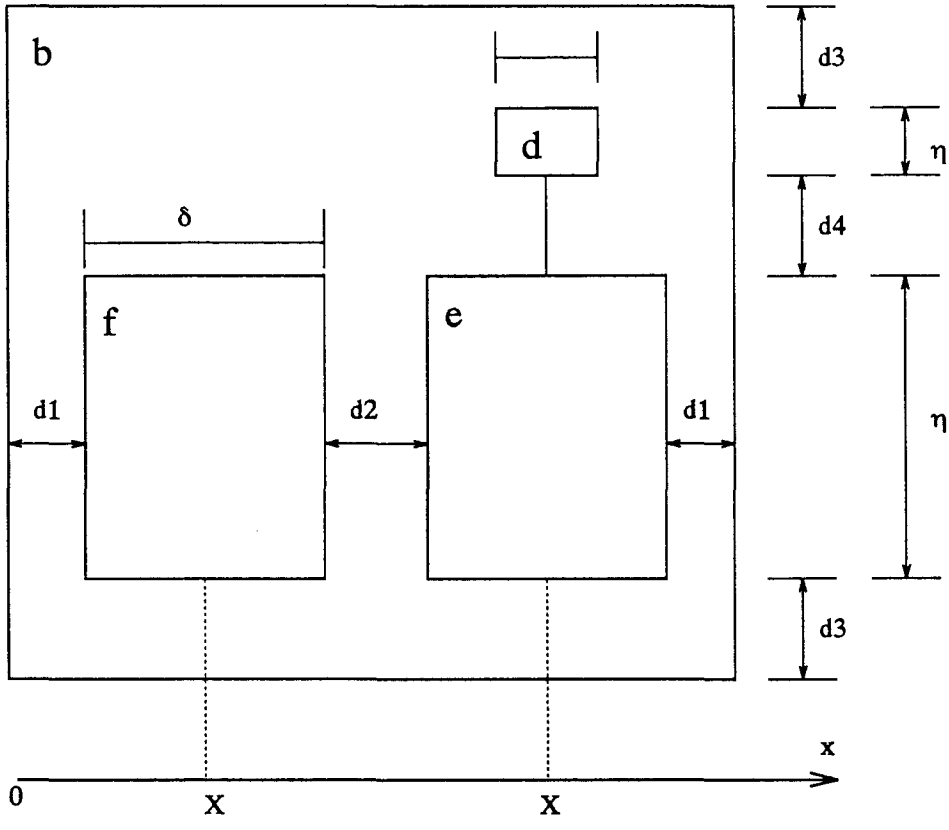


Figura 5.17: Partes de um Mapa

A Figura 5.17 ilustra a definição de métricas de leiaute de um dígrafo. O algoritmo de Sugiyama concentra-se no problema horizontal, pois o problema vertical não possui nenhuma relação com as regras de traçado estabelecidas. Portanto, as métricas de leiaute de um dígrafo é definido como:

$$MD = MD(\chi, \delta) = (V, E, F, ncp, \iota, \chi, \delta, d_1, d_2)$$

Para calcular a MD de um dígrafo, o algoritmo de Sugiyama decompõe o problema MD em subproblemas denominados métricas de um hierarquia local definida como:

$$MD = MH(v_1) \ \ldots \ MH(v_N)$$

Métricas para uma Hierarquia Local

Seja $D = (v, E, f, ncp, \iota)$ para todo vértice $v \in V - \{folhas\}$ é definido por:

$$MH(v) = MH(x, \delta) = (Ch(v), F^d(v), n(v), \iota(v), x, \delta, d_1, d_2)$$

onde:

1. $x : Ch(v) \rightarrow R^+$ indica a coordenada (x) do centro de um retângulo correspondente ao vértice $w \in Ch(w)$;

2. $\delta : Ch(v) \rightarrow R^+$ indica o comprimento do retângulo correspondente ao vértice $w \in Ch(w)$.

Em $MH(v)$ seja $Ft(w) \subset F^d$ o conjunto de arestas com destino em w e $Fo(w) \subset F^d$ o conjunto de arestas com origem em w, tem-se que:

$$Ft(w) = \{(u, w) \in F^d(v) \mid u \in Ch(v)\}$$

$$Fo(w) = \{(w, u) \in F^d(v) \mid u \in Ch(v)\}$$

e $\mid Ft(w) \mid$ e $\mid Fo(w) \mid$ são denominados de conectividade superior e inferior, respectivamente. Define-se, também, $B_u(w)$ e $B_l(w)$, tal que:

$$B_u(w) = \begin{cases} \frac{\sum_{(u,w) \in Ft(w)} x(u)}{|Ft(w)|} & \text{Se } \mid Ft(w) \mid > 0 \\ x(w) & \text{caso contrário} \end{cases}$$

$$B_l(w) = \begin{cases} \frac{\sum_{(w,u) \in Fo(w)} x(u)}{|Fo(w)|} & \text{Se } \mid Fo(w) \mid > 0 \\ x(w) & \text{caso contrário} \end{cases}$$

O método de programação quadrática resolve o problema $MH(v)$, porém este método consome muito tempo para ser resolvido, portanto o algoritmo de Sugiyama desenvolveu um método heurístico denominado método de leiaute de prioridade (PR). O método de PR é semelhante à ordenação local, na etapa ordenação de vértices, onde aplica seqüencialmente operações, denominadas melhoramento de posições horizontais, em cada nível. No caso da ordenação local, os baricentros de cada vértice são utilizados para reordenação de vértices, enquanto no método PR o melhoramento das posições são realizados de acordo com o B_u e B_l e a prioridade de cada vértice. Onde:

- As posições dos retângulos em cada nível são determinadas de acordo com suas prioridades. A maior prioridade, um número inteiro maior do que a máxima conectividade de todos os vértices, é atribuído aos vértices fictícios para minimizar a segmentação de retas. A prioridade dos outros vértices são determinadas pela conectividade de cada vértice, de forma a balancear a estrutura do mapa;

- O princípio para melhorar a posição de um retângulo é minimizar a diferença entre a posição inicial do retângulo e os valores de B_u e B_l do respectivo vértice de modo que:

  1. a coordenada x do retângulo deve ser um número inteiro e um retângulo não pode sobrepor outro retângulo no mesmo nível;
  2. a ordem dos vértices em um nível deve ser mantida para preservar a minimização do cruzamento de linhas;
  3. as posições dos vértices com menor prioridade devem ser deslocadas de modo que o deslocamento seja o mínimo possível para minimizar a proximidade de cada vértice.

5.7.1 Algoritmo

MetricalLayout $(D(\iota) = (V, E, F, ncp, \iota))$
início
1. LocalLayout$(D(\iota), raiz, x, \delta)$
2. $\chi(raiz) = \delta(raiz)/2$
3. GlobalLayout$(D(\iota), raiz, x, \chi, \delta)$

fim

MetricalLayout calcula, de forma recursiva, as métricas de leiaute para todos os vértices de $D(\iota)$ através da hierarquia de vértices formados pelos filhos do vértice raiz. *MetricalLayout* reposiciona a coordenada que expressa a posição do centro do vértice raiz e de todos os demais vértices de acordo com as coordenadas do nó pai.

Locallayout $(D(\iota), v, x, \delta)$
início
1. **se** $Ch(v) \neq 0$ **então**
   1.1 **para** $\forall\, w \in Ch(v)$
      1.1.1 LocalLayout$(D(\iota), w, x, \delta)$
2. Priority$(MH(v), x, \delta)$

fim

LocalLayout calcula as coordenadas que expressam as posições dos vértices de uma hierarquia de acordo com o método *Priority*.

GlobalLayout $(D(\iota), v, x, \chi, \delta)$
início
1. **se** $Ch(v) \neq 0$ **então**
   1.1 **para** $\forall\, w \in Ch(v)$
      1.1.1 $\chi(w) = (\chi(v) - \delta(v)/2) + \chi(w)$
      1.1.2 GlobalLayout$(D(\iota), w, x, \chi, \delta)$

fim

GlobalLayout calcula as coordenadas que expressam as posições do centro de todos os vértices filhos em relação ao vértice pai.

Priority $(MH(v), x, \delta)$
início
1. **para** $i = 1 \ldots n(v)$
   1.1 **para** $\forall\, w = Ch(v_i)$
      1.1.1 $x(w) = \delta_1^i + (d_2 + \delta_2^i) + \ldots + (d_2 + \delta_w^i/2)$
      1.1.2 CalculatePriority(w)
2. **para** $i = 1 \ldots n(v)$
   2.1 $K \in Ch(v_i)$ com maior prioridade em i
   2.2 Move($w = Ch(v_i), B_u(K)$)
3. **para** $i = n(v) - 1 \ldots 1$
   3.1 $K \in Ch(v_i)$ com maior prioridade em i
   3.2 Move($w = Ch(v_i), B_l(K)$)
4. **para** $i = 2 \ldots n(v)$
   4.1 $K \in Ch(v_i)$ com maior prioridade em i
   4.2 Move($w = Ch(v_i), B_u(K)$)
5. $x_0 = min\{x(u) - \delta(u)/2 \mid u \in Ch(v)\}$
6. **para** $\forall\, w = Ch(v)$
   6.1 $x(w) = x(w) - (x_0 - d - 1)$
7. $\delta(v) = max\{x(w) + \delta(w)/2\} - min\{x(w) + \delta(w)/2\} + 2d_1 \forall\, w \in Ch(v)$

Priority efetua o balanceamento dos vértices de uma hierarquia de acordo com a prioridade dos vértices, tal que vértices fictícios apresentam maior prioridade em relação aos demais vértices de uma hierarquia, enquanto os outros vértices apresentam como prioridade o número de conectividades. O balanceamento é efetuado de forma a posicionar as coordenadas que expressam a posição dos vértices com maior prioridade o mais próximo o possível do valor dos seus baricentros e após reposiciona os demais vértices deste nível na hierarquia.

5.7.2 Exemplo

A Figura 5.18 ilustra o mapa após realizar o leiaute métrico do dígrafo $D = (V, E, F, ncp, \iota)$ da Figura 5.16. Note-se que os vértices e arestas fictícias inseridas na normalização foram eliminadas.

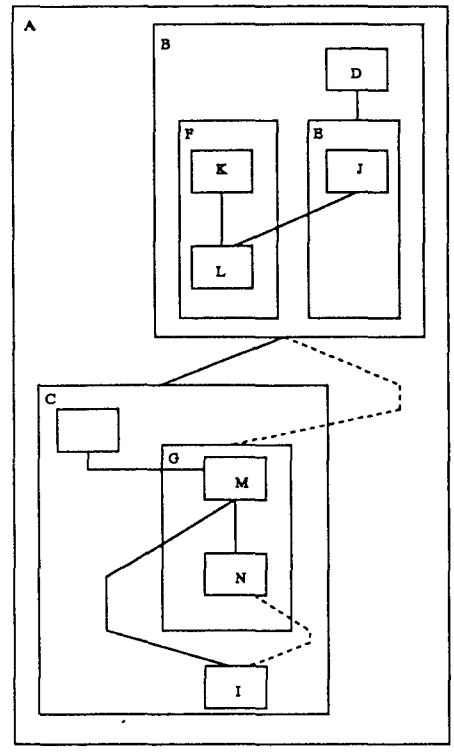


Figura 5.18: Mapa após calcular as métricas de leiaute

5.8 Xgraph e o Cronômetro

Esta seção traz a representação do componente de diálogo do cronômetro gerada pelo componente *Xgraph* do *Smart*. O componente de diálogo do cronômetro é ilustrado na Figura 4.13 e é capturado pelo componente *Tree* do *Smart*. A Figura 5.19 ilustra o grafo gerado por *Smart* a partir dos dados capturados por *Tree*.

5.9 Resumo

Este Capítulo apresentou a componente *Xgraph* responsável pelo leiaute de um *Xchart* na sua representação usual, um grafo. As etapas do algoritmo de leiaute automático para traçados de diagrama são representadas por classes distintas, na arquitetura do *Xgraph* e, estabelecem todos os atributos do grafo neutro do *Xgraph*. *Xgraph* é composto por quatro classes: *Hierarchization*, *Normalization*, *VertexOrder* e *MetricalLayout*. A classe *Hierarchization* atribui um nível para cada vértice da árvore de estados. A classe *Normalization* redefine o grafo de transições de forma que os níveis do vértice origem e destino de cada aresta de transição difiram de apenas 1. A classe *VertexOrder* ordena os vértices que apresentam o mesmo nível na árvore de estados. A classe *MetricalLayout* estabe-

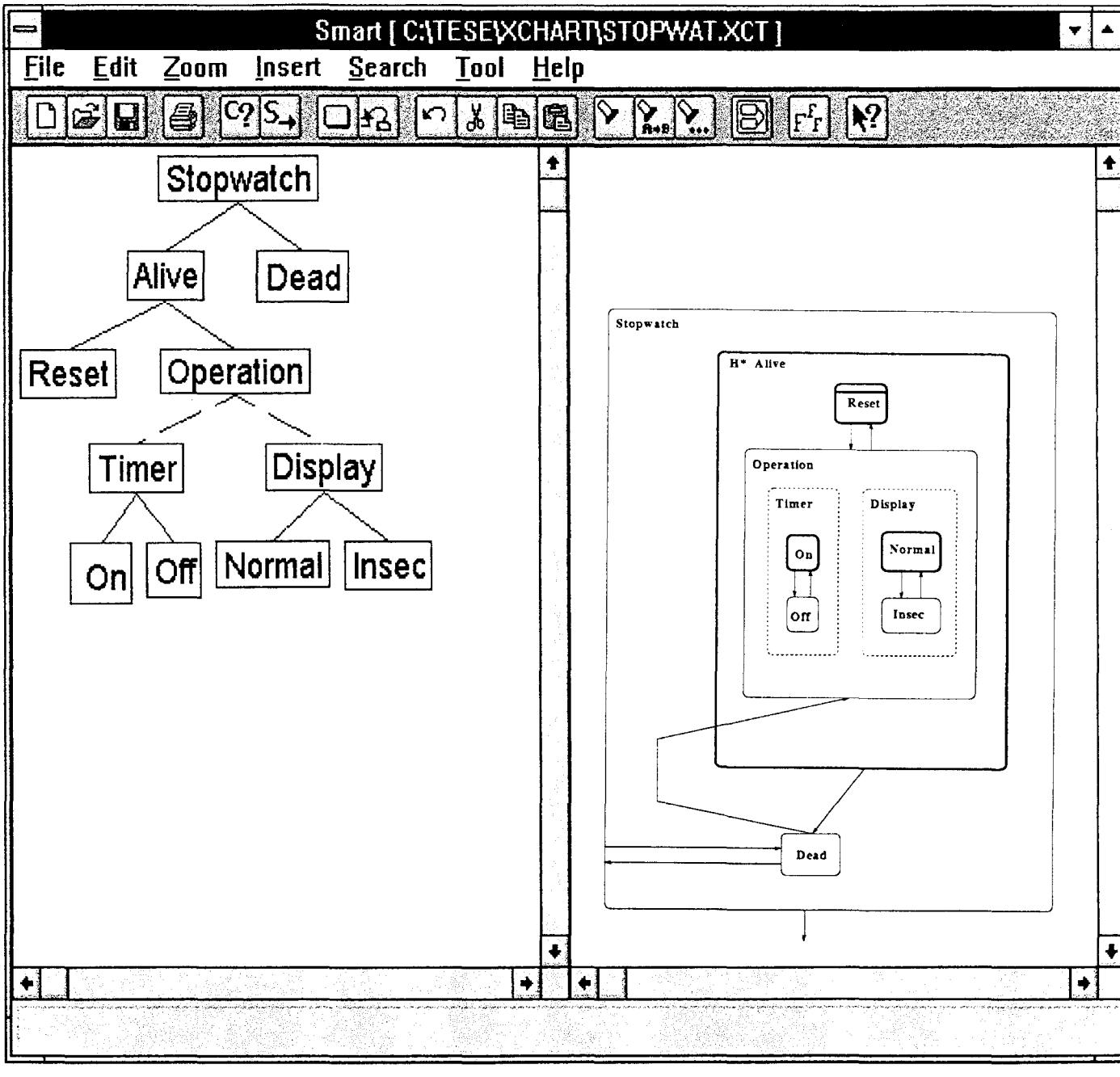


Figura 5.19: Representação do cronômetro na forma de grafo

lece as coordenadas de traçado para cada vértice da árvore de estados. *Xgraph* herda as características das classes *Hierarchization*, *Normalization*, *VertexOrder* e *MetricalLayout*. A execução dos métodos das classes *Hierarchization*, *Normalization* e *VertexOrder*, nesta ordem, permite a classe *MetricalLayout* efetuar o leiaute da estrutura neutra do *Smart*. O próximo capítulo apresenta as conclusões finais do trabalho e as propostas de extensões para o trabalho.

Capítulo 6

Conclusões

Até o último Capítulo foram apresentados os diversos componentes da arquitetura de *software* do editor gráfico *Smart*. Verificou-se que *Smart* foi projetado e implementado seguindo princípios de orientação a objetos e de acordo com a metodologia para o desenvolvimento de interfaces homem-computador proposta pelo projeto *Xchart*. Estas duas diretrizes permitiram a construção de um sistema modular e, portanto, mais fácil de ser mantido e alterado. Podemos concluir que as contribuições deste trabalho ocorrem em duas áreas principais:

1. **interfaces homem-computador.** Desenvolveu-se uma interface de programação amigável e de alto nível para o ambiente *Xchart*. Esta interface de programação é materializada pela *interface homem-computador* do *Smart*, que permite a captura, edição e geração de código para especificações de controle de diálogos escritas na linguagem visual Xchart.

2. **Algoritmos para leiaute automático de grafos.** Adaptação e implementação do algoritmo de leiaute automático para traçado de árvores, proposto por Moen [17], e do algoritmo de leiaute automático para traçado de dígrafos compostos, proposto por Sugiyama [7]. Acredita-se que *Smart* seria menos útil se não fosse capaz de traçar *Xcharts* automaticamente. Neste sentido, o estudo de algoritmos de leiaute automático de diagramas foi fundamental para o projeto do editor.

Trabalhos Futuros

Em breve, Smart deverá ser integrado em definitivo ao ambiente *Xchart*. Isto possibilitará a realização de testes mais complexos com o editor e, certamente, permitirá a sua reavaliação e aperfeiçoamento.

Hoje, *Smart* somente permite a captura de um *Xchart* através do fornecimento de sua configuração (estados e arestas) gostaríamos que a captura fosse também possível através de diagramas.

A Passagem de dados para o compilador pode ser otimizada, com a transferência da descrição do *Xchart* diretamente, via memória, para o compilador, e não via arquivo como realizado atualmente.

Bibliografia

[1] J. E. Hopcroft e J. D. Ullman A. V. Aho. *The Design and Analysis of Computer System Algoritms.* Addison-Wesley, 1974.

[2] Edilmar Lima Alves. Port System: Sistema de Comunicação em Grupo para o Ambiente Xchart. Master's thesis, Instituto de Computação - Unicamp, February 1996.

[3] Luciana de Paula Brito. Sistema de Ações Atômicas Distribuídas para Xchart. Master's thesis, Instituto de Computação - Unicamp, 1996.

[4] Deborah Hix e H. Rex Harstson. *Developing User Interface.* John Wiley & Sons, Inc., 1993.

[5] Fábio Lucena e Hans Liesemberg. Reflection on Using Statecharts to Capture User Interface Behaviour. In *XIV Int. Conf. of the Chilean Computer Science Society*, November 1994.

[6] A. Lempel e I. Cederbaum. Minimum Feedback Arc and Vertex Sets of a Directed Graph. *IEEE Transaction Circuit Theory*, 13(4):339–403, 1966.

[7] K. Sugiyama e K. Misue. Visualization of Structural Information: Automatic Drawing of Compound Digraphs. *IEEE Transactions on Software Engineering*, 21(4):876–892, 1991.

[8] Fábio Lucena, Hans Liesemberg e Luiz Buzato. Xchart Based Complex Dialogue Development. In *Simpósio Nipo-Brasileiro de Ciência e Tecnologia*, pages 387–396, August 1995.

[9] Jürgen Herczeg Hubertus Hohl e Mathias Ressel. Progress in Buildings User Interface ToolKits: The World According to XIT. *ACM*, pages 15–18, November 1992.

[10] J. Rumbaugh e outros. *Object-Oriented Modeling and Design.* Pretice Hall, Englewood Cliffs, New Jersey, 1991.

[11] S. Even. *Graph Algoritms.* Computer Science Press, 1979.

[12] Mark Green. *Report on Dialogue Specification*, chapter User Interface Management System, pages 9–20. 1985.

[13] D. Harel. Statecharts: A Visual Approach to Complex Systems. Technical Report CS84-05, Department of Applied Mathematics, The Weizman Institute of Science, 1984.

[14] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 5(3):244–275, July 1987.

[15] D. Harel. Biting the Silver Bullet. *IEEE Computer*, pages 8–20, January 1992.

[16] D. S. Johnson. The NP-Completeness Column: An Ongoing Guide. *Journal Algorithms*, 3:89–99, 1082.

[17] Sven Moen. Drawing Dynamic Trees. *IEEE Transactions on Software Engineering*, pages 21–28, July 1990.

[18] J. Rumbaugh. State Trees as Structured Finite Machines for User Interfaces. *ACM SIGGRAPH Symposium on User Interfce Software*, pages 17–19, October 1988. Banff, Alberta.