UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
ÁREA DE CONCENTRAÇÃO: ENGENHARIA DE COMPUTAÇÃO

ProVisual - Um Modelo para a Programação de Matrizes

Autor: Silvio Roberto Medeiros Evangelista
Orientadora: Dra. Beatriz Mascia Daltrini

Tese de Doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos exigidos para obtenção do título de Doutor em Engenharia Elétrica.

Campinas, São Paulo
Julho - 2002

UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

Departamento de Engenharia de Computação e Automação Industrial

Tese de Doutorado defendida por Silvio Roberto Medeiros Evangelista e aprovada em 25 de Julho de 2002 pela banca constituída pelos doutores:

Orientador :

Dra. Beatriz Mascia Daltrini

Banca:

Dr. Adilson Marques da Cunha / ITA

Dr. Dalton Francisco de Andrade / UFSC

Dr. Kleber Xavier Sampaio de Souza / EMBRAPA

Dr. Ivan Luiz Marques Ricarte

Dr. Mario Jino

Resumo e Abstract

Resumo

Este trabalho de pesquisa objetiva estudar e especificar um modelo para a programação visual de matrizes fundamentado nos paradigmas de fluxo de dados e planilhas eletrônicas. O fluxo de dados e a planilha formam o embasamento semântico do modelo proposto, enquanto o grafo direcionado forma seu embasamento sintático. O modelo consiste em um conjunto de diagramas no plano bidimensional associado a regras de transformação, implementadas em redes de fluxo de dados. Os dados são instanciados com planilhas. As planilhas podem ser vistas como funções ou como variáveis do tipo matriz. No papel de variáveis, armazenam dados bidimensionais. Encaradas como funções, recebem valores via parâmetros de entrada e produzem valores que serão utilizados por processos a elas conectados. Neste caso, as planilhas são programadas segundo o paradigma de programação por exemplos, que embute um poderoso construtor de iteração, o que permite a redução significativa da utilização de recursões e repetições.

Abstract

This research studies and specify a model for matrix visual programming based on data flow paradigm and spreadsheets. The data-flow and the spreadsheet are the semantical basis of the language, whereas the graphical representation of directed graph and spreadsheet are the syntactical fundaments. This model consists of a set of bi-dimensional diagrams and transformation rules in which processes are implemented as data flow networks and the data are represented by spreadsheets. The spreadsheets can be seen as variables of matrix type, which store bi-dimensional data, or as functions, which receive and produce values, to be used by other processes. In this case, the spreadsheets are programmed following the programming-by-example paradigm, which embeds a powerful iterator constructor, greatly reducing the usage of recursions and repetitions.

"Um gráfico não é desenhado de uma única vez; ele é construído e reconstruído até revelar todos os relacionamentos constituídos pelas interações de seus dados... Um gráfico nunca é um fim em si mesmo; ele é um momento no processo de decisão".

- *Jacques Bertin [Bertin, 1981]*

À minha família

Agradecimentos

Agradeço à Empresa Brasileira de Pesquisa Agropecuária – EMBRAPA, pelo apoio, incentivo e suporte financeiro, fundamentais à realização deste trabalho de pesquisa.

À Profa. Dra. Beatriz Mascia Daltrini, pela confiança, pela paciência, pelo encorajamento e pela dedicação, que foram fundamentais ao desenvolvimento deste trabalho.

Ao Prof. Dalton Francisco de Andrade, pelo incentivo no início de minha carreira na EMBRAPA, mas acima de tudo por sua inspiração profissional.

Ao Prof. Adilson Marques da Cunha, pelo exemplo de competência profissional.

Aos amigos do CNPTIA, pelo apoio e companheirismo constantes.

Agradeço, finalmente, à minha querida esposa Adriana, pelo imenso carinho, apoio e, também, pelas extenuantes revisões neste texto.

À Adriana e ao Luca, peço desculpas por minha ausência e por terem sido obrigados a aturar o meu mau humor nos meses finais da conclusão deste trabalho.

Aos meus pais, por tudo.

Campinas, Julho de 2002 Silvio Roberto Medeiros Evangelista

.

Sumário

1 – Introdução 1

1.1 Contexto do Trabalho de Pesquisa 2

1.2 Motivação do Trabalho de Pesquisa 4

1.3 Enunciado do Problema e a Solução Escolhida 5

1.4 Ordem de Apresentação 5

2 - Linguagens Visuais 7

2.1 Definição de Linguagem de Programação Visual 8

2.2 Fundamentação das Linguagens Visuais 8

2.3 Linguagens de Programação Visual 12

2.4 Vantagens e Desvantagens das Linguagens Visuais 29

2.5 Conclusão 33

3 - Um Modelo para Programação Visual de Matrizes– ProVisual 35

3.1 Definição do ProVisual 36

3.2 A Matriz 40

3.2.1 Criação de uma Matriz de dados a partir de outras Matrizes 44

3.2.2 Criação de uma Matriz de dados a partir da Entrada Direta de Dados..... 47

3.2.3 Definição de uma Função a partir de uma Planilha:
Abstração Procedimental 58

3.2.4 Definição de Visões sobre os Dados de uma Matriz 66

3.2.5 Explicitando os Fluxos de Dados das Fórmulas 72

3.3 Sintaxe Visual e Semântica do ProVisual 75

3.3.1 Expressões simples 76

3.3.2 Expressões Estruturadas 80
3.3.2.1 Operador de Decisão 80
3.3.2.2 Operador de Iteração 84
3.3.3 Sincronização 88
3.3.4 Operador de Fusão 91
3.3.5 Inserção de Documentação 91
3.3.6 Abstração Procedimental 92
3.4. Comparação do ProVisual com outras Linguagens Visuais 96
3.5 Conclusão 98

4 - Descrição Formal 101
4.1 Uma Abordagem para a Descrição Formal do ProVisual 102
4.1.1 Um Resumo da Formalização Espacial Lógica 103
4.2 Descrição Formal do ProVisual 105
4.2.1 Quadros 109
4.2.2 Barras de Entrada e de Saída 109
4.2.3 Porta 110
4.2.4 Conexão (arcos) 111
4.2.5 Operador Simples 112
4.2.6 Operador *Junção* 112
4.2.7 Operador de Repetição 113
4.2.8 Operador de Decisão 113
4.2.9 Comentário ou Documentação 115
4.2.10 Matriz 115

4.3 Exemplo da Semântica Operacional do ProVisual 116
4.4 Conclusão 120
5 - Implementação e Verificação 121
5.1 A Arquitetura da Implementação do ProVisual 122
5.2 O Editor Visual 124
5.2.1 O Editor de Programas 124
5.2.2 O Editor de Matrizes 134
5.3 A Máquina de Execução do APP 136
5.3.1 Componentes da Máquina de Execução 138
5.4 Verificação do ProVisual 141
5.5 Conclusão 148
6 – Conclusões 149
6.1 Conclusões 150
6.2 Contribuições 153
6.3 Sugestões para Trabalhos Futuros 155
Bibliografia 159

Lista de Figuras

Figura 2.1: Classificação de Myers das linguagens visuais 9

Figura 2.2: Classificação de Burnett para as linguagens visuais 10

Figura 2.3: Um exemplo de programa de aplicação utilizando o ambiente de programação ThingLab..... 14

Figura 2.4: Um exemplo de aplicação implementado em PROGRAPH 16

Figura 2.5: Um exemplo de painel frontal no LabView 17

Figura 2.6: Um exemplo de diagrama de blocos no LabView..... 17

Figura 2.7: Um exemplo de aplicação em Forms/3 19

Figura 2.8: Um exemplo de aplicação implementado em Fabrick 20

Figura 2.9: Um exemplo de segmentação de uma Matriz em LPM..... 21

Figura 2.10: Exemplo de aplicação em LPM 21

Figura 2.11: Um exemplo de aplicação implementado em Show and Tell 22

Figura 2.1: Um exemplo de aplicação utilizando a árvore AND/OR 23

Figura 2.13: Um exemplo de aplicação de regras para um cenário no Cocoa. 25

Figura 2.14: Um exemplo de aplicação em Formulate..... 26

Figura 2.15: Um exemplo de aplicação implementado em V 27

Figura 2.16: Um exemplo de aplicação implementado em Stella 28

Figura 3.1: Representação gráfica de uma matriz e de um processo no ProVisual. 39

Figura 3.2: Interface principal do ProVisual. 40

Figura 3.3: Representação gráfica de uma matriz. 42

Figura 3.4: Operações realizadas nas planilhas do ProVisual 43

Figura 3.5: Criação de uma matriz por composição de quatro partições e quatro parâmetros (matrizes) de entrada. 44

Figura 3.6: Passos para a criação de uma determinada matriz a partir de dados fornecidos por outras 4 matrizes 45

Figura 3.7: Geração iterativa de uma matriz. 46

Figura 3.8: Soma de sucessivos blocos de tamanho 3 orientados horizontalmente 47

Figura 3.9: Exemplos de padrões para geração de valores em uma planilha. 48

Figura 3.10: Variáveis envolvidas no processo de cópia de fórmulas 49

Figura 3.11: Exemplos não expressáveis pelo esquema usual de cópia das planilhas tradicionais. 52

Figura 3.12: Exemplos de utilização de parâmetros nas fórmulas. 55

Figura 3.13: Seqüência esquemática do processo de indução de fórmulas. 58

Figura 3.14: Exemplos de inserção de blocos, células e fórmulas indutoras 61

Figura 3.15: Exemplo de uma função baseada em planilha 62

Figura 3.16: Programação de uma planilha para produzir a multiplicação de Kronecker .. 63

Figura 3.17: Uma partição de uma matriz. 67

Figura 3.18: Dimensionamento das partições. 68

Figura 3.19: Obtenção de quatro partições com a mesma dimensão 69

Figura 3.20: Exemplo de padrão de iteração. 70

Figura 3.21: Partição em quatro regiões..... 70

Figura 3.22: Obtenção dos elementos de cada região. 70

Figura 3.23: Obtenção de uma região de cada vez. 71

Figura 3.24: Obtenção da diagonal de uma matriz. 71

Figura 3.25: Exemplo de representação de fluxo de dados em planilhas com fórmulas... 73

Figura 3.26: Exemplo de visualização global 75

Figura 3.27: Elementos visuais simples 76

Figura 3.28: Pictogramas de operações simples 77

Figura 3.29: Exemplo de um diagrama ProVisual 77

Figura 3.30: Expressão textual 78

Figura 3.31: Nomeação das portas de saída 78

Figura 3.32: Exemplos de interseção entre pictogramas 79

Figura 3.33: Pictograma do operador de decisão 81

Figura 3.34: Alternativas para as cláusulas verdadeira e falsa 82

Figura 3.35: Seleção de elementos de uma matriz 83

Figura 3.36: Seleção de linhas pares de uma matriz 84

Figura 3.37: Um exemplo de representação gráfica da iteração 85
}

Figura 3.38: Corpo da iteração da Figura 3.36 86

Figura 3.39: Soma total dos 30 primeiros números de Fibonacci 87

Figura 3.40: Iteração paralela 88

Figura 3.41: Sincronização entre dois processos 89

Figura 3.42: Sincronização entre duas matrizes 89

Figura 3.43: Sincronização entre um processo iterativo e um processo não-iterativo 90

Figura 3.44: Sincronização entre dois processos iterativos 90

Figura 3.45: Operador de fusão 91

Figura 3.46: Exemplo de abstração procedimental 94

Figura 3.47: Procedimento para cálculo de soma matricial 95

Figura 4.1: As oito relações topológicas básicas entre duas regiões 104

Figura 4.2: Ilustração dos elementos sintáticos do ProVisual 107

Figura 5.1: Arquitetura do Ambiente de Programação ProVisual (APP) 122

Figura 5.2: Interface principal do editor de programas do APP 125

Figura 5.3: Definição do operador soma do APP 126

Figura 5.4: Seqüência de ações para a definição de um operador condicional 128

Figura 5.5: As duas funções desempenhadas pelo operador de repetição 129

Figura 5.6: Imposição de um sincronismo entre os processos A e B 131

Figura 5.7: Exemplo de sincronismo em um bloco de duas matrizes 132

Figura 5.8: Criando um módulo local no APP 133

Figura 5.9: Resultado do encapsulamento de um conjunto de nós no APP 133

Figura 5.10: Interface do editor de matrizes e suas principais funcionalidades 134

Figura 5.11: Representação interna de uma matriz 135

Figura 5.12: Formato de uma instrução em APP 139

Figura 5.13: Ilustração dos possíveis estados de uma instrução na máquina de execução do APP 140

Figura 5.14: Cálculo do determinante de uma matriz com dimensão maior do que 1 142

Figura 5.15: Cálculo do determinante de uma matriz unitária e nula 143

Figura 5.16: Especificação da concatenação horizontal das submatrizes M1 e M2 da Figura 5.14 143

Figura 5.17: Determinação dos parâmetros de uma regressão linear múltipla 147

Lista de Tabelas

Tabela 2-1: Resultados para o problema de concatenação matricial. 31
Tabela 2-2: Resultados para o problema da série de Fibonacci. 31
Tabela 3-1: Lista de elementos visuais utilizados para a geração de valores nas 56
Tabela 3-2: Símbolos visuais utilizados para delimitar o número de células geradas pelo processo de indução. 57
Tabela 4-1: Definições das relações básicas entre duas regiões [Randell, 1992]..... 104
Tabela 4-2: Definição informal dos elementos sintáticos do ProVisual..... 108

Capítulo 1
Introdução

Neste capítulo, o autor apresenta o contexto, o problema, a solução escolhida e a ordem de apresentação dos demais capítulos.

1.1 Contexto do Trabalho de Pesquisa

A programação pode ser definida como a descrição de um algoritmo em termos de uma notação formal. Ela é uma tarefa desafiadora e intelectual, envolvendo uma complexidade equivalente à de outras atividades de Engenharia [Blackwell, 1998].

Os primeiros programas de que se tem notícia foram elaborados no século XIX, sendo Charles Babbage e Augusta Ada Lovelace considerados os seus pioneiros da programação. No entanto, a atividade de programação só se tornou mais difundida com o advento dos computadores eletrônicos, a partir da década de 40. Inicialmente, as linguagens de programação refletiam apenas a arquitetura do computador. Em 1950, a IBM desenvolveu o FORTRAN (FORmula TRANslator), considerada a primeira linguagem simbólica para a programação de computadores, destinada a cientistas, físicos e matemáticos.

Atualmente, a grande maioria das linguagens de programação desenvolvidas é textual, com suas expressões construídas a partir de cadeias de caracteres de algum alfabeto. Seu desenvolvimento ocorre paralelamente ao desenvolvimento do hardware, influenciado por sua arquitetura [Cox, 1989]. Em conseqüência, as linguagens textuais de programação tornam-se difíceis de utilizar, pois a descrição de um determinado problema ou algoritmo relaciona-se à maneira pela qual os computadores operam e não ao processo cognitivo de programação [Brown, 1994].

Nos últimos anos, diferentes tipos de gráficos e diagramas têm sido utilizados por desenvolvedores de software em resposta à lacuna existente entre processos cognitivos[1] e computacionais. O uso de diagramas para mostrar inter-relacionamentos entre sub-rotinas e rotinas hierarquicamente superiores passou a representar uma prática comum.

A documentação de um programa qualquer descrito numa linguagem de programação modular é considerada incompleta se não utilizar um diagrama apresentando o inter-relacionamento entre os seus módulos. Da mesma maneira, documentações de sistemas

[1] Processo intelectual de análise do problema a ser implementado com alguma linguagem de programação.

orientados a objetos incluem, inevitavelmente, diagramas que expõem relações existentes entre classes e subclasses.

Notações bidimensionais e linguagens visuais têm sido utilizadas livremente na Matemática e na Lógica Simbólica. Grafos e árvores têm sido empregados para definir e/ou ilustrar relacionamentos entre objetos, bem como o conhecido diagrama de Venn, adaptado por Harel [1988], com o propósito de representar visualmente diagramas de estado.

A idéia de transformar diagramas em programas computacionais não é nova. Pesquisadores do Massachusetts Institute of Technology (MIT) exploraram a programação interativa, por meio de *flowcharts*, no início da década de 60 [Marriott, 1998]. Todavia, a programação visual desenvolveu-se somente após o advento dos computadores pessoais e de uma significativa redução do custo do hardware gráfico. Desde então, linguagens visuais têm sido propostas, desenvolvidas e comercializadas [Glinert, 1990a, 1990b].

Para muitos usuários, a programação visual tem significado uma atraente alternativa para as linguagens usuais de programação, principalmente porque a representação visual de um problema encontra-se muito mais próxima da forma pela qual a solução é obtida ou entendida pelo usuário, se comparada à representação textual [Brown, 1994].

De acordo com Shu [1988a], a programação visual tem o objetivo de propiciar aos programadores e usuários um entendimento do que o programa pode realizar, como ele funciona, porque ele funciona e quais são os seus efeitos.

A busca constante da melhoria dos processos envolvidos nos desenvolvimentos de software sinaliza para a necessidade da incorporação de novos procedimentos robustos, que permitam aumentar a produtividade e a confiabilidade dos softwares produzidos.

Dentro deste contexto, a investigação de soluções mais apropriadas e eficientes para o desenvolvimento de sistemas de software representa uma necessidade constante e de extrema importância.

1.2 Motivação do Trabalho de Pesquisa

O autor deste trabalho de pesquisa vem atuando como analista de sistemas e engenheiro de software na Empresa Brasileira de Pesquisa Agropecuária (Embrapa) desde 1983, onde desenvolveu e implantou diversos módulos estatísticos e matemáticos para diversos softwares científicos da empresa, no período de 1983 a 1995.

Ao longo desse período, algumas dificuldades relacionadas com interpretações, adequações e desenvolvimentos de seus módulos foram constatadas.

Dentre as principais dificuldades encontradas, o que mais chamou a atenção do autor deste trabalho de pesquisa foi o fato de que os desenvolvedores de software não dominavam a teoria matemática necessária para o bom entendimento das especificações de requisitos dos problemas passados pelos pesquisadores e especialistas dos domínios das aplicações.

Em conseqüência, o autor detectou que existiam dificuldades de conexão entre as exigências dos domínios das aplicações, as reais necessidades dos pesquisadores e a forma de desenvolvimento dos módulos do software.

Uma outra dificuldade era a falta de conhecimento em programação por parte dos pesquisadores dos domínios das aplicações, impedindo-os de desenvolver os módulos estatísticos requeridos pela instituição de pesquisa.

A presença dessas dificuldades na maioria dos projetos de desenvolvimento de software científico da Embrapa Informática Agropecuária, em Campinas, constituiu-se no principal fator motivacional para o desenvolvimento deste trabalho de pesquisa.

Ao realizar as matérias teóricas do Programa de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação – UNICAMP, foram identificadas as sub-áreas de Engenharia de Software e de Linguagem de Programação que propiciaram o embasamento teórico necessário ao desenvolvimento deste trabalho de pesquisa.

A integração do conhecimento adquirido durante a realização das matérias teóricas do programa de doutorado com a experiência profissional do autor fundamentou a proposta de uma

solução para as dificuldades detectadas, visando contribuir para a comunidade científica do domínio de conhecimento abordado nesta investigação.

1.3 Enunciado do Problema e a Solução Escolhida

O problema investigado neste trabalho de pesquisa é aumentar a eficiência de implementação de modelos estatísticos e matemáticos por pesquisadores com pouca experiência em programação.

A solução escolhida pelo autor é criar uma arquitetura visual de desenvolvimento de software para a modelagem de métodos estatísticos e matemáticos que permita aumentar a eficiência da implementação de algoritmos em computador por pesquisadores com pouca experiência em programação.

1.4 Ordem de Apresentação

Neste capítulo são apresentados o contexto, o problema, a solução escolhida e a ordem de apresentação dos demais capítulos.

No Capítulo 2 são descritos os conceitos básicos das Linguagens de Programação Visual, que fundamentam a formulação do Modelo para Programação Visual de Matrizes desenvolvido no Capítulo 3. Destacam-se a definição do termo Programação Visual e a apresentação dos paradigmas mais utilizados em seu desenvolvimento, assim como exploram-se alguns exemplos de ambientes computacionais que aplicam tais conceitos. Finalmente, são apresentadas as principais vantagens e desvantagens das Linguagens de Programação Visual.

No Capítulo 3 propõe-se um Modelo para a Programação Visual de Matrizes (ProVisual) fundamentado no paradigma de fluxo de dados e de planilhas. O fluxo de dados e a planilha constituem a base semântica do modelo proposto, enquanto o grafo direcionado constitui sua base sintática. O modelo consiste em um conjunto de diagramas no plano bidimensional

associado a regras de transformação, representadas por uma rede de fluxo de dados. Os dados são representados por planilhas.

No Capítulo 4 desenvolve-se a descrição formal dos elementos da sintaxe e da semântica estática do ProVisual, utilizando a lógica espacial para o cálculo de conexões de regiões. A descrição da formalização considera a geometria básica dos objetos e o inter-relacionamento estático entre os elementos visuais da linguagem.

No Capítulo 5 apresenta-se a arquitetura da implementação do ProVisual. Em primeiro lugar, descrevem-se a arquitetura e os requisitos básicos do Ambiente de Programação ProVisual (APP). Em segundo lugar, expõem-se os editores visuais de programas e de matrizes. Em terceiro lugar, relata-se a forma de execução de um programa ProVisual na máquina virtual de fluxo de dados. Por fim, é desenvolvido um exemplo para verificação do modelo proposto.

Finalmente, no Capítulo 6 apresentam-se as principais conclusões, contribuições e recomendações de trabalhos futuros.

Capítulo 2
Linguagens Visuais

Neste capítulo, o autor descreve os conceitos básicos de Linguagens de Programação Visual (LPV), que fundamentam a formulação do Modelo para Programação Visual de Matrizes (ProVisual) desenvolvido no Capítulo 3. Ele destaca a definição do termo Programação Visual e a apresentação dos paradigmas mais utilizados em seu desenvolvimento, assim como explora alguns exemplos de ambientes computacionais que aplicam tais conceitos. Finalmente, o autor descreve as principais vantagens e desvantagens das Linguagens de Programação Visual.

2.1 Definição de Linguagem de Programação Visual

Os processos computacionais são tradicionalmente especificados por cadeias unidimensionais de caracteres. A programação visual, em contraste, utiliza diagramas e ícones em pelo menos duas dimensões [Brown, 1994]; ou mais formalmente, um conjunto de diagramas e ícones que correspondam às sentenças válidas em uma determinada linguagem. Cada diagrama, por sua vez, representa uma coleção de símbolos no espaço bi ou tridimensional. A determinação da validade de uma sentença e o seu significado dependem do relacionamento espacial entre estes símbolos.

Shu [1988b] relata que objetos manipulados na programação visual não possuem uma representação visual intrínseca. Entre eles, encontram-se os tipos de dados tradicionais (arranjos, pilhas, listas, filas, etc) e as aplicações orientadas por tipos de dados mais complexos. Neste último caso, pode-se citar as aplicações orientadas por formulários, documentos, banco de dados, etc. Naturalmente, estes objetos precisam ser apresentados visualmente para se produzir uma interface amigável. Da mesma maneira, a linguagem de programação precisa ser retratada graficamente. Em resumo, os construtores de um programa, os seus tipos e as regras de combinação dos construtores necessitam de uma notação visual.

2.2 Fundamentação das Linguagens Visuais

Diferentes classificações têm sido exploradas para a definição da base semântica de uma linguagem de programação visual. Uma das primeiras classificações das linguagens visuais foi realizada por Myers [1986], que organizou os sistemas de programação em oito categorias diferentes, seguindo um critério ortogonal (Figura 2.1).

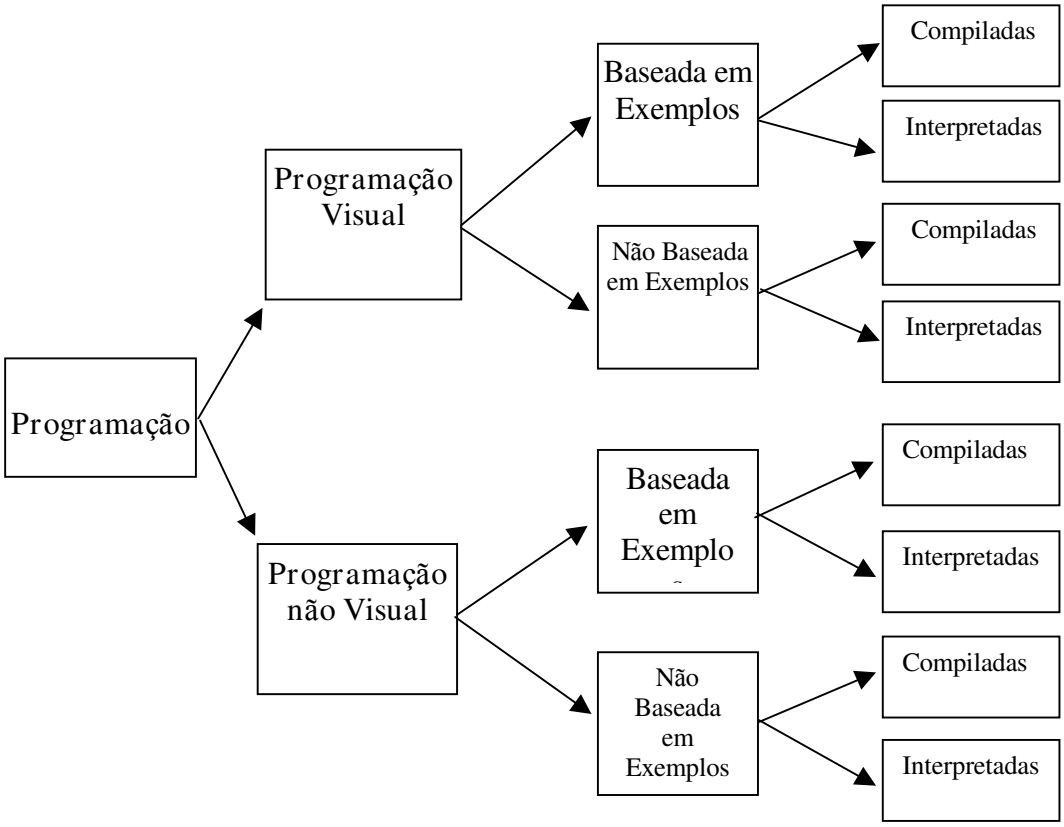


Figura 2.1: Classificação de Myers para os sistemas de programação.

Esta classificação foi suplantada pela proposta de [Burnett, 1994], que categoriza as linguagens visuais em dois grupos. O primeiro grupo agrega as linguagens visuais derivadas ou adaptadas das linguagens textuais tradicionais. Isto é, linguagens não intrinsecamente visuais, que possuem notações visuais impostas e podem ser classificadas como linguagens visualmente transformadas. O segundo grupo reúne as linguagens que possuem representação naturalmente visual. A Figura 2.2 ilustra as diferentes linguagens dos dois grupos.

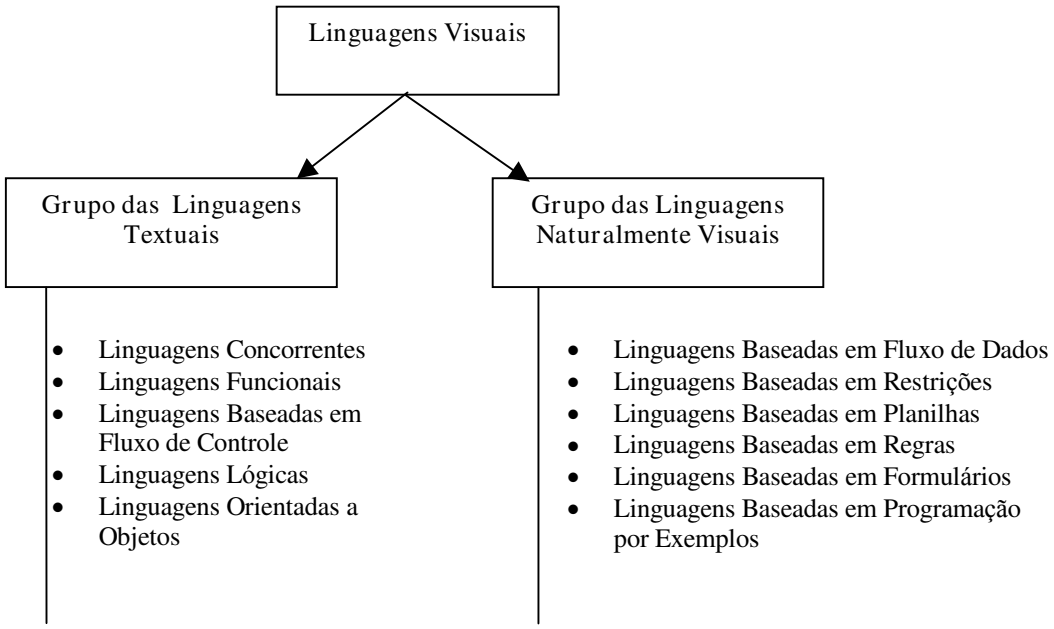


Figura 2.2: Classificação de Burnett para as linguagens visuais.

As primeiras propostas de linguagens para programação visual consistiram de editores visuais para as linguagens textuais tradicionais. Baseadas em fluxo de controle, essas linguagens utilizavam-se de diagramas para representar os seus construtores, como os diagramas *flowchart* e *Nassi-Schneiderman* [Trip, 1988]. Pode-se citar com exemplo o ambiente computacional PECAN [Reiss, 1985], que permitia a visualização de um programa PASCAL através de uma série de diagramas. Mais tarde, surgiram as Linguagens Visuais, que não eram vinculadas a qualquer notação textual anterior. Estas linguagens, baseadas em paradigmas tradicionais, não correspondiam a uma simples tradução de uma linguagem textual préexistente.

O segundo grupo inclui linguagens de programação que possuem representações naturalmente visuais de suas expressões. Como primeiro exemplo, podem-se citar as linguagens visuais fundamentadas no paradigma de fluxo de dados. Nestas linguagens, não há especificação explícita da seqüência de execução das operações e funções. Especifica-se apenas a fonte dos dados de uma operação. Uma determinada operação ocorre caso todos os seus dados de entrada estejam disponíveis. Note-se, contudo, que a maioria das Linguagens de Programação Visual, baseadas em fluxo de dados, utiliza alguns construtores de fluxo de controle [Hils, 1992].

Uma alternativa às Linguagens baseadas em Fluxo de Dados é a utilização de Linguagens baseadas em Restrições[2]. Os ambientes de programação desenvolvidos segundo esta metáfora permitem que o usuário especifique visualmente as propriedades invariantes dos objetos, bem como os seus inter-relacionamentos [Graf, 1995]. Devem-se atribuir valores para as variáveis de maneira que todas as restrições sejam verdadeiras. Ambientes que utilizam esta metáfora são, em alguns aspectos, análogos aos sistemas de programação lógica.

Existem linguagens visuais baseadas na Programação por Exemplos. Nestas linguagens, o usuário manipula amostras de dados ou representações visuais de suas estruturas para "ensaiar" o sistema em relação às operações que devem ser executadas. Nestes casos, o sistema emula as operações ensaiadas sobre um novo conjunto de dados.

Uma outra Linguagem de Programação Visual é fundamentada em formulários. Ambler e Burnett, que lideram as pesquisas deste tipo de linguagem, consideram a Programação baseada em Formulários como uma generalização da Programação em Planilhas Eletrônicas [Ambler, 1989]. Utilizando a idéia básica de expansão visual dos conceitos empregados nas planilhas, esta generalização possibilita a ampliação dos domínios de aplicação. Uma definição para as classes de problemas por ela resolvidos foi apresentada por Ambler [1987]. Segundo o autor, qualquer problema solucionado com a utilização de um lápis e folhas de papel torna-se um bom candidato para esta linguagem de programação. É importante observar que o sucesso das planilhas eletrônicas deve-se aos métodos visuais utilizados para representar os relacionamentos entre os dados [Ambler, 1989].

[2] Restrições referem-se a uma relação booleana. Normalmente, uma relação de igualdade ou desigualdade entre os valores de uma ou mais variáveis. Por exemplo, x>3 é uma restrição em x.

Finalmente, as Linguagens Visuais baseadas em Regras possuem um conjunto de estados e regras, que transformam o estado atual em algum estado futuro [Ambler, 1997]. Cada regra possui uma pré-condição e uma especificação de transformação. Caso satisfeita a pré-condição, executa-se a especificação de transformação. Na sua forma mais simples, assume-se que todas as regras são mutuamente exclusivas. Todavia, se o número de regras aumentar muito, não se pode garantir a sua ortogonalidade, o que requer um mecanismo de resolução mais complexo. Os sistemas fundamentados neste paradigma são particularmente efetivos para expressar controle sobre eventos que exijam algum tipo de reação.

2.3 Linguagens de Programação Visual

Uma vez escolhida a base semântica para uma linguagem visual, deve-se selecionar a fundamentação sintática. Esta fundamentação sintática determina a representação real dos programas.

Nas Linguagens Baseadas em Fluxo de controle ou de dados, utiliza-se o grafo direcionado como abordagem mais comum [Brown, 1994]. Os nós destes grafos indicam operações ou células de dados, enquanto os arcos indicam fluxos de controle ou de dados. Algumas linguagens baseadas em fluxo, como por exemplo a HI-Visual [Hirakawa, 1988], utilizam a justaposição de nós para indicar fluxo. Os nós podem ser representados por caixas, ícones ou qualquer outra forma.

As Linguagens Baseadas em Restrições são normalmente representadas por caixas e linhas. Todavia, existem implementações que utilizam representações fortemente acopladas ao domínio de aplicação, como por exemplo para desenho de circuitos elétricos. Desde que as linguagens de programação baseadas em formulários são uma extensão das planilhas eletrônicas, as suas representações visuais se assemelham às interfaces das planilhas eletrônicas. A representação visual e sintática das linguagens de programação fundamentados em Exemplos é usualmente ligada a domínios específicos de aplicação.

Após as definições das bases semântica e sintática, deve-se criar um conjunto de construtores básicos. Pode-se incluir representações para iterações, execuções seqüenciais, abstrações de procedimentos e recursões, teste de tipos, etc.

Os ambientes de programação visual descritos a seguir ilustram algumas alternativas que os projetistas destas linguagens adotaram para os seus desenvolvimentos.

ThingLab

O ambiente de programação ThingLab, desenvolvido a partir de uma pesquisa de doutorado de Borning [1981], foi originalmente projetado para um ambiente de simulação gráfica de experimentos em Física e Geometria. Todavia, a maior contribuição do ThingLab para a Ciência da Computação foi a incorporação de uma Linguagem de Programação Visual Baseada em Restrições.

O ThingLab utiliza linhas e retângulos para a representação dos seus objetos. Esses objetos enviam e recebem mensagens, podendo ser visualizados e editados pelo usuário. As restrições em ThingLab são representadas como regras associadas a um conjunto de métodos. As regras são utilizadas pelo ThingLab na construção de testes para verificar o atendimento às restrições. Os métodos descrevem as formas alternativas para satisfazer a restrição. O usuário responsabiliza-se pela definição das restrições de um objeto, enquanto o ThingLab é obrigado a satisfazê-las. A Figura 2.3 apresenta um exemplo de aplicação de uma Linguagem de Programação Visual Baseada em Restrições, utilizando-se o ambiente ThingLab, que faz a conversão de temperatura de graus Centígrados em Fahrenheit. Neste programa, os retângulos conectados a um ícone que simboliza uma âncora são constantes, enquanto os demais representam variáveis ou operações.

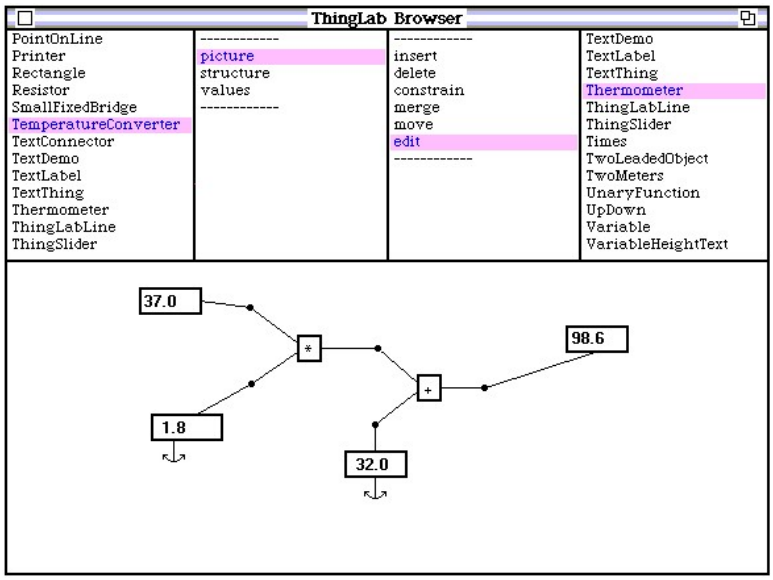


Figura 2.3: Um exemplo de programa de aplicação utilizando o ambiente de programação ThingLab [Borning, 1981].

Cabe ressaltar que o ThingLab é bidirecional na solução das restrições. Considerando-se o exemplo da Figura 2.3, se o valor que representa graus Centígrados fosse editado e alterado, o valor de graus Fahrenheit seria automaticamente calculado, visando satisfazer à expressão. O mesmo ocorreria, caso fosse editado e alterado o valor que representa a temperatura em graus Fahrenheit.

PICT/D

O PICT/D [Glinert, 1984] foi um dos primeiros esforços de desenvolvimento de uma Linguagem de Programação Visual fundamentada em Fluxo de Controle. A base sintática do PICT é um grafo direcionado. O ambiente computacional da linguagem incorpora um editor interativo, que permite a criação de programas pelo usuário, apenas utilizando um *joystick*. Uma vez iniciado o ambiente computacional PICT/D, ele não necessita do teclado para entrada de dados. A linguagem permite a criação e manipulação de *flowcharts* com a habilidade de geração de novos ícones para representar sub-rotinas.

Todavia, a semântica da linguagem é convencional, exigindo a utilização de conceitos tradicionais para o ato de programar. Pode-se afirmar que a complexidade de um programa PICT equivale à complexidade de um programa textual com os mesmos objetivos, mas o PICT aplica-se de forma mais adequada para o entendimento de programas.

Rehearsal World

Uma das mais conhecidas linguagens de Programação Visual por Exemplos é o *Rehearsal World* [Gould, 1984], projetado para permitir que professores, sem experiência em programação, pudessem criar aulas ou lições computadorizadas.

Ele utiliza o teatro como metáfora, isto é, existem as figuras do palco, dos atores e do diretor de cena. O monitor corresponde ao "palco". O processo de programação consiste na movimentação dos atores pelo palco e no ensinamento de sua interação com os outros atores, pelo envio de mensagens e na resposta a mensagens recebidas. O *Rehearsal World* possui inicialmente um conjunto de atores primitivos (cada um respondendo a mensagens predefinidas) que podem ser compostos para montar um cenário e ensinados a responder a novas mensagens.

Prograph

Um outro exemplo de linguagem de programação visual é o Prograph [Matwin, 1985]. Uma das melhores características do Prograph é a integração das técnicas de programação por objetos em uma Linguagem de Programação Visual baseada em Fluxo de Dados. Nele, classes representadas por ícones podem ser visualizadas a partir de um *browser* específico. Os métodos de uma classe são implementados como diagramas de fluxo de dados.

O ambiente de programação Prograph inclui mecanismos predefinidos de iteração e de paralelismo. Ele também fornece um editor para criação de classes e programas, um interpretador para executar e depurar o programa e um construtor de aplicação para construir e testar a interface com o usuário. A Figura 2.4 mostra um exemplo de um programa Prograph para o cálculo do teorema de Pitágoras.

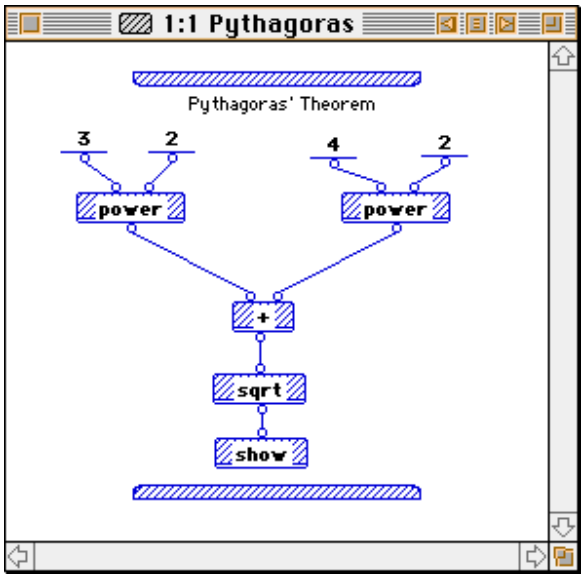


Figura 2.4: Um exemplo de aplicação implementado em PROGRAPH [Matwin, 1985].

LabView

O ambiente de programação LabView [Vose, 1986] foi concebido para utilização por engenheiros e cientistas com pouca ou nenhuma experiência em programação tradicional. O componente básico de qualquer programa em LabView é um instrumento virtual, que consiste de um painel frontal e de um diagrama de blocos. Como o próprio nome indica, o instrumento virtual deve imitar o comportamento do instrumento real do laboratório (por exemplo, um osciloscópio).

O painel frontal corresponde à interface pela qual o usuário interage com o LABVIEW em tempo de execução. O diagrama de blocos representa o código fonte que descreve o comportamento do instrumento virtual em questão. A Linguagem de Programação do diagrama de blocos baseia-se em Fluxo de Dados. Ela possui um conjunto especial de estruturas que implementa construtores para fluxo de controle (repetição e decisão). As Figuras 2.5 e 2.6 retratam, respectivamente, o painel frontal e o diagrama de blocos de um programa LABVIEW para um instrumento virtual de monitoramento da temperatura.

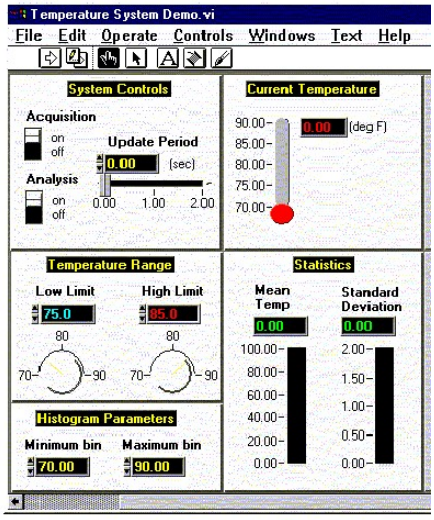


Figura 2.5: Um exemplo de painel frontal no LabView [Whitley, 2000].

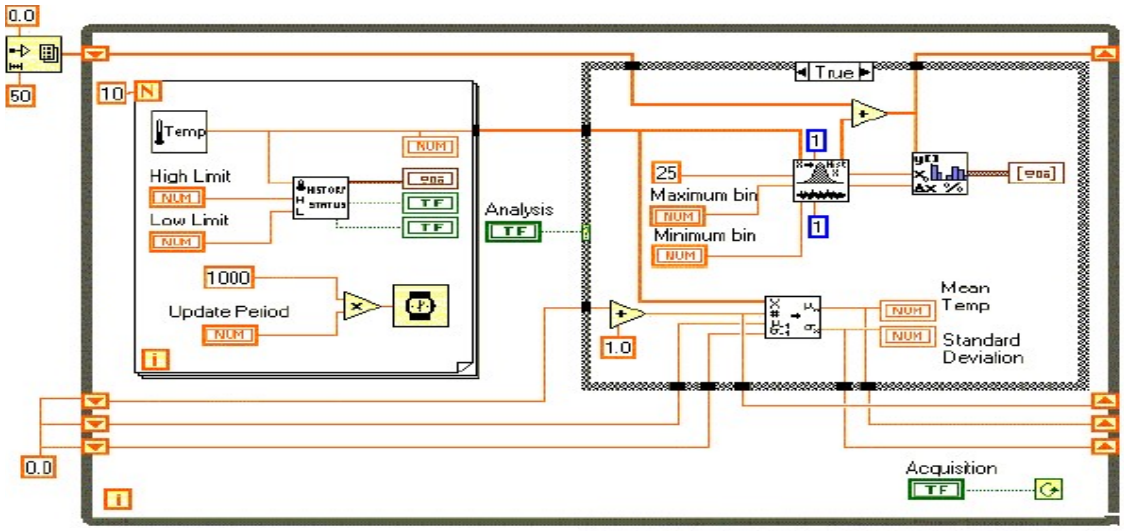


Figura 2.6: Um exemplo de diagrama de blocos no LabView [Whitley, 2000].

Forms

O ambiente de programação FORMS [Ambler, 1987], como o próprio nome sugere, fundamenta-se em formulários. Como descrito anteriormente, as Linguagens Baseadas em Formulários expandem alguns conceitos relacionados às planilhas eletrônicas.

Um dos aspectos abordados por este ambiente é a sua estrutura de células, em forma de matriz das planilhas eletrônicas, que se encontram livremente arranjadas no formulário.

Um programa FORMS compõe-se de vários formulários utilizados pelo programador para realizar manipulações diretas - por exemplo, inserir células em matrizes (objetos) e definir uma fórmula ou valor para cada uma. Uma expressão de uma célula pode fazer referência a qualquer célula (ou células) em qualquer objeto dentro do formulário. A única restrição é a impossibilidade de se fazer referências circulares. O FORMS permite a definição de subformulários com conteúdos similares a um formulário. Certos objetos herdam valores de seus formulários originais. Cada célula é avaliada apenas uma vez na avaliação de um formulário ou subformulário. Recursividades e iterações são construídas por repetidas invocações a um formulário, cada qual gerando uma nova instância do formulário. Desta forma, o conjunto de todos os formulários e subformulários, usados para produzir uma determinada computação, provê um histórico completo dos passos computacionais realizados para resolver uma tarefa.

Os valores produzidos pelas computações anteriores são recuperáveis pela instância corrente da computação. Baseado nas várias versões do FORMS, desenvolveu-se o FORMS/3 [Burnett, 1994], que introduziu uma série de novos conceitos na programação visual. Um desses conceitos foi a abstração visual de dados. A Figura 2.7 mostra uma implementação para o cálculo fatorial realizada em Forms/3.

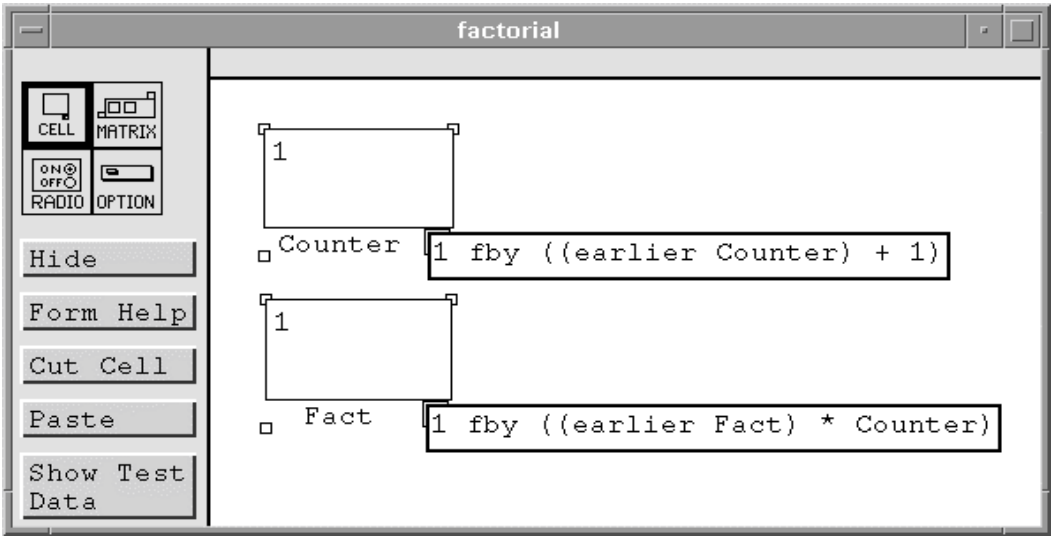


Figura 2.7: Um exemplo de aplicação em Forms/3 [Ambler, 1987].

Fabrick

Um outro desenvolvimento de uma linguagem de programação visual é o Fabrick [Ingalls, 1988]. Ele foi idealizado para tornar a tarefa de programação um processo mais natural e acessível aos programadores novatos ou casuais. Desenvolvido nos laboratórios da *Apple Computer,* o Fabrick baseia-se no modelo de Fluxo de Dados. Um programa Fabrick é composto de componentes conectados, que correspondem a funções e são retratados como ícones.

O Fabrick estendeu o uso normal de fluxo de dados pela adição de um fluxo bidirecional. Cada componente possui um conjunto de pontos de conexão ou pinos, aos quais conectam-se as linhas de fluxo. Triângulos são utilizados para indicar a direção do fluxo, enquanto diamantes simbolizam um fluxo bidirecional. O elemento pino é apenas um dos elementos da metáfora eletrônica utilizada pelo Fabrick. Os programadores constroem seus programas, ou componentes, a partir de uma biblioteca de componentes preexistentes. A Figura 2.8 retrata um exemplo de implementação de um programa para conversão de temperatura.

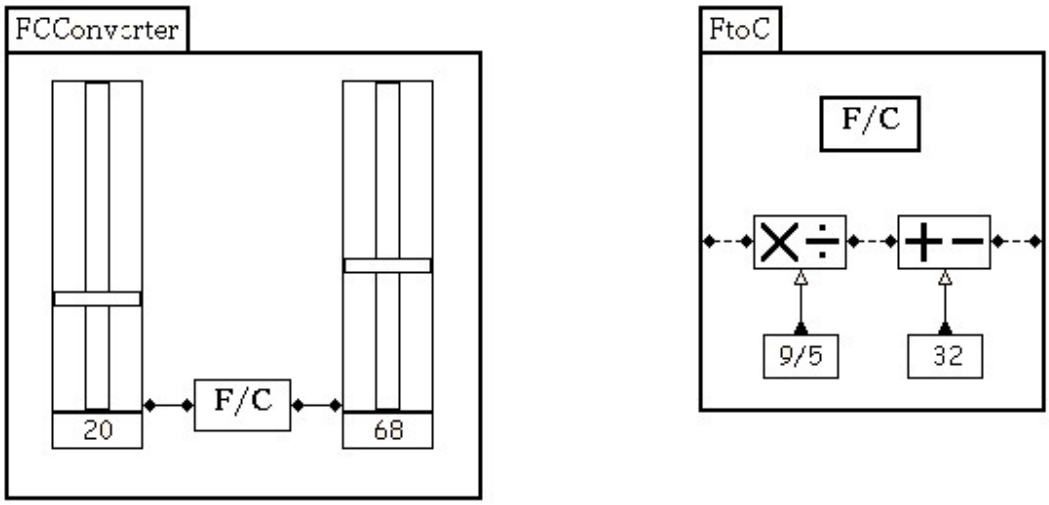


Figura 2.8: Um exemplo de aplicação implementado em Fabrick [Ingalls, 1988].

O Fabrick implementa a noção explícita de tipos de dados, tais como: enumeração, registros, arranjos, *bitmap*, etc. Uma outra característica interessante do Fabrick é o seu ambiente dinâmico. Se qualquer conexão, valor ou componente for alterado, o restante do grafo atualiza-se automaticamente de forma a refletir a mudança efetuada.

LPM

Um exemplo fundamentado nas Linguagens Baseadas em Restrições e na Linguagem Lógica é a Linguagem de Programação de Matrizes (LPM) [Yeung, 1988]. A LPM é uma linguagem híbrida aplicada à manipulação de matrizes. Ela provê um conjunto de elementos para a notação gráfica de uma matriz, enquanto as cláusulas lógicas são informadas textualmente. A LPM compõe um ambiente integrado ao sistema de Programação Lógica *Constraint Logic Programming* (CLP) [Jaffar, 1987].

Em LPM, uma matriz consiste de um retângulo fechado e composto por um conjunto de sub-retângulos internos que delimitam várias submatrizes. A Figura 2.9 exemplifica uma matriz formada por várias submatrizes (subdivisões). O programador cria um programa (cláusulas em Prolog) com um editor de texto modificado, no qual uma matriz pode ser manipulada como um

texto ordinário. Conseqüentemente, um diagrama (matriz) pode ser mesclado com textos que representam as cláusulas. Dentro da matriz, o programador pode mover, alterar dimensões, remover, duplicar e rotular sub-retângulos, como mostrado na Figura 2.9.

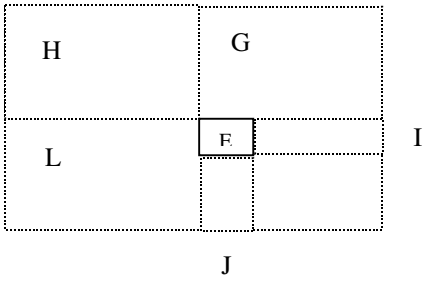


Figura 2.9: Um exemplo de segmentação de uma Matriz em LPM.

Uma vez criado o programa, o programador invoca o LPM para traduzir os diagramas em predicados PROLOG de primeira ordem. Após a sua tradução, o programador precisa informar uma cláusula de averiguação, para que o sistema possa executá-la. A Figura 2.10 ilustra um cenário de execução para o cálculo do coeficiente binomial dos dados de uma matriz.

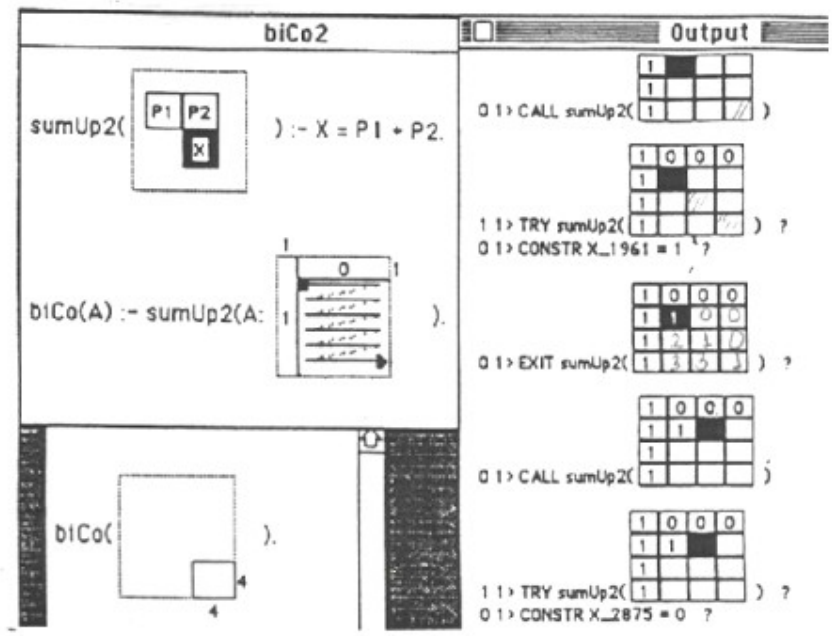


Figura 2.10: Exemplo de aplicação em LPM [Yeung, 1988].

Show and Tell

A Linguagem Visual *Show and Tell* [Kimura, 1990] baseia-se em Fluxo de Dados para programações de uso geral. Como na maioria das Linguagens baseadas em Fluxo de Dados, o grafo direcionado serve como base semântica. Caixas representam funções, constantes, variáveis, etc. Arcos indicam o fluxo de dados entre as caixas. Nesta linguagem, existe uma construção especial que permite a criação de caixas aninhadas e fluxos que partem de qualquer nível hierárquico. Uma figura composta por caixas e arcos é chamada de peça de um "quebra-cabeça".

O ambiente de programação tenta completar este "quebra-cabeça" pela execução de todos os fluxos de dados possíveis. Se um dado que flui para dentro de uma caixa encontra um valor diferente já existente, a caixa torna-se inconsistente. O conceito de consistência é fundamental para esta linguagem.

A inconsistência não está limitada a uma determinada caixa, podendo, em termos mais abrangentes, fluir de uma determinada caixa, o que torna inconsistente todo o ambiente abrangido espacialmente pela caixa. As caixas inconsistentes são pintadas de cinza e consideradas removidas do diagrama. Esta propriedade de inconsistência pode ser utilizada de diferentes maneiras, principalmente para prover a funcionalidade de seleção existente nas linguagens textuais. A Figura 2.11 mostra um programa Show and Tell para computar o fatorial de um número.

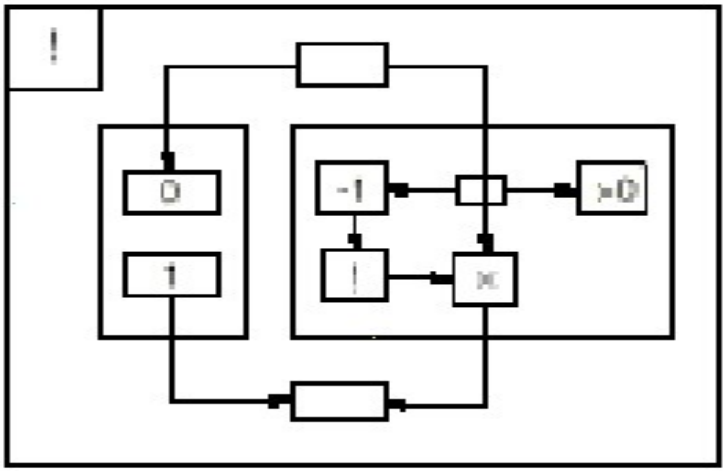


Figura 2.11: Um exemplo de aplicação implementado em Show and Tell [Kimura, 1990].

Senay e Lazzeri

Em relação às Linguagens Visuais para Programação Lógica, pode-se citar o ambiente de programação desenvolvido por Senay e Lazzeri [Hikmet, 1991]. Este ambiente fundamenta-se numa árvore AND/OR, que visa explicitar o inter-relacionamento entre a raiz de uma cláusula e seus sub-objetivos, para visualizar graficamente um programa escrito em PROLOG. A Figura 2.12 mostra um trecho de programa escrito em Prolog e a sua representação gerada com o ambiente desenvolvido por Senay e Lazzeri.

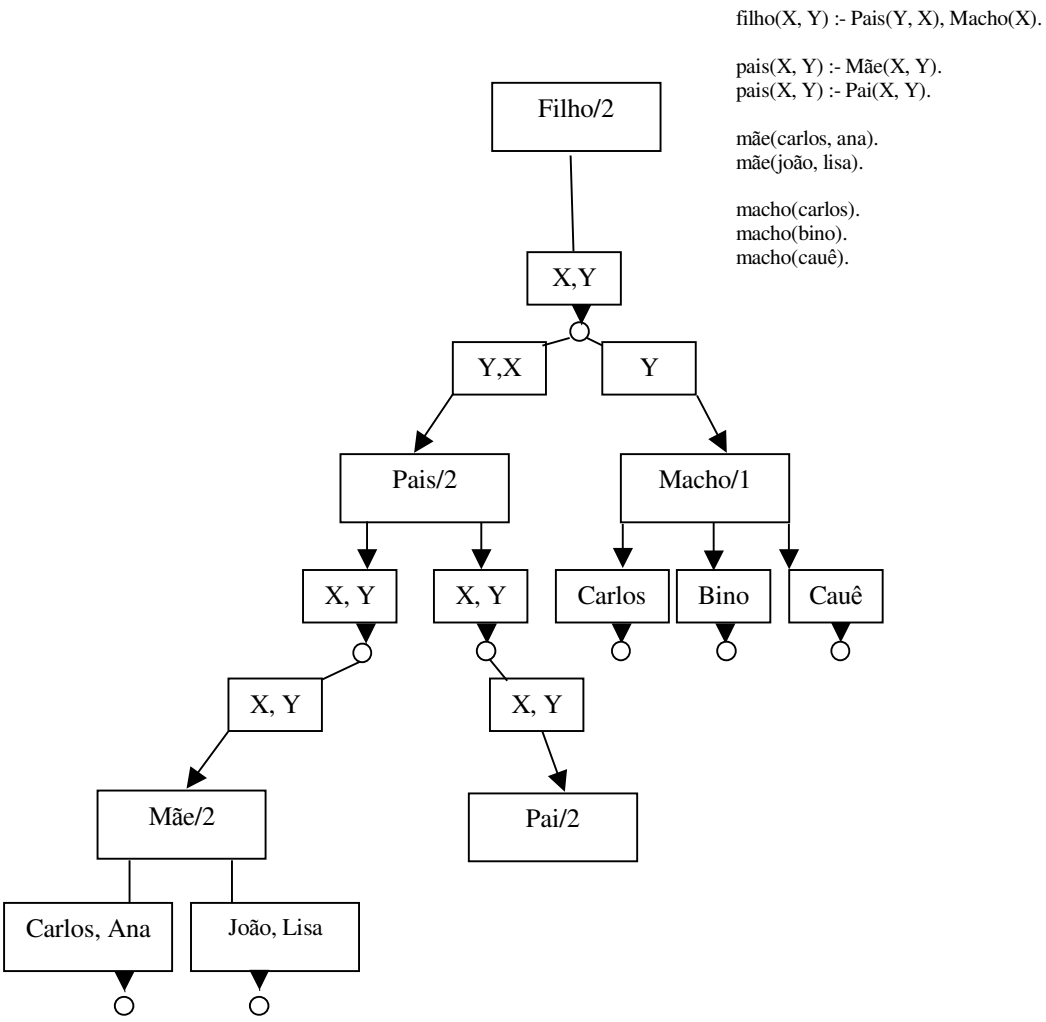


Figura 2.12: Um exemplo de aplicação utilizando a árvore AND/OR [HikMet, 1991].

HyperFlow

A Hyperflow [Kimura, 1993] é uma Linguagem de Programação Visual projetada para trabalhar com o periférico *pen* para a entrada de dados. Ela é uma extensão da linguagem *Show and Tell* descrita anteriormente.

O bloco de montagem básico da Hyperflow é o Processo Interativo Visual (PIV), que representa um processo concorrente com uma interface com o usuário. Esta interface é normalmente uma caixa ou um grupo de caixas dispostas na tela do computador. Pode-se fazer um paralelo entre um PIV e uma janela de um ambiente multijanelas.

Cada PIV deve possuir uma parte reservada para descrição de um processo, que contém, além de outras informações, a especificação de como um PIV deve responder à mensagem recebida dos usuários e/ou de outros PIVs. Esta troca de mensagens entre os PIVs representa o cerne da capacidade de computação distribuída existente dentro de um grupo de PIVs.

Os PIVs são conectados por arcos direcionados que indicam o caminho do fluxo de comunicação. O corpo de um PIV, que representa a parte do processamento, pode ser codificado com uma sintaxe bidimensional própria do Hyperflow ou com qualquer linguagem textual (C, Pascal, Assembler). Cada módulo funcional do Hyperflow, incluindo os de baixo nível (por exemplo: *drivers),* possui um PIV correspondente. Um dos objetivos do projeto do Hyperflow é torná-lo uma linguagem que possa ser utilizada em vários níveis de programação.

Cocoa

A Linguagem de Programação Visual Cocoa, originalmente conhecida como Kidsim, é um ambiente de programação baseado em regras associadas a classes [Smith, 1994]. O ambiente de execução do Cocoa consiste de objetos instanciados de suas classes. Para cada classe de objetos, existe uma lista de regras, que define as ações herdadas por todas as instâncias da classe.

Cada regra é dividida em duas partes separadas por uma seta. A parte da esquerda especifica a pré-condição, enquanto a parte da direita especifica graficamente a ação a ser aplicada decorrente da regra selecionada. Tanto a regra como a ação são especificadas graficamente. A Figura 2.14 ilustra um exemplo em Cocoa de um conjunto de regras que ensina um objeto, de um determinado cenário, a pular um muro. Note-se que as regras definidas permitem que muros de qualquer altura sejam pulados.

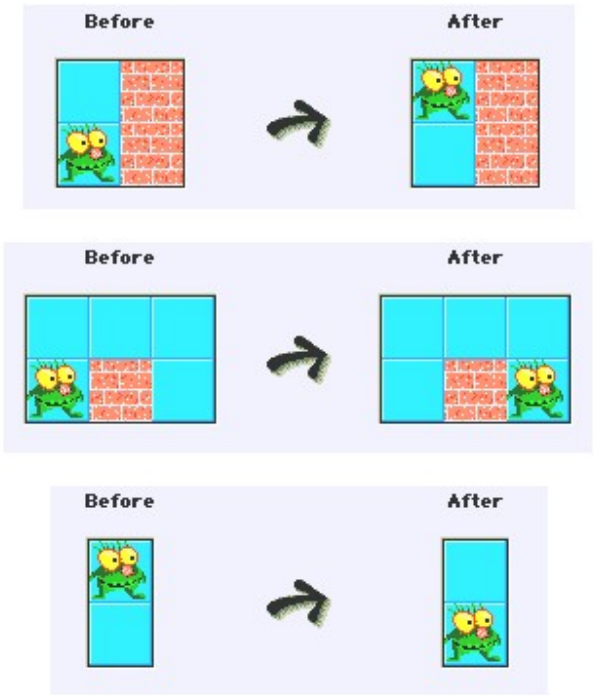


Figura 2.13: Um exemplo de aplicação de regras para um cenário no Cocoa [Smith, 1994].

Formulate

Ambler e Leopold propuseram o ambiente de programação Formulate [Leopold, 1996] para visualizar e manipular arranjos (*array)* de dados. Este ambiente é baseado em planilhas, que suportam divisões em partes lógicas. Essas partes lógicas são regiões de um arranjo e podem ser utilizadas para descrever outros arranjos.

As funções básicas deste ambiente de programação, bem como as definidas pelo usuário, podem ser aplicadas a qualquer região de um arranjo. Uma referência a uma região pode ser inserida numa expressão de outra região. Uma importante distinção entre o Formulate e as planilhas usuais é a associação de uma expressão a sua região e não a cada célula individualmente.

Em Formulate, um objeto de dados pode ser uma célula, uma lista, uma matriz ou uma tabela. Uma célula é o componente primitivo de outras estruturas. Cada tipo de estrutura pode possuir um conjunto de atributos que auxiliam a sua definição, por exemplo, uma lista possui um atributo que indica o número de elementos da lista. Um procedimento é implementado como um formulário que recebe parâmetros e possui células que correspondem a variáveis. A Figura 2.15 ilustra um exemplo de um programa escrito em Formulate que define uma função (denominada *Choose*) que retorna o valor de A ou zero, dependendo do valor de B.

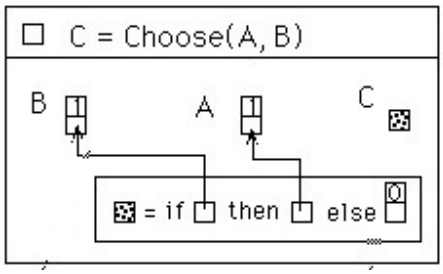


Figura 2.14: Um exemplo de aplicação em Formulate [Leopold, 1996].

V

O ambiente de programação experimental V [Auguston, 1997] baseia-se em fluxo de dados e implementa uma estratégia mais direta para manipulação de estruturas de dados típicas das linguagens visuais (arranjos, listas, etc). O ambiente V propõe uma solução para o processamento iterativo de fluxo de dados baseada nos conceitos de fluxo de dados condicional e de construtor iterativo especializado para vetores e listas. A Figura 2.16 exemplifica um programa em V para gerar os primeiros 100 números (determinado pelo valor de *size).* Neste exemplo, o variável *a* representa o valor da célula imediatamente anterior, enquanto a variável *X*

retrata e determina o valor do primeiro item do vetor. Todos os outros itens são gerados pelas fórmulas associadas ao construtor de seleção, dependendo da avaliação da expressão booleana *'a mod 2=0'*.

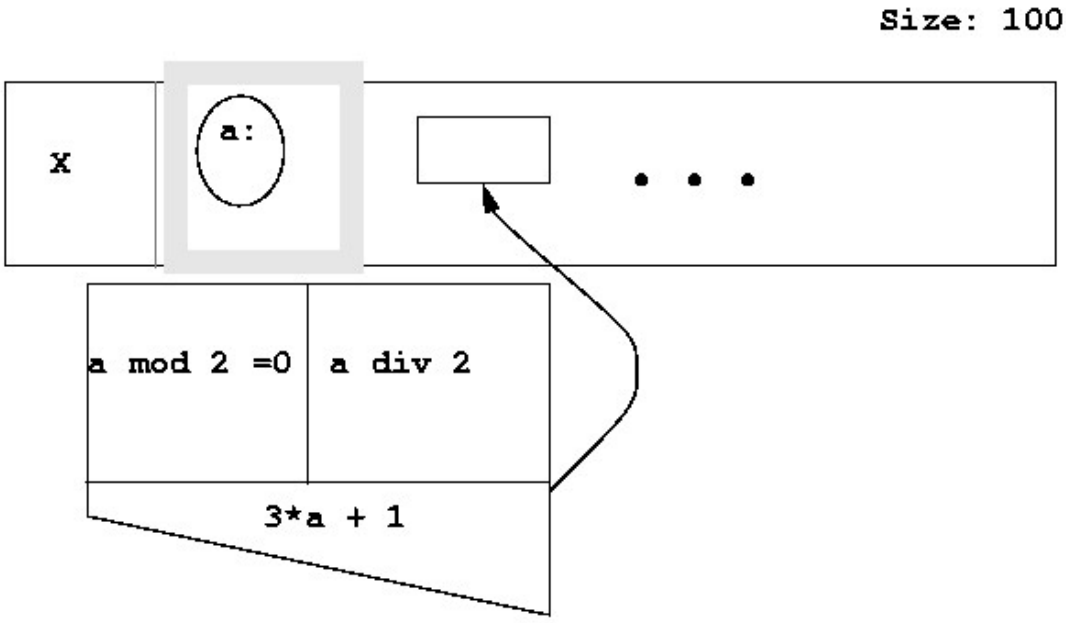


Figura 2.15: Um exemplo de aplicação implementado em V [Auguston, 1997]

Stella

Um outro exemplo de Linguagem Visual Baseada em Fluxo de Dados é o Stella [2001]. Este ambiente de programação foi concebido para facilitar a construção de modelos matemáticos de simulação. Ele emprega conceitos de armazenadores (caixas) e constantes (círculos, fórmulas), conectados, via arcos, para simular modelos discretos. A Figura 2.17 mostra um exemplo de um sistema de componentes interdependentes que implementa um modelo de simulação para presa/predador.

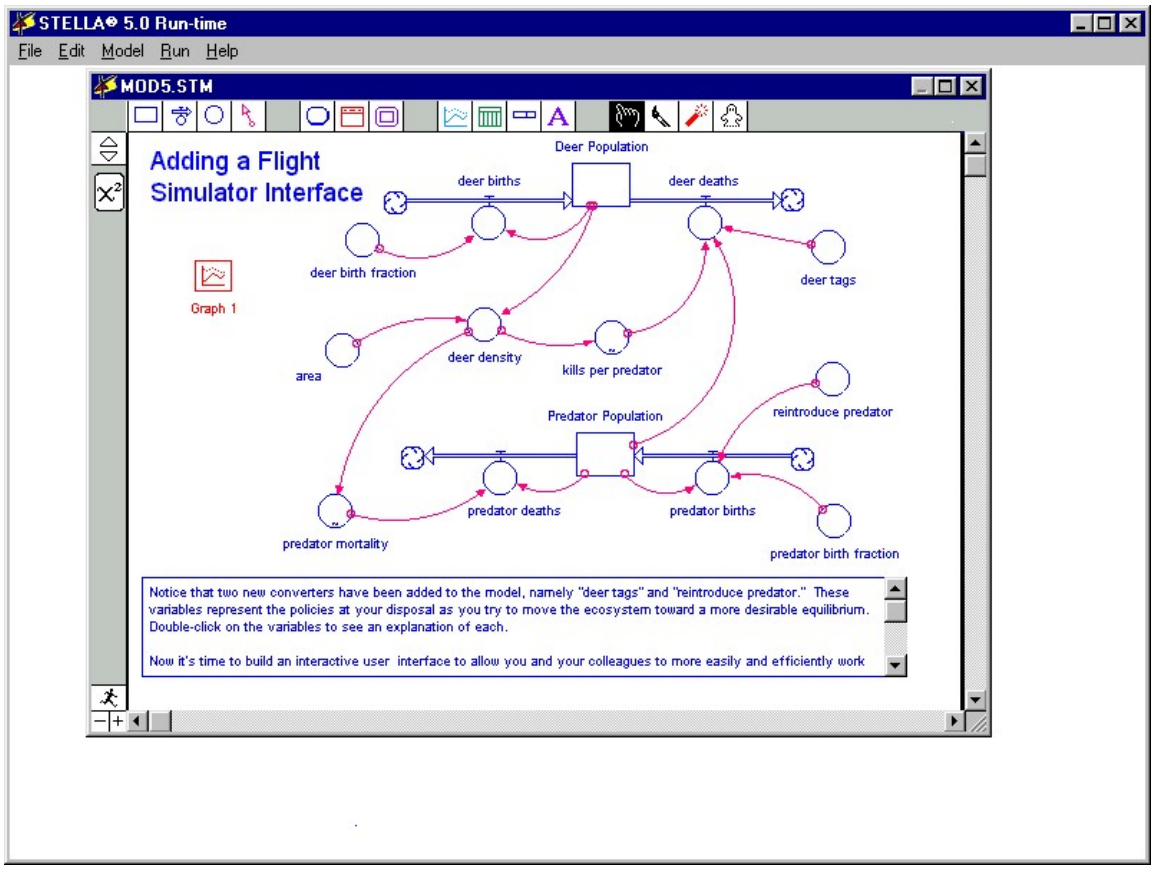


Figura 2.16: Um exemplo de aplicação implementado em Stella [Stella, 2001]

2.4 Vantagens e Desvantagens das Linguagens Visuais

Vantagens das Linguagens Visuais

A mente humana é visualmente orientada, permitindo que as pessoas assimilem novas informações pelo descobrimento de inter-relações gráficas em figuras complexas [Raeder, 1985]. As principais vantagens das linguagens visuais estão freqüentemente relacionadas à maior facilidade de interpretação oferecidas pelas figuras e diagramas nelas utilizados. Esta premissa básica é bastante plausível, pois os programadores usualmente utilizam diagramas para desenvolver algoritmos ou estruturas de dados.

De acordo com os argumentos levantados por Raeder, as principais vantagens das linguagens visuais são:

- Acesso aleatório;
- Maior taxa de transferência de informação;
- Multidimensionalidade das expressões visuais; e
- Descrição concreta ou abstrata.

Uma das vantagens das linguagens visuais é o acesso aleatório à informação. Uma figura permite acesso aleatório a qualquer uma de suas partes, além de viabilizar sua observação de forma detalhada ou sumária, enquanto um texto é de natureza seqüencial.

Ademais, as figuras podem ser acessadas e decodificadas mais rapidamente, tornando sua captação pelos sensores humanos mais adequada, porque estão aptos a receber e processar imagens em tempo real. Desta forma, as linguagens visuais permitem uma maior taxa de transferência de informação ao usuário.

As figuras são também multidimensionais e permitem a definição de uma linguagem mais rica e compacta, através de propriedades visuais, como cor, forma, tamanho, direção, conexão, etc. Por sua vez, um texto é, por definição, unidimensional. Sua leitura e seu entendimento são

seqüenciais. Neste caso, a única propriedade relevante é saber a ordem das expressões representadas pelo texto.

Outra vantagem das linguagens visuais é a possibilidade de realização de uma descrição concreta ou abstrata. As figuras podem descrever objetos de forma concreta ou abstrata. Em outras palavras, as figuras tanto possibilitam a criação de ícones ou diagramas específicos ao domínio de aplicação, quanto permitem a caracterização gráfica de algum conceito mais abstrato.

A verificação dos argumentos apresentados por Raeder dependem de estudos empíricos. Vários estudos foram realizados com o objetivo de medir os benefícios das linguagens visuais. Pandley e Burnett [Pandley, 1993], por exemplo, conduziram uma comparação empírica entre a Linguagem Visual Form/3, o Pascal e a Linguagem APL. Eles investigaram a utilidade dessas linguagens para resolução de problemas de um domínio restrito. Em particular, eles averiguaram a habilidade dos estudantes que participaram do teste em resolver dois problemas: 1) concatenação matricial, ou seja, a concatenação de duas matrizes de dimensões compatíveis; e 2) série de Fibonacci, ou seja, o cálculo dos n primeiros números da série de Fibonacci.

O teste foi realizado por 60 estudantes com conhecimento prévio de 'C' e Pascal. Todavia, apenas um possuía experiência em APL e nenhum deles possuía qualquer noção a respeito do Forms/3. Foi ministrada aos estudantes apenas uma aula de 40 minutos sobre os conhecimentos básicos que permitiriam a implementação dos programas em APL e Forms/3. Cada estudante implementou os dois problemas em cada uma das linguagens em apenas 10 minutos (totalizando 6 programas). Cada participante decidia a ordem de aplicação das linguagens.

As Tabelas 2.1 e 2.2 resumem a compilação dos resultados obtidos no experimento. Para o primeiro problema de concatenação matricial, Forms/s e APL mostraram-se superiores ao Pascal, enquanto para a solução da série de Fibonacci, Forms/3 e Pascal foram considerados superiores.

Tabela 2-1: Resultados para o problema de concatenação matricial [Pandley, 1993]

	Completamente Correto	Quase Correto	Conceitualmente correto, mas logicamente errado	Incorreto
Pascal	7	1	21	31
Forms/3	53	0	2	5
APL	49	3	2	6

Tabela 2-2: Resultados para o problema da série de Fibonacci [Pandley, 1993]

	Completamente Correto	Quase Correto	Conceitualmente correto, mas logicamente errado	Incorreto
Pascal	38	5	4	13
Forms/3	35	9	7	9
APL	15	3	6	36

O domínio de aplicação do experimento foi reconhecidamente restrito, o que impediu qualquer generalização dos resultados. Entretanto, este estudo sugeriu que, para certas tarefas, uma linguagem visual apropriada pode superar a produtividade das linguagens textuais.

Um outro estudo plenamente favorável às linguagem visuais foi relatado por Baroth [Baroth, 1995]. Este estudo envolveu o desenvolvimento de um analisador de telemetria por dois grupos independentes de programadores do *Measurement Technology Center* (MTC). O primeiro grupo desenvolveu o sistema utilizando o LABVIEW, enquanto o segundo utilizou uma linguagem textual (C). Os dois grupos receberam as mesmas especificações de requisitos e o mesmo prazo para desenvolvimento (3 meses). Ao final dos três meses, o grupo que utilizou o LABVIEW estava muito mais adiantado do que o grupo que desenvolveu o sistema na linguagem C. O segundo grupo não chegou nem a completar todos os requisitos necessários para o desenvolvimento do sistema.

Esses dois estudos indicam que a representação visual, comparada com a representação textual, possibilita uma informação mais organizada e explícita, que facilita o desenvolvimento da solução dos problemas. Informações desorganizadas e implícitas acarretam trabalhos adicionais. Portanto, a principal razão dos resultados satisfatórios dos estudos empíricos é que as pessoas conseguem desenvolver melhor uma solução para um problema quando as informações são apresentadas de maneira consistente, organizada e explícita.

Desvantagens das Linguagens Visuais

As principais desvantagens das linguagens visuais são:

- notação visual esparsa;
- elevado custo de desenvolvimento e dificuldade de portabilidade;
- inadequação para soluções genéricas;
- inexistência de uma fundamentação teórica consolidada.

As linguagens visuais caracterizam-se por uma notação esparsa, que gasta mais espaço do monitor do que a notação das linguagens textuais equivalentes. Este problema relaciona-se com o fato das notações visuais não suportarem uma boa densidade de informação. Desta forma, pequenos programas ocupam muito espaço [Nardi, 1993]. Isso pode ser aliviado pelo uso de abstrações procedimentais, isto é, pelo colapso de subdiagramas em símbolos mais simples, que são tratados como uma sub-rotina. Todavia, este problema ainda não está satisfatoriamente resolvido.

Além disso, as linguagens visuais geralmente apresentam maior custo de desenvolvimento e maior dificuldade de portabilidade para outros ambientes de hardware e software [Nardi, 1993]. A maioria das linguagens desenvolvidas é projetada para trabalhar em domínios específicos, muitas vezes não aproveitando trabalhos já realizados. Ademais, os aspectos gráficos da linguagem podem depender de recursos de software ou hardware não facilmente portáveis para outros ambientes.

As linguagens visuais não são adequadas para soluções genéricas. Aquelas relacionadas a domínios específicos tiveram maior aceitação por parte dos usuários finais. Todavia, se elas forem utilizadas em domínio não inicialmente previsto, observa-se uma queda em seu

desempenho. É bom salientar que existem linguagens visuais desenvolvidas para serem aplicadas a um domínio genérico (por exemplo, Prograph). Contudo, na opinião do autor deste trabalho de pesquisa, essas linguagens não são facilmente utilizadas por usuários não treinados. Por outro lado, elas são excessivamente prolixas para usuários programadores.

As linguagens visuais não possuem fundamentação teórica consolidada e alguns aspectos teóricos não são totalmente entendidos. A teoria e o entendimento das linguagens visuais encontram-se muito menos avançados, parcialmente em decorrência da falta de integração entre as diferentes comunidades científicas pesquisadoras de linguagens visuais [Marriott, 1998]. O mesmo não se pode dizer das linguagens textuais, cujas técnicas de formalização e especificação se encontram atualmente mais desenvolvidas [Marriott, 1998]. além de existir um substancial progresso no entendimento dos aspectos cognitivos de seu processamento.

2.5 Conclusão

Em primeiro lugar, o autor apresentou os paradigmas mais utilizados pela comunidade científica para descrever a base semântica das linguagens visuais. Em segundo lugar, expôs alguns ambientes de programação, que aplicam cada um dos paradigmas de programação apresentados. Em último lugar, o autor destacou algumas vantagens e desvantagens das linguagens visuais, que explicam os limites de sua aceitação por parte de usuários finais mais especializados.

Relatos recentes mostram que linguagens visuais, quando bem projetadas, possuem inúmeras vantagens sobre as textuais, tais como: acesso aleatório à informação, maior capacidade de transferência de informação, expressividade da linguagem e descrição de um problema de forma concreta ou abstrata.

Na investigação realizada sobre linguagens visuais, constatou-se que as linguagens relacionadas a domínios específicos tiveram maior aceitação por parte dos usuários finais.

Capítulo 3

Um Modelo para Programação Visual de Matrizes - ProVisual

Neste capítulo, o autor propõe um Modelo para a Programação Visual de Matrizes (ProVisual) fundamentado no paradigma de fluxo de dados e de planilhas. O fluxo de dados e a planilha constituem a base semântica do modelo proposto, enquanto o grafo direcionado constitui sua base sintática.

3.1 Definição do ProVisual

As linguagens visuais permitem que, de maneira simples e direta, os programadores esbocem e demonstrem os relacionamentos entre os dados e suas transformações, em vez de os traduzirem em comandos textuais, ponteiros e variáveis. Em outras palavras, as linguagens visuais podem simplificar o ato de programar.

Neste capítulo o autor propõe o Modelo para Programação Visual de Matrizes – ProVisual, fundamentado nos paradigmas de fluxo de dados e de planilha[3], procurando um estilo de programação simples e concreto.

A busca da simplicidade dificulta a construção de uma linguagem visual efetiva para resolver problemas de larga escala[4] [Burnett, 1995]. Para contornar essa dificuldade, procurou-se um equilíbrio na utilização de conceitos tanto concretos como abstratos na modelagem do ProVisual. O método utilizado para atingir este objetivo pode ser resumido nos seguintes pontos:

1. Explicitação dos inter-relacionamentos entre os elementos sintáticos da linguagem - Evitando, assim, a construção de programas com dependências escondidas, ou seja, aquelas que não se encontram totalmente visíveis ou explícitas nos programas [Green, 1996]. Por exemplo, efeitos colaterais podem ocorrer em muitas linguagens de programação como resultado da não explicitação formal de relações entre as variáveis no processo de comunicação com os procedimentos;

2. Utilização de poucos conceitos para a programação -
   Visando a simplicidade e a legibilidade de um programa desenvolvido com uma linguagem visual. Esses devem ser os compromissos principais de um processo de modelagem. Por exemplo, faz-se necessário minimizar a utilização de ponteiros, alocação de memória, declaração de arquivo e de variáveis. Além disso, deve-se procurar reduzir a necessidade de recursão, um mecanismo amplamente utilizado nas linguagens visuais, pois não é facilmente aplicada por usuários inexperientes;

[3] Neste trabalho de pesquisa, os termos *matriz* e *planilha* são considerados sinônimos.

3. Fornecimento de recursos para *feedback i*mediato -
   Permitindo a atualização imediata dos resultados a partir de uma alteração no programa, o que possibilita a localização de eventuais erros de forma mais rápida e mais consistente;

4. Provisão de recursos concretos para programação -
   Tornando o programa mais legível e compacto. Os recursos concretos são instâncias particulares do domínio de interesse, utilizadas para expressar alguns aspectos do programa [Burnett, 1999].

5. Dotação de recursos para manipulação direta -
   Possibilitando a leitura, a exploração e a alteração do programa e de seus dados em tempo de desenvolvimento ou de execução. A manipulação direta pode ser definida como o sentimento experimentado pelo programador ao manipular diretamente um objeto [Shneiderman, 1983].

6. Utilização de texto em expressões matemáticas -
   Combinando a conveniência da notação visual com a força de abstração da notação textual [Erwig, 1995], particulamente útil para escrever expressões matemáticas;

7. Adoção de estratégias contra a limitação do espaço físico do monitor -
   Permitindo a resolução de problemas de maior porte. Os problemas relacionados ao espaço físico do monitor independem do domínio de aplicação [Burnett, 1995]. Possíveis soluções devem incluir recursos para a programação e para a representação visual de um programa. No que se refere à programação, devem-se adotar a abstração procedimental e a persistência dos dados, bem como reduzir a necessidade de recursão e de iteração. Por sua vez, no que diz respeito à representação visual, deve-se endereçar soluções para a representação estática do programa, para o uso físico do monitor e para a documentação.

[4] Neste relatório de pesquisa o problema de larga escala está relacionado à extensão da solução obtida.

8. Verificação do modelo –
   Viabilizando a avaliação dos resultados de aplicação da linguagem em termos de legibilidade (entendimento) da solução obtida.

O ProVisual pode ser considerado um resultado de aplicação do método acima descrito. O modelo considera o fluxo de dados e a planilha como sua base semântica e o grafo direcionado como sua base sintática. Ele constitui-se por um conjunto de diagramas no plano bidimensional, associado a um conjunto de regras de transformação. Os dados são representados por planilhas e as regras de transformação das planilhas são representadas por uma rede de Fluxo de dados.

O modelo prevê a manipulação direta para a construção e a interação com cada componente da rede. As planilhas são construídas a partir da definição de fórmulas ou valores para suas células. As fórmulas podem incluir constantes ou referências a outras células (da mesma matriz ou não). Ademais, podem ser executadas instantaneamente com os dados atuais, ou posteriormente (em *batch)* com os dados de outras planilhas ou matrizes, segundo um estilo de programação por exemplos. Deste modo, a planilha pode ser visualizada como uma abstração funcional e não apenas como um repositório de dados de uma matriz.

Os programas, concebidos de acordo com o modelo ProVisual, devem ser produzidos como diagramas de fluxo de dados, contendo pictogramas[5] que, por sua vez, simbolizam dados e processos[6]. Os dados são representados por retângulos. Os processos também são descritos por retângulos e possuem duas barras verticais em suas extremidades, que indicam, respectivamente, o ponto de entrada dos parâmetros e o ponto de saída dos valores produzidos internamente. Nos dois casos, pode-se usar um rótulo para nomear os pictogramas. A Figura 3.1 ilustra os dois pictogramas básicos da linguagem. O retângulo, representando um processo, pode simbolizar tanto a chamada de uma sub-rotina, quanto uma expressão textual. A utilização de texto permite o desenvolvimento de programas mais sucintos.

[5] O pictograma é uma representação gráfica de algum recurso da linguagem visual.
[6] Neste trabalho de pesquisa, o termo processo designa um encapsulamento de regras de transformação. Pode ser visto como um módulo ou procedimento das linguagens textuais.

O ProVisual provê um conjunto de recursos computacionais, retratados com ícones próprios, que visam dotá-lo do instrumental necessário para o desenvolvimento de programas relacionados à manipulação matricial.

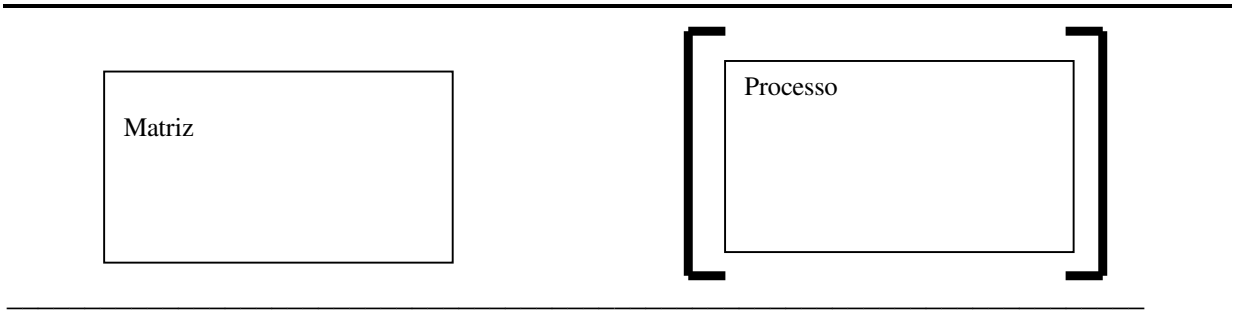


Figura 3.1: Representação gráfica de uma matriz e de um processo no ProVisual.

Ademais, o modelo permite a abstração funcional por meio de módulos[7] construídos a partir da composição funcional de outros módulos internos (disponíveis na linguagem) ou externos (construídos pelo programador). A Figura 3.2 mostra a interface principal do ProVisual com quatro janelas:

1) a janela de persistência, que contém a lista de matrizes com dados preservados durante e após a execução do programa;
2) a janela de módulos, que contém a lista de módulos implementados para o programa;
3) a janela de funções, que retrata a lista de funções reutilizadas; e
4) a janela de módulos em desenvolvimento, que recebe a rede de fluxo de dados (representada por ícones e arcos). Esta janela representa um módulo e possui uma barra à esquerda para os parâmetros de entrada e uma barra à direita para os parâmetros de saída.

A interface principal do ProVisual também possui um conjunto de botões e um menu de opções. O conjunto de botões indica os elementos sintáticos que podem ser utilizados na composição de um programa.

[7] Neste trabalho de pesquisa módulo e sub-rotina são sinônimos.

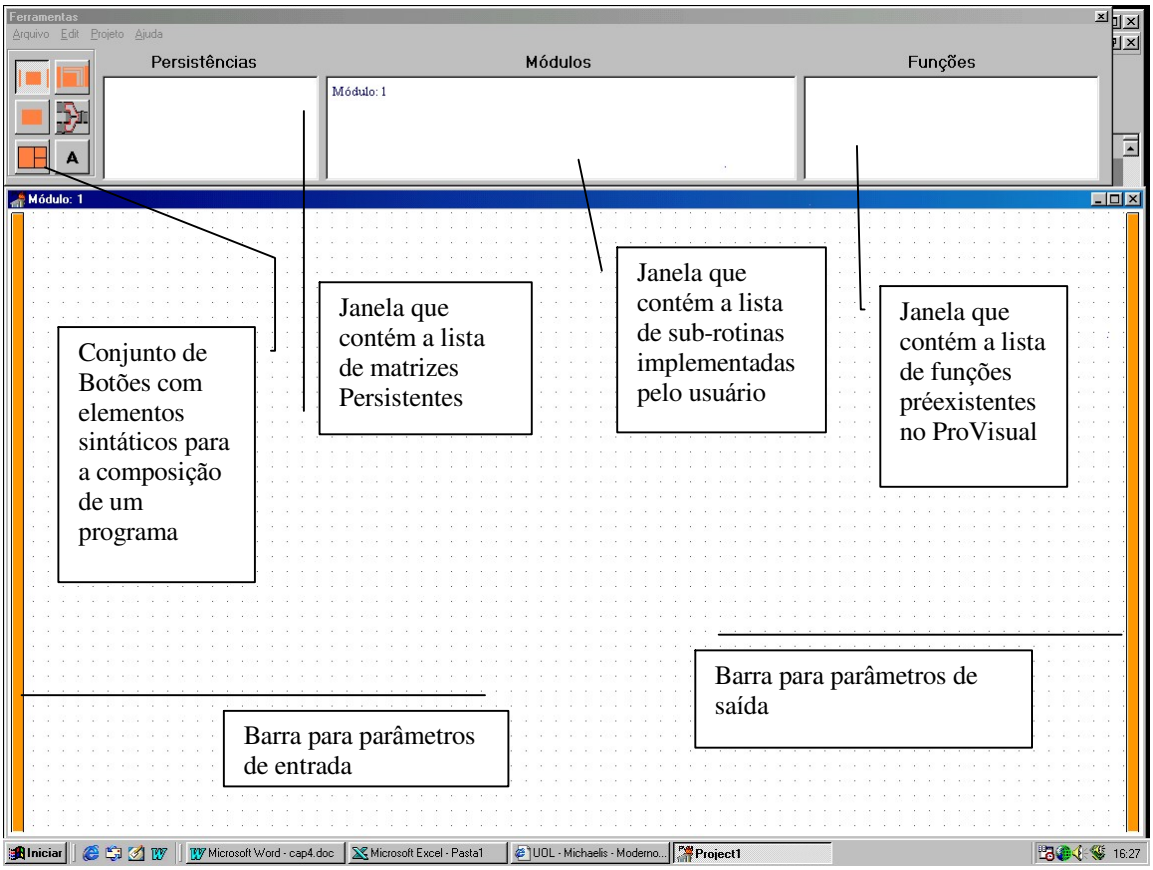


Figura 3.2: Interface principal do ProVisual.

3.2 A Matriz

A representação visual de uma matriz é geralmente bem entendida e, muitas vezes, utilizada com propósitos didáticos. Por sua vez, a representação textual de uma matriz pode ser um tanto abstrata e ininteligível para usuários inexperientes e, principalmente, sem treinamento em programação. Entretanto, para que a representação visual seja efetiva, o usuário deve ser capaz de manipulá-la sem a necessidade de recorrer a uma notação textual.

Tratando-se do ProVisual, cada matriz é representada conforme mostrado na Figura 3.3. Durante o processo de edição da matriz, ela revela-se graficamente sob a forma de duas planilhas

posicionadas lado a lado, descrevendo, respectivamente, os dados da matriz e uma visão sobre os mesmos.

A planilha de dados permite a construção de uma matriz a partir de duas estratégias diferentes. A primeira estratégia gera uma matriz a partir de cópias e reordenações espaciais de outras matrizes ou submatrizes. A segunda estratégia atribui valores/fórmulas para as células de uma matriz/planilha.

A planilha que representa a visão dos dados da matriz serve para descrever como eles serão enviados para os processos subseqüentes. Esta visão pode ser a própria matriz, um elemento, um vetor ou até mesmo um conjunto de submatrizes, em um processo iterativo ou não.

As planilhas podem assumir o papel de funções ou variáveis do tipo matriz. Como funções, recebem valores via parâmetros de entrada e produzem resultados que serão utilizados por outros processos. Como variáveis, armazenam dados bidimensionais. Contudo, o ProVisual não distingue qualquer um dos dois papéis assumidos por uma planilha.

Uma matriz pode ser referenciada dentro do procedimento, de forma semelhante a uma variável local, ou pode ser acessada globalmente. Neste último caso, faz-se necessário informar que ela é uma matriz persistente e pode ser manipulada, não apenas dentro do próprio programa, mas também por outros programas.

A Figura 3.3 ilustra a estrutura geral de uma matriz (N x M). A planilha da esquerda descreve os dados da matriz, que pode conter, opcionalmente, um nome temporário, dois atributos, simbolizando seu número de linhas e colunas (representados por retângulos na parte superior, à esquerda da figura) e setas rotuláveis, utilizadas para informar ou recuperar o número da linha e/ou da coluna em que se está atuando. A planilha da direita contém uma barra de saída e duas setas, que recuperam, se necessário, o endereço da célula ativa em um processo iterativo.

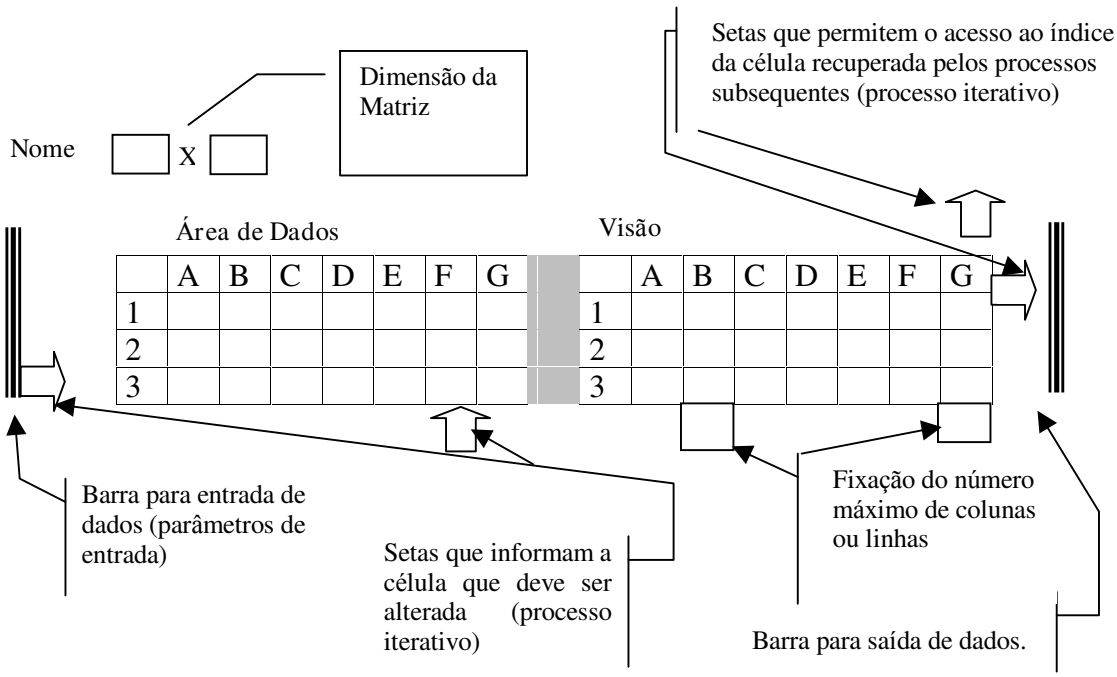


Figura 3.3: Representação gráfica de uma matriz.

Área de Dados

Área reservada para a definição dos dados de uma matriz.

Visão

Área reservada para a definição de uma visão dos dados da matriz. Esta visão permite que os dados da matriz possam preparados convenientemente e enviados para outros processos.

Parâmetros de Entrada

As planilhas podem receber dados de outras planilhas ou processos, bastando para tanto fazer a conexão de um ou mais fluxos de dados à barra de entrada de dados da planilha. Cada conexão recebe, automaticamente, uma determinada cor e, opcionalmente, um rótulo. Esta cor servirá para identificar a planilha de origem dos dados de entrada.

Parâmetros de saída

Uma planilha também pode enviar dados para outros processos ou planilhas, via barra de saída. Neste caso, cada identificador (rótulo ou cor) de algum elemento (escalar, submatriz ou índice) da parte correspondente à visão de uma planilha pode ser utilizado por qualquer processo conectado à barra de saída.

O ProVisual permite que várias operações sejam realizadas nas planilhas relacionadas aos dados e à visão de uma matriz. A planilha de dados é utilizada para definir uma matriz como um repositório de dados ou uma função. Neste caso, pode-se criar ou alterar uma matriz a partir da reordenação de dados de outras matrizes ou a partir da entrada direta de valores. A planilha da visão sobre os dados é utilizada para segmentar uma matriz em diferentes regiões e/ou para criar um padrão de iteração. A Figura 3.4 ilustra as operações que podem ser realizadas nas planilhas de dados e de visão de uma matriz do ProVisual .

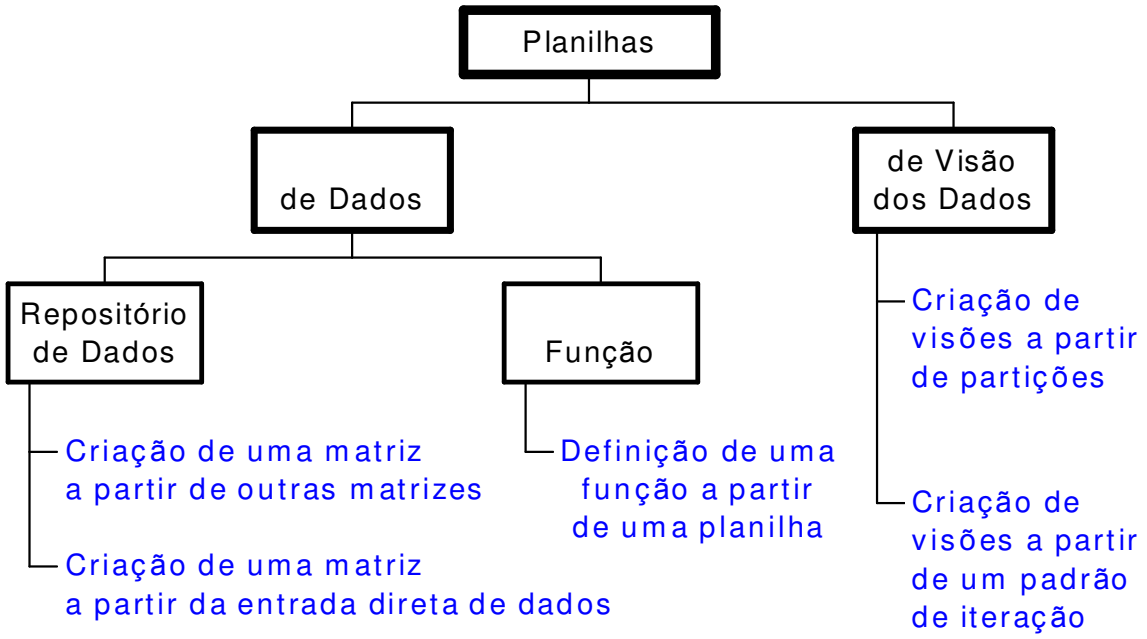


Figura 3.4: Operações realizadas nas planilhas do ProVisual .

3.2.1 Criação de uma Matriz de dados a partir de outras Matrizes

O ProVisual permite que uma matriz seja criada ou montada a partir de dados de outras matrizes. Neste caso, cada parâmetro de entrada deve ser nomeado e/ou colorido para que seja identificado de forma única. Cada parâmetro alimentará com dados uma respectiva partição da matriz que possua a mesma cor do arco que representa o parâmetro. Cada partição ocupará uma área retangular que não poderá ser sobreposta a outras áreas.

No processo de edição da matriz, o programador informa se deseja entrar com fórmulas ou proceder a uma montagem de matriz via composição com outras matrizes. Caso a segunda opção seja a escolhida e as matrizes de entrada ainda não existam, pois só serão criadas em tempo de execução, verifica-se a consistência da operação, isto é, se as matrizes possuem dimensões compatíveis com a composição desejada, apenas quando estiverem todas disponíveis.

A Figura 3.5 retrata um exemplo de criação de matriz por composição de partições de outras matrizes. Nesta figura, as matrizes verde e cinza devem possuir o mesmo número de linhas, enquanto a soma do número de linhas das matrizes amarela e azul deve ser igual ao número de linhas existentes na matriz de cor verde. Esta verificação de compatibilidade é imediatamente efetivada se as matrizes de entrada estiverem à disposição em tempo de composição (edição).

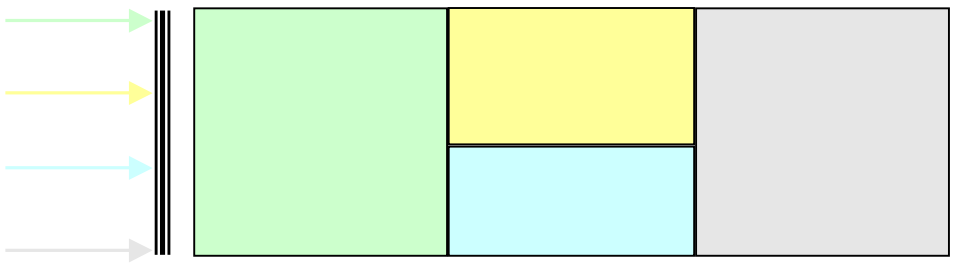


Figura 3.5: Criação de uma matriz por composição de quatro partições e quatro parâmetros (matrizes) de entrada.

Com o intuito de simplificar esta verificação, as partições precisam ser criadas de forma hierárquica. Inicialmente, devem ser criadas as partições maiores com o toque do *mouse* em qualquer uma das linhas fronteiriças. Se a linha da fronteira for vertical, cria-se uma partição horizontal. Caso contrário, obtém-se uma nova partição vertical. Estas novas partições podem ser divididas novamente, seguindo-se o mesmo processo. Após a definição de todas as partições, deve-se atribuir a cada uma delas um dos identificadores dos parâmetros de entrada (cor ou nome). A Figura 3.6 exemplifica processo de criação de uma matriz a partir de outras matrizes.

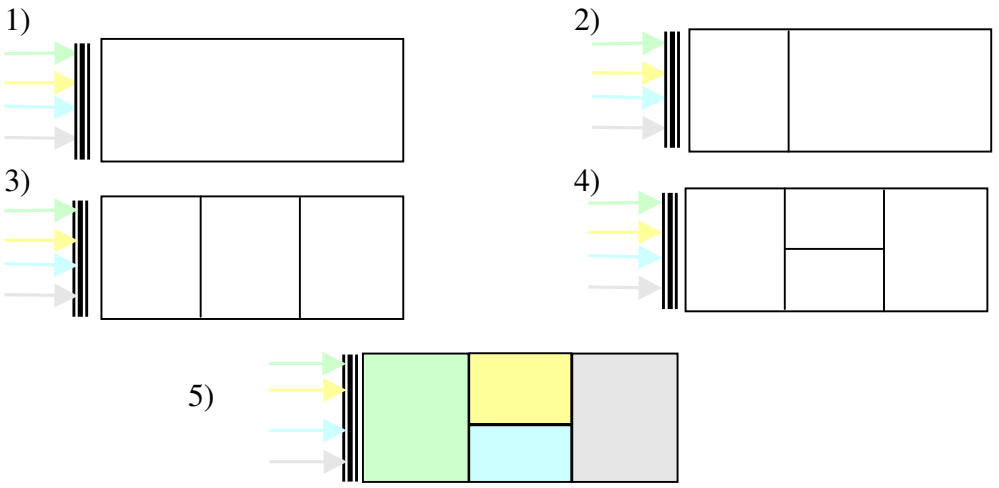


Figura 3.6: Passos para a criação de uma determinada matriz a partir de dados fornecidos por outras 4 matrizes.

Na Figura 3.6, os seguintes passos foram realizados:

1) Criação da matriz e seus parâmetros de entrada;
2) Criação da primeira partição, com a indicação em uma determinada posição da linha superior ou inferior do retângulo, representando a criação de duas partições da matriz;
3) Aplicação do passo dois em outra posição para a obtenção de outra partição vertical;
4) Geração de duas novas partições horizontais a partir da indicação de linhas verticais que delimitam a partição central; e
5) Atribuição de parâmetros de entrada (cores) às partições obtidas.

Uma matriz também pode ser criada iterativamente por meio de dados recebidos de outras matrizes ou funções. Esses dados são repassados via algum processo iterativo. Em outras palavras, os dados são recebidos paulatinamente e um elemento ou bloco de elementos é inserido na nova matriz para cada um deles.

Neste sentido, uma matriz pode conter um padrão de iteração para geração de seus elementos. Este padrão pode incluir os valores e seus índices, as dimensões envolvidas e, primordialmente, a ordem de ocorrência da geração dos elementos da matriz no processo de iteração. Um padrão de iteração, baseada em [Auguston, 1997], contém os seguintes símbolos:


representa o valor de um único elemento;


representa um conjunto de valores (linhas, colunas, …);

… indica a direção do padrão de iteração; e

indica a célula que será gerada.

A Figura 3.7 ilustra uma matriz criada a partir do envio de vários elementos (A e B). Cada par de elementos recebido será multiplicado e uma nova célula do vetor será definida. Quando não existirem mais dados a serem recebidos, a geração do vetor será finalizada e, com isso, poderá alimentar algum outro processo.

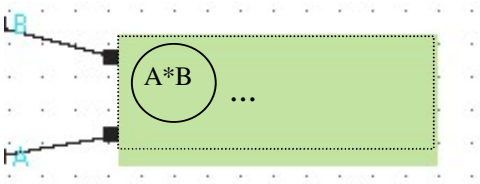


Figura 3.7: Geração iterativa de uma matriz.

3.2.2 Criação de uma Matriz de dados a partir da Entrada Direta de Dados

O ProVisual admite que uma matriz seja criada ou editada manualmente pelo usuário com o auxílio de uma planilha. Cada célula da planilha pode conter constantes ou fórmulas. As fórmulas podem possuir constantes, referências a outras células da mesma planilha ou referências às células de outras planilhas representadas pelos parâmetros de entrada.

A planilha do ProVisual é diferente das planilhas tradicionais[8]. Uma das diferenças é a não utilização do conceito de endereçamento absoluto, tornando mais simples e intuitivo o processo de cópia de células que possuem fórmula.

O processo de cópia de fórmulas adotado pelas planilhas tradicionais não é muito adequado às necessidades de usuários discricionários[9], isto é, usuários que necessitam dividir a planilha em uma série de blocos e aplicar uma função a cada um dos blocos.

A Figura 3.8 mostra a dificuldade do processo de cópia de fórmulas por planilhas tradicionais. Por exemplo, considerando uma série de números colocados na primeira linha de uma planilha comercial, deve-se proceder ao agrupamento desses dados em blocos de três e computar sua soma. A resolução deste problema requer uma boa capacidade de programação por parte dos usuários das planilhas, uma vez que várias funções precisam ser combinadas em uma fórmula razoavelmente complexa.

	A	B	C	D	E	F	G	H	I	J	K	L
1		3	3	4	2	1	3	4	2	1	4	...
2												
3		10	6	7	...							
4		B1	E1	H1	...							
5		D1	G1	J1	...							

Fórmulas associadas às células da coluna B que serão copiadas para outras colunas (c,d,...):

B3 → =SOMA(INDIRETO(B4):INDIRETO(B5))

B4 → =ENDEREÇO(COL(A1);COL(A1)+3*(COL()-COL(A1));4)

B5 → =ENDEREÇO(COL(A1);COL(A1)+3*(COL()-COL(A1))+3-1;4)

Figura 3.8: Soma de sucessivos blocos de tamanho 3 orientados horizontalmente (baseado em [Hendry, 1994])

[8] Nesta pesquisa consideram-se como planilhas tradicionais aquelas baseadas no padrão do VISICALC.
[9] Usuários que trabalham com dados discretos ou descontínuos.

A Figura 3.8 ilustra uma solução particular do problema acima apresentado para a planilha Excel. Soluções similares são requeridas por planilhas de outros desenvolvedores. Esta solução exige a criação ou programação de uma fórmula complexa, obrigando o usuário a conhecer detalhes acerca de funções e seus parâmetros para resolver o problema sugerido acima. Outras soluções igualmente complexas precisam ser formuladas para outros tipos de problemas relacionados à aplicação de fórmulas em diferentes formatos de blocos.

A Figura 3.9 retrata graficamente alguns padrões de manipulação algébrica de blocos que não poderiam ser resolvidos com as planilhas tradicionais. Ademais, retrata a direção dos blocos para a formação de novos valores da planilha. No exemplo 1, a distância entre os blocos é maior que 1. Nos exemplos 2, 3 e 5, esta distância é exatamente igual a 1. Finalmente, no exemplo 4 e 6, a distância é menor que 1.

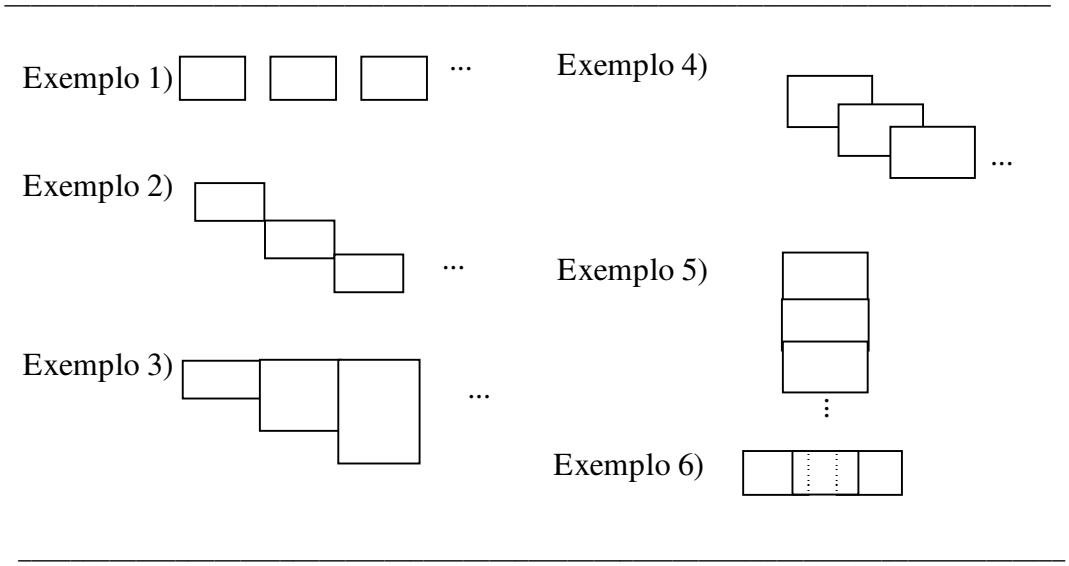


Figura 3.9: Exemplos de padrões para geração de valores em uma planilha (baseado em [Hendry, 1994]).

A dificuldade de utilização dos recursos tradicionais para resolver a manipulação algébrica de blocos está relacionada à forma de atualização das referências das células no processo de cópia.

As planilhas tradicionais ajustam automaticamente os endereços existentes nas fórmulas pela distância entre a fórmula (fonte da cópia) e a posição de sua colagem. Esta estratégia pode

ou não proporcionar o resultado desejado. Para inibir esses ajustes, o usuário pode especificar uma célula com endereçamento absoluto (com a utilização do símbolo '$'). Todavia, a colocação deste símbolo não é uma tarefa intuitiva, o que exige primeiro a verificação do resultado obtido.

A formalização deste processo, baseada no trabalho de Hendry [1995], mostra a limitação da notação tradicional utilizada pelas planilhas e, em adição, sugere um caminho para aperfeiçoá-la.

A Figura 3.10 mostra a representação esquemática das variáveis envolvidas no processo de cópia de fórmulas nas planilhas tradicionais. A fórmula a ser copiada é representada pela região com endereço Cf e Lf, que indicam a linha e a coluna da fórmula de entrada (por exemplo, para referência à célula B2, tem-se: Cf = B e Lf = 2).

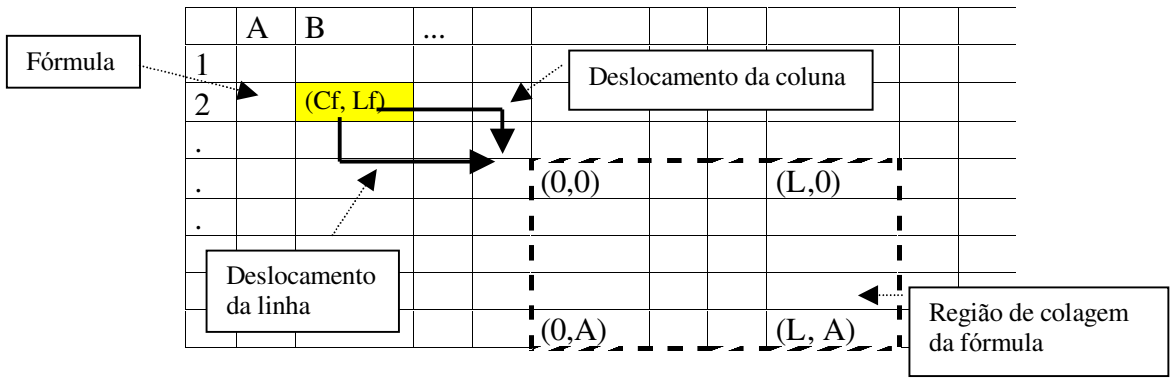


Figura 3.10: Variáveis envolvidas no processo de cópia de fórmulas nas planilhas tradicionais.

Os deslocamentos de linha e coluna indicam a distância da fórmula à região de saída (colagem da fórmula), enquanto L e A representam, respectivamente, a largura e a altura da região de saída.

Quando uma fórmula é colada, ela é assentada em cada célula da região de colagem, mas as referências são atualizadas para refletir a distância desta célula à fórmula de origem.

Assumindo que cada nova referência nas células da região de saída é representada pelo par ordenado (Ci, Lj), a derivação da semântica da operação de cópia pode ser especificada por equações que computam valores corretos de C_0, C_1, C_2, ..., C_L e L_0, L_1, L_2, ..., L_A da região de colagem:

$$C_0 = C_f + \text{deslocamento da coluna} \qquad (1)$$

Os endereços das colunas à direita de C_0 podem ser computados a partir de C_0, produzindo-se:

$$C_1 = C_0 + 1,\ C_2 = C_0 + 2,\ ..., C_L = C_0 + L\text{-}1. \qquad (2)$$

O mesmo procedimento pode ser adotado para computar os endereços das linhas na região de saídas:

$$L_0 = L_f + \text{deslocamento da linha}, \qquad (3)$$

$$L_1 = L_0 + 1,\ L_2 = L_0 + 2,\ ...,\ L_A = L_0 + A\text{-}1 \qquad (4)$$

As fórmulas acima são adequadas para descrever o processo de atualização dos endereços relativos. Por outro lado, se o endereço na fórmula é absoluto, a distância de colagem da cópia não tem qualquer significado:

$$C_0 = C_f,\quad C_1 = C_f,\quad ...,\ C_L = C_f \qquad (5)$$

$$L_0 = L_f,\quad L1 = L_f,\quad ...,\ L_L = L_f \qquad (6)$$

A partir das fórmulas acima, pode-se derivar as seguintes relações para cálculo dos endereços das colunas e linhas:

$$C_i = \begin{cases} C_f + \text{deslocamento coluna} + i & \text{Caso } C_f \text{ com endereço relativo} \\ C_f & \text{Caso } C_f \text{ com endereço absoluto} \end{cases} \qquad (7)$$

$$Li = \begin{cases} L_f + \text{deslocamento linha} + j & \text{Caso } L_f \text{ com endereço relativo} \\ L_f & \text{Caso } L_f \text{ com endereço absoluto} \end{cases} \quad (8)$$

Onde: $0 \le i \le L\text{-}1$ e $0 \le j \le A\text{-}1$.

Pode-se produzir as seguintes fórmulas análogas a partir de (7) e (8):

$$C_i = C_f + \sum_{k=1}^{DeslocamentoColuna} dc + \sum_{k=1}^{i} dc \quad (9)$$

$$L_j = L_f + \sum_{k=1}^{DeslocamentoLinha} dl + \sum_{k=1}^{j} dl \quad (10)$$

Se endereçamento relativo, $dc = 1$ e $dl = 1$;
Se endereçamento absoluto, $dc = 0$ e $dl = 0$;
$0 \le i \le L\text{-}1$ e $0 \le j \le A\text{-}1$

Analisando as fórmulas acima e considerando as planilhas tradicionais, pode-se observar que os valores das variáveis *dc* e *dl* estão restritos a 0 e 1, indicando o endereçamento absoluto ou relativo (inserção ou não de $ nos endereços das fórmulas). Todavia, essas mesmas variáveis podem ser utilizadas para produzir um processo de cópia mais expressivo, que consiga gerar progressões, conforme os exemplos da Figura 3.9.

Hendry propôs uma técnica baseada em um estilo simples de programação por exemplos [Hendry, 1995]. Nesta técnica, três passos são necessários:

1. Escrever as duas primeiras fórmulas da progressão;
2. Especificar a fronteira da região de geração da progressão; e
3. Executar a cópia da fórmula para as células da região demarcada.

Para aplicar esta técnica, basta calcular os valores de *dc* e *dl*, derivados das duas primeiras fórmulas fornecidas no passo 1, desde que possuam o mesmo número de termos. Por exemplo, dadas duas fórmulas seqüenciais (C_0, L_0) e (C_1, L_1), pode-se obter os valores de *dc* e *dl,* a partir das seguintes expressões:

$$dc = C_1 - C_0 \qquad (11)$$

$$dl = L_1 - L_0 \qquad (12)$$

Todo o processo de indução pode ser iniciado a partir deste cálculo. A Figura 3.11 exemplifica alguns casos de progressão que, baseados nos dois valores iniciais, admitem a produção de uma série de computações. Neste exemplo, o retângulo tracejado indica o resultado da cópia.

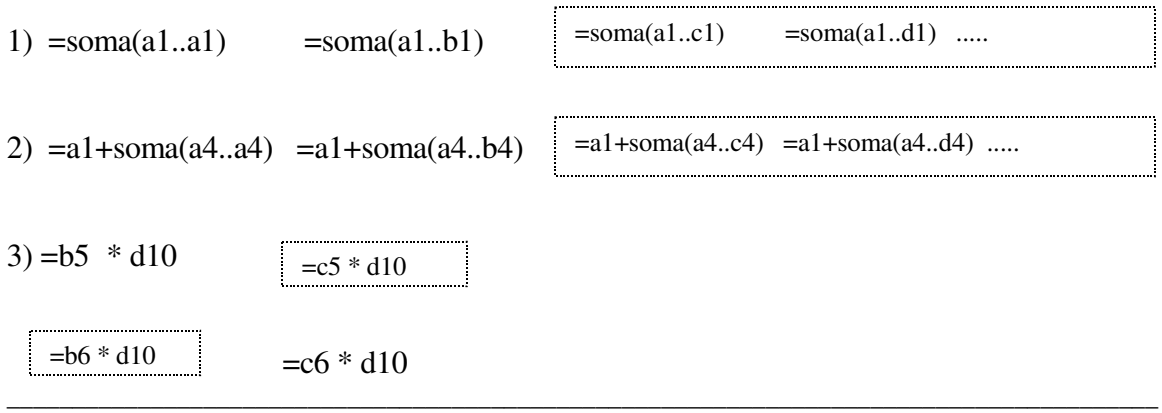


Figura 3.11: Exemplos não expressáveis pelo esquema usual de cópia das planilhas tradicionais (baseado em [Hendry, 1994]).

No primeiro exemplo, uma soma cumulativa é computada por coluna. O cálculo de *dc, dl* por diferença de termos produz:

termo1, (*dc, dl*) = (a-a, 1-1)= (0,0);
termo2, (*dc, dl*) = (b-a,1-1) = (1,0).

No segundo exemplo, uma soma cumulativa é calculada, adicionando-se uma constante a cada termo:

termo1, (*dc, dl*) = (a-a,1-1) = (0,0);

termo2, (*dc, dl*) = (a-a,4-4) = (0,0);

termo3, (b-a, 4-4) = (1,0).

Finalmente, no terceiro exemplo, uma constante é multiplicada a cada célula do bloco de saída:

termo1, (*dc, dl*) = (c-b, 6-5) = (1, 1);

termo2, (*dc, dl*) = (d-d, 10-10) = (0,0).

Neste último exemplo uma observação deve ser feita com relação às fórmulas (9) e (10). A mudança do componente coluna (C*i*) somente ocorre se *i* for alterado. Similarmente, Lj se modifica se *j* for alterado. Isto significa que, se as duas primeiras fórmulas forem colocadas horizontalmente (ou verticalmente), o passo 2 do processo de indução só poderá ser horizontal (ou vertical). Todavia, se o processo de indução for diagonal, tanto Ci como Lj devem ser alterados.

Esta estratégia de indução elimina totalmente a necessidade de se utilizar algum símbolo especial (por exemplo, $) para fixar uma referência num processo de cópia. Por outro lado, introduz a dificuldade de se saber, de maneira direta, se um determinado termo da fórmula é uma constante. Este aspecto negativo pode ser minimizado pela utilização de algum recurso visual para indicar a constância do termo.

O ProVisual propõe um processo de indução gráfica apoiado na proposta de Hendry, contudo estendido para trabalhar com o conceito de abstração procedimental. Neste processo, o usuário deve editar dois exemplos consecutivos para apresentar uma proposta gráfica de indução, que pode ser aceita ou modificada. Esta proposta, gerada com base nas duas primeiras fórmulas, deve ser estruturada em três partes: a fórmula-base, o fluxo de dados e a região de preenchimento. Esta última pode possuir opcionalmente um parâmetro de limitação, que se torna fundamental quando este processo de indução é utilizado como função.

O ProVisual deve trabalhar com os seguintes passos:

1. Edição textual das duas primeiras fórmulas indutoras-
   As duas primeiras fórmulas do processo de indução devem ser escritas. O número de termos das duas fórmulas deve ser igual. Entretanto, o número total de termos pode ser desconhecido;

2. Sugestão do terceiro elemento do processo de indução –
   A fórmula para o terceiro elemento do processo de indução deve ser sugerida a partir das duas primeiras fórmulas. Neste ponto, pode-se reeditar as duas primeiras fórmulas;

3. Aceitação do terceiro elemento do processo de indução-
   Uma área para preenchimento da seqüência deve ser proposta. Esta área pode ser editada e delimitada de forma gráfica ou textual;

4. Aplicação da fórmula a todo o bloco-
   O esquema de indução pode ser aplicado a todo o bloco. Esta aplicação pode se proceder linha a linha ou coluna a coluna, de acordo com a direção das duas primeiras fórmulas do passo 1. Neste caso, será necessária a edição textual da fórmula de transição de linha para coluna ou de coluna para linha (ver Seção 3.2.3);

5. Execução do preenchimento em tempo de desenvolvimento –
   O preenchimento deve ser executado em tempo de desenvolvimento se os dados de entrada estiverem disponíveis. Caso contrário, devem ser preenchidos no futuro com dados de outra planilha.

No passo 3, necessita-se da confirmação ou da alteração da proposta de região de preenchimento. Neste último caso, pode-se seguir três caminhos:

1) Modificar o ponto final de indução;

2) Definir o número total de células da série geradas. Esta informação pode ser um número absoluto, uma fórmula, um parâmetro recebido pela matriz na estrutura de fluxo de dados do programa ou uma condição; e
3) Não estabelecer qualquer limite pré-definido.

Em qualquer um dos dois últimos caminhos, deve-se utilizar o símbolo visual, que indica a direção da geração com ou sem limite estabelecido. A Tabela 3.1 exemplifica os elementos visuais utilizados para a geração de valores nas planilhas, definindo a direção e a condição de parada. A Tabela 3.2 apresenta os símbolos de delimitação.

A utilização do construtor de indução é particularmente útil quando os termos das fórmulas iniciais dependem de matrizes externas, isto é, matrizes fornecidas como parâmetros para a planilha das fórmulas. Neste caso, cada termo da fórmula precisaria ter a sua cor definida de acordo com as cores dos parâmetros ou possuir um identificador extra de acordo com o rótulo do parâmetro de entrada desejado. Por outro lado, os termos da fórmula que possuam a cor preta e sem identificador de parâmetro são referências à célula de uma mesma matriz.

A Figura 3.12 retrata dois exemplos de fórmulas que manipulam dados de duas matrizes de entrada. O primeiro exemplo utiliza cores para distinguir a origem dos termos utilizados na fórmula, enquanto o segundo utiliza os nomes das matrizes de entrada. Em ambos os casos, no entanto, a direção da indução é préestabelecida pela direção das duas primeiras fórmulas.

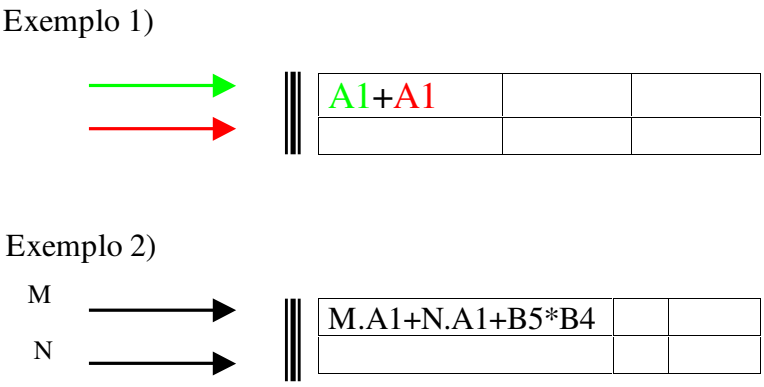


Figura 3.12: Exemplos de utilização de parâmetros nas fórmulas.

Tabela 3.1: Exemplo de elementos visuais utilizados para delimitar a geração de valores nas planilhas.

Exemplos	Descrição
... {10}	Aplica a fórmula de indução em dez colunas consecutivas.
...{N}	Aplica a fórmula de indução em N colunas consecutivas. O valor de N deve ser um dos parâmetros de entrada da planilha.
...	Aplica a fórmula de indução até a última coluna da planilha. Observe-se que, às vezes, a dimensão da planilha somente é conhecida em tempo de execução, dependendo das dimensões das planilhas de entrada.
⋮ {N*2}	Aplica a fórmula de indução em N x 2 linhas consecutivas da planilha.
⋱	Aplica a fórmula de indução até o último elemento da diagonal pretendida.
...{← = 1}	Aplica a fórmula de indução até que a célula horizontal imediatamente anterior tenha valor igual a 1.
⋮ {↑=0}	Aplica a fórmula de indução até que a célula vertical imediatamente anterior tenha valor igual a zero.
...{•=←}	Aplica a fórmula de indução até que dois valores consecutivos sejam iguais. O valor da célula atual deve ser igual ao valor da célula anterior.
...{←+ > 100}	Aplica a fórmula até que a soma de todos os valores gerados na horizontal seja maior que 100.
⋮ {↑+ > 100}	Aplica a fórmula até que soma de todos os valores gerados na vertical seja maior que 100.

Tabela 3.2: Símbolos visuais utilizados para delimitar o número de células geradas pelo processo de indução.

Símbolo	Derivação	Descrição
...		Indica a direção horizontal da indução. Se colocado após as duas fórmulas-base, a direção será horizontal da esquerda para a direita.
⋮		Indica a direção vertical da indução. Se colocado após as duas fórmulas-base, a direção será vertical de cima para baixo.
{P}		O processo de indução termina quando o predicado P é satisfeito.
	P :: Nulo |	O processo de indução não tem limite preestabelecido, depende apenas da quantidade dos dados de entrada.
	P :: Número |	O processo de indução termina quando o número de células geradas é igual à variável ou ao número indicado.
	P :: Variável |	O processo de indução termina quando o número de células geradas é igual ao valor armazenado pela variável indicada. Esta variável pode ser um parâmetro explícito ou uma das variáveis internas de uma matriz ou região (LI, LF, CI, CF).
	P :: Fórmula |	O processo de indução termina se o número de células geradas é igual ao valor produzido pela fórmula.
	P :: Condição	O processo termina se a condição é satisfeita.
←		Indica valor da célula imediatamente anterior na horizontal.
↑		Indica valor da célula imediatamente anterior na vertical.
•		Indica valor da fórmula atual, ou seja, da célula que está sendo gerada.
←+		Representa o somatório de todos os valores horizontais anteriores calculados pela aplicação da fórmula de indução.
↑+		Representa a soma de todos os valores gerados, que antecedem, na vertical, a célula atual.

A Figura 3.13 mostra um exemplo criado pelo mecanismo de indução proposto pelo ProVisual. Em primeiro lugar, o usuário precisa informar as duas primeiras fórmulas da indução (1). Em segundo lugar, o ProVisual deve propor uma fórmula como terceiro elemento da série a partir dos dois primeiros (2). Em terceiro lugar, o usuário deve acatar ou modificar a sugestão da região de preenchimento (retângulo tracejado em 3). Finalmente, o usuário deve informar o momento de realização do preenchimento (4).

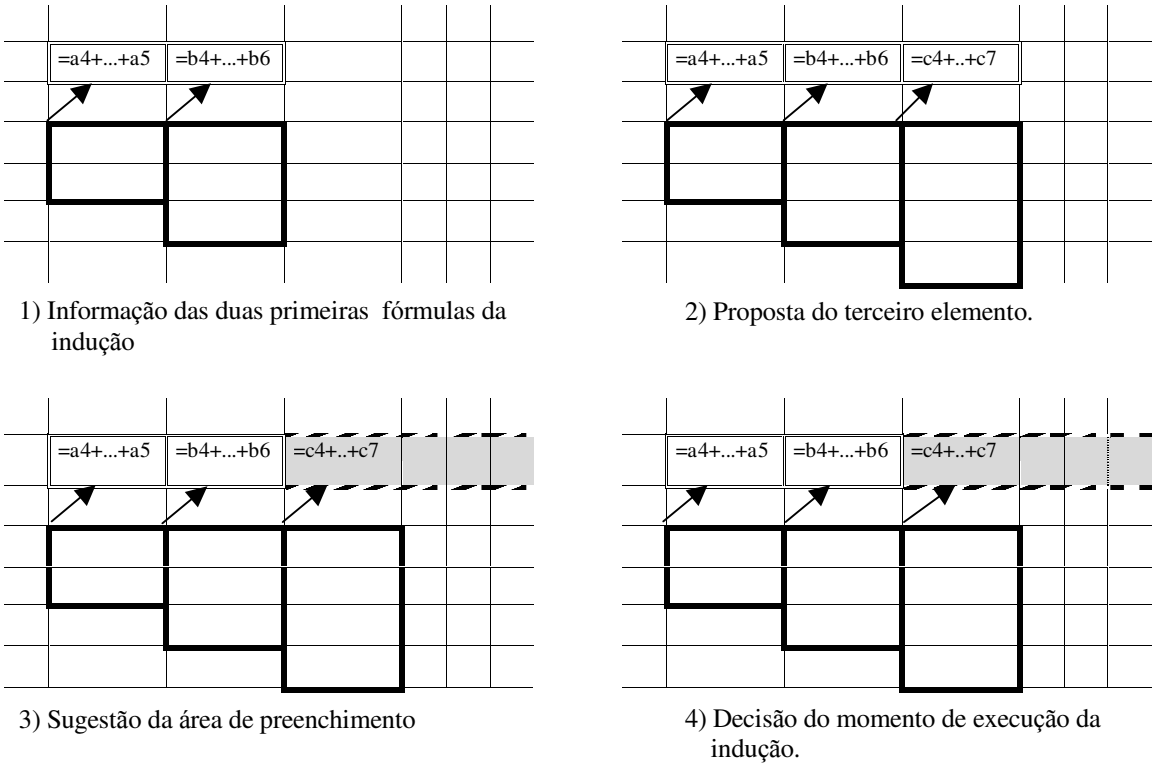


Figura 3.13: Seqüência esquemática do processo de indução de fórmulas.

3.2.3 Definição de uma Função a partir de uma Planilha: Abstração Procedimental

O ProVisual permite que uma matriz seja definida pelo usuário como uma função que recebe e processa valores genéricos, produzindo uma outra matriz como resultado. O usuário pode programar as células desta matriz com o objetivo de produzir a funcionalidade desejada. A programação pode ser estática ou genérica.

Na programação estática, a dimensão da matriz é determinada em tempo de programação e os valores de suas células são calculados a partir dos parâmetros de entrada. As matrizes correspondentes aos parâmetros de entrada podem estar disponíveis em tempo de compilação ou de execução. A validação de conformidade somente será possível quando as matrizes de entrada forem criadas.

A programação genérica envolve a criação dinâmica de uma matriz. A dimensão final da matriz pode ou não ser determinada pelo usuário e as células são inseridas na planilha em tempo de execução. Esta inserção depende de um processo indutivo baseado em exemplos definidos para algumas células pré-inseridas. Além disso, o processo de cálculo dos valores das células pode envolver alguma relação de recorrência que precisa ser programada pelo usuário.

Com o intuito de facilitar o processo de programação das relações de recorrência da planilha, o ProVisual utiliza um método de programação por exemplos, baseado na estrutura geral de cópias de células já descrita na Seção 3.2.2.

No método de programação por exemplo, a regra a ser seguida é a definição das duas primeiras fórmulas de um conjunto de valores a serem gerados. A consistência desta regra baseia-se na suposição de que as duas fórmulas (exemplos) programadas pelo usuário possuem os mesmos tipos de objetos (operadores, endereços e constantes), dispostos na mesma ordem, isto é, as fórmulas devem possuir um padrão estrutural regular. As únicas diferenças admitidas entre essas fórmulas são os valores das constantes e os endereços envolvidos nos cálculos. A principal conseqüência desta regra é que o ProVisual deve repetir o exemplo com possíveis mudanças de valores, mantendo-se, contudo, a estrutura das fórmulas iniciais.

Baseando-se no mesmo princípio, o conceito de geração de blocos foi acrescentado à programação por exemplos. Segundo este conceito, uma matriz pode ser definida como uma composição de blocos aninhados ou hierarquizados que possuem o seu próprio modelo de geração de valores para as células. Neste contexto, convém considerar a questão relacionada à multiplicação de duas matrizes. Uma possível solução em ProVisual seria a programação das duas primeiras células da primeira linha da matriz e a posterior geração da fórmula indutora. Esta

solução limitaria as situações em que a matriz poderia ser utilizada como função. Por exemplo, é perfeitamente possível a criação de uma função para multiplicação de duas matrizes, usando-se o método descrito na Seção 3.2.2. Porém, a construção de uma função para o produto de Kronecker[10] não seria viável sem a introdução dos blocos.

A generalização de uma matriz é uma extensão do conceito de geração de fórmula por indução. A principal diferença encontra-se na forma de utilização da planilha. O processo de indução descrito anteriormente é aplicado a uma planilha já estruturada, mesmo que sua dimensão não seja conhecida. A generalização permite a estruturação da planilha por meio da inserção, em uma planilha vazia, de blocos, de sub-blocos, de células e, finalmente, da especificação das fórmulas indutoras para as células dos blocos mais internos.

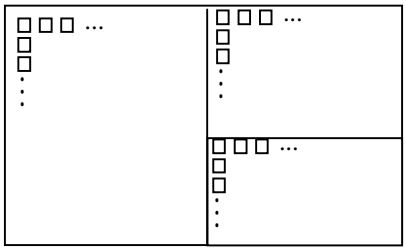
Os blocos, por sua vez, podem ser estáticos ou dinâmicos. Os blocos estáticos não são replicados, apenas as células internas aos mesmos. Por outro lado, os blocos dinâmicos são replicados de uma maneira similar à replicação das células. A Figura 3.14 ilustra a generalização do método de indução.

Em qualquer dos casos descritos anteriormente, podem-se criar blocos aninhados, obedecendo às mesmas regras de formação. Convém observar que este método de geração de matrizes baseado em exemplos visa reduzir a utilização do instrumento de recursão. Ele também funciona como um poderoso construtor de repetição, uma vez que pode embutir uma série de padrões de repetição, aninhados ou não, que atuam nos dados de uma matriz, conforme os comandos de repetição mostrados na Figura 3.14.

A Figura 3.14 merece alguns comentários. A matriz da esquerda utiliza três blocos estáticos, havendo um processo indutor responsável pela geração das células. A matriz final é obtida após o processamento dos três blocos. A matriz da direita possui apenas um bloco, replicado por linha e por coluna, seguindo-se algum critério definido. Um indutor para geração dos valores dos blocos foi inserido em cada um deles. Abaixo de cada matriz, existe um

[10] Produto matricial de duas matrizes (An_xm e Bp_xq) no qual cada elemento da matriz A é multiplicado por todos elementos da matriz B, produzindo uma matriz de dimensão np x mq.

fragmento de código, que retrata o esquema de repetição embutido em cada matriz do exemplo. Note-se que na figura 3.14 os elementos visuais ... e ⋮ indicam um nível de aninhamento do construtor de repetição.

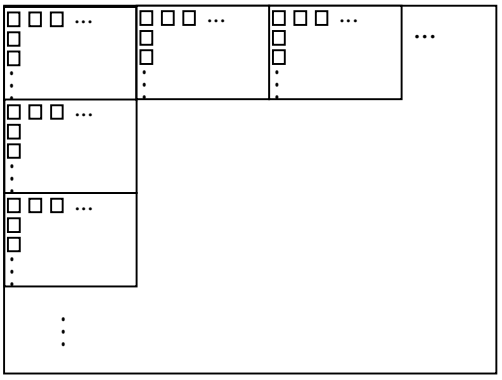


Esquema de repetição implementado por esta matriz:

```
For .... do begin
...
end;

For ... do begin
...
end;

For ... do begin
...
end;
```



Esquema de repetição implementado por esta matriz:

```
For .... do begin
        For ... do begin
                For ... do begin
                        For ... do begin
                                                        ...
                        end;
                end;
        end;
end;
```

Figura 3.14: Exemplos de inserção de blocos, células e fórmulas indutoras para criação de matrizes.

A Figura 3.15 ilustra o exemplo para multiplicação matricial. Note-se, neste exemplo, a existência de três diferentes fórmulas de indução: a primeira fórmula relaciona a geração de fórmulas para uma linha; a segunda trata da transição de uma linha para outra; e, por fim, a terceira fórmula atua dentro de uma célula e indica como será a geração de novas fórmulas para cada célula, uma vez que ainda não se sabe as dimensões das matrizes de entrada.

A programação desta planilha deve obedecer a uma seqüência de passos. O primeiro passo consiste em escrever as fórmulas que formam a base da indução. O segundo passo consiste na resposta do ProVisual com uma proposta de fórmula. O terceiro passo consiste na determinação do número de células geradas pelo usuário (no caso, número indeterminado) e no fornecimento da fórmula de transição de uma linha da matriz para outra. O último passo consiste no requerimento do encapsulamento e da nomeação da função pelo usuário.

O ProVisual responde com um ícone na janela de funções, com os respectivos pontos para ancoradouro dos parâmetros e com suas respectivas cores derivadas das fórmulas (em (2)). A seta da direita constitui apenas uma indicação visual de que o processo de indução deve ser executado linha a linha. Convém notar, ainda, que as caixas com bordas vermelhas retratam as fórmulas indutoras geradas automaticamente.

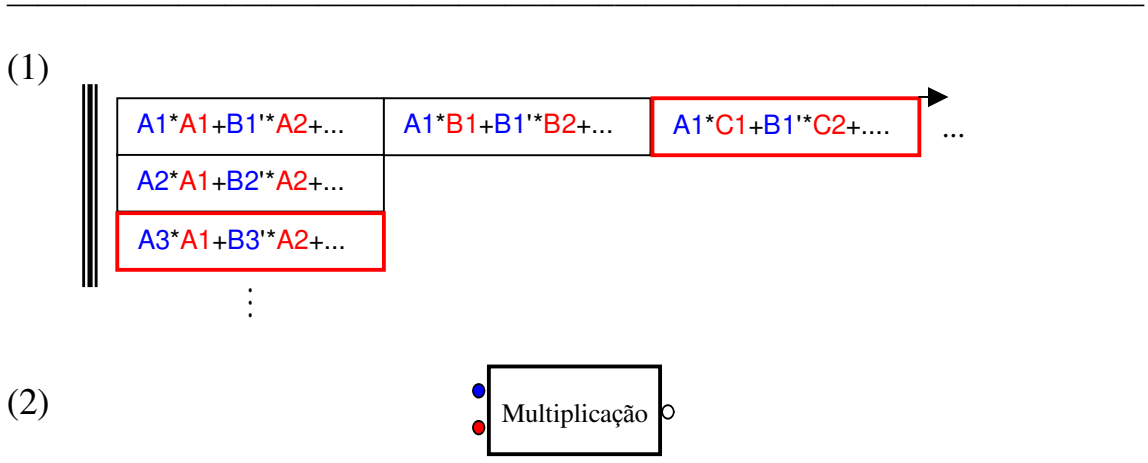


Figura 3.15: Exemplo de uma função baseada em planilha para a multiplicação de duas matrizes.

Um outro fator relevante é a verificação da consistência das dimensões envolvidas na operação desejada. Novamente, considerando-se o exemplo anterior, convém notar que as referências às células das duas matrizes de entrada ocorrem em pares. Como o número de termos gerados depende das dimensões das matrizes, a fórmula só é expandida a bom termo, se e somente se, o número de colunas da matriz identificada pela cor azul for igual ao número de

linhas da matriz registrada em vermelho. Em outras palavras, a verificação da conformidade das dimensões das matrizes de entrada com a operação desejada somente será válida se as fórmulas internas puderem ser expandidas.

A Figura 3.16 exemplifica a programação de uma planilha para produzir a multiplicação de Kronecker de duas matrizes. O usuário precisa definir, primeiramente, três blocos externos com as respectivas direções das iterações. Para cada um dos blocos, define células, fórmulas de indução e direção da indução. Observe-se que a seta indica que todo o processo deve ser executado por linha. Novamente, os diagramas com bordas em vermelho simbolizam fórmulas e/ou blocos, gerados automaticamente para que se possa validar o indutor. Se a matriz azul possuir dimensão *m x n* e a matriz vermelha possuir dimensão *r x q*, a dimensão da matriz resultado será: *mr x nq*.

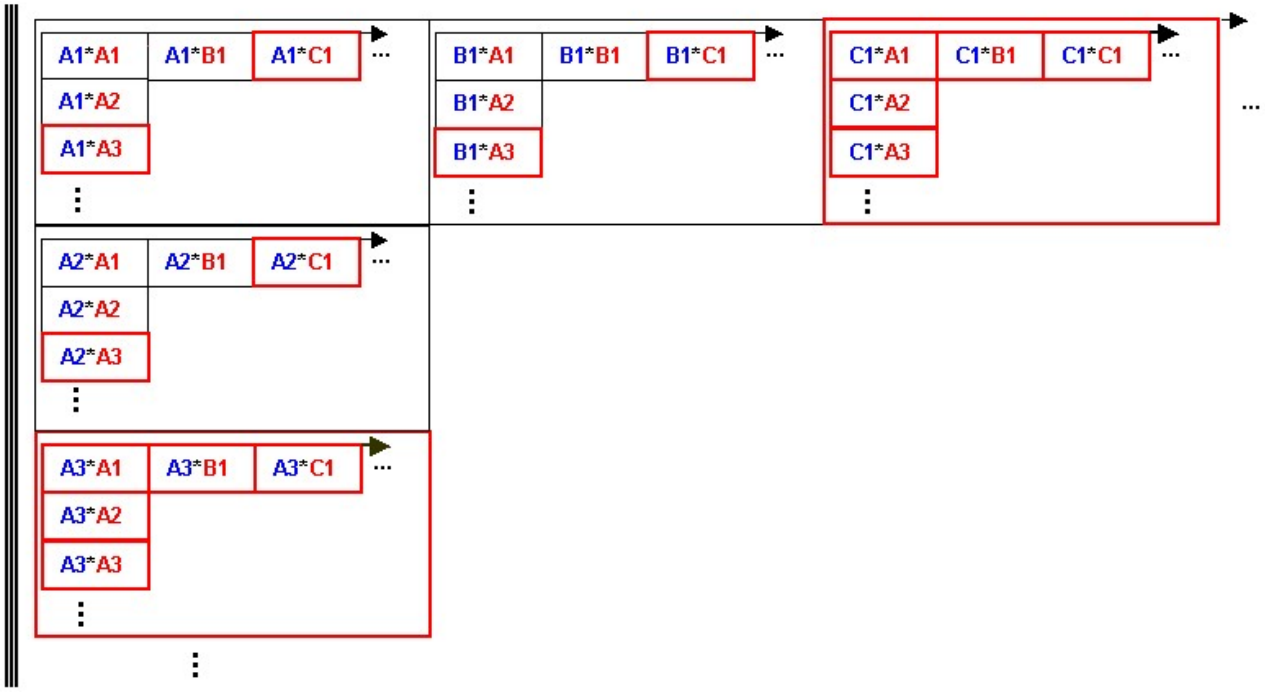


Figura 3.16: Programação de uma planilha para produzir a multiplicação de Kronecker.

A dimensão final da matriz produzida deve ser diretamente proporcional ao número de iterações aninhadas executadas. Este número de iterações pode ser delimitado com alguma condição de parada. Caso nenhum delimitador tenha sido especificado, o sistema deve prever quantas iterações serão executadas. Além do mais, mesmo que uma determinada condição de parada tenha sido fornecida, é necessário calcular o máximo de iterações geradas por um determinado modelo de indução sem causar erros de endereçamento.

O número máximo de iterações, ou seja, o número máximo de vezes que a fórmula indutora será executada, pode depender do número de blocos aninhados, das dimensões das matrizes de entrada, do número de termos existentes na fórmula indutora e da distância dos índices envolvidos nos termos de duas fórmulas consecutivas do modelo de indução. Assim, deve-se considerar a seguinte notação:

- j — representa um termo da fórmula de indução. Este termo não é uma constante e deve se referir a um determinado endereço de uma matriz;
- i — representa o bloco indutor i. Caso não existam blocos, a matriz será considerada um bloco;
- k — representa um endereço das fórmulas-exemplos. Os possíveis valores assumidos são 1, 2 ou 3. O valores 1 e 2 indicam fórmulas exemplos (base da indução) O valor 3 corresponde à fórmula de indução gerada pelo sistema;
- Ni — indica o número de termos existentes na fórmula indutora do bloco i;
- NCij — indica o número de colunas da matriz a que o termo j, do bloco i, faz referência;
- NLij — indica o número de linhas da matriz a que o termo j, do bloco i, faz referência;
- L_{ijk} — exprime o endereço da linha do termo j, contido no bloco i e na célula k (1,2 ou 3); e
- C_{ijk} — exprime o endereço da coluna do termo j, contido no bloco i e na célula k (1,2 ou 3)

A determinação do número de blocos que devem ser gerados depende dos termos variáveis nas fórmulas de indução do bloco em questão. Como este bloco pode estar gerando valores de uma matriz (linha e coluna) e não apenas vetores, as fórmulas indutoras da linha e da coluna devem ser consideradas. Desta maneira, deve-se calcular a distância de cada termo da fórmula indutora em relação à mesma fórmula do bloco indutor, adotando os seguintes passos:

1. Calcular o número de blocos (NBij) gerados por cada termo j do bloco i:

$$NBij = \text{minímo} \left(\begin{array}{l} \text{Abs}\left(\dfrac{NLij}{Lij3 - L(i-1)\,j3}\right) \\ \\ \text{Abs}\left(\dfrac{NCij}{Cij3 - C(i-1)\,j3}\right) \end{array} \right.$$

onde: $0 < j \leq Ni$

Produzindo $NB_{i1}, NB_{i2}, ..., NB_{iNi}$

2. Calcular o número de blocos (NBi) gerados a partir dos NBij´s:

$NBi =$ mínimo $(NB_{i1}, NB_{i2}, ..., NB_{iN})$

Este processo deve ser executado hierarquicamente, partindo-se dos blocos mais internos para os mais externos. Dentro de cada bloco, por sua vez, o processo deve ser aplicado horizontalmente (linhas) e verticalmente (colunas), dependendo da ordem de geração que foi especificada pelo usuário.

Os blocos mais internos devem possuir apenas células. Neste caso, o cálculo do número de células geradas por cada um desses blocos é baseado na segunda fórmula de indução e na terceira fórmula gerada pelo sistema dentro de cada um dos blocos. Assim, adotando-se a notação anterior, tem-se:

$$NEij = \text{mínimo} \left(\begin{array}{l} \text{Abs}\left(\dfrac{NLij}{Lij3 - Lij2}\right) \\ \\ \text{Abs}\left(\dfrac{NCij}{Cij3 - Cij2}\right) \end{array} \right.$$

onde: $0 < j \leq Ni$

Produzindo $NEi_1, NEi_2, ...NEi_{Ni}$

O número total de células é dado pela seguinte fórmula:

$$NEi = \text{mínimo}(NEi_1, NEi_2, ..., NEi_{Ni})$$

O mesmo processo deve ser aplicado, se for o caso, a cada dimensão e, assim, o número de linhas e colunas do bloco i é fornecido por:

NEi_L = NEi calculado na direção horizontal (colunas); e
NEi_C = NEi calculado na direção vertical (linhas).

Desta forma, é possível calcular tanto o número máximo de iterações executadas, pela multiplicação de NE_{IL} * $NE_{IC,}$ quanto a dimensão final da matriz produzida.

3.2.4 Definição de Visões dos Dados de uma Matriz

Uma matriz é composta por duas planilhas, que permitem, respectivamente, a manipulação dos dados de entrada e a construção de visões apropriadas para o processamento. Uma visão pode ser um simples elemento, uma partição (submatriz), um conjunto de partições ou mesmo toda a matriz dentro de um processo iterativo ou não. Este processo pode ser utilizado de maneira que elementos ou submatrizes sejam preparados e enviados à saída da planilha, obedecendo-se a ciclos de um controle interno de repetição.

O componente visão de uma matriz pode ser considerado um poderoso comando de repetição, que tenta reduzir a utilização dos recursos de iteração e de recursão na implementação de um programa. Todavia, a confecção de uma visão diferente dos dados de entrada é opcional. Se nada diferente for requerido, os dados de entrada serão enviados integralmente dentro de uma única matriz para os processos conectados à saída da planilha.

A partição divide uma matriz, inicialmente definida, em várias regiões independentes. Estas regiões podem ser utilizadas para descrever outras matrizes ou para auxiliar outros processamentos ou fórmulas.

O ProVisual associa o conceito de partição a um processo iterativo, visando fornecer uma alternativa ao conceito do construtor de repetição existente nas linguagens de programação imperativas. Isto é possível porque, na maioria das vezes, as repetições são utilizadas para atribuir valores a um conjunto de elementos de uma estrutura ou para selecionar este conjunto de elementos.

Criação de Visões a partir de Partições:

Uma matriz pode ser seccionada em áreas retangulares (chamadas de regiões) na parte da planilha reservada para diferentes visões. Estas regiões não podem ser sobrepostas. A Figura 3.17 retrata um exemplo de partição de uma matriz em seis distintas regiões. A partição de uma matriz ocorre de maneira hierárquica. Uma região pode ser dividida em regiões menores, que a cortam vertical e horizontalmente. Três cortes verticais e dois cortes horizontais foram realizados para produzir a partição da Figura 3.17. Qualquer uma dessas regiões pode ser subdividida em outras regiões, utilizando-se o mesmo conceito.

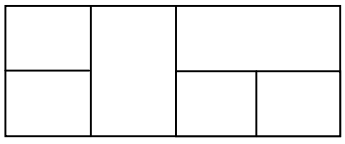


Figura 3.17: Uma partição de uma matriz.

As dimensões dessas partições podem ser informadas ou obtidas da mesma forma que a matriz da Figura 3.18, isto é, utilizando-se as células de dimensionamento (quadrados externos

desenhados nas bordas das regiões) para informar a dimensão de cada uma das partições efetuadas. Note-se que nem todas as regiões desta figura possuem células de dimensionamento

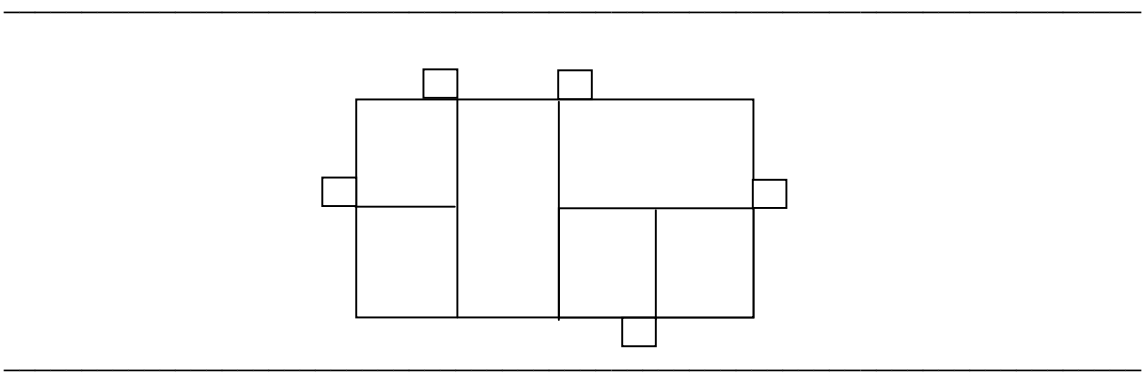


Figura 3.18: Dimensionamento das partições.

O usuário pode determinar se todas as células de dimensionamento serão mostradas ou não, o que se aplica no caso de partições complexas. Quando as células de dimensionamento não são mostradas, pode-se descobrir as dimensões das regiões a partir das células de dimensionamento de regiões adjacentes.

As células de dimensionamento podem ser nomeadas para que seus valores possam ser utilizados em computações posteriores. Os valores associados às células de dimensionamento podem ser constantes, variáveis ou uma expressão aritmética. Os retângulos que representam as células de dimensionamento podem ser preenchidos com uma determinada cor para facilitar a associação entre diferentes células de dimensionamento que devem obrigatoriamente possuir os mesmos valores.

Uma partição pode ser nomeada ou colorida para ser referenciada pelos nós diretamente conectados a esta matriz no diagrama. Neste caso, o pictograma (matriz ou processo) pode conter uma expressão que utilize as cores ou nomes definidos anteriormente no diagrama. A Figura 3.19 retrata a geração de uma nova matriz a partir dos dados de quatro partições de outra matriz. Estas quatro regiões são recebidas pela matriz da direita, que troca a ordem de duas delas (A1 e A4).

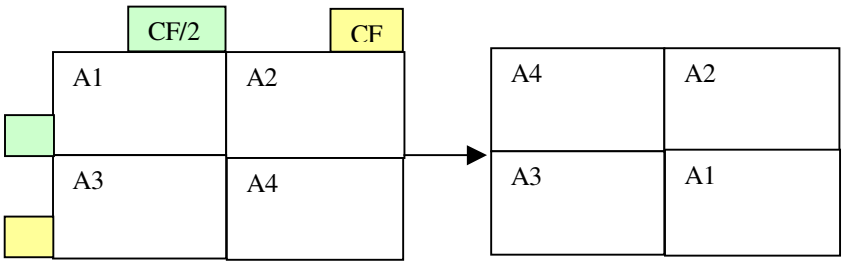


Figura 3.19: Obtenção de quatro partições com a mesma dimensão (A1, A2, A3 E A4).

Criação de Visões a partir de um Padrão de Iteração:

Uma matriz ou região pode conter, também, um padrão de iteração para que seja possível acessar os valores de seus elementos. Este padrão, baseado na notação proposta por Auguston [1997], pode incluir valores de determinadas células, os seus índices, as dimensões envolvidas e, primordialmente, a ordem segundo a qual ocorrerá a iteração na matriz. Um padrão de iteração contém os seguintes símbolos :

 representa o valor de um único item;

 representa um conjunto de valores (linhas, colunas, …);

… indica a direção do padrão de iteração;

permite o acesso ao índice da matriz durante a iteração.

A Figura 3.20 mostra um exemplo em que o padrão de iteração possibilita o acesso a todos as células da matriz (variável *e*) e ao número da linha correspondente (variável I) na ordem especificada. Neste exemplo, a matriz é percorrida da esquerda para direita e de cima para baixo.

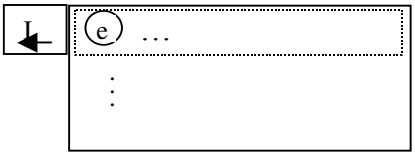


Figura 3.20: Exemplo de padrão de iteração.

A exemplificação de algumas partições mostra a força e a flexibilidade deste conceito do ProVisual. A Figura 3.21 apresenta um exemplo de partição em quatro regiões de dimensão (A1, A2, A3 e A4), enviadas para o processamento posterior:

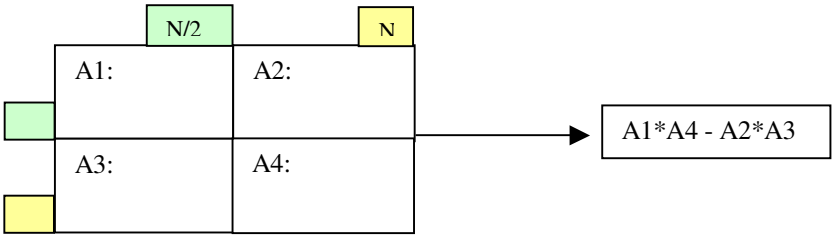


Figura 3.21: Partição em quatro regiões.

A Figura 3.22 ilustra um exemplo de obtenção e de posterior processamento dos elementos de cada região a partir de uma preparação da partição anterior. Note-se que os elementos devem ser lidos linha a linha para cada região.

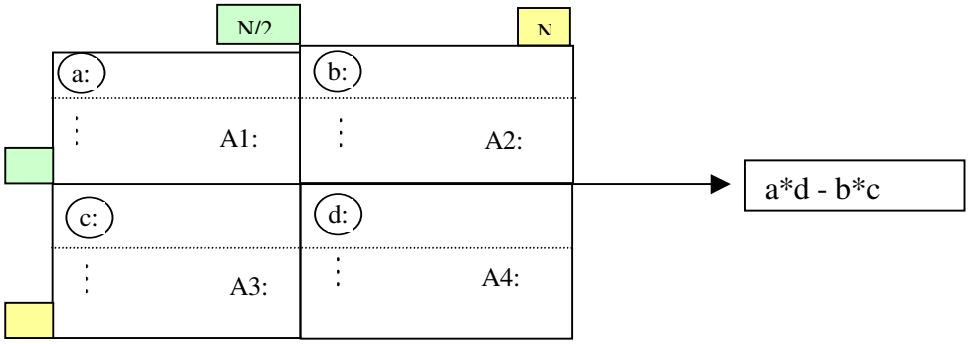


Figura 3.22: Obtenção dos elementos de cada região.

Finalmente, a Figura 3.23 representa um exemplo de obtenção e de posterior processamento de uma partição dentro de um processo iterativo. Note-se que as partições devem ser geradas segundo o padrão de iteração especificado no diagrama. Neste caso, cada uma das quatro regiões (uma por vez) será repassada para o próximo nó da rede.

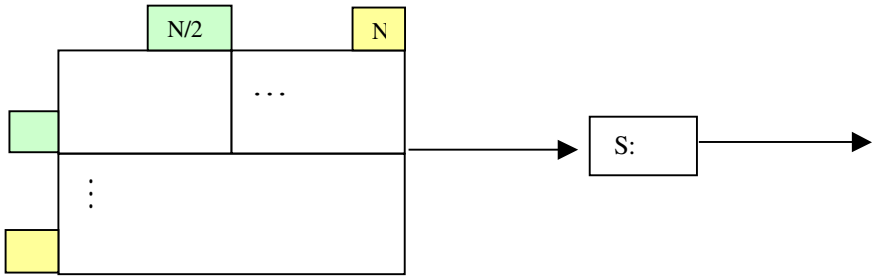


Figura 3.23: Obtenção de uma região de cada vez.

Note-se a diferença entre os três exemplos. O primeiro exemplo criou e nomeou quatro regiões, passando-as para a expressão seguinte. O segundo dividiu a matriz em quatro regiões independentes, criou um padrão de iteração para cada uma e enviou quatro elementos, um de cada região, para o nó seguinte. A iteração deve ser repetida até que todos os elementos sejam enviados para o devido processamento. Finalmente, o terceiro exemplo obteve cada região (uma por vez) de dimensão *N/2* x *N/2*, copiando a região gerada para a matriz *S*.

O ProVisual possui ainda um gerador de partições diagonais. O exemplo abaixo mostra a obtenção da diagonal principal (*D*) de uma matriz:

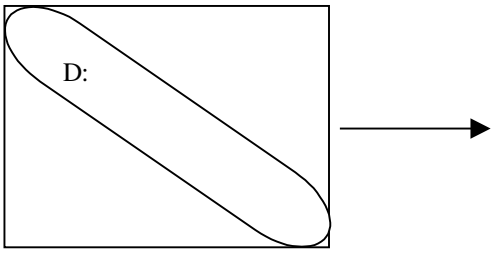


Figura 3.24: Obtenção da diagonal de uma matriz.

A utilização de planilhas, padrões de iteração e partições possibilita uma vasta gama de aplicações que podem ser implementadas sem a utilização de construtores computacionais relacionados à repetição e à recursão. Desta forma, o autor acredita que a implementação de programas de domínio matricial possam ser desenvolvidas de maneira mais fácil, mais rápida e com um nível de abstração mais alto.

3.2.5 Explicitando os Fluxos de Dados das Fórmulas

A planilha do ProVisual explicita o fluxo de dados envolvido entre as células e as fórmulas de uma planilha, procurando superar a questão da invisibilidade do fluxo de dados das planilhas tradicionais.

Esta invisibilidade é uma fonte potencial de problemas, uma vez que o usuário não sabe, sem investigar o conteúdo da célula, quais células afetam e são afetadas por uma determinada fórmula. Um problema correlato ocorre quando o usuário precisa entender uma planilha criada por outro usuário.

Várias soluções têm sido propostas e implementadas para tornar mais perceptíveis as relações existentes numa fórmula. Por exemplo, a planilha Excel da Microsoft™ [Excel, 2000], uma das planilhas com mais recursos existentes no mercado, inclui duas técnicas que permitem uma visualização limitada do fluxo de dados existente numa dada célula. A primeira técnica é invocada quando o campo que mostra o conteúdo da célula é editado. Nesta situação, cada endereço envolvido na fórmula recebe uma cor e as regiões correspondentes aos endereços recebem a mesma cor. A segunda técnica é ativada quando se utiliza a ferramenta de auditoria, que desenha arcos entre a célula da fórmula e seus ascendentes e descendentes.

Estes recursos tentam tornar o fluxo de dados, que se encontra escondido, visualmente acessível. Todavia, eles precisam ser explicitamente ativados, via menu ou barra de ferramentas,

e limitados a uma única célula por vez. Não existe qualquer forma de se mostrar todo o fluxo de dados da planilha de uma só vez.

A planilha do ProVisual aplica duas novas técnicas: 1) visualização transitória; e 2) visualização gobal. Estas técnicas visam minorar os problemas acarretados pela invisibilidade do fluxo de dados. A primeira técnica permite a visualização transitória para retratar a estrutura de fluxo de dados associada a células individuais. A segunda técnica compreende a visualização global com o objetivo de permitir a representação de todo o fluxo de dados da planilha de uma só vez.

Técnica de Visualização Transitória

A técnica de visualização transitória, baseada na proposta de Igarashi [1998], permite que o usuário visualize parte do fluxo de dados associado à célula com fórmula com a qual está interagindo. Ela mostra tanto o fluxo das células que afetam a fórmula, quanto as células que são por ela afetadas.

O ProVisual distingue visualmente estes dois tipos de células com a utilização de arcos e cores. Regiões adjacentes a células que afetam uma fórmula são agrupadas em um retângulo e pintadas com uma determinada cor. A Figura 3.25 ilustra um exemplo da estrutura de fluxo de dados envolvido em uma fórmula. Nesse exemplo, a célula E1 foi criada pela soma da primeira linha A1 até D1, enquanto a célula E5 foi obtida pela multiplicação por 2 do resultado da célula E1.

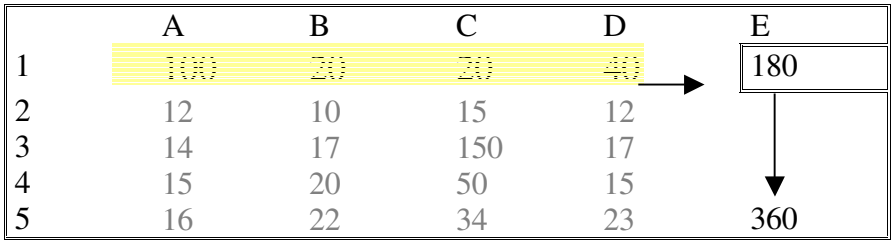


Figura 3.25: Exemplo de representação de fluxo de dados em planilhas com fórmulas.

O termo transitória que acompanha esta técnica de visualização está relacionado à forma de ativação da estrutura de fluxo de dados. Nas planilhas convencionais, o usuário move o cursor até a célula de interesse e ativa uma alternativa do menu de opção para desenhar o fluxo de dados envolvido em uma fórmula. No ProVisual, o usuário especifica a célula de interesse movendo o cursor sobre a célula. Quando o cursor está em cima de uma célula, o grafo do fluxo de dados aparece gradualmente. Os valores envolvidos nos cálculos da fórmula são realçados, enquanto os valores não-envolvidos são levemente apagados. Este grafo desaparece gradualmente quando o cursor é movido para outra célula. Desta maneira, o usuário pode explorar a estrutura de fluxo de dados da planilha através do movimento do cursor por suas células, controlando o nível de informação disponibilizado pelo sistema.

A visualização transitória não requer qualquer operação especial para mostrar graficamente o fluxo de dados, permitindo que o usuário mantenha seu foco de atenção apenas no conteúdo da planilha e no movimento do cursor. Todavia, fica limitada a uma única célula por vez, não permitindo a visualização da estrutura de fluxo de dados de toda a planilha. O ProVisual estende a proposta de Igarashi, possibilitando que toda a matriz, independentemente de sua dimensão, esteja representada na janela correspondente.

Técnica de Visualização Global

A técnica de visualização global permite que o usuário visualize todo o fluxo de dados existente na planilha, tendo acesso a uma rápida revisão de toda a estrutura de uma planilha montada.

Nesta técnica, todas as dependências de dados são representadas na tela com cores diferentes. As células podem ser representadas como grupos para minimizar o número de cores,. Estes grupos podem ser arranjados de maneira vertical, horizontal ou diagonal. Cada um dos arranjos possui uma determinada cor e fornecem uma idéia de como a planilha se encontra estruturada. A Figura 3.26 retrata um exemplo de planilha, na qual existem quatro blocos e uma diagonal independentes. Neste exemplo, cada bloco e diagonal possuem determinada cor, indicando que existe uma interdependência de células dentro de cada bloco e da diagonal.

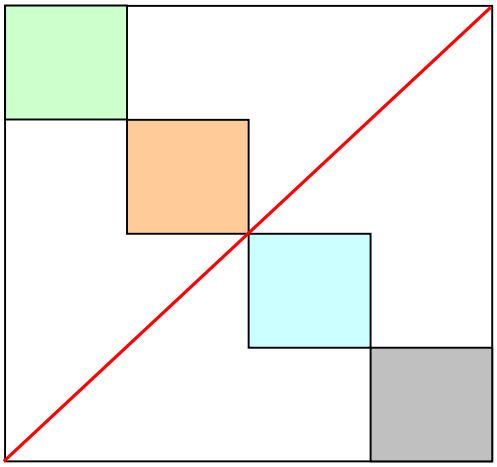


Figura 3.26: Exemplo de visualização global.

A visualização global pode ser requisitada para toda a planilha ou somente para uma parte da mesma, bastando para isso selecionar um conjunto de linhas, colunas ou células individuais e requerer que se mostre os fluxos de dados de entrada e de saída das células escolhidas. Esta é uma adaptação da proposta de Igarashi [1998]. O avanço do ProVisual se encontra na visualização de uma matriz grande. Neste caso, o usuário teria uma espécie de fotografia dos padrões das relações existentes entre todas as células.

A sobreposição de várias partes do grafo é inevitável, mesmo com o agrupamento de células, podendo dificultar a determinação do fluxo de dados de uma célula. No entanto, a técnica de visualização global pode fornecer um esboço geral da estrutura de uma planilha para facilitar seu entendimento.

3.3 Sintaxe Visual e Semântica do ProVisual

Um programa no ProVisual é composto por elementos visuais distribuídos no monitor do computador. A superfície na qual os elementos são arranjados é chamada de *canvas*. Qualquer fragmento válido de programa é um quadro, recursivamente composto de quadros mais simples e, em última instância, de pictogramas primitivos: as matrizes, os operadores simples e de controle, os arcos, as portas e os textos.

A diagramação e a composição dos ícones do ProVisual utilizam os elementos visuais descritos na Figura 3.27. As portas servem como pontos de conexões entre os pictogramas (A). Os arcos conectam pictogramas por meio de suas portas. Existem dois tipos de arcos: o arco de dados e o arco de sincronismo (B).

As três formas de composição de um quadro são:

1) Composição de interseção de dois pictogramas (C).
2) Composição de inserção de dois pictogramas (D).
3) Composição por conexão das portas de saída e entrada de dois pictogramas (E).

Nickerson [1994] advoga que estas três formas de composição aparecem, juntas ou não, na maioria dos diagramas utilizados na Ciência da Computação. Considerando-se o ProVisual, cada tipo de composição possui uma semântica particular. A composição de interseção permite a aplicação de computações regulares a uma matriz, representadas pelo pictograma não desenhado completamente. A composição de inserção permite a construção de operações complexas na determinação dos valores a serem atribuídos aos elementos de uma matriz ou de um vetor. A composição por conexão possibilita o envio de dados de um processo para outro.

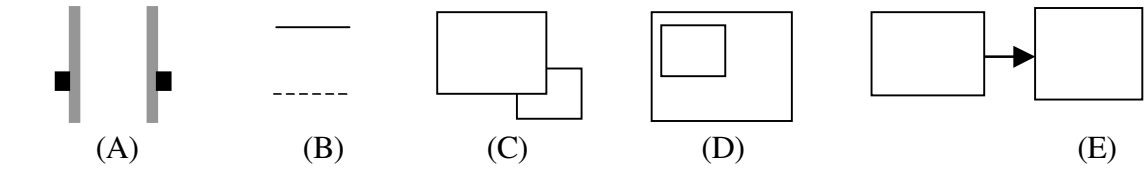


Figura 3.27: Elementos visuais simples.
A) Portas de Entrada e Saída; B) Arcos de Conexão de Dados e de Sincronismo; C) Composição de Interseção; D) Composição de Inserção; E) Composição por Conexão.

3.3.1 Expressões simples

O pictograma que representa uma operação (interna ou definida pelo usuário) é um retângulo, contendo internamente um ícone, um símbolo ou uma descrição da operação interna. A Figura 3.28 retrata dois pictogramas de operações simples.

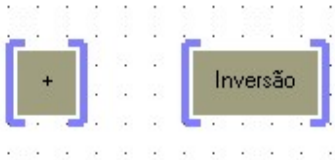


Figura 3.28: Pictogramas de operações simples.

As operações possuem portas de entrada e saída que as conectam, respectivamente, com os dados de entrada e saída. Uma operação somente é ativada, quando todos os dados de entrada estiverem disponíveis nas respectivas portas de entrada. Em caso de operações (diagramas) definidas pelo próprio usuário, é necessário, ainda, que não exista qualquer nó ativo dentro do diagrama. Esta pré-condição exige que todas as computações existentes no diagrama terminem antes que ele possa ser invocado novamente.

Os operadores internos têm um número fixo de portas de entrada e saída, dependendo do tipo do operador. Por exemplo, os operadores internos: *+*, ***, *-*, */*, *&* (e booleano) e *|* (ou booleano) possuem duas portas de entrada e uma única porta de saída.

Os diagramas devem ser definidos da esquerda para a direita. Por exemplo, a Figura 3.29 representa a multiplicação de duas matrizes (esquerda), cujo resultado é repassado para duas funções (direita).

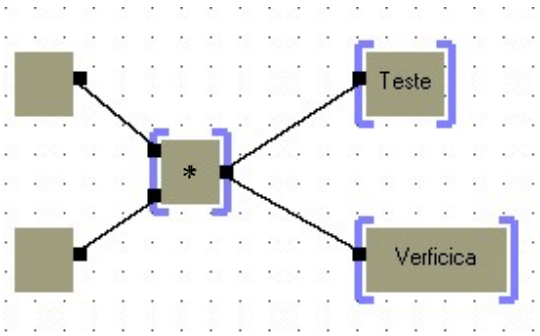


Figura 3.29: Exemplo de um diagrama ProVisual.

As portas de entrada podem receber apenas um fluxo de dados[11]. Por outro lado, qualquer porta de saída pode enviar várias cópias da matriz ali disposta para diversos processos a ela conectados.

Com o intuito de produzir diagramas mais concisos, a versão acima poderia ser abreviada, conforme ilustrado na Figura 3.30.

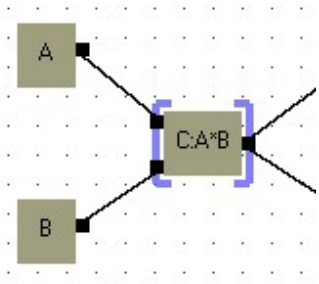


Figura 3.30: Expressão textual.

Os valores de entrada *A* e *B* são multiplicados com a operação pré-definida (*). Seu resultado é atribuído a *C*. Esta notação abreviada preserva a dependência de dados entre os nós e torna o diagrama produzido mais conciso. O mesmo efeito pode ser obtido, nomeando-se as portas de saída, conforme representado na Figura 3.31. Os nomes definidos para operações ou portas de saída somente permanecem visíveis aos nós imediatamente conectados a elas.

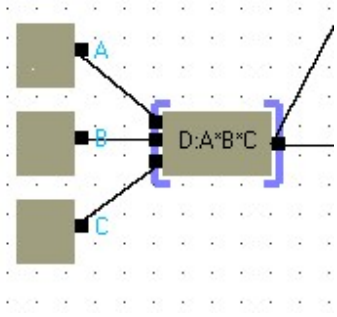


Figura 3.31: Nomeação das portas de saída.

[11] A única exceção é um tipo especial de porta do operador de repetição, conforme descrito na Seção 3.1.2.2..

O ProVisual possui um conjunto de operações que podem ser aplicadas a algumas ou a todas as dimensões de uma matriz, dependendo do tipo de interseção especificado pelo diagrama. Estas operações possuem os mesmos rótulos dos operadores binários básicos: *+*, ***, *-*, */*, *&* e */*. A diferenciação desses operadores (se binário ou unário) é verificada pela utilização do conceito de interseção ou não de diagramas. A Figura 3.32 exemplifica possíveis variações da aplicação da interseção entre dois pictogramas.

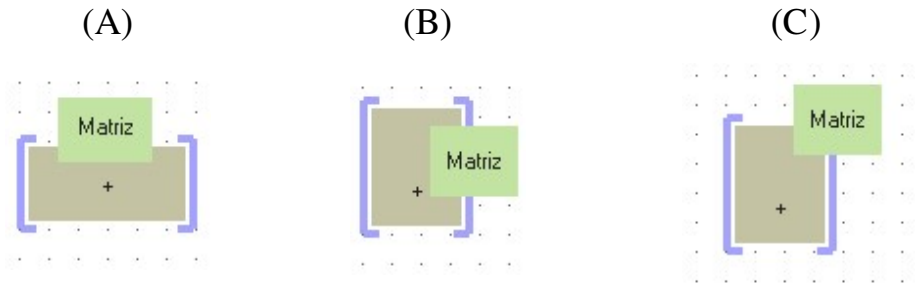


Figura 3.32: Exemplos de interseção entre pictogramas.

O diagrama (A) permite a produção de um vetor (coluna) a partir da soma das linhas de uma matriz. Caso a interseção do operador (*+*) com uma matriz seja realizada conforme o diagrama (B), produz-se um vetor (linha) com as somas das colunas de uma matriz. Finalmente, o diagrama (C) calcula um escalar com a soma de todos os elementos de uma matriz. Vale ressaltar que cada tipo de interseção permite a identificação unívoca do operador unário interno, que deve ser invocado.

No contexto do ProVisual, qualquer operador (interno ou não), que possua apenas um ponto de entrada e um ponto de saída, pode utilizar este conceito de interseção para a obtenção de uma computação que deva ser executada para cada linha, coluna ou elemento de uma matriz de entrada. Este tipo de construção, em última instância, pode ser considerado um processo iterativo com um alto nível de abstração.

3.3.2 Expressões Estruturadas

A experiência no desenvolvimento do ProVisual mostrou, mais uma vez, que um modelo baseado puramente em fluxo de dados precisa ser enriquecido com alguma forma de fluxo de controle. Isto ocorre porque o fluxo de dados não provê os construtores computacionais necessários para a manipulação de problemas complexos no domínio de aplicação pretendido [Hils, 1992].

A dificuldade de se alcançar uma solução satisfatória deve-se ao fato da notação utilizada ser inconsistente com o paradigma de fluxo de dados. De uma maneira geral, os operadores de decisão e de repetição dificultam seu entendimento e utilização.

3.3.2.1 Operador de Decisão

O operador de decisão possui dois objetivos: 1) controle do fluxo de dados; e 2) seleção de valores de uma matriz. O primeiro objetivo permite o controle da execução de um programa, seja admitindo o controle do fluxo de dados disponíveis para outros nós da rede, seja permitindo o cancelamento do programa ou subprograma. O segundo objetivo seleciona um conjunto de valores de uma matriz para obter uma submatriz ou para controlar os valores atribuídos às células.

Controle do Fluxo de Dados

O operador de decisão permite o controle do fluxo de dados pela aplicação de operadores booleanos aos dados conectados à porta de entrada. O fluxo de dados é desviado para a parte verdadeira do operador se a expressão for avaliada como verdadeira. Caso contrário, o fluxo é enviado para a parte falsa. Todo o fluxo de entrada é disponibilizado na saída tanto da parte verdadeira quanto da parte falsa, ficando a cargo do usuário a conexão dos fluxos.

Este operador também permite o controle da execução do programa. Neste caso, pode-se alterar o comportamento da parte verdadeira ou falsa do operador com o objetivo de parar a execução ou de desviar todo o fluxo de entrada do módulo para uma outra alternativa de execução.+

O operador condicional possui vários pontos de entrada. Para cada ponto de entrada definido, é gerada uma porta de saída em suas partes verdadeira e falsa. Uma expressão booleana avalia se o fluxo de dados da entrada deve seguir pela porta verdadeira ou falsa do operador. Algumas portas de saída podem permanecer desconectadas. A Figura 3.33 retrata o pictograma deste operador:

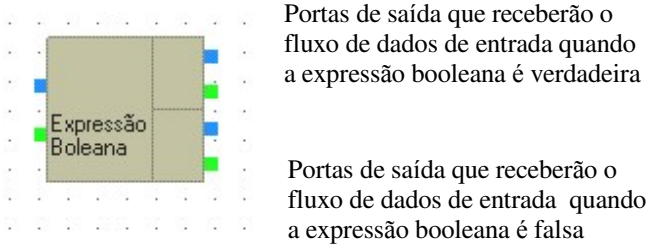


Figura 3.33: Pictograma do operador de decisão.

O autor deste trabalho de pesquisa acredita que esta notação é mais condizente com os princípios de fluxo de dados do que a maioria das notações utilizadas[12], porque ela não interrompe o fluxo de dados, apenas o redireciona em função de uma avaliação booleana. Esta notação é baseada no conceito do operador de distribuição apresentado no trabalho de Davis [1982]. A diferença se encontra no dado de controle, que não é separado dos dados controlados. Este tipo de construtor também é fundamental no controle de processos iterativos.

Este operador pode controlar a execução do programa, incluindo o controle sobre o cancelamento da execução de um programa ou subprograma, bem como sobre a transferência de todo o fluxo de dados de entrada para uma outra alternativa de execução. A Figura 3.34 mostra as três alternativas de utilização do operador de decisão.

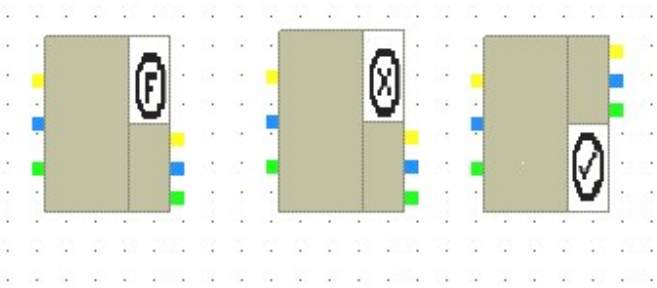


Figura 3.34: Alternativas para as cláusulas verdadeira e falsa.

O primeiro diagrama finaliza a execução de um programa ou subprograma, se a avaliação da expressão booleana for verdadeira. Caso contrário, o fluxo é desviado para o lado falso do construtor. O segundo diagrama termina a execução do programa ou subprograma, se a expressão booleana for verdadeira. Finalmente, o terceiro diagrama indica a interrupção da execução do subprograma e a transferência de todo o seu fluxo de dados de entrada para uma outra alternativa de execução caso a expressão booleana seja avaliada como falsa. Esta alternativa de execução é uma forma de abstração funcional do ProVisual, detalhada na Seção 3.3.6.

Seleção de Valores de uma Matriz

O operador de decisão pode ser utilizado para selecionar células, linhas ou colunas de uma matriz que satisfaçam a alguma condição definida. Esta condição pode ser baseada nos valores das células ou em suas posições (índices).

Esta funcionalidade é obtida pela aplicação do operador de decisão dentro de uma matriz, visando a seleção dos valores da matriz. Neste caso, apenas um ponto de saída é permitido em cada parte do construtor. A eliminação de conectores de saída não desejados é necessária quando o operador é utilizado na seleção de dados de uma matriz, enquanto basta deixar um ponto de saída desconectado quando utiliza-se o operador para o controle de fluxo de dados.

[12] Por exemplo, notações utilizadas pelo *Prograph* e pelo *Show-and-Tell*.

O exemplo da Figura 3.35 mostra a criação de uma nova matriz, cujos valores foram obtidos pela seleção de elementos maiores ou iguais a zero de uma matriz de entrada. Caso a expressão booleana seja falsa, o valor selecionado será a constante *0*.

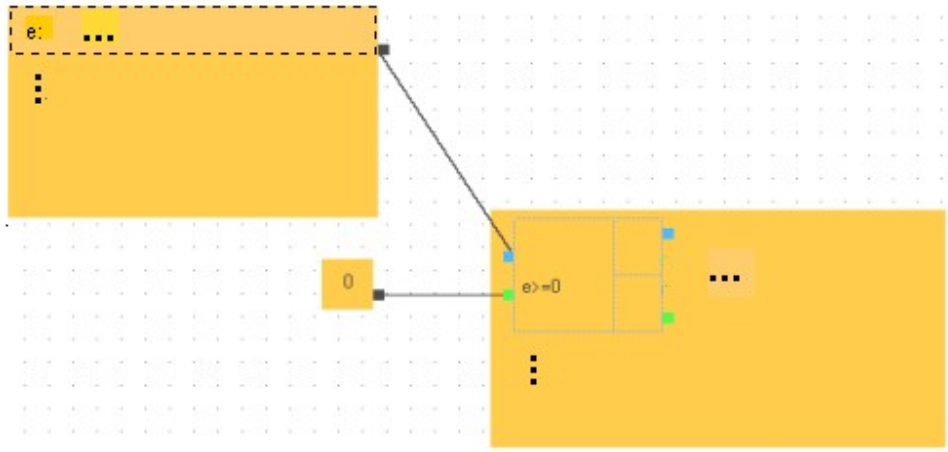


Figura 3.35: Seleção de elementos de uma matriz.

Cabe salientar, entretanto, que a geração de uma nova matriz baseada na seleção de elementos pode produzir uma matriz inválida, como uma matriz com números diferentes de elementos em cada uma de suas linhas. Este problema pode ser contornado pela restrição adotada pelo ProVisual, que obriga a existência de valores conectados na parte verdadeira e falsa do construtor. Esta restrição não se aplica à seleção de toda uma linha ou coluna.

A seleção baseia-se em algum critério de comparação com a posição relativa do elemento na matriz de entrada, bastando a utilização explícita do índice desta matriz na expressão booleana. A Figura 3.36 reflete esta situação.

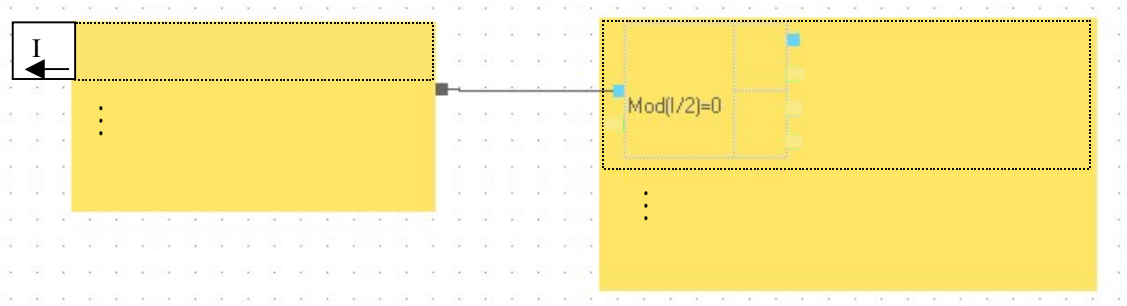


Figura 3.36: Seleção de linhas pares de uma matriz.

No exemplo da Figura 3.36, a matriz à direita retrata a seleção das linhas pares de uma matriz de entrada. Observa-se que a parte falsa do operador de decisão não possui qualquer porta ativa. Neste caso, uma linha com valores nulos (NIL) será gerada caso a expressão booleana seja avaliada como falsa.

Quando a matriz de entrada não possuir mais dados para serem enviados, uma linha preenchida com um símbolo especial (EOM, indicador de final de matriz) apontará que o processo de geração da nova matriz pode ser finalizado. Para finalizar a geração, todas as linhas e/ou colunas preenchidas com o símbolo NIL devem ser filtradas e eliminadas da matriz.

3.3.2.2 Operador de Iteração

O operador de iteração, em qualquer linguagem de programação visual ou textual, pode ser caracterizado como um programa executado um número finito de vezes, até que uma condição de parada seja satisfeita. Neste sentido, podem ser distinguidas duas formas básicas de iteração: iteração horizontal (paralela) e iteração temporal (seqüencial) [Ambler, 1990]. Na iteração horizontal, o resultado gerado por um determinado ciclo não afeta o resultado do ciclo seguinte. Por outro lado, a iteração seqüencial implica em uma dependência dos resultados produzidos em cada ciclo.

Existem duas estratégias principais para dotar a programação baseada em fluxo de dados de iterações seqüenciais [Shür, 1997]. A primeira estratégia admite a presença de arcos de retorno explicitamente representados no grafo do programa. Cada fluxo de dados possui um par de valores (atual e anterior) a ele associado, ao invés de um único valor atual. Estes valores podem ser utilizados em expressões que os exijam. A segunda estratégia baseia-se na criação de uma nova cópia dos nós que representam o corpo da iteração. O valor de entrada desta cópia é o valor produzido pela cópia anterior. Essas estratégias são necessárias porque o modelo computacional do fluxo de dados é fundamentado no princípio de atribuição simples [Milosko, 1984], segundo o qual uma variável pode ser alterada apenas uma vez.

A estratégia adotada pelo ProVisual é similar à primeira descrita no parágrafo anterior. Todavia, ao invés de suportar apenas um par de valores, suporta uma série de valores temporais que podem fluir de um processo iterativo e ser convenientemente acessados. Nesse sentido, o construtor de iteração possui um tipo especial de porta de entrada que armazena em uma fila os dados temporais produzidos pelo processo iterativo. Esses dados podem ser acessados pela atribuição de um índice negativo a um determinado arco. Este índice retrata o valor temporal requerido para uma determinada expressão. No corpo do nó que representa a iteração é definido um subgrafo, cuja execução deve ser repetida até que uma condição de parada seja satisfeita. A Figura 3.37 mostra um exemplo de representação gráfica da iteração.

Figura 3.37: Um exemplo de representação gráfica da iteração.

O exemplo da Figura 3.37 possui dois arcos de entrada: um representa o valor inicial do fluxo que alimenta a iteração, enquanto o outro representa um dos fluxos de saída reenviado para a mesma porta que recebeu o valor inicial.

A Figura 3.38 apresenta uma implementação do programa para o corpo da iteração do exemplo da Figura 3.37. A execução da iteração termina quando a expressão booleana (A<10) torna-se verdadeira.

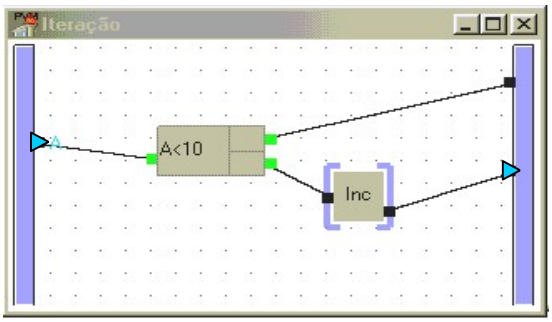


Figura 3.38: Corpo da iteração da Figura 3.36.

O nó que representa a iteração pode possuir várias portas de recirculação. Cada uma delas é unicamente identificada por uma cor automaticamente atribuída. O subgrafo que representa o corpo da iteração possui o mesmo número de portas do ícone que simboliza a iteração no grafo que lhe deu origem.

O exemplo mostrado na Figura 3.39 ilustra uma iteração que calcula a soma total dos 30 primeiros números de Fibonacci. É oportuno observar que a função de soma do corpo da iteração possui um dos parâmetros rotulados com o valor *-1*. Isso indica que o dado ali postado foi produzido pelo ciclo anterior da iteração. Naturalmente, não existe qualquer valor anterior para o primeiro ciclo da iteração. Neste caso, foi necessária a definição dos dois primeiros valores da fila que armazena os dados daquela porta de entrada (aqui, os valores iniciais foram *1* e *1*).

Uma porta de entrada da recirculação é uma fila de tamanho *1*. O tamanho da fila pode ser redefinido automaticamente a partir dos parâmetros rotulados com o valor *negativo.*

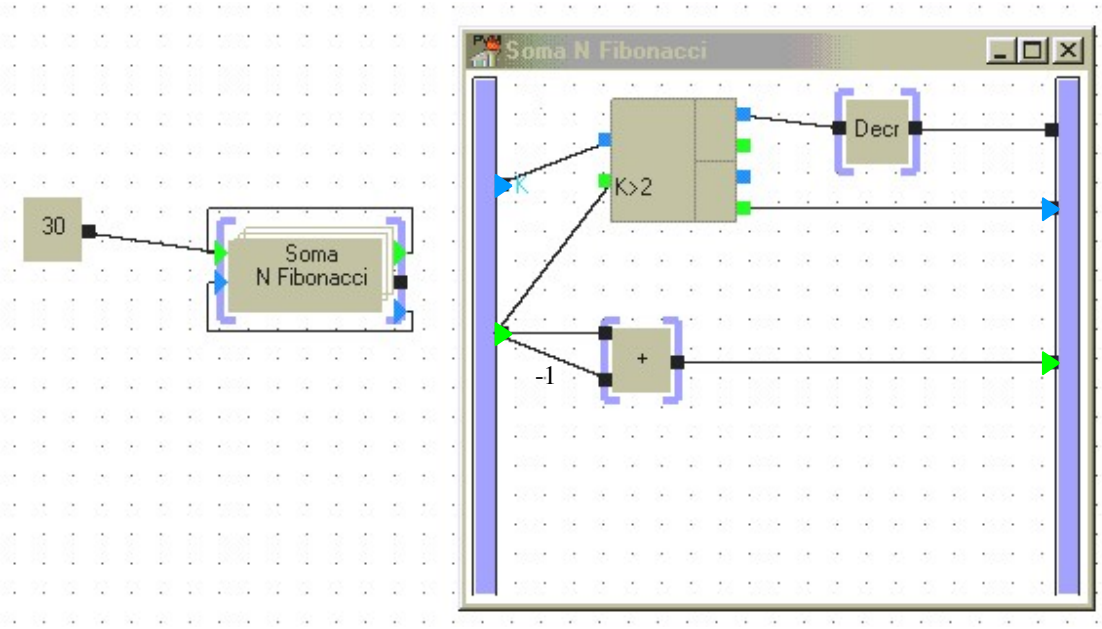


Figura 3.39: Soma total dos 30 primeiros números de Fibonacci.

O Capítulo 2 apresentou algumas implementações de linguagens visuais que incorporam algum construtor visual para iteração. Vale ressaltar, portanto, algumas vantagens importantes do modelo para iteração concebido para o ProVisual.

O tipo de representação adotado pelo modelo para a iteração evita a confusão gerada pela criação de ciclos para implementar uma repetição. Além disso, não gera uma nova cópia do construtor a cada ciclo da iteração. Uma outra vantagem é a possibilidade de se definir o número de ciclos temporais (ciclos anteriores) que precisam ser lembrados. Se não existe dependência temporal entre os ciclos da iteração e uma iteração paralela precisa ser especificada, pode-se determinar o número de iterações que devem ser executadas como complemento ao rótulo do ícone. A Figura 3.40 retrata esta situação.

Figura 3.40: Iteração paralela.

Esta Figura simboliza uma iteração, denominada *Teste*, que deve ser repetida vinte vezes (1 até 20). O número do ciclo em execução é armazenado em um pseudoparâmetro de entrada, que pode ser referenciado no corpo da iteração via fluxo de dados,. Evidentemente, cada ciclo pode ser executado em paralelo para este tipo de iteração.

A possibilidade de paralelismo da execução, que o paradigma de fluxo de dados admite, exige alguns cuidados com respeito à sincronização entre ciclos de uma iteração. Apesar do objetivo principal do ProVisual não ser a execução paralela de programas matriciais, esta possibilidade existe e não pode ser menosprezada, sob o risco de sua inviabilidade no futuro, caso alguns cuidados de sincronização não sejam observados.

3.3.3 Sincronização

O ProVisual admite dois tipos de sincronização:

1) Sincronização entre processos, quando existe a necessidade de se garantir que um determinado processo seja executado antes de outro, na situação em que ambos possam ser executados;
2) Sincronização entre matrizes, quando duas ou mais matrizes precisam enviar dados a um processo e os dados precisam necessariamente vir de maneira sincronizada.

Com respeito à sincronização entre processos, o ProVisual explora sinais de controle que podem ser enviados de um processo a outro. Neste caso, portas de sincronização precisam ser definidas e conectadas nos processos envolvidos na sincronização. A Figura 3.41 apresenta a

sincronização de dois processos (*A* e *B*). Como existe uma porta de sincronização na saída do processo *A* conectada a uma porta de sincronização na entrada do processo *B*, o processo *B* só pode ser executado após o término de *A*. Uma porta de sincronização de entrada pode receber mais de um sinal de controle, bastando a chegada de apenas um dos sinais para habilitar este processo.

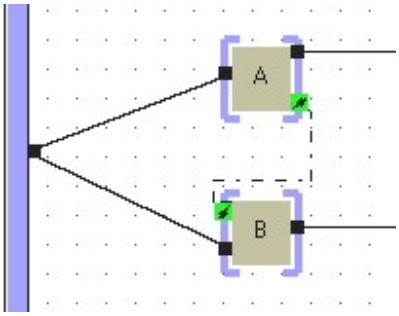


Figura 3.41: Sincronização entre dois processos.

No que se refere à sincronização entre matrizes, é importante ressaltar a forma de sincronização do processo de iteração de matrizes dentro de um diagrama. Três situações são consideradas. A primeira situação refere-se ao diagrama com pelo menos dois processos iterativos conectados. Neste caso, as matrizes que possuem padrão iterativo são sincronizadas, isto é, os elementos de cada uma delas são obtidos na mesma ordem. Como regra, o valor do segundo agregado sincronizado é construído a partir do valor do primeiro. Esta circunstância é ilustrada na Figura 3.42.

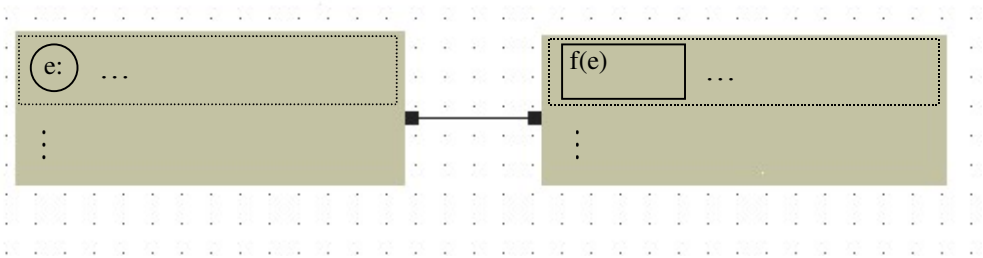


Figura 3.42: Sincronização entre duas matrizes.

A segunda situação relaciona-se à conexão entre dois nós em que apenas o primeiro representa um processo iterativo. Neste caso, a iteração produz uma seqüência de itens, um por

vez, passando cada um deles para o nó seguinte. O diagrama da Figura 3.43 esclarece esta circunstância. Neste exemplo, os itens *e* e *f* são dois valores consecutivos do vetor em questão.

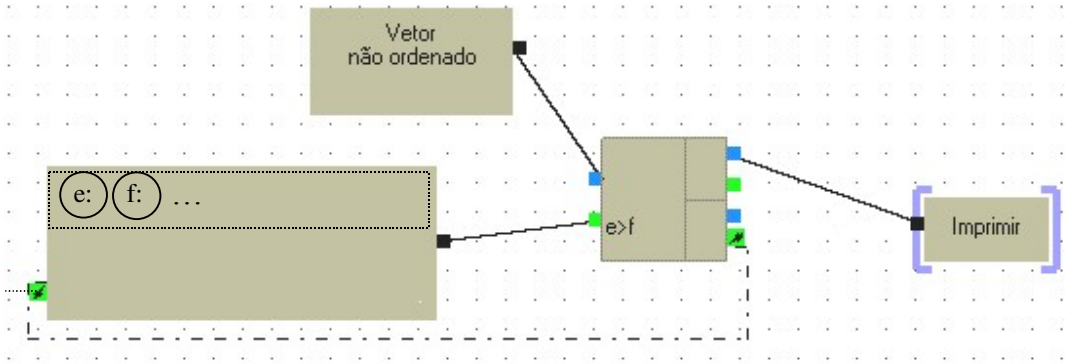


Figura 3.43: Sincronização entre um processo iterativo e um processo não-iterativo.

A terceira situação refere-se à ocorrência de duas ou mais matrizes paralelas, não diretamente conectadas, que precisam sincronizar o processo de iteração. Esta é uma situação na qual mais de um agregado é envolvido no processo iterativo. Neste caso, as iterações de vários agregados precisam ser sincronizadas. O diagrama da Figura 3.44, verifica se dois vetores possuem os mesmos números em ordem inversa:

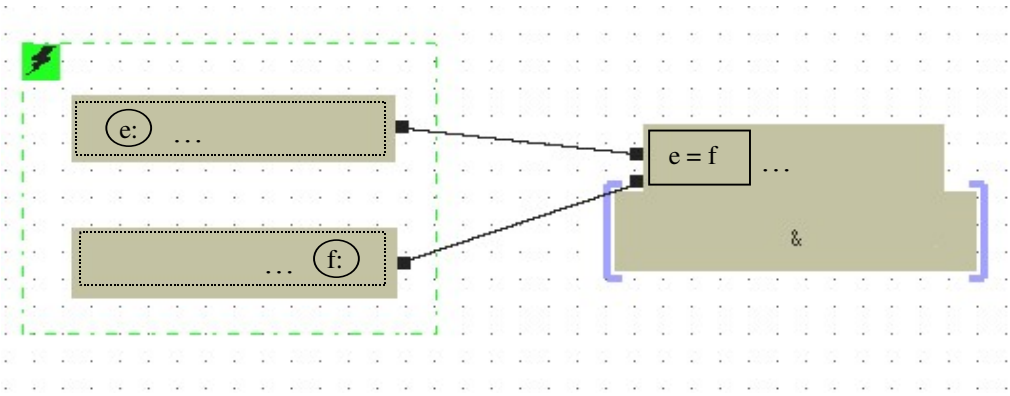


Figura 3.44: Sincronização entre dois processos iterativos.

O número total de iterações será igual ao número de elementos da matriz de menor dimensão. O sinal de final de matriz é expedido após o envio do último elemento de uma das matrizes sincronizadas para que o processo ou a matriz que esteja recebendo os dados saiba que não existem mais dados a serem recebidos.

3.3.4 Operador de Fusão

O nó que representa a operação de fusão é ativado quando pelo menos uma das portas de entrada recebe algum valor. Este valor é repassado para a porta de saída. A Figura 3.45 ilustra o operador de fusão conectado a três processos.

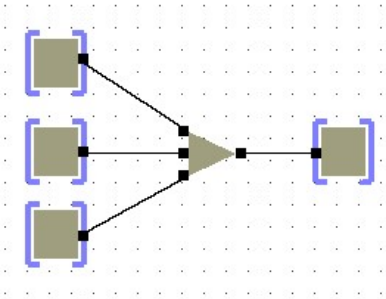


Figura 3.45: Operador de fusão.

O operador de fusão preserva a ordem temporal dos dados que chegam em suas portas de entrada. Esta ordem não é determinística, se dois dados chegam ao mesmo tempo, qualquer um pode ser enviado para a porta de saída. Este operador possui uma fila, cujo tamanho é igual ao número de portas de entrada, para garantir o processamento de todos os dados ali postados.

3.3.5 Inserção de Documentação

A documentação do código-fonte permite que o programador forneça informação adicional para facilitar o entendimento do grafo do programa. Apesar da documentação das linguagens de programação visual não ser muito desenvolvida, várias implementações suportam documentação textual, funcionalmente equivalente à utilização de comentários nas linguagens textuais. Por exemplo, a linguagem Prograph permite que o programador insira anotações no grafo de fluxo de dados.

A inserção de comentários agrava o problema relacionado ao espaço físico disponível para a implementação de uma função. Algumas soluções foram adotadas para minimizar este

problema. O Prograph, por exemplo, permite que o usuário controle quando a documentação será mostrada. Neste caso, só existem duas possibilidades: mostrar ou não o comentário. O Fabrik estende a solução adotada pelo Prograph com a adoção de diferentes níveis de detalhe selecionáveis. Além disto, o comentário só é mostrado quando o cursor se encontra sobre a área reservada para o mesmo.

O ProVisual adota a estratégia do Fabrik, adicionando-se a possibilidade de mostrar o comentário relacionado a uma função no ponto de invocação. O usuário não precisa editar o grafo da função para ler sua documentação inicial. Com o cursor em cima do ícone que representa a chamada da função, é mostrado, caso exista, um retângulo com o comentário sobre sua funcionalidade.

A documentação numa linguagem de programação visual constitui, na opinião do autor deste trabalho de pesquisa, mais uma oportunidade do que um problema, uma vez que essas linguagens possuem características dinâmicas não encontradas nas linguagens textuais, permitindo que a documentação da implementação supere a inserção de um simples comentário textual.

3.3.6 Abstração Procedimental

Um importante avanço da programação foi a adoção da abstração procedimental, isto é, a habilidade para a criação de módulos que encapsulam detalhes de uma subtarefa. A abstração procedimental e a modularização são tão importantes nos projetos de programação que se torna difícil imaginar uma tarefa de programação sem aplicar tais conceitos. Um atributo importante do suporte a este tipo de abstração nas linguagens de programação visual é a sua consistência com os outros recursos sintáticos da mesma linguagem [Burnett, 1999].

O ProVisual admite quatro estratégias que viabilizam a utilização da abstração procedimental:

- Seleção de uma parte de um grafo (subgrafo) e sua associação a um ícone (um único nó no grafo original);

- Criação de um programa (grafo) que pode ser armazenado e invocado por outros programas;
- Utilização de casos de execução, permitindo a divisão dos procedimentos em duas ou mais alternativas de execução. Durante a execução do programa, apenas um dos casos pode ser ativado; e
- Definição de uma função a partir de uma planilha, utilizando-se o recurso de programação por exemplos das planilhas do ProVisual (Seção 3.2.3).

As duas primeiras estratégias são amplamente utilizadas pelas linguagens visuais de programação baseadas em fluxo de dados. A seleção de um subgrafo apenas substitui uma série de nós por um único nó, tornando o grafo mais conciso. A criação de um grafo como um subprograma está relacionada à criação de bibliotecas de funções reutilizáveis, que podem ser invocados por outros programas.

Por sua vez, a terceira estratégia, a utilização de casos de execução, permite a estruturação de um procedimento em termos das circunstâncias que ativam um determinado processamento. Finalmente, uma planilha pode ser vista como uma função que recebe valores, processa-os e envia-os para outros nós do grafo que representa o programa.

Procedimentos

O conceito de abstração funcional é facilmente encaixado no modelo de fluxo de dados, bastando definir a chamada do procedimento como um nó do grafo. Os dados de entrada deste nó são os dados repassados para o procedimento. A Figura 3.46 exemplifica um programa com uma chamada para o procedimento *A* que, por sua vez, invoca um procedimento *B*. Neste exemplo, a função soma as duas entradas do procedimento *A*, se o valor de seu primeiro parâmetro for maior ou igual a zero. Caso contrário, a execução do programa será finalizada.

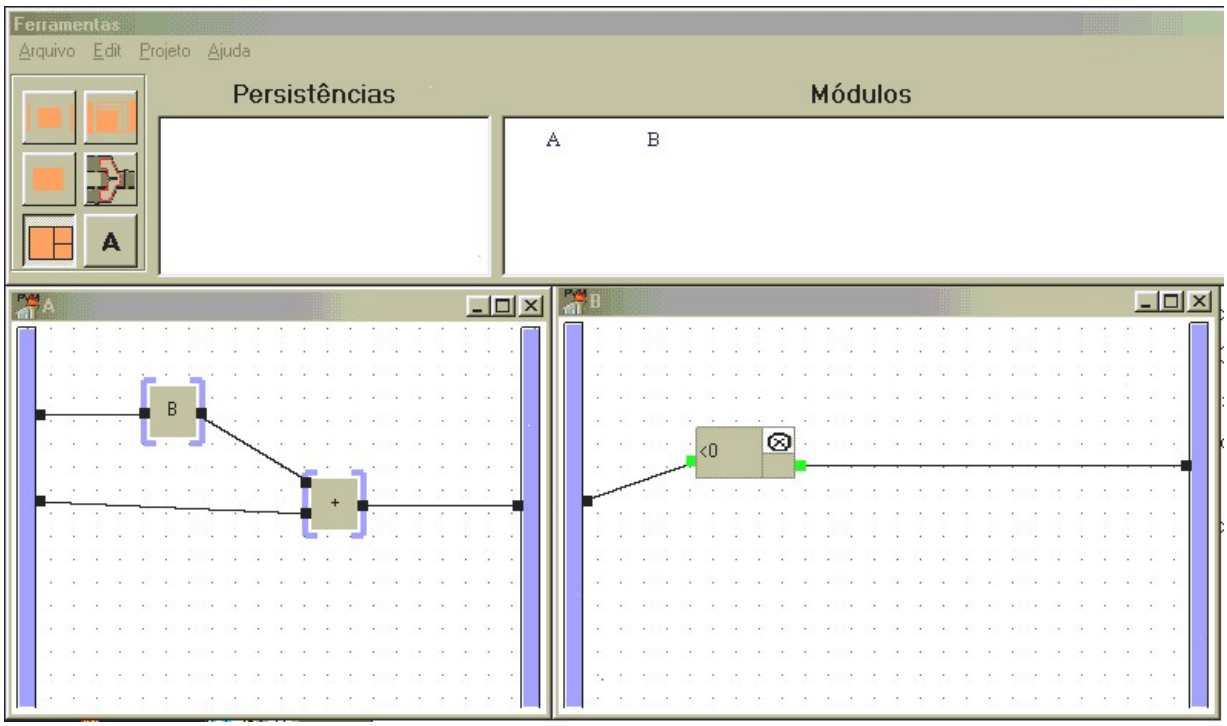


Figura 3.46: Exemplo de abstração procedimental.

Os procedimentos em ProVisual podem ser de dois tipos. O primeiro representa um subprograma, que pode ser invocado por outro subprograma ou programa. É equivalente ao procedimento das linguagens imperativas e visa dotar o ProVisual de recursos para a criação de bibliotecas de funções pré-definidas. O segundo tipo de procedimento é criado a partir de um subgrafo. Neste caso, uma parte do grafo, que representa um subprograma, é iconizada e encapsulada num nó local. Este nó não pode ser chamado em qualquer outro lugar do programa. O principal motivo para a adoção de subprogramas locais é o fornecimento de meios para a criação de programas mais concisos, diminuindo, desta forma, o problema relacionado ao espaço requerido para se representar um programa visual.

A associação dos parâmetros decorre do uso das barras de entrada e de saída. Todos os parâmetros de entrada necessários são associados à barra de entrada, enquanto os parâmetros de saída são conectados à barra de saída. Não existe limite para o número de portas de entrada e de

saída. Todavia, as funções internas do ProVisual possuem um número portas pré-fixado. Ou seja, não se pode alterar o número de portas das funções já existentes no ProVisual.

Casos de execução

Existem procedimentos que consistem em um ou mais casos de execução, que possuem os mesmos números de portas de entrada e de saída. Cada caso é formado por uma série de operações e matrizes. Estas operações podem ser referências a outros procedimentos ou funções. A execução de um caso é controlada por uma condição colocada no corpo do programa. Dependendo da condição contida neste caso, a execução do procedimento pode prosseguir ou a execução é desviada para o próximo caso, se porventura existir. A Figura 3.47 retrata um programa para o cálculo da soma matricial entre duas matrizes.

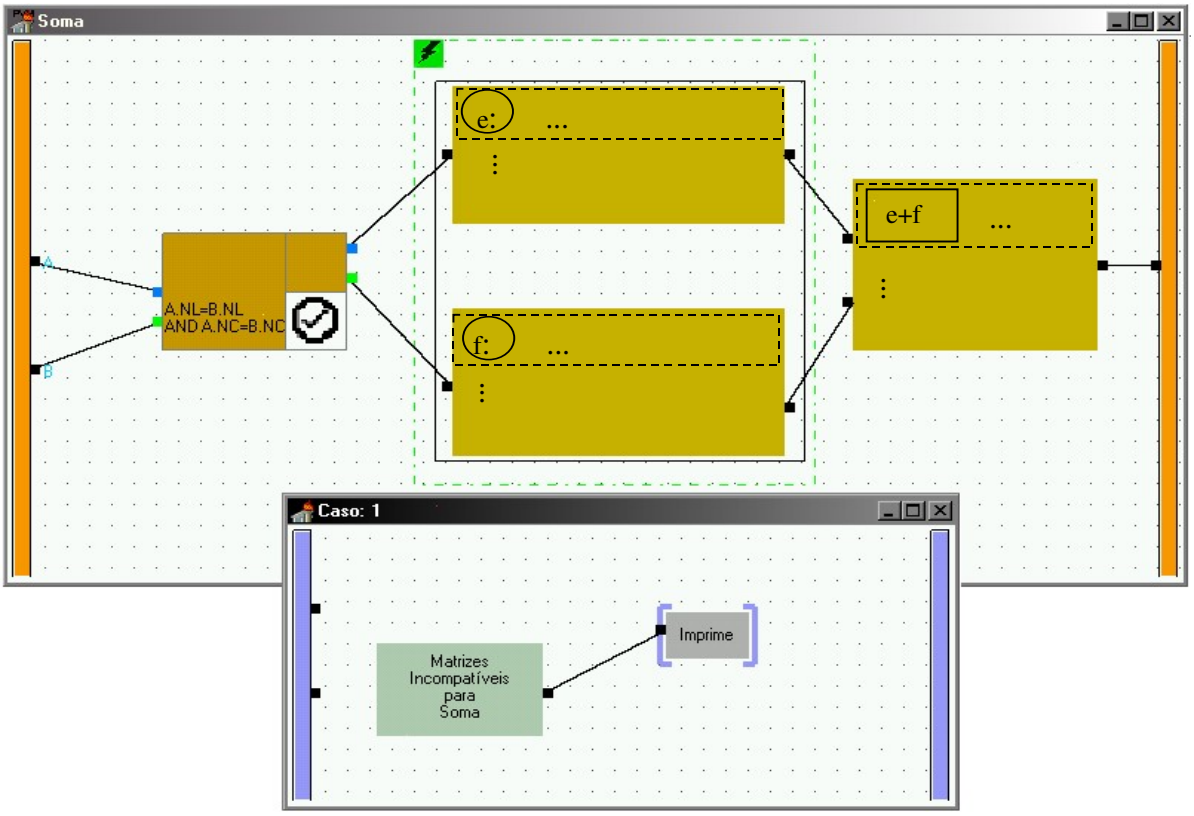


Figura 3.47: Procedimento para cálculo de soma matricial com dois casos de execução.

No programa representado pela figura, a implementação foi dividida em dois casos de execução. O primeiro caso possui um operador condicional e três matrizes, associadas a processos iterativos. A função *soma* implementada percorre duas matrizes, elemento a elemento, e repassa dois de seus elementos para uma terceira matriz. Esta matriz calcula a soma dos dois elementos e o atribui a uma determinada célula. O segundo caso de execução é ativado se as matrizes não possuírem dimensões compatíveis para a operação desejada. Nesta situação, o fluxo de dados é transferido para o segundo caso, que apenas imprime uma mensagem, indicando que as matrizes são incompatíveis. Cabe ressaltar que o segundo caso não utiliza nenhum de seus parâmetros.

3.4. Comparação do ProVisual com outras Linguagens Visuais

Algumas linguagens (APL [Iverson, 1973] e MathLab [Kajler, 1995]) incorporam operadores para a construção e a manipulação de matrizes. Estas linguagens são textuais e possuem uma notação densa, o que freqüentemente dificulta o entendimento do programa. A estratégia adotada nesta pesquisa procura eliminar esta deficiência pela utilização da notação visual e da manipulação direta de uma matriz.

A planilha eletrônica é possivelmente a forma mais conhecida para a manipulação direta de matrizes. Ela introduz o conceito de referência de células pelo apontamento direto das mesmas, além da replicação de fórmulas com a utilização do recurso de cópia. Todavia, ela não é adequada para a manipulação de submatrizes e, ademais, não incorpora o conceito de abstração funcional.

A linguagem MPL (Yeung, 1988) é uma linguagem de manipulação de matrizes fundamentada nos paradigmas de restrições e lógica. Ela provê um conjunto de elementos para a notação gráfica de uma matriz, enquanto as cláusulas lógicas são informadas textualmente. Em MPL, uma matriz é formada por um retângulo fechado e um conjunto de sub-retângulos internos que delimitam várias submatrizes. O programador cria um programa (cláusulas Prolog) com um

editor de texto modificado, no qual uma matriz pode ser manipulada como um texto. Conseqüentemente, um diagrama (matriz) pode ser mesclado com textos que representam as cláusulas. Uma vez criado o programa, o programador invoca o MPL para traduzir os diagramas em predicados Prolog. Após a tradução, o programador precisa informar uma cláusula de averiguação para que o sistema possa executá-la. Pode-se afirmar que um programa em MPL é consideravelmente mais complexo do que o proposto por este trabalho. Isto decorre da necessidade de aprendizado anterior do Prolog para a utilização do MPL.

Uma outra linguagem para a manipulação de matrizes é o FORMS/3 [Ambler, 1997]. Existem algumas similaridades entre a proposta deste trabalho e o FORMS/3. Ambos utilizam a manipulação da representação visual de matrizes para a descrição de algoritmos matriciais. Os dois fazem uso de cores para identificar e selecionar partes de uma matriz. No entanto, a principal diferença relaciona-se à estratégia adotada para a programação de uma matriz. Neste trabalho de pesquisa é utilizado o paradigma de programação por exemplos com o intuito de aumentar a produtividade na implementação de algoritmos matriciais. O ProVisual adota, ainda, estratégias que procuram reduzir a necessidade de utilização da recursão na programação, fartamente utilizada no FORMS/3. Uma outra diferença significativa é a explicitação dos fluxos de dados envolvidos nas fórmulas de uma planilha. O autor acredita que a visualização dos fluxos de dados entre as fórmulas reduzem a ocorrência de efeitos colaterais no desenvolvimento e na manutenção de um programa ProVisual.

A linguagem de programação visual V utiliza um construtor iterativo especializado para atuar com dados vetoriais. Este mesmo construtor foi utilizado pelo ProVisual para dotar uma matriz de padrões de iteração. Todavia, o ProVisual permite também a utilização de padrões de iteração dentro de regiões. Estas regiões podem ser estruturadas de forma aninhada ou adjacente, permitindo um maior poder de aplicação dos padrões de iteração do modelo proposto. Além disso, ele emprega planilhas para a representação de uma matriz e permite um padrão de iteração para dados tanto recebidos quanto enviados de outros processos.

A definição do ProVisual foi influenciada por algumas linguagens de programação visual. No entanto, ele estende os conceitos incorporados e propõe novas soluções que visam facilitar a tarefa de programação de algoritmos matriciais. Nesse contexto, vale ressaltar a utilização de uma planilha como função, aplicando a programação por exemplos, e a adoção de um modelo de

programação visual híbrido, combinando a facilidade e a flexibilidade das planilhas com a expressividade do fluxo de dados. Portanto, o ProVisual constitui-se em uma nova proposta para a programação e a manipulação de matrizes.

3.5 Conclusão

Neste capítulo, o autor apresentou um Modelo para Programação Visual de Matrizes – ProVisual, como base da arquitetura visual para a modelagem de métodos estatísticos e matemáticos do domínio matricial. O principal objetivo do modelo proposto pelo autor é permitir o aumento da eficiência na implementação de algoritmos por pesquisadores sem experiência em programação.

Neste sentido, o autor descreveu, em primeiro lugar, o conceito de planilha, que representa o objeto de dados suportado pelo modelo. O conceito de planilha foi introduzido para manipular uma matriz sem a necessidade de recorrer a uma notação textual, considerando que a representação visual tende a facilitar seu entendimento e sua manipulação por usuários não treinados em programação.

Em segundo lugar, o autor apresentou a criação de visões sobre uma matriz a partir de partições e de padrões de iteração. A partição de uma matriz associada ao padrão de iteração possibilita a utilização de uma planilha como um robusto construtor de iteração. Desta forma, a planilha pode representar um processo iterativo em um nível mais alto de abstração, ao mesmo tempo em que facilita a implementação de algoritmos matriciais.

O autor propôs a utilização de uma planilha como uma função a partir da utilização da técnica de programação por exemplos. A programação por exemplos facilita a tarefa de implementação de algoritmos matriciais sem a necessidade, em muitos casos, de se recorrer a construtores usuais de uma linguagem de programação. Neste caso, uma ou mais matrizes de entrada são aplicadas a uma máscara pré-programada. Esta máscara permitirá, por exemplo, que métodos matemáticos possam ser experimentados em matrizes de dimensões menores e, em seguida, aplicados a outras matrizes maiores.

Em quarto lugar, o autor descreveu a técnica de visualização transitória dos fluxos de dados entre as fórmulas de uma planilha, permitindo a diminuição de possíveis efeitos colaterais decorrentes da existência de relações escondidas entre as fórmulas. Esta técnica mostra os fluxos de dados tanto entre as células que afetam a fórmula quanto entre as células que são por ela afetadas.

Por último, o autor expôs a linguagem de composição de planilhas baseada em fluxo de dados. Esta linguagem fornece suporte gráfico para os construtores de expressão simples, repetição, decisão, fusão e documentação. Ademais, permite o controle da sincronização de processos e viabiliza a abstração funcional a partir da definição de procedimentos (globais e locais) e de casos de execução.

A linguagem de composição admite três formas para compor dois *pictogramas*: interseção, composição e conexão. As diferentes formas de composição, associadas ao conceito de abstração funcional, fornecem meios para tornar um programa ProVisual mais conciso, permitindo a redução de problemas de escalabilidade e de espaço físico do monitor.

Capítulo 4

Descrição Formal

Neste capítulo, o autor desenvolve a descrição formal dos elementos da sintaxe e da semântica estática do ProVisual, utilizando a lógica espacial para o cálculo de conexões de regiões. A descrição da formalização considera a geometria básica dos objetos e o inter-relacionamento estático entre os elementos visuais da linguagem.

4.1 Uma Abordagem para a Descrição Formal do ProVisual

Toda especificação de linguagem textual seqüencial considera a distinção entre os aspectos sintáticos e semânticos da linguagem. A sintaxe descreve a estrutura das sentenças, enquanto a semântica determina o seu significado. A sintaxe é normalmente especificada pela utilização de uma gramática composta por regras que retratam a composição dos componentes gramaticais em frases ou sentenças. A semântica de uma sentença deriva-se do significado de cada palavra e suas posteriores combinações, seguindo-se a estrutura sintática.

Antes de definir a abordagem empregada na formalização do ProVisual, é conveniente estabelecer as principais diferenças existentes entre uma linguagem visual e uma linguagem textual. Em primeiro lugar, a seqüência de desenho dos componentes não é importante nas linguagens visuais, pois não altera o significado da sentença. Em segundo lugar, uma linguagem visual emprega relacionamentos de adjacência e de conexão. Em geral, existe um grande número de relacionamentos importantes empregados em uma linguagem visual. Por exemplo: 'conectividade', 'à esquerda de', 'perto de', 'dentro de', etc. No caso de uma linguagem textual, o único tipo de relacionamento existente é o imediatamente precedente.

Atualmente, existem três abordagens mais pesquisadas para a formalização de uma linguagem visual: a gramatical, a algébrica e a lógica [Marriott, 1998].

A abordagem gramatical baseia-se no formalismo das gramáticas utilizadas para especificar linguagens textuais. O formalismo gramatical para especificação de linguagens visuais normalmente difere do formalismo textual pela utilização de conjuntos ou multiconjuntos no lugar de seqüências e, principalmente, pela especificação dos relacionamentos geométricos entre os objetos da linguagem [Fu, 1973].

A abordagem algébrica utiliza a especificação algébrica, que consiste na composição funcional de elementos gráficos mais simples para obter construções mais complexas. O processo de análise (*parsing)* pode ser considerado análogo à busca da seqüência funcional, que constrói um diagrama mais complexo a partir da composição de objetos mais simples. A semântica pode ser capturada pela utilização de especificações algébricas paralelas para os diagramas utilizados na linguagem [Wang, 1993].

Finalmente, a abordagem lógica utiliza a lógica matemática de primeira ordem, usualmente baseada na lógica espacial para fornecer os axiomas dos relacionamentos topológicos (geométricos) entre objetos [Gooday, 1996]. Uma das vantagens deste tipo de abordagem é que o mesmo formalismo pode ser utilizado para especificar a parte tanto sintática como semântica de um diagrama.

A formalização do ProVisual é estruturada segundo a proposta de Gooday [1996], que descreve uma abordagem para a descrição formal de linguagens visuais, na qual a sintaxe e a semântica podem ser especificadas a partir da lógica espacial.

4.1.1 Um Resumo da Formalização Espacial Lógica

O Cálculo de Conexões de Regiões (CCR) foi adotado neste trabalho de pesquisa como fundamentação teórica para a formalização do ProVisual. O CCR baseia-se no trabalho de Clarke [1981] que propôs, utilizando as teorias da Lógica de Primeira Ordem e de Conjuntos, um cálculo topológico de indivíduos e conexões, cujo conceito primitivo é uma região. As regiões não precisam ser contínuas. A primitiva básica é a existência de uma conexão entre duas regiões, denotada por C(x,y), significando que a região 'x' conecta-se à região 'y'.

Utilizando-se a relação C, outras relações são definidas: DC(x,y) (“x é desconectado de y”), P(x,y) (“x é parte de y”), PP(x,y) (“x é uma parte própria de y”) , O(x,y) (“x sobrepõe y”), EC(x,y) (“x externamente conectado com y”), PO(x,y) (“x sobrepõe parcialmente y”), TPP(x,y) (“x é parte própria tangencial de y”), EQ(x,y) (“x é idêntico à y”), NTPP(x,y) (“x é uma parte própria não tangencial de y”).

As relações P, PP, TPP, NTPP possuem inversa (simbolizadas por: Pi, PPi, TPPi e NTPPi). Dessas relações definidas, DC, EC, PO, TPP, TPPi, NTPP, NTPPi e EQ formam um conjunto exaustivo de oito relações topológicas básicas [Randell, 1992]. A Figura 4.1 mostra, esquematicamente, o significado dessas oito relações, enquanto a Tabela 4.1 as descreve.

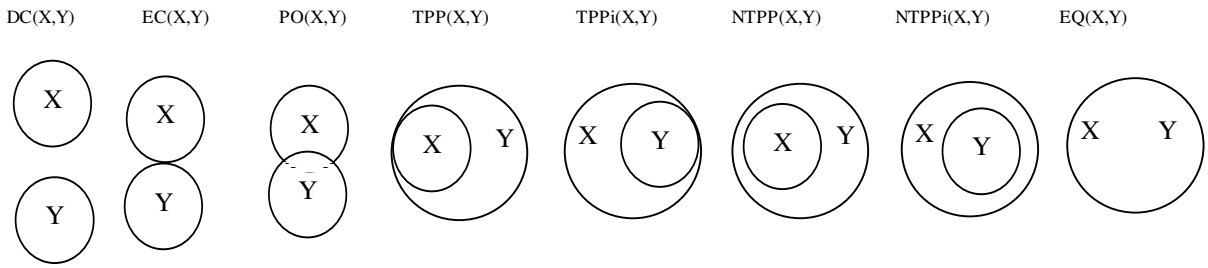


Figura 4.1: As oito relações topológicas básicas entre duas regiões [Randell, 1992].

Tabela 4.1: Definições das relações básicas entre duas regiões [Randell, 1992].

DC(X,Y)	$\neg C(X,Y)$
P(X,Y)	$\square\ Z[\ C(Z, X) \Rightarrow C(Z,Y)]$
PP(X,Y)	$P(X,Y) \wedge \neg P(Y,X)$
O(X,Y)	$\exists\ Z\ [\ P(Z,X) \wedge P(Y,Z)]$
EC(X,Y)	$C(X,Y) \wedge \neg O(X,Y)$
PO(X,Y)	$O(X,Y) \wedge \neg P(X,Y) \wedge \neg P(Y,X)$
TPP(X,Y)	$PP(X,Y) \wedge\ [\exists\ Z\ [EC(Z,X) \wedge EC(Z,Y)]]$
TPPi(X,Y)	TPP(Y,X)
NTPP(X,Y)	$PP(X,Y) \wedge \neg\ [\exists\ Z\ [EC(Z,X) \wedge EC(Z,Y)]]$
NTPPi(X,Y)	NTPP(Y,X)
EQ(X,Y)	$P(X,Y) \wedge P(Y,X)$

Além de C(X,Y), outras primitivas, relacionadas a algumas propriedades de orientação física dos objetos gráficos, são necessárias para capturar todas as características sintáticas do ProVisual. Essas primitivas adicionais referem-se à determinação do posicionamento relativo de duas regiões (“à direita de” , “à esquerda de”, “acima de”, e “abaixo de”) e foram baseadas na proposta de [Cui, 1993].

Neste caso, faz-se necessária a definição das primitivas B1(x,y) (para o eixo cartesiano "x") e B2(x,y) (para o eixo cartesiano "y") que são verdadeiras se a região "x" for inteiramente anterior à região "y". Alguns axiomas para B1 e B2 são descritos a seguir:

$\forall x \neg Bi(x, x)$

$\forall xyz [Bi(x,y) \wedge Bi(y,z) \rightarrow Bi(x,z)]$

$\forall xy[Bi(x,y) \rightarrow \forall x_1y_1[(P(x_1,x)) \wedge P(y_1,y) \rightarrow Bi(x_1, y_1)]]$

Estas relações não conseguem indicar se uma região é ligeiramente anterior à uma outra, nem indicar que uma linha vertical intercepta duas regiões. Para esses casos, define-se:

D-B1(x,y) e D-B2(x,y):$D\text{-}B_1(x,y) \Rightarrow B_1(x,y) \wedge \neg B_2(x,y) \wedge \neg B_2(y,x)$

$D\text{-}B_2(x,y) \Rightarrow B_2(x,y) \wedge \neg B_1(x,y) \wedge \neg B_1(y,x)$

4.2 Descrição Formal do ProVisual

Como a lógica espacial CCR é fundamentada em regiões, faz-se necessária a definição das regiões de interesse para a formalização do ProVisual e de todos os elementos visuais utilizados para a construção de um programa. Cada elemento visual pode ser escrito ou desenhado com a utilização dos seguintes elementos primitivos: texto, linhas e curvas fechadas.

Neste contexto, é conveniente definir os conjuntos que simbolizam os elementos visuais do ProVisual (conjunto ProVisualE descrito a seguir), bem como as primitivas utilizadas na aconstrução dos elementos visuais (conjunto EPD descrito a seguir).

Elementos visuais do ProVisual

O conjunto ProVisualE é formado por quadros (subprogramas), portas, conexões e operadores. Estes operadores estão disponíveis para a composição de uma programa.

$$\text{ProVisualE}(x) \Rightarrow \text{Quadro}(x) \vee \text{Portas}(x) \vee \text{Conexão}(x) \vee \text{OPER\_LOOP}(x) \vee \text{OPER\_MERGE}(x) \vee \text{OPER\_DECISÃO}(x) \vee \text{OPER\_SIMPLES}(x) \vee \text{Matriz}(x) \vee \text{BarraParâmetros}(x) \vee \text{COMENTÁRIO}(x)$$

Elementos visuais que podem receber porta de entrada e saída

O conjunto ProVisualC representa todos os elementos visuais que podem receber portas de entrada e de saída. Ele foi estabelecido apenas para reduzir e facilitar a definição da sintaxe e da semântica estática do ProVisual.

$$\text{ProVisualC}(x) \Rightarrow \text{OPER\_LOOP}(x) \vee \text{OPER\_MERGE}(x) \vee \text{OPER\_DECISÃO}(x) \vee \text{OPER\_SIMPLES}(x) \vee \text{Matriz}(x) \vee \text{BarraParâmetros}(x)$$

Primitivas utilizadas para construção dos elementos visuais:

O conjunto EPD representa as primitivas básicas utilizadas para definir graficamente os elementos do conjunto ProVisualE.

$$\text{EPD}(x) \Rightarrow \text{CurvaFechada}(x) \vee \text{Linha}(x) \vee \text{Texto}(x) \vee \text{LinhaTracejada}(x) \vee \text{Ícone}(x) \vee \text{TrêsPontosHorizontais}(x) \vee \text{TrêsPontosVerticais}(x)$$

A Figura 4.2 retrata um programa que possui a maior parte dos elementos sintáticos do ProVisual, enquanto a Tabela 4.2 apresenta a definição informal da linguagem.

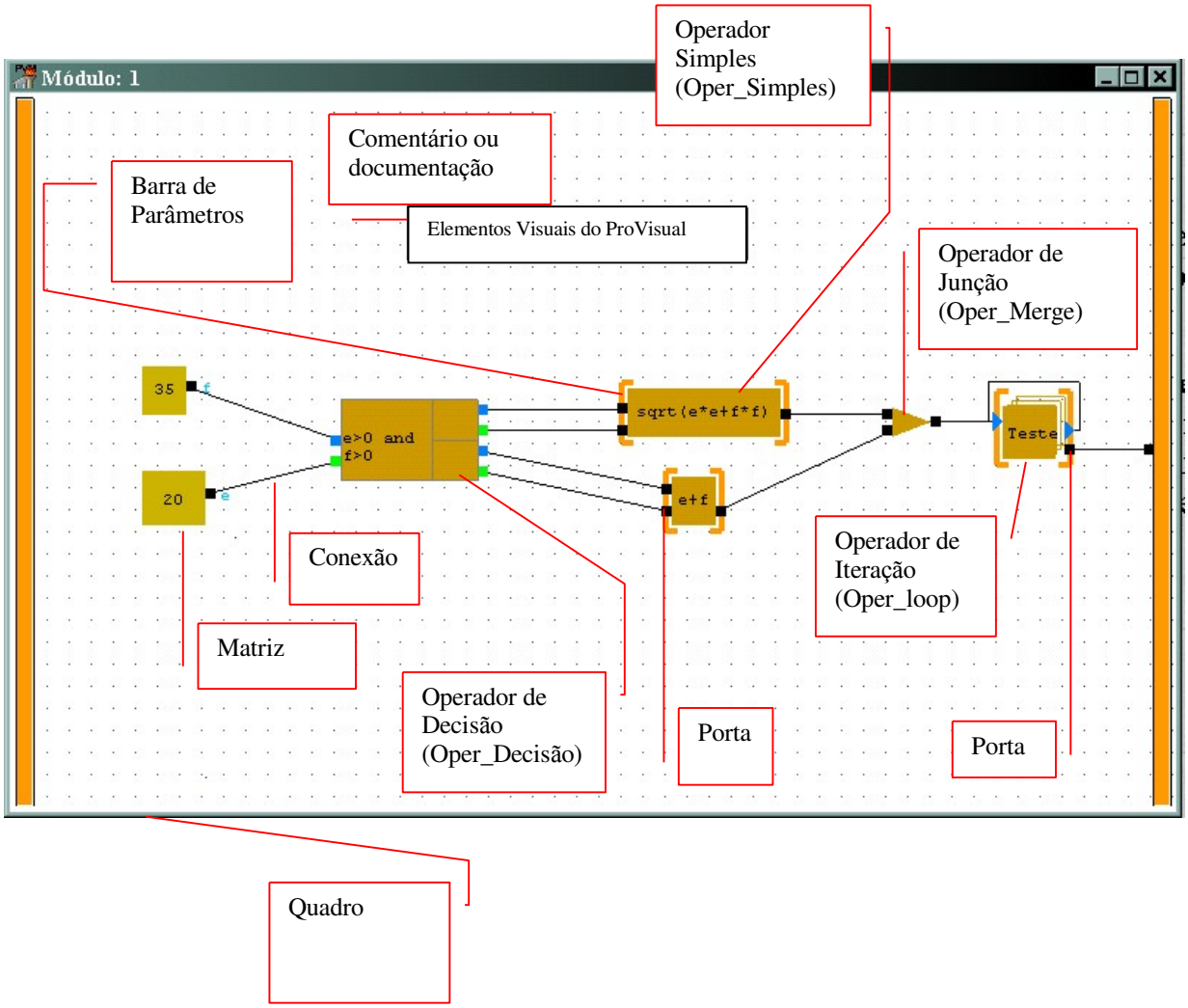


Figura 4.2: Ilustração de elementos sintáticos do ProVisual

Tabela 4.2: Definição informal dos elementos sintáticos do ProVisual.

Definição 1	Um programa P é formado por um conjunto de quadros.
Definição 2	Um quadro, de um programa P, é uma tupla (ID, Tipo, Nome do módulo, Entrada, Saída), na qual: ID: Identificador único de um quadro em P; Tipo: Identificador do tipo do quadro (processo ou planilha) Nome do módulo : ID.Rótulo se o quadro é local Rótulo caso contrário Entrada: Conjunto de portas de entrada; Saída: Conjunto de portas de saída.
Definição 3	Um quadro é montado pela composição dos elementos sintáticos – as planilhas, os operadores simples e de controle, os operadores do usuário, os arcos, as portas e os textos.
Definição 4	Porta de entrada: cada porta de entrada é uma tupla (ID, Cor, End, Posição) ID: Nome do módulo + Posição (Ordem) da porta de entrada End: Endereço da origem do arco conectado à porta Cor: Identificação da porta. Esta cor, se utilizada, é priorizada em relação à cor do arco.
Definição 5	Porta de Saída: cada porta de saída é uma tupla (ID, Nome, Cor, Planilha) ID: Nome do módulo + Posição (Ordem) da porta de saída Nome: Rótulo da porta Cor: Cor opcional para identificação da porta Planilha: Dados de saída
Definição 6	Uma planilha é uma tupla (ID, Nome, Cor, conjunto de células, atributos) Os atributos de uma planilha são: 1. Índices da linha e da coluna da submatriz: Se 0, indica todas as linhas e/ou colunas; 2. Dimensão (número de linhas e de colunas) da planilha em questão.
Definição 7	Uma célula em uma planilha é uma tupla (Linha, Coluna, Valor) Linha/Coluna representam a posição da célula na planilha; Valor : representado pelo conteúdo da célula P[Linha, Coluna]
Definição 8	Um operador simples: +, -, *, /, &, <.>, =, ...
Definição 9	Um operador do usuário representa uma chamada a um quadro implementado pelo usuário (subprograma).
Definição 10	Um operador de controle pode ser: decisão, repetição ou *merge.*
Definição 11	A composição de dois pictogramas resulta da inserção de um arco entre uma porta de saída de um dos pictogramas e uma porta de entrada do segundo pictograma.

4.2.1 Quadros

Um quadro representa um subprograma. Ele é uma região, inicialmente vazia, que possui uma barra de entrada, uma barra de saída e um nome.

Tipo: Quadro(ProVisualE, ProVisualE, ProVisualE, EPD)

$Quadro(x, e, s, tx) \Rightarrow (CurvaFechada(x) \wedge BarraParâmetros(e, s) \wedge Texto(tx)) \wedge [ProVisualE(e)]^*$

4.2.2 Barras de Entrada e de Saída

As barras de entrada e saída representam os pontos de ancoramento das portas de entrada e de saída de um subprograma, de um operador simples ou de um operador de iteração. As barras são apenas elementos sintáticos que permitem a diferenciação de outros elementos gráficos do ProVisual. A especificação a seguir indica que um elemento gráfico "Y", pertencente ao conjunto ProVisualC, possui dois locais para ancoramento dos parâmetros de entrada e de saída. O local para os parâmetros de entrada é formado por uma região fechada, não conectada a qualquer outro elemento, anterior a "Y". O local para os parâmetros de saída é formado por uma região fechada, não conectada a qualquer outro elemento, posterior a "Y".

Tipo: BarraParâmetros(EPD, EPD)

$BarraParâmetros(e, s) \Rightarrow (CurvaFechada(e) \wedge Curvafechada(s)) \wedge \neg\exists(y)[ProVisualC(y) \wedge DC(y, e) \wedge DC(y,s) \wedge B1(e,y) \wedge B1(s,y)]$

4.2.3 Porta

Uma porta é uma região vazia fisicamente conectada a uma outra região. Uma região R é classificada como região vazia, se não existir qualquer elemento do ProVisual no seu interior.

Tipo: Porta(ProVisualE)

Porta(*p)* $\Rightarrow$ CurvaFechada(p) $\wedge \neg\exists$(y) [EPD(y) $\wedge$ PP(y, *p*) $\wedge \neg$ ProVisualC(p)]

O ProVisual distingue as portas a partir de seus relacionamentos com outros elementos sintáticos. Uma porta serve como:

- porta argumento (PortaA) para representar passagem de parâmetro; e
- porta para retorno (PortaR) de resultado.

Tipo: PortaA(ProVisualE, ProVisualE)

Tipo: PortaR(ProVisualE, ProVisualE)

Um elemento gráfico "p" é uma porta de argumento caso ele seja uma porta tangencial ao elemento visual "x" e esteja situado à sua esquerda.

PortaA(p, x) $\Rightarrow$ Porta(p) $\wedge$ EC(p, x) $\wedge$ B1(p,x)

Um elemento gráfico "p" é uma porta de argumento caso ele seja uma porta tangencial ao elemento visual "x" e esteja situado à sua direita.

PortaR(p, x) $\Rightarrow$ Porta(p) $\wedge$ EC(p, x) $\wedge$ B1(x,p)

Os dois próximos axiomas são necessários para assegurar que uma porta de entrada ou de saída pertença a apenas um particular ProVisualE:

$\forall$ (*porta,* x) [PortaA(*porta*, x) $\Rightarrow \neg\exists$(y) [x $\neq$ y $\wedge$ PortaA(*porta*, y)]]

$\forall$ (*porta,* x) [PortaR(*porta,* x) $\Rightarrow \neg\exists$(y) [x $\neq$ y $\wedge$ PortaR(*porta*, y)]]

4.2.4 Conexão (arcos)

A conexão pode ser de dois tipos: passagem de dados (conexãoD) ou de sincronismo (conexãoS).

Tipo: Conexão(ProVisualE)
Tipo: ConexãoS(ProVisualE, ProVisualE, ProVisualC, ProVisualE, ProVisualC, EPD)
Tipo: ConexãoD(ProVisualE, ProVisualE, ProVisualC, ProVisualE, ProVisualC, EPD)

Uma conexão é uma linha que começa numa porta de resultado (saída) e termina numa porta de argumento (entrada). Essas portas não podem pertencer ao mesmo operador ($x \neq y$)) ou matriz, exceto para o operador de repetição ($x = y \wedge x = OPER\_LOOP$) . Uma porta de entrada pode receber apenas uma conexão, enquanto uma porta de saída pode enviar várias conexões.

$Conexão(L) \Rightarrow \exists$ (porta1, x, porta2, y, tx)
$[ConexãoS(L, porta1, x, porta2, y, tx) \vee ConexãoD(L, porta1, x, porta2, y, tx)]$
$ConexãoD(L, porta1, x, porta2, y, tx) \Rightarrow$

$Linha(L) \wedge EC(L, porta1) \wedge EC(L, porta2) \wedge Porta(porta1) \wedge$
$Porta(porta2) \wedge DC(porta1, porta2) \wedge PortaR(porta2, y) \wedge$
$[PortaA(porta1, x) \wedge \neg\exists(p,w) [p \neq porta2 \wedge [ConexãoD(L, porta1, x, p, w) \vee$
$ConexãoS(L, porta1, x, p, w)]]] \wedge [x \neq y \vee [x = y \wedge x = OPER\_LOOP]] \wedge$
$Texto(tx)$

$ConexãoS(L, porta1, x, porta2, y, tx) \Rightarrow$

$LinhaTracejada(L) \wedge EC(L, porta1) \wedge EC(L, porta2) \wedge Porta(porta1) \wedge$
$Porta(porta2) \wedge DC(porta1, porta2) \wedge PortaR(porta2, y) \wedge$
$[PortaA(porta1, x) \wedge \neg\exists(p,w) [p \neq porta2 \wedge [ConexãoD(L, porta1, x, p, w) \vee$
$ConexãoS(l, porta1, x, p, w)]]] \wedge [x \neq y \vee [x = y \wedge x = OPER\_LOOP]] \wedge$
$Texto(tx)$

A condição DC(porta1, porta2) garante que duas portas diferentes estão sendo utilizadas.

4.2.5 Operador Simples

Um operador simples possui um contorno fechado com uma ou mais portas de entrada e/ou de saída. Em adição, ele possui um rótulo que pode ser utilizado para armazenar um nome ou uma expressão.

Tipo: OperadorSimples(ProVisualE)

Tipo: OperadorSimples1(ProVisualE, ProVisualE, ProVisualE, EPD, EPD, EPD)

OperadorSimples(func) $\Rightarrow$

$\exists$ (contorno, portaA*, portaR*, rótulo, barraE, barraS)

[OperadorSimples(contorno, portaA*, portaR*, rótulo, barraE, barraS) $\wedge$

func = CurvaFechada+PortaA*+PortaR*]

OperadorSimples1(contorno, portaA*, portaR*, rótulo, barraE, barraS) $\Rightarrow$

CurvaFechada(contorno) $\wedge$ PortaA(portaA, contorno)* $\wedge$

Texto(rótulo) $\wedge$ P(rótulo, contorno) $\wedge$ PortaR(portaR, contorno)* $\wedge$

barraParâmetro(barraE, barraS)

4.2.6 Operador *Junção*

Um operador de junção possui um contorno fechado com uma ou mais portas de entrada e uma única porta de saída.

Tipo: OperadorMerge(ProVisualE)

Tipo: OperadorMerge(ProVisualE, ProVisualE, ProVisualE)

OperadorMerge(merge) $\Rightarrow$ $\exists$ (contorno, portaA*, portaR)[

OperdorMerge(contorno, portaA*, portaR) $\wedge$ merge =

CurvaFechada+PortaA*+PortaR

OperadorMerge (contorno, portaA*, portaR) $\Rightarrow$ CurvaFechada(contorno) $\wedge$

PortaA(portaA, contorno)* $\wedge$ PortaR(portaR, contorno)

4.2.7 Operador de Repetição

Um operador de repetição possui um contorno fechado, acrescido de duas regiões com sobreposição parcial, com uma ou mais portas de entrada e saída.

Tipo: OperadorRepetição(ProVisualE)
Tipo: OperadorRepetição1(ProVisualE, ProVisualE, ProVisualE,
EPD, EPD, EPD, EPD, EPD)
OperadorRepetiçao(loop) $\Rightarrow$ $\exists$ (contorno, portaA*, portaR*, rótulo, contorno1,
contorno2, barraE, barraS)
[OperdorRepetição1(contorno, portaA*, portaR*, rótulo, contorno1, contorno2, barraE, barraS) $\wedge$ loop = curvaFechada+PortaA*+PortaR*
OperadorRepetição1(contorno, portaA*, portaR*, rótulo, contorno1, contorno2), barraE, barraS] $\Rightarrow$
CurvaFechada(contorno) $\wedge$ PortaA(portaA, contorno)* $\wedge$ PortaR(portaR, contorno)* $\wedge$[Texto(rótulo) $\wedge$ P(rótulo, contorno)]$\wedge$[CurvaFechada(contorno1) $\wedge$ PO(contorno, contorno1)] $\wedge$ [CurvaFechada(contorno2) $\wedge$
PO(contorno, contorno2)] $\wedge$ barraParâmetro(barraE, barraS)

4.2.8 Operador de Decisão

Um operador de decisão possui um contorno fechado, acrescido de duas regiões internas que representam a parte verdadeira e falsa do operador, podendo receber ações específicas para cancelar o processamento ou desviar o fluxo de dados para um outro caso de execução. Este operador possui N portas de entrada e o mesmo número de portas em suas partes verdadeira e falsa, caso nenhuma ação específica lhes tenha sido atribuída.

Tipo: OperadorDecisão(ProVisualE)
Tipo: OperadorDecisão1(ProVisualE, ProVisualE, ProVisualE, EPD, EPD,
EPD, EPD)

OperadorDecisão(decisão) $\Rightarrow$ $\exists$ (contorno, portaA*, portaR*, contorno1, contorno2, ícone1, ícone2)

[OperadorDecisão1(contorno, portaA*, portaR*, contorno1, contorno2, ícone1, ícone2) $\wedge$ decisão = CurvaFechada+PortaA*+PortaR*]

OperadorDecisão(decisão) $\Rightarrow$ $\exists$ (contorno, portaA*, portaR*, contorno1, contorno2, ícone1,ícone2)[OperadorDecisão1(contorno, portaA*, portaR*, contorno1, contorno2, ícone) $\wedge$ decisão = CurvaFechada+PortaA*+PortaR*]

OperadorDecisão1 (contorno, portaA*, portaR*, contorno1, contorno2, ícone) $\Rightarrow$

CurvaFechada(contorno) $\wedge$ PortaA(portaA, contorno)* $\wedge$

[Texto(rótulo) $\wedge$ P(rótulo, contorno)] $\wedge$

[CurvaFechada(contorno1) $\wedge$ PO(contorno, contorno1) $\wedge$

B1(contorno1,contorno2) $\wedge$ B2(contorno1,contorno2) $\wedge$

[$\neg\exists$(y) [EPD(y) $\wedge$ PP(y, *contorno1*)] $\wedge$ [[(porta) [portaA(porta, y) $\rightarrow$

$\exists$ (porta´) [PortaR(porta´, contorno1)] $\wedge$ Ícone1=NULO] $\vee$

[EPD(ícone) $\wedge$ P(ícone, contorno1)]] ^

[CurvaFechada(contorno2) $\wedge$ PO(contorno, contorno2) $\wedge$

B1(contorno2,contorno) $\wedge$ B2(contorno1,contorno2) $\wedge$ [$\neg\exists$(y) [EPD(y) $\wedge$ PP(y, *contorno2*)] $\wedge$ [[(porta) [portaA(porta, y) $\rightarrow$ $\exists$ (porta´)[PortaR(porta´, contorno2)] $\wedge$ Ícone2=NULO] $\vee$

[EPD(ícone) $\wedge$ P(ícone, contorno2)]]

4.2.9 Comentário ou Documentação

Um comentário possui um contorno fechado com um texto em seu interior.

Tipo: Comentário(ProVisualE, EDP)

Comentário(contorno, texto) $\Rightarrow$

CurvaFechada(contorno) $\wedge$ Texto(rótulo) $\wedge$ P(rótulo, contorno)

4.2.10 Matriz

Uma matriz é representada por um contorno fechado com uma ou mais portas de entrada e/ou de saída. Ela possui um rótulo, que pode ser utilizado para armazenar um nome ou uma expressão, e um conjunto de subdivisões, que permite sua construção com dados de outras matrizes (portas de entrada). A matriz pode ser, ainda, subdividida, visando o envio de submatrizes pelas portas de saída.

Tipo: Matriz(ProVisualE)

Tipo: Matriz1(ProVisualE, ProVisualE, ProVisualE, EPD)

Tipo: SubMatriz(ProVisualE, ProVisualE, EPD, EPD, EPD, EPD)

Matriz(func) $\Rightarrow$ $\exists$ (contorno, portaA*, portaR*, rótulo)[

Matriz1(contorno, portaA*, portaR*, rótulo) $\wedge$ func =

CurvaFechada+PortaA*+PortaR*]

Matriz1(contorno, portaA*, portaR*, rótulo) $\Rightarrow$

CurvaFechada(contorno) $\wedge$ PortaA(portaA, contorno)* $\wedge$ Texto(rótulo) $\wedge$

P(rótulo, contorno) $\wedge$ PortaR(portaR, contorno)* $\wedge$

[SubMatriz(contorno, contorno1, dirV, dirH, rótulo1, rótulo2) $\wedge$

SubMatriz(contorno, contorno2, dirV1, dirH1, rótulo11, rótulo21) $\vee$ NULO]

Submatriz1(contorno, contorno1, dirV1, dirH1, rótulo1, rótulo2) ⇒

[CurvaFechada(contorno)∧[CurvaFechada(contorno1)∧TPP(contorno1, contorno)]

∧ [NULO ∨ [[TrêsPontosHorizontais(dirH1) ∧ TPP(dirH1, contorno)] ∨

[TrêsPontosVerticais(dirV1) ∧ TPP(dirH1, contorno)] ∨

[Texto(rótulo1) ∧ NTPP(rótulo1, contorno1]] ∨

[SubMatriz(contorno1, contorno3, dirV2, dirH2, rótulo21, rótulo22) ∧

SubMatriz(contorno1, contorno4, dirV3, dirH3, rótulo31, rótulo32)]]

4.3 Exemplo da Semântica Operacional do ProVisual

A semântica operacional do ProVisual será demonstrada para a execução do nó, que representa um módulo ou operador simples. O especificação da semântica dar-se-á via a técnica de Traços prposta por Parnas [1978]. Esta especificação consiste em duas partes. A primeira parte, chamada de Sintaxe, fornece a assinatura das interfaces de acesso as funcionalidades do programa. A segunda parte da especificação fornece a semântica dos operadores definidos na sintaxe.

O operador simples é formado por um corpo asssociado a um conjunto de portas de entrada (PortaE) e um conjunto de portas de saída (PortaS).

Porta de Entrada:

Sintaxe:	ADD:	<operando> x PortaE[i]	→	PortaE[i]
	Remove:	PortaE[i]	→	PortaE[i]
	Front:	PortaE[i]	→	<operando>

Legalidade:

1. $\lambda(T) \rightarrow \lambda(T.ADD(a))$
2. $\lambda(T) \rightarrow \lambda(T.ADD(a).Remove)$
3. $\lambda(T.Remove) \rightarrow \lambda(T.Front)$

Equivalência:

1. $\lambda(T.Front) \rightarrow T.Front \equiv T$
2. $\lambda(T.Remove) \rightarrow T.ADD(a).Remove \equiv T.Remove.ADD(a)$
3. $ADD(a).remove \equiv \varnothing$

Valores:

1. $V(ADD(a).Front) = a$
2. $\lambda(T.Front) \rightarrow V(T.ADD(a).Front) = V(T.Front)$

Porta de Saída:

Sintaxe:

ADD:	<operando> * PortaS[i]	→	PortaS[i]
Remove:	PortaS[i]	→	PortaS[i]
Front:	PortaS[i]	→	<operando>

Legalidade:

1. $\lambda(T)$ ➔ $\lambda(T.ADD(a).Remove)$
2. $\lambda(T.Remove)$ ➔ $\lambda(T.Front)$
3. $\lambda(T = \varnothing)$ ➔ $\lambda(ADD(a))$
4. $\lambda(T.ADD(a))$ ➔ $\lambda(T.Remove)$
5. $\lambda(T.ADD(a)$ ➔ $\lambda(T.Front = \varnothing)$

Equivalência:

1. $\lambda(T.Front)$ ➔ $T.Front \equiv T$
2. $\lambda(T.Remove)$ ➔ $T.ADD(a).Remove \equiv T.Remove.ADD(a)$
3. $ADD(a).remove \equiv \varnothing$

Valores:

1. $V(ADD(a).Front) = a$
2. $\lambda(T.Front)$ ➔ $V(T.ADD(a).Front) = a$

<u>Operador Simples:</u>

Sintaxe:

Execute:	PortaE x OperadorSimples	→	PortaE
GeraSaída:	PortaS x OperadorSimples	→	PortaS
estáEsperando:	OperadorSimples	→	booleano

estáPronto:	OperadorSimples	→	booleano
estáGerando:	OperadorSimples	→	booleano
PortaE[i]_tem_dado:	OperadorSimples	→	booleano

Legalidade:

1. λ(T) → λ(T.estáEsperando)
2. λ(T) → λ(T.estáPronto)
3. λ(T) → λ(T.estáExecutando)
4. λ(T) → λ(T.estáGernado)
5. λ(T) → λ(T.PortaE[i]_tem_dado)
6. V(T.estáPronto) = True → λ(T.Execute)
7. V(T.Execute.estaExecutando)=True → λ(T.GeraSaída)

Equivalência:

1. V(T.estáPronto) = True ≡ V(T. PortaE[i]_tem_dado) = True
   (∀ i | i >=1 e i <= número de Portas)
2. V(T.estáPronto) = False ≡ V(T.estáEsperando) = True
3. V(T.estáGerando) = True ≡ V(T.Execute.estáExecutando) = False
4. V(T.Execute.estáExecutando) = False ≡ V(T.estáGernado) = True
5. (V(T.Execute.estáExecutando) =False) e (V(T.Execute. estáGerando)= False)
   ≡ V(T.estáEsperando)

Valores:

1. V(T.execute.estáExecutando) = True

4.4 Conclusão

A descrição formal de uma linguagem visual é muito importante para sua implementação e verificação. Diferentemente de uma especificação de uma linguagem textual, ela deve levar em consideração as relações topológicas entre os objetos sintáticos da linguagem. A relação topológica mais importante para este trabalho de pesquisa é a conectividade, uma vez que todo o fluxo de dados do programa é controlado pela conexão entre operadores. Por este motivo, o ProVisual foi descrito formalmente segundo a lógica espacial de regiões. Uma das vantagens deste tipo de abordagem para a especificação de uma linguagem visual é a possibilidade de utilizar o mesmo formalismo para a sintaxe e a semântica dos diagramas da linguagem.

Capítulo 5

Implementação e Verificação

Neste capítulo, o autor apresenta a arquitetura da implementação do ProVisual. Em primeiro lugar, o autor descreve a arquitetura e os requisitos básicos do Ambiente de Programação ProVisual (APP). Em segundo lugar, expõe os editores visuais de programas e de matrizes. Em terceiro lugar, relata a forma de execução de um programa ProVisual na máquina virtual de fluxo de dados. Por fim, o autor desenvolve um exemplo para verificar o modelo proposto.

5.1 A Arquitetura da Implementação do ProVisual

O Ambiente de Programação ProVisual (APP) foi implementado inteiramente em Delphi [Borland, 2000] pelo rico conjunto de recursos oferecidos, tornando a tarefa de desenvolvimento mais fácil e rápida. Ele é voltado para a criação de interfaces gráficas de usuário (*Graphic User Interface* - GUI), em contraponto às linguagens clássicas orientadas a objetos (por exemplo, C++). O Delphi oferece, ainda, controle de exceções e gerenciamento de *threads,* fundamentais para a implementação da máquina virtual de execução.

Existem dois processos principais integrados ao APP: 1) o processo de codificação; e 2) o processo de execução de um programa. A Figura 5.1 apresenta a arquitetura do APP.

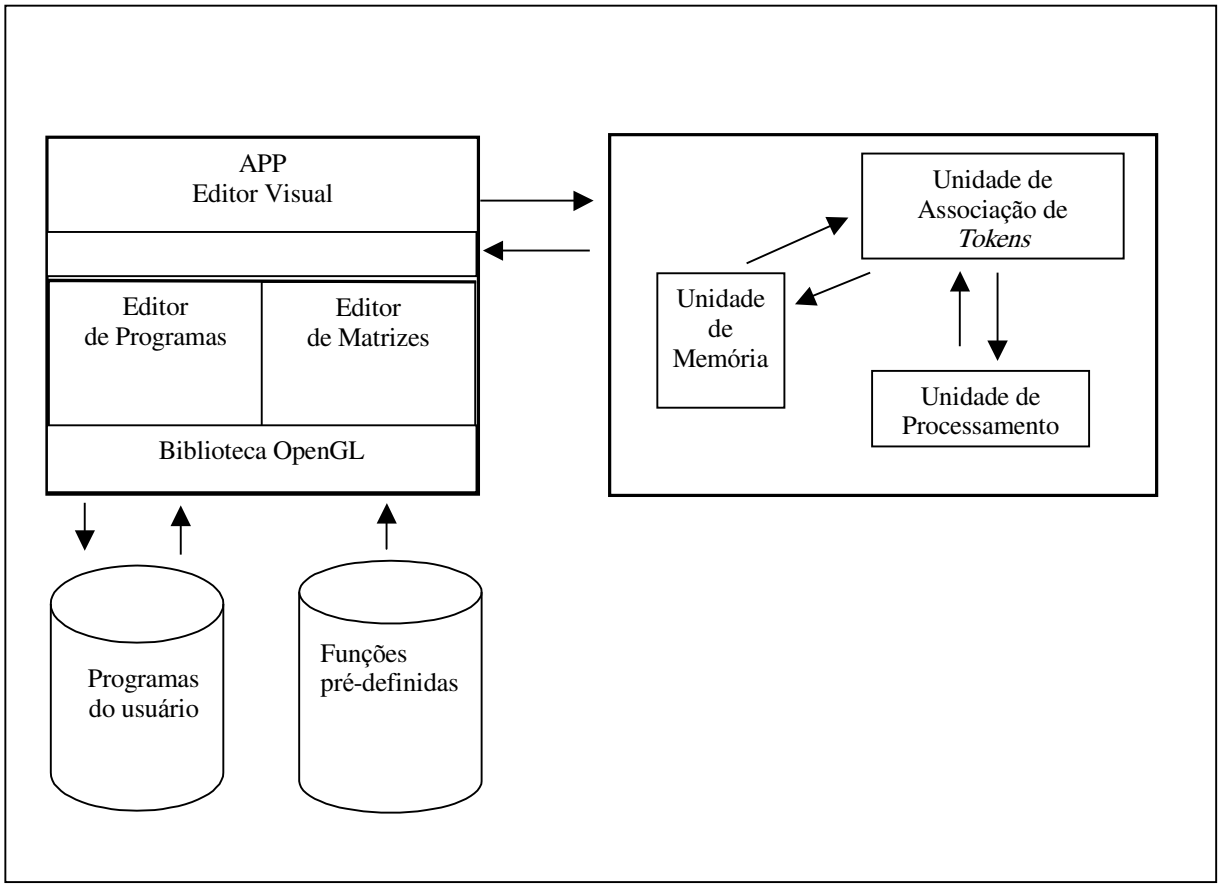


Figura 5.1: Arquitetura do Ambiente de Programação ProVisual (APP).

O processo de codificação do programa é dirigido pelo editor visual, que possui recursos distintos para a composição de programas e para a manipulação de matrizes: o editor de

programas e o editor de matrizes. O autor utiliza a biblioteca OpenGL [OpenGL, 2000] para dar suporte gráfico ao editor visual do APP, especificamente para representações gráficas em seu editor de programas.

O programa, uma vez implementado pelo uso do editor, pode ser executado a partir da sua invocação direta. Neste caso, *tokens*[13] são gerados e enviados para a máquina de execução. Esta máquina é dividida em três seções: 1) a unidade de processamento; 2) a unidade de associação de *tokens; e* 3) a unidade de memória.

A unidade de processamento recebe pacotes, contendo uma instrução e seus operandos. A instrução é executada com os respectivos operandos e o resultado, após uma rotulação apropriada, é enviado para a unidade de associação de *tokens.* Esta unidade procura por *tokens* que possuam o mesmo rótulo e destinação, verificando se o destino dos *tokens,* ou seja, se a instrução a ser executada, possui todos os valores de entrada disponíveis na unidade de memória. Caso afirmativo, a operação e seus *tokens* de entrada são retirados da memória, empacotados e enviados para a unidade de processamento. Caso contrário, o *token* em questão é armazenado na memória.

Assim que a execução de um programa se inicia, cria-se uma *thread* para avaliar e, ao mesmo tempo, controlar sua execução. Neste caso, abre-se uma janela que mostra quantas soluções (ou ciclos completados de um módulo em execução) foram encontradas. Ademais, habilita-se um botão para cancelar a execução. A adoção desta estratégia de *multi-threading* permite a intervenção do usuário em caso de computações infinitas.

[13] Um *token* representa uma unidade de informação básica em uma máquina de fluxo de dados. Ele encapsula os dados a serem executados e um conjunto de informações sobre seu contexto de execução.

5.2 O Editor Visual

O editor visual executa duas funções principais: 1) a composição de programas (editor de programas); e 2) a manipulação, estática ou dinâmica, de matrizes (editor de matrizes).

5.2.1 O Editor de Programas

O editor de programas destina-se ao processo de codificação de programas pela composição de ícones gráficos (que representam operações) e planilhas.

A interface principal do editor de programas consiste de: 1) um formulário com uma barra de menu na parte superior; 2) três janelas de controle das matrizes persistentes, dos subprogramas criados pelo usuário e das funções externas ao programa; e 3) um reduzido número de ícones gráficos disponíveis para a programação. A Figura 5.2 ilustra a interface principal do editor de programas.

O processo de edição de um programa é dirigido pela sintaxe e pela semântica estática do programa. O editor de programas do APP é totalmente orientado pela sintaxe válida para a composição de um programa, de acordo com as regras especificadas no Capítulo 4. Para cada tipo de operação, o APP habilita apenas as composições factíveis e controla o número máximo e mínimo de portas.

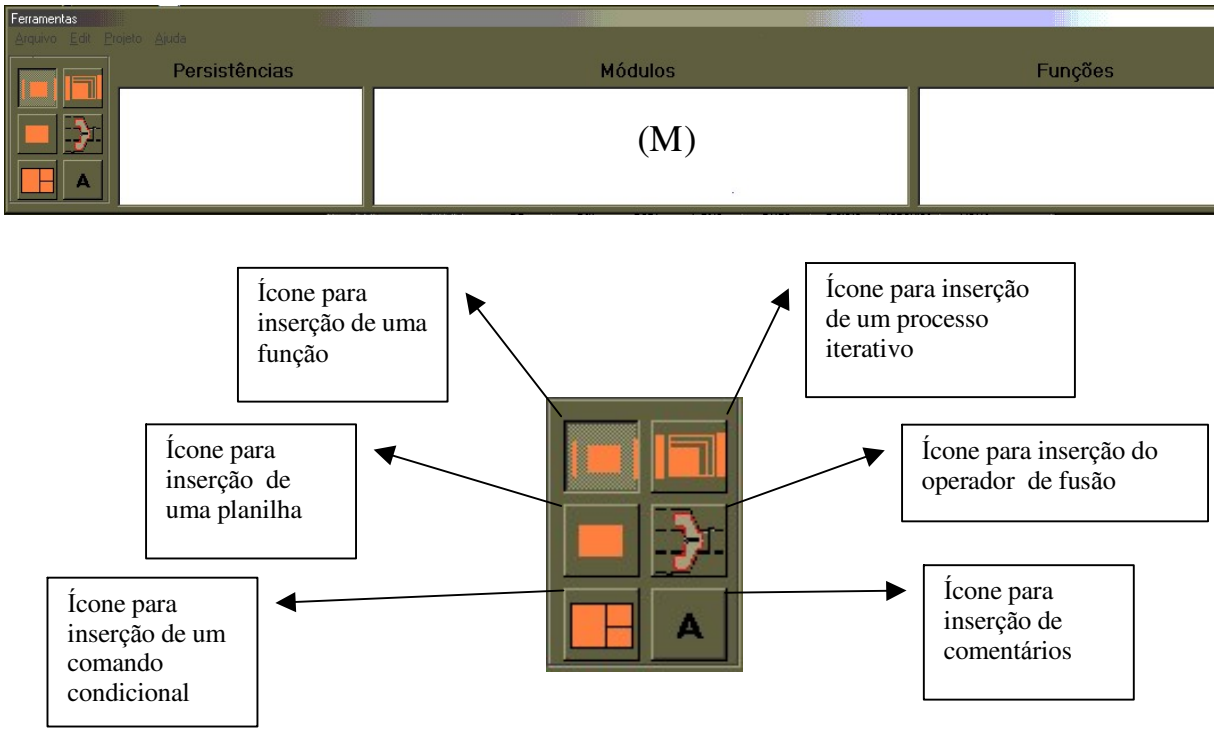


Figura 5.2: Interface principal do editor de programas do APP.

Quando o usuário deseja compor um programa, deve teclar duas vezes o primeiro botão do *mouse* na janela de módulos (M). Caso o usuário tenha teclado em cima de um módulo já implementado ou em implementação, a janela de edição se abre com este programa; caso contrário, um novo programa é aberto.

Na composição de um novo programa, o usuário escolhe e tecla sobre um dos ícones gráficos disponíveis. Em seguida, tecla em algum lugar na janela de edição para adicioná-lo à rede de fluxo de dados do programa.

O editor de programas suporta as funções básicas de edição gráfica, permitindo ao usuário mover, redimensionar, remover ou nomear, se for o caso, os ícones da rede. Cada ícone possui um conjunto de regras de composição que são validadas em tempo de construção.

As subseções seguintes descrevem o modo de interação entre o usuário e o editor de programas, ressaltando a forma de inserção de funções e de operadores, bem como de criação de conexões e de módulos locais.

Inserindo uma Função

Este ícone representa a chamada de uma função, de um procedimento ou de operadores básicos. A forma de definição do papel assumido pelo ícone depende de sua nomeação.

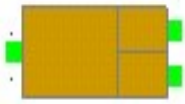
Em primeiro lugar, o usuário pode adotar o mesmo nome de funções ou operadores préexistentes na biblioteca do APP, permitindo o reuso de funções. A nomeação do ícone dar-se-á pela seleção, via botão da direita do *mouse,* de um dos itens da lista de operações disponíveis, mostrada em uma janela aberta do editor. A Figura 5.3 ilustra esta situação.

Figura 5.3: Definição do operador soma do APP.

Em segundo lugar, o usuário pode rotular um ícone com o mesmo nome de um procedimento ou de uma função já implementada, viabilizando sua invocação. Finalmente, o usuário pode escrever uma operação matemática mais complexa para expressar o atributo-nome do ícone. Nesta situação, o ícone funciona como um encapsulamento de uma rede de fluxo de dados.

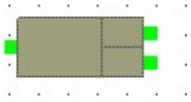
Em cada uma das situações descritas acima, o APP impõe algumas restrições aos parâmetros de entrada e de saída. Quando os parâmetros de entrada e de saída já são conhecidos (funções e operadores préexistentes), são oferecidas as respectivas portas de entrada e de saída, que não podem ser removidas, mas apenas rotuladas e reposicionadas. Quando o ícone representa uma operação matemática mais complexa, ela somente pode ser estruturada após a definição dos parâmetros de entrada, que representam operandos da expressão desejada.

Por fim, quando o ícone representa uma chamada a um procedimento, a ser implementado ou já implementado pelo usuário, os parâmetros de entrada ou de saída podem ser definidos, teclando-se duas vezes sobre as respectivas barras de entrada e de saída do procedimento. Conseqüentemente, novas portas de dados são adicionadas ao ícone. O APP mapeia automaticamente cada nova porta criada no procedimento representado pelo ícone. Se novos parâmetros são inseridos no procedimento em questão, e não na sua chamada, o mesmo processo é estabelecido. Desta forma, a inserção e a remoção de portas de entrada e de saída são fortemente acopladas aos ícones e aos procedimentos que eles representam.

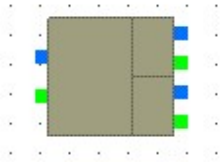


Inserindo um Operador Condicional

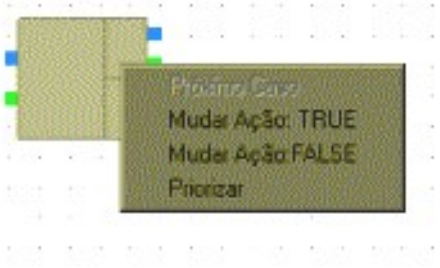
Este ícone representa o operador condicional do APP para controle do fluxo de dados. Uma vez inserido no programa, faz-se necessário definir seus pontos de entrada. Para cada nova inserção de um ponto de entrada, cria-se automaticamente um ponto de saída nos compartimentos que representam a parte verdadeira e falsa do operador. A expressão que representa a condição é construída com as cores ou nomes dos parâmetros de entrada. A Figura 5.4 mostra uma seqüência de ações para a definição de uma determinada configuração de um operador condicional: com duas portas de entrada e duas portas de saída em sua parte verdadeira. Cabe observar a parada obrigatória do programa na parte falsa do construtor.



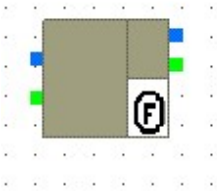
Inserção do ícone do operador
de decisão.



Inserção de mais um ponto de
de entrada, teclando-se duas vezes na
borda de entrada do operador



Mudança da ação relacionada à parte falsa,
ativando a caixa de opções com o botão
direito do *mouse* e selecionando
a opção desejada (Mudar a ação: FALSE).



Até que a ação desejada
apareça na tela.

Figura 5.4: Seqüência de ações para a definição de um operador condicional.

Inserindo um Operador de Repetição

Este ícone representa o operador de repetição do APP. Este operador encapsula a invocação de uma função até que uma condição de parada seja satisfeita. Este operador pode desempenhar dois papéis, funcionalmente equivalentes aos operadores "*while*" e *"for".* Sua nomeação determina o papel por ele desempenhado.

Caso o usuário apenas rotule o ícone com um nome, ele desempenha o papel do operador "*while".* Sendo assim, qualquer porta de entrada pode receber um fluxo de retroalimentação. Caso o usuário determine o número de iterações executadas, utilizando a notação: Nome: Número Inicial .. Número Final, ele funciona como o operador "*for".* Neste caso, não se permite a retroalimentação das portas de entrada. Em contrapartida, o APP insere automaticamente uma

nova porta de entrada no procedimento que implementa o corpo da iteração. Esta porta possui o valor da "variável" interna de controle da iteração. A Figura 5.5 ilustra os dois casos relatados.

Dentro do círculo (A) encontra-se um exemplo de iteração do tipo "*while*", juntamente com a implementação da rede de fluxo de dados para o seu corpo. Observa-se uma porta de entrada com dados de retroalimentação. Por sua vez, dentro do círculo (B), tem-se uma implementação de uma iteração do tipo "*for*". Nota-se, neste caso, a existência de uma porta de entrada no corpo da iteração, ícone representado por uma seta. Esta porta não é representada na chamada da iteração, apenas em seu corpo. Ela armazena o valor da variável de incremento da iteração.

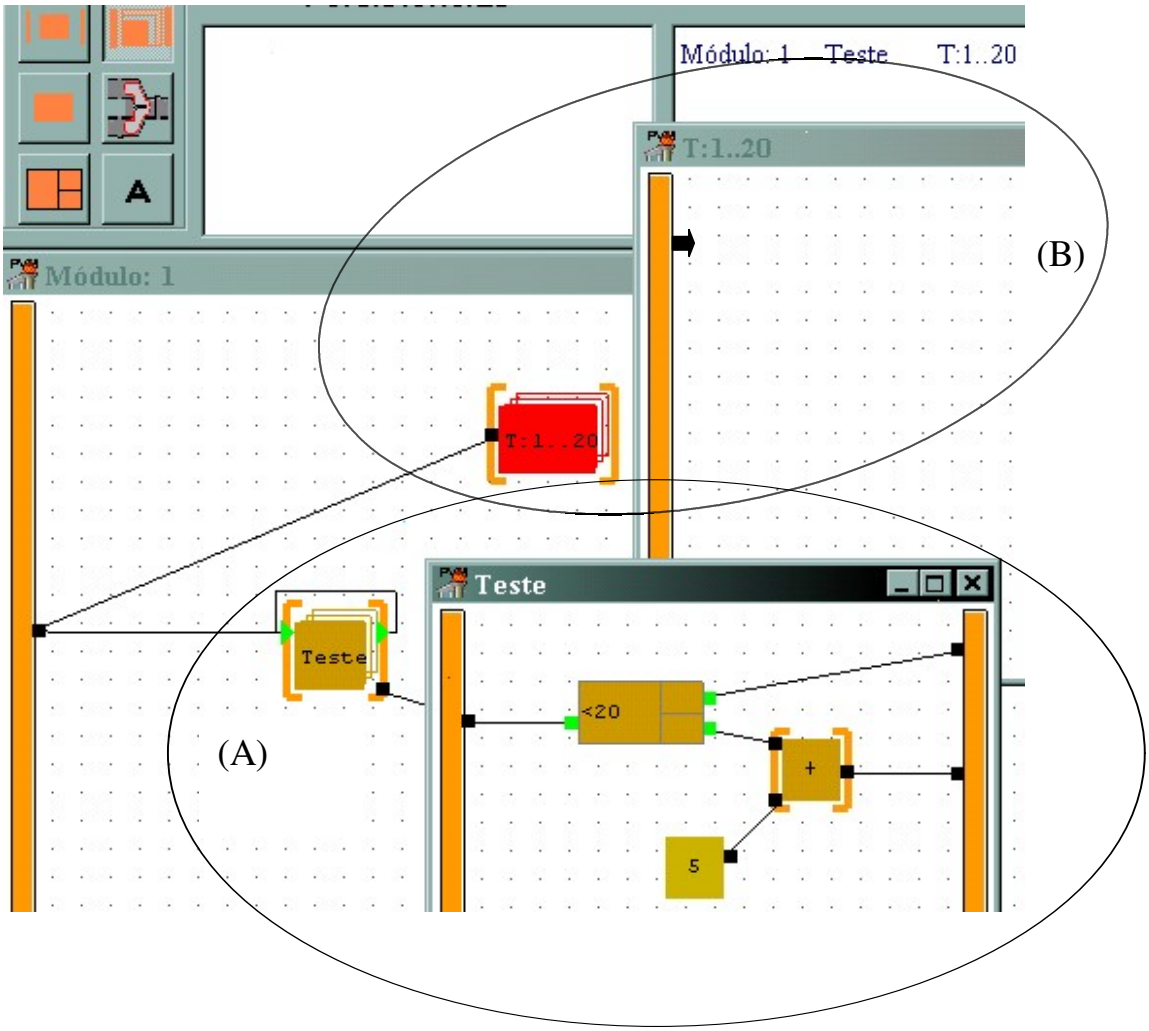
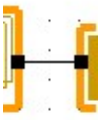


Figura 5.5: As duas funções desempenhadas pelo operador de repetição.

Inserindo um Operador de Fusão

Este ícone representa um operador de fusão para agregar mais de um fluxo de dados à entrada e apenas um fluxo à saída do operador. Este operador permite N portas de entrada e apenas uma porta de saída. O fluxo que parte da porta de saída é o primeiro a chegar na entrada do operador. O usuário pode inserir quantas portas de entrada desejar, teclando duas vezes em algum ponto da fronteira de entrada do ícone.



Criando Conexões

Uma conexão é representada por meio de uma linha de fluxo não pontilhada. Esta linha conecta uma porta de saída a uma porta de entrada de dois operadores ou matrizes. Não se pode conectar a porta de saída de um operador a uma porta de entrada do mesmo operador (exceto no caso do operador de iteração). Ademais, uma porta de entrada recebe apenas uma linha de fluxo, enquanto a porta de saída pode enviar mais do que uma linha de fluxo para outros processos.

Para inserção de uma linha de fluxo no programa, o usuário necessita apenas colocar e teclar o primeiro botão do *mouse* em uma das portas (entrada ou saída) e, segurando o primeiro botão, puxar a linha até a porta desejada. A conexão é estabelecida quando o usuário soltar o botão do *mouse*. Uma vez estabelecida a conexão, a porta pode ser removida ou movimentada. Caso uma porta seja eliminada, a linha de conexão é suprimida.

Duas portas também são conectadas para determinar o sincronismo entre dois processos. Neste caso, um determinado processo somente pode começar após o término de um outro. A forma de implementação de uma linha de sincronismo é praticamente a mesma adotada para a conexão de uma linha de fluxo, bastando alterar o tipo de uma das portas de entrada ou de saída após o estabelecimento da conexão da linha de fluxo. Neste caso, o usuário deve escolher uma das duas portas e teclar o segundo botão do *mouse* para ativar as opções disponíveis para a porta escolhida, selecionando, a seguir, a opção de sincronismo.

A Figura 5.6 ilustra a determinação de um sincronismo entre os processos A e B. Na parte (I), tem-se uma conexão normal e a ativação do *menu* de opções de uma das portas. Após a seleção da opção de sincronismo, obtém-se o resultado retratado em (II).

(I)

(II)

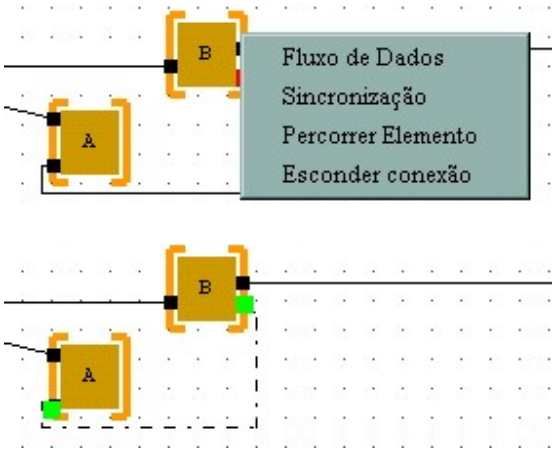


Figura 5.6: Imposição de um sincronismo entre os processos A e B.

O APP admite ainda o estabelecimento de um rótulo ou de uma cor para as portas de entrada ou de saída, bastando selecionar a porta desejada e teclar *enter.* Neste caso, uma pequena janela se abre, permitindo a definição do rótulo ou de uma cor.

Estabelecendo Sincronismos

O APP permite o estabelecimento de sincronismos em um bloco que contenha uma ou mais matrizes, de maneira que, a cada ciclo de execução, um dado de cada matriz seja necessariamente enviado para o processo subseqüente. O usuário seleciona, via *mouse,* as matrizes que serão sincronizadas e escolhe a opção de sincronismo do menu "projeto", segundo a Figura 5.7.

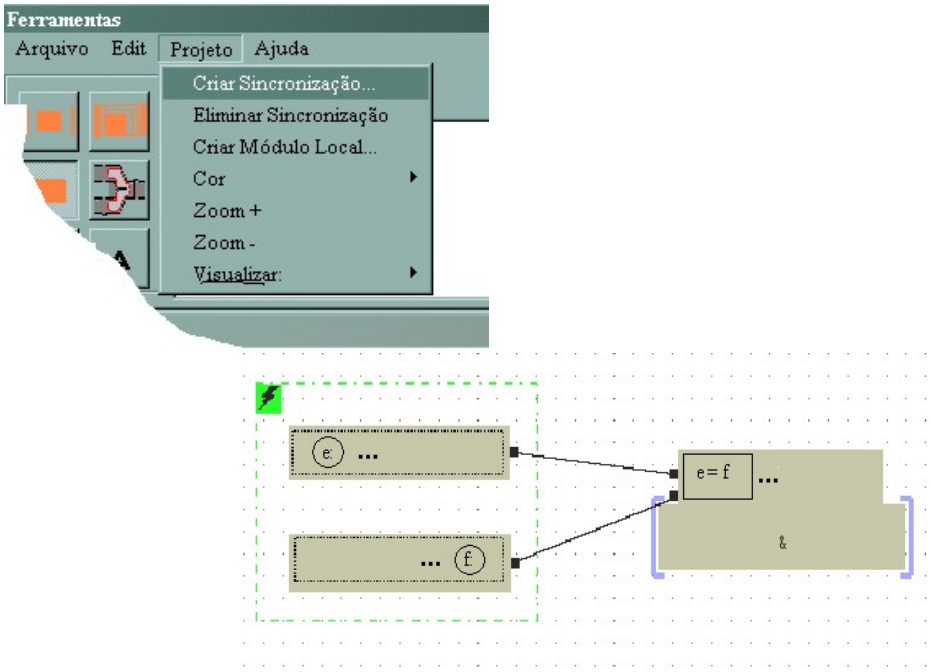
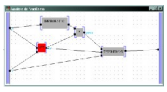


Figura 5.7: Exemplo de sincronismo em um bloco de duas matrizes.



Criando um Módulo Local

O editor de programa do APP permite o encapsulamento de um conjunto de nós em um único nó. Este nó possui as mesmas características de uma função ou de um procedimento, mas sendo visível apenas localmente. Ademais, ele não pode ser invocado a partir de outros módulos. As Figuras 5.8 e 5.9 retratam, respectivamente, a criação de um módulo local e o resultado do encapsulamento de um conjunto de nós no APP.

Em primeiro lugar, o usuário delimita a área de encapsulamento, definindo um retângulo que envolva todos os nós desejados. Em segundo lugar, ele ativa o menu "projeto" e a opção “Criar Módulo Local...”. Por último, ele define um nome para o encapsulamento.

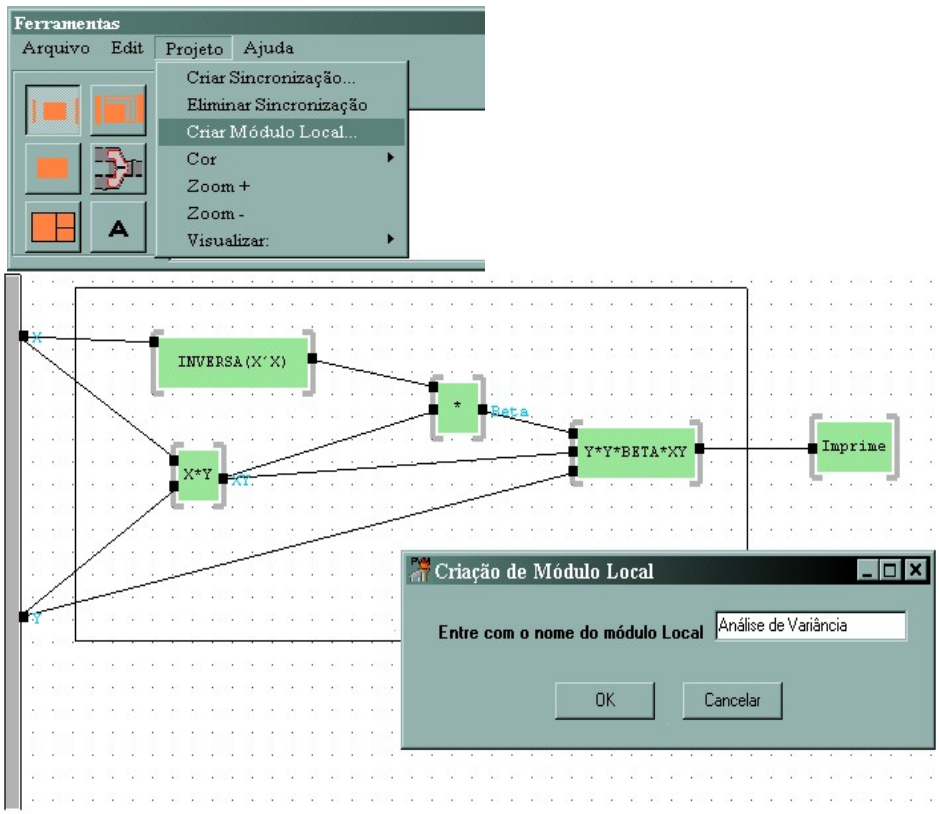


Figura 5.8: Criando um módulo local no APP.

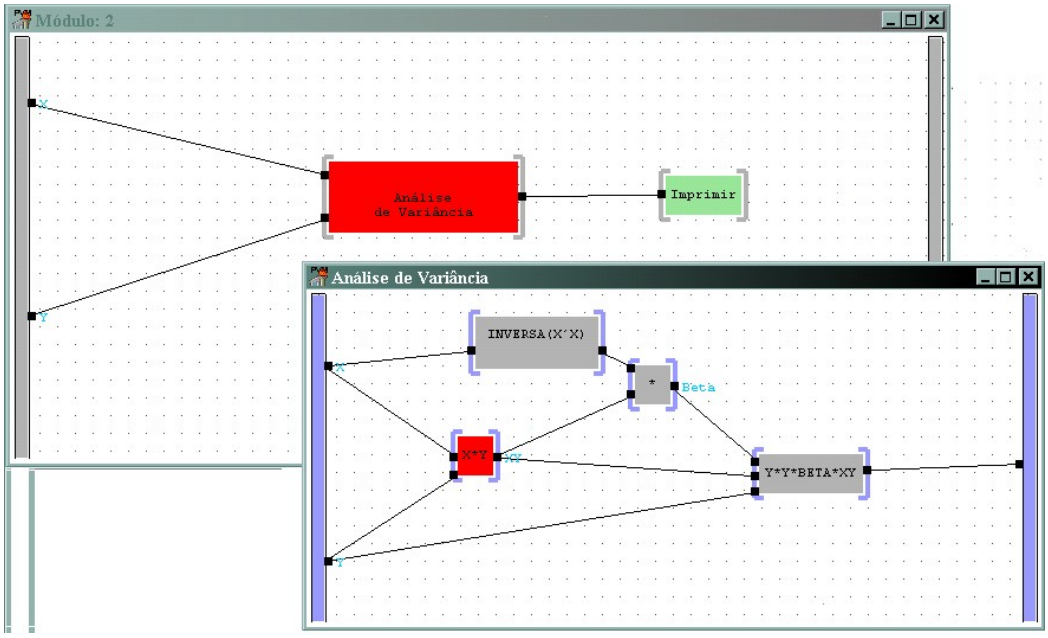


Figura 5.9: Resultado do encapsulamento de um conjunto de nós no APP.

5.2.2 O Editor de Matrizes

O editor de matrizes destina-se à manipulação de matrizes via operadores específicos. Este editor é visualmente dividido em duas partes. A primeira parte atua sobre os dados de entrada, seja pela composição de outras planilhas, seja pela inserção de dados ou fórmulas nas células da planilha. A segunda parte permite a criação de uma visão customizada dos dados da planilha. Esta visão pode ser de toda a matriz, um conjunto de submatrizes ou um escalar, dentro de um processo iterativo ou não. A Figura 5.10 retrata a interface do editor visual de matrizes.

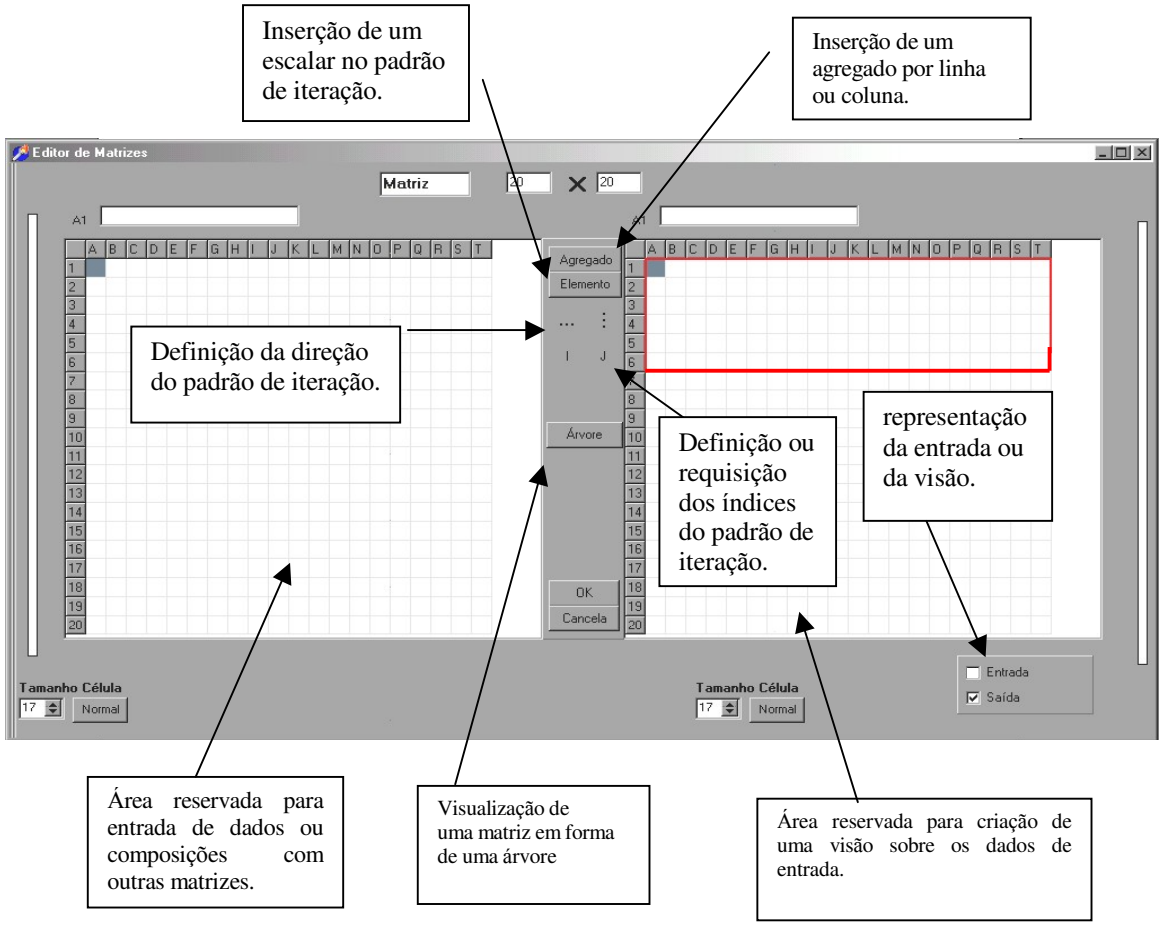


Figura 5.10: Interface do editor de matrizes e suas principais funcionalidades.

A representação interna de cada matriz é uma árvore binária. Cada nó/folha da árvore é uma partição com os seguintes elementos: identificador da região, tipo da partição (Vertical ou Horizontal), Linha Inicial (LI), Linha Final (LF), Coluna Inicial (CI), Coluna Final (CF), Número de Linhas (NL) e Número de Colunas (NC). A Figura 5.11 retrata, esquematicamente, a partição de uma matriz e a árvore correspondente. O rótulo V indica que os dois nós gerados representam partições verticais. O rótulo H indica que os nós descendentes são partições horizontais.

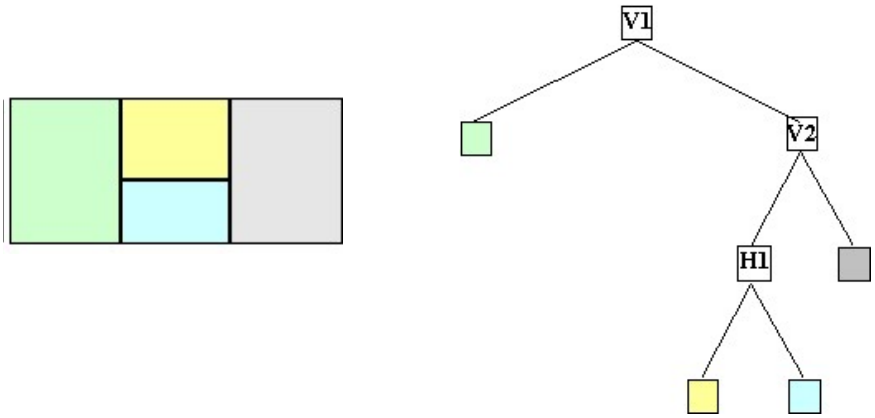


Figura 5.11: Representação interna de uma matriz.

Os atributos de cada nó/folha também são utilizados para validar a conformidade dimensional da operação desejada. Por exemplo, a partição horizontal H1, que gerou os nós/folhas amarelo e azul, torna-se válida se, e somente se, amarelo.CI = azul.CI e amarelo.CF = azul.CF. Nem sempre é possível a verificação das dimensões em tempo real, pois as dimensões dos parâmetros de entrada nem sempre se encontram disponíveis.

A composição da dimensão final da matriz gerada depende do tipo de partição de cada nó da árvore. Caso a partição seja horizontal, deve-se somar o número de linhas e manter o número de colunas. Caso a partição seja vertical, o valor constante deve ser o número de linhas.

5.3 A Máquina de Execução do APP

A máquina de execução do APP destina-se a executar um programa produzido com o editor visual. Esta execução se inicia pelo envio de instruções e dados para a máquina de execução, que simula o comportamento de um computador baseado em fluxo de dados.

Cabe ressaltar alguns aspectos fundamentais da dinâmica de execução do APP: as regras de dependência entre os dados, o comportamento das portas de entrada e de saída e as planilhas.

As regras de dependência entre os dados possibilitam uma sintonia fina do grau desejado de paralelismo da execução e explicitam o inter-relacionamento dos dados. Um nó está pronto para ser executado quando todos os seus terminais de entrada contêm dados. A ordem de execução é ditada pela dependência entre os dados. Se vários nós estão prontos para execução, eles podem ser executados em qualquer ordem. Em uma máquina com mais de um processador, estes nós poderiam ser executados paralelamente. A execução de um subdiagrama também não pode começar até que todos os dados de entrada estejam disponíveis. Ademais, um subdiagrama é executado separadamente do diagrama a que pertence.

O comportamento das portas de entrada e de saída do APP merece alguns comentários. Para programadores de linguagens imperativas, o comportamento destas portas lembra o funcionamento dos comandos "*Read" e "Print",* no sentido de que os valores de entrada e de saída são consumidos e fornecidos pelos nós, ocorrendo em períodos de tempo discretos. Uma determinada porta de entrada pode receber dados de apenas um único nó, enquanto a porta de saída pode enviar um resultado para mais de uma porta de entrada.

As planilhas podem embutir um padrão de iteração na geração dos dados, em sua entrada e/ou em sua saída. O padrão de iteração da saída só começa a ser executado após a criação da matriz. O dado enviado por uma planilha é uma matriz, seja ela de dimensão unitária ou não. Em outras palavras, o dado que corre na rede de fluxo de dados do APP é sempre uma matriz. A planilha envia pelo menos uma matriz a cada ciclo do padrão de iteração da saída. O fim da execução do padrão de iteração da planilha é anunciado pelo envio de uma matriz nula.

No APP, a semântica de execução de um programa e de seus nós pode ser descrita da seguinte forma:

Execução de um Programa

- invocação do programa: um programa pode ser invocado explicitamente ou chamado como um subprograma;
- início da execução do programa: a execução de um programa pode ser iniciada quando cada terminal de saída propagar uma cópia da matriz na linha de conexão a ele atrelada;
- fim da execução do programa: a execução de um programa termina quando o último nó for executado;

Execução de um Nó do Programa

- condição para a execução do nó: um nó pode ser executado quando todos os dados de entrada forem colocados nas respectivas portas de entrada;
- execução do nó, que representa:
  - um subprograma: quando um nó que representa um subprograma é escolhido para ser executado, todos os subnós de seu grafo são adicionados a um conjunto de nós à espera de sua execução. Todos os nós do subprograma devem ser executados antes de sua finalização;
  - uma iteração: quando o nó que representa uma iteração é escolhido para ser executado, a matriz inicial da porta de retroalimentação é recebida pela conexão externa do construtor. Durante sua execução, alguma matriz pode ser enviada para a porta de saída que alimenta a entrada do mesmo construtor. Entre as iterações do construtor, a matriz enviada para a porta de saída é disponibilizada na porta de entrada para o início do próximo ciclo.
  - fim da execução do nó: quando a execução do nó termina, os dados de saída são propagados para terminais de saída e para as correspondentes linhas de conexão;

5.3.1 Componentes da Máquina de Execução

A máquina de execução do APP consiste de uma unidade de memória, de uma unidade de processamento e de uma unidade de procura e de associação de dados a uma instrução. A unidade de memória armazena as instruções e os dados que passam pela rede de fluxo de dados. A unidade de processamento recebe e executa uma instrução, utilizando seus próprios operandos. Após a execução, o resultado é enviado para a unidade de procura e de associação. Esta unidade verifica se existe alguma operação que possa ser executada pelo dado que acabou de chegar. Caso afirmativo, a instrução e seus dados são retirados da memória e enviados para o processamento. Caso contrário, o dado em questão é armazenado na memória.

As subseções seguintes descrevem os *tokens* e as instruções, principais componentes da máquina de execução.

Tokens

Tokens representam dados que trafegam pela rede de fluxo de dados da máquina de execução do APP. Mais especificamente, eles representam resultados de operações executadas. Um *token* é uma unidade independente, ou seja, que não é atrelada a qualquer endereçamento de memória. Além disso, ele não faz qualquer referência a outros *tokens*.

Um *token* contém três tipos de informações: 1) um nome ou uma cor de identificação; 2) uma matriz; e 3) um rótulo de controle do contexto de execução.

O nome ou a cor de identificação do *token* são especificados no momento de rotulagem de uma porta de saída, sendo utilizados para identificação do *token* em relação aos outros operandos. A matriz encapsulada no *token* carrega, além de seus dados, um nome ou uma cor, sua dimensão e valores agregados, como, por exemplo, os índices de origem de um escalar.

Por sua vez, o rótulo de controle possui campos para descrever seu escopo e seu endereço de origem. O escopo de um rótulo lhe é atribuído automaticamente pela máquina de execução, visando descrever o escopo de execução do *token.* Este escopo depende não somente do nó em

execução, mas também, se for o caso, dos ciclos de uma iteração ou de uma recursão. Ademais, o endereço de origem do rótulo indica o nó e a porta de origem do dado.

Instruções

Uma instrução é formada por um conjunto de valores, que descrevem, respectivamente, a operação, seus operandos e seus resultados. A Figura 5.12 ilustra o formato típico de uma instrução que encapsula uma operação do APP.

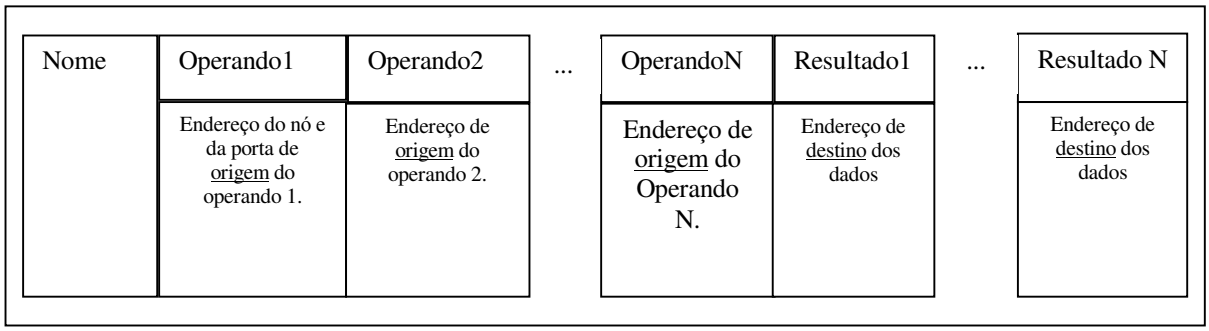


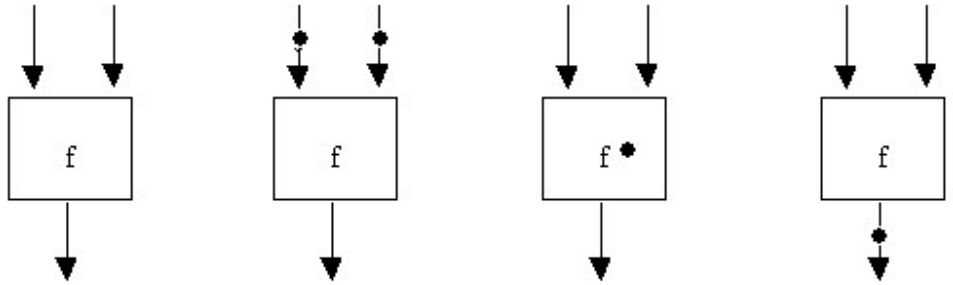
Figura 5.12: Formato de uma instrução em APP.

Esta instrução representa um nó do grafo e pode ser descrita simbolicamente como uma aplicação de uma função a seus operandos, isto é, Resultados = Função(Operandos).

Uma instrução pode assumir os seguintes estados:

- Esperando: quando pelo menos um dos *tokens* de entrada ainda não está disponível;
- Pronto: quando todos os *tokens* de entrada estão disponíveis para execução;
- Executando: quando a operação está em execução; e
- Enviando: quando a instrução foi executada e um *token* de saída é enviado para o próximo operador.

A Figura 5.13 ilustra os possíveis estados de uma instrução na máquina de execução do APP.



Esperando Pronto Executando Enviando

Figura 5.13: Ilustração dos possíveis estados de uma instrução na máquina de execução do APP.

O APP não permite que uma instrução passe do estado "pronto" para o estado "executando" enquanto existirem *tokens* em sua porta de saída. Isto significa a ausência de qualquer fila nas portas de saída. A não implementação de uma fila na porta de saída simplifica a implementação da máquina de execução.

5.4 Verificação do ProVisual

A implementação do modelo proposto pelo autor (ProVisual) pode ser verificado a partir de dois exemplos. O primeiro implementa o cálculo do determinante de uma matriz. Este exemplo escolhido pode ser justificado pelo alto grau de complexidade da implementação do cálculo determinante em linguagens textuais. O segundo exemplo implementa o cálculo para a determinação dos parâmetros de uma regressão linear múltipla.

Cálculo determinante de uma matriz:

A solução foi dividida em três diferentes casos de execução, representados nas Figuras 5.14 e 5.15.

O primeiro caso (Figura 5.14) refere-se a uma matriz de entrada com dimensão (coluna ou linha) maior do que 1. Nesta situação, a matriz é subdividida com a utilização do recurso de partição de matrizes e de padrão de iteração. Cada submatriz é obtida pela eliminação da primeira linha e da coluna correspondente à posição do elemento "e:" no processo iterativo. As submatrizes M1 e M2 são enviadas para o próximo nó do diagrama. Este nó deve concatenar as submatrizes e enviá-las para o nó de geração de elementos de um vetor. Os elementos desta nova matriz ("M") são computados a partir da fórmula ali especificada, que contém uma chamada recursiva para a função determinante. Esta função determinante recebe uma matriz formada pela concatenação horizontal das submatrizes M1 e M2. O resultado final será obtido pela soma de todos os elementos contidos na matriz "M".

O segundo caso aplica-se a uma matriz unitária e retorna o valor da única célula da matriz de entrada. Finalmente, o terceiro caso apresenta como resultado o valor 1 para uma matriz de dimensão zero (Figura 5.15).

As matrizes que simbolizam o padrão de iteração e a concatenação horizontal merecem alguns comentários. No ambiente de programação proposto pelo autor (APP), uma matriz é

representada no editor de programas através de um ícone que possui as mesmas cores, regiões e nomes a ela atribuídos no editor de matrizes. O usuário deve escolher, contudo, se deseja a representação da parte destinada à entrada ou à visão de uma planilha. Na Figura 5.14, a matriz que simboliza o padrão de iteração ilustra a parte da planilha reservada à visão. Na mesma figura, a matriz que representa a concatenação horizontal ilustra a parte reservada à entrada.

A Figura 5.16 mostra a especificação da concatenação horizontal das submatrizes M1 e M2 da Figura 5.14 no editor de matrizes, que produz uma matriz de saída identificada com a cor amarela.

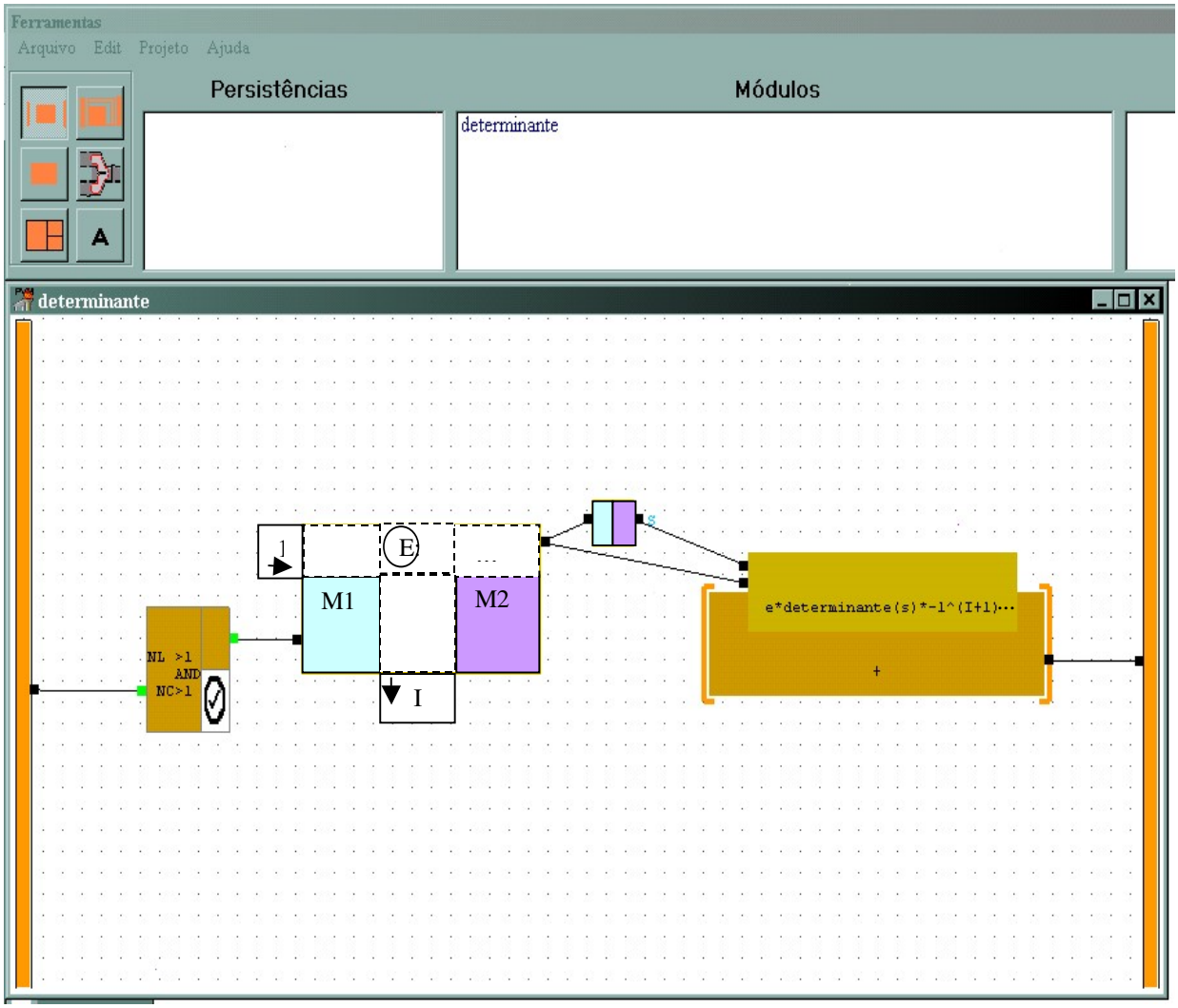


Figura 5.14: Cálculo do determinante de uma matriz com dimensão maior do que 1.

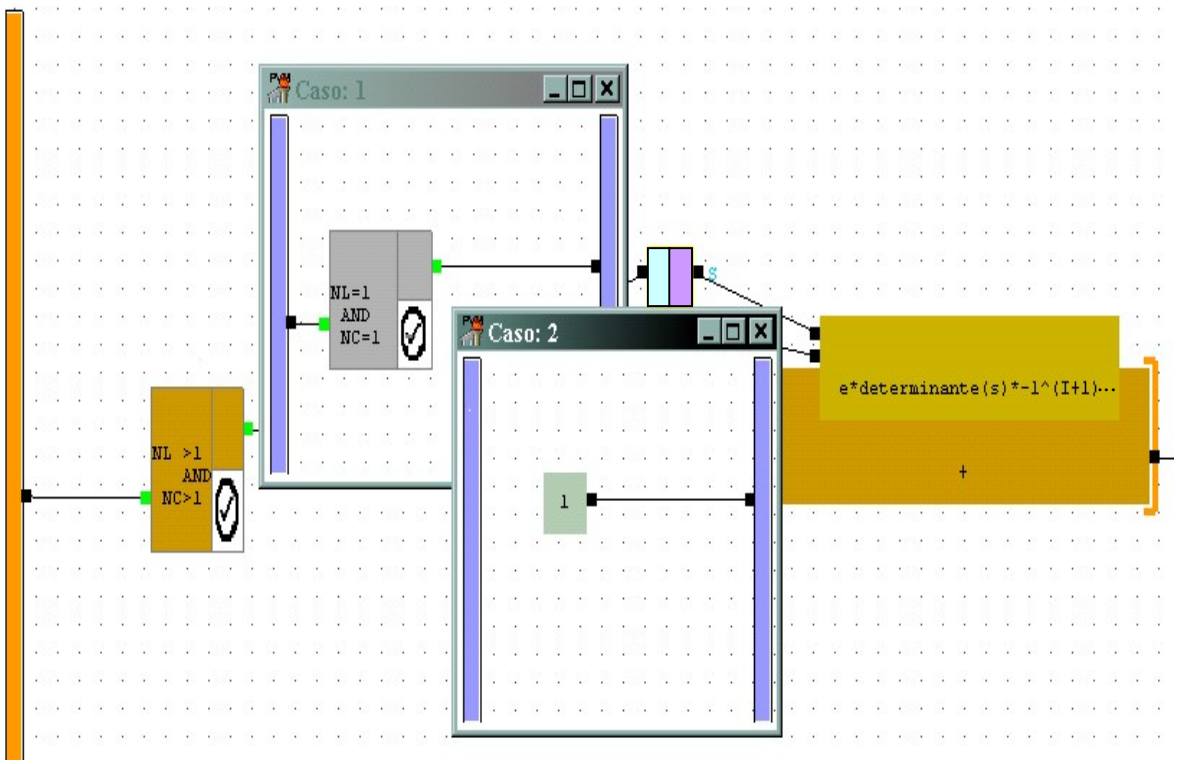


Figura 5.15: Cálculo do determinante de uma matriz unitária e nula.

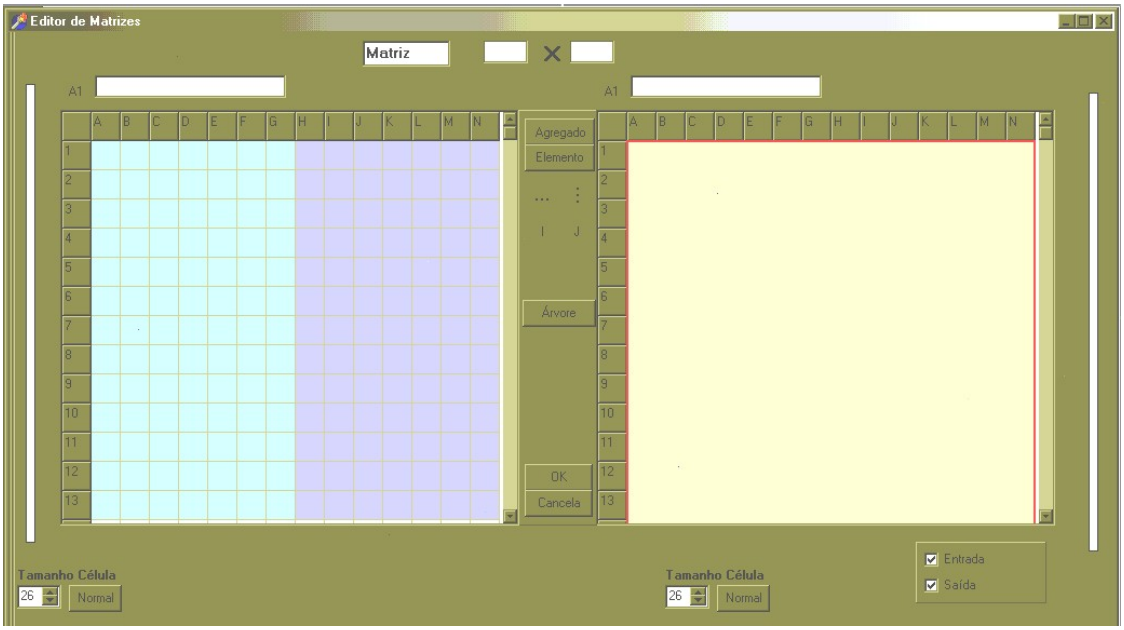
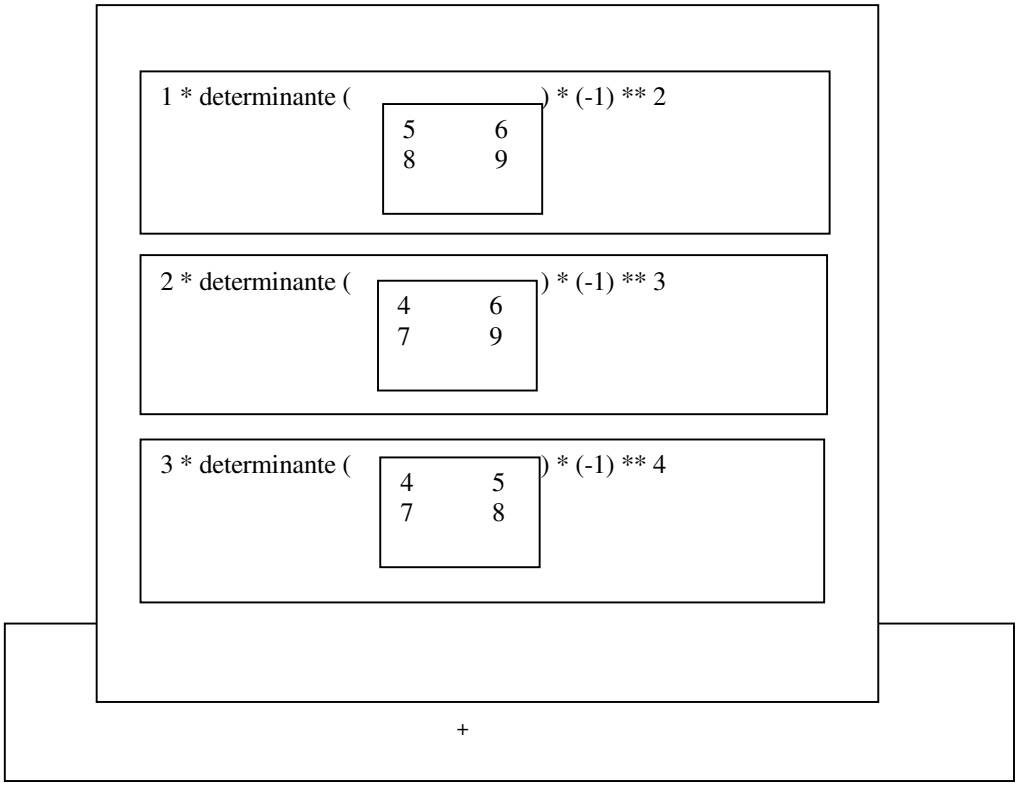


Figura 5.16: Especificação da concatenação horizontal das submatrizes M1 e M2 da Figura 5.14.

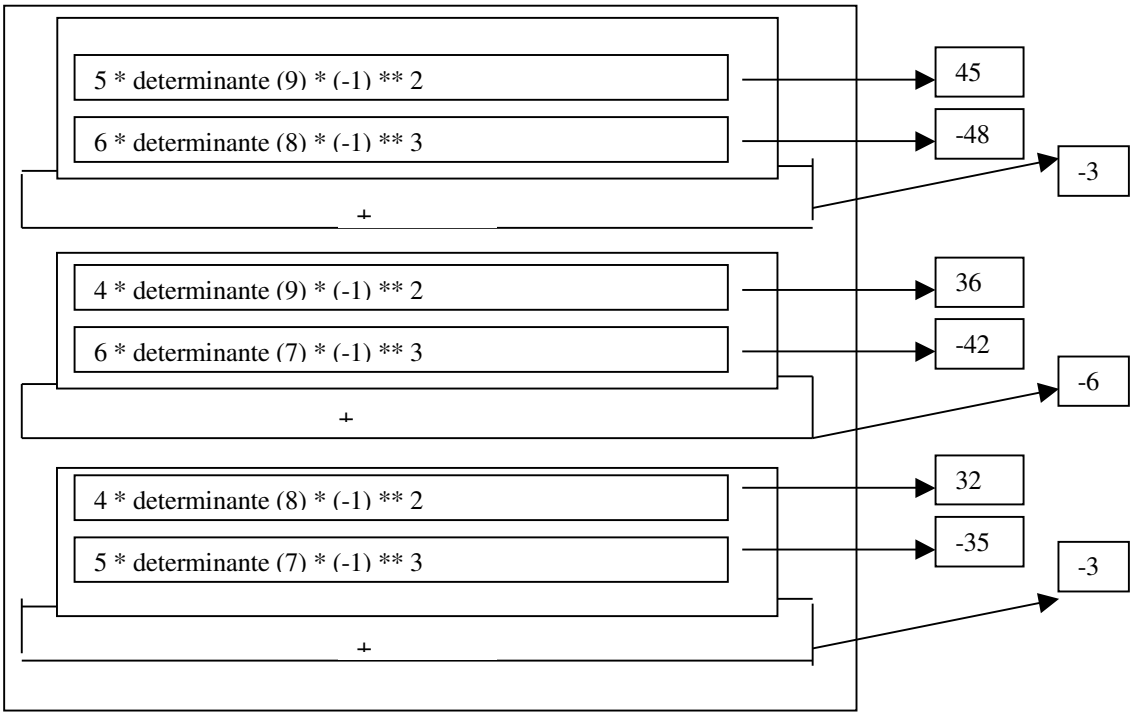
Cabe considerar a execução do programa da Figura 5.14 para a seguinte matriz, com três colunas e três linhas:

1	2	3
4	5	6
7	8	9

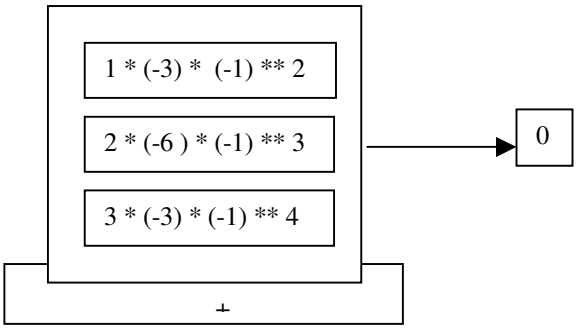
O primeiro caso de execução é ativado, gerando, inicialmente, três computações independentes:



A chamada recursiva do determinante, das três computações anteriores, gera seis expressões que devem ser executadas.



Os resultados obtidos retornam aos respectivos locais das chamadas recursivas do primeiro ciclo da computação, produzindo o seguinte resultado:



Após a execução e a soma das três expressões, chega-se ao resultado do determinante, representado por um escalar de valor zero.

O ponto mais importante deste exemplo é a segmentação da matriz em diversas submatrizes dentro de um processo iterativo. Esta segmentação permite que o programa resultante seja compacto e expressivo, pois a estratégia adotada para a resolução do determinante está bem definida no padrão de iteração. Portanto, a utilização da planilha como um construtor de iteração foi fundamental para este resultado.

Cálculo para determinação dos parâmetros de uma regressão linear múltipla:

A solução deste problema está representada na Figura 5.17. Neste exemplo, é verificado se uma matriz de entrada possui número de colunas maior do que 1 e o número de linhas maior que o número de colunas. Nesta situação, a matriz é subdividida com a utilização do recurso de partição de matrizes. Cada submatriz obtida é rotulada para que possa ser referenciada por outros processos. A variável *ind* indica qual é a coluna que armazena a variável dependente do modelo linear (Y). As submatrizes obtidas (M1, M2 e Y) são processadas e os valores dos parâmetros estimados são retornados pelo módulo em questão.

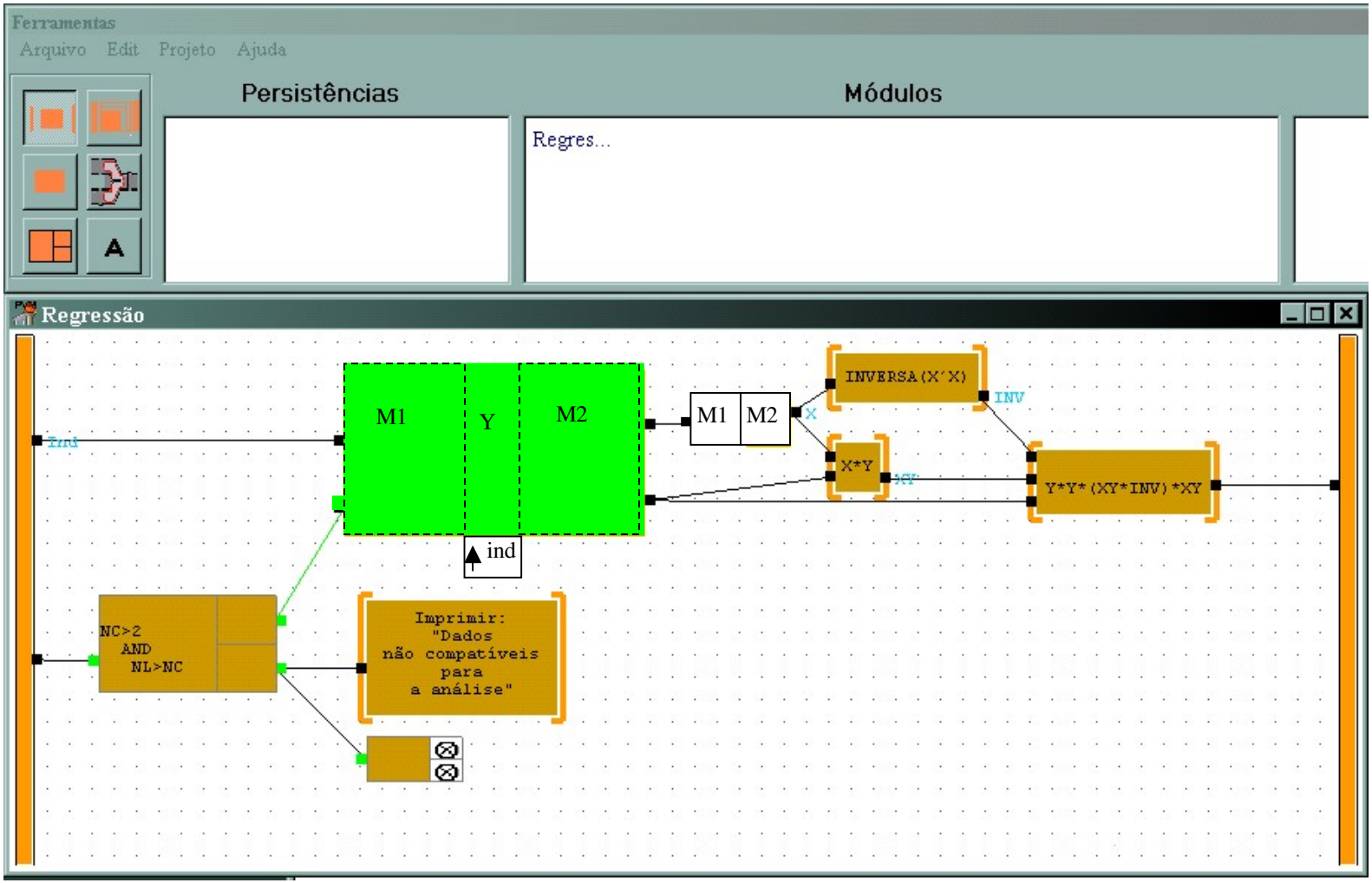


Figura 5.17: Determinação dos parâmetros de uma regressão linear múltipla

5.5 Conclusão

O autor apresentou a arquitetura da implementação do ProVisual, mostrando que o ambiente de programação ProVisual (APP) é um único programa formado por dois editores visuais e uma máquina de execução de fluxo de dados.

Ele descreveu os editores visuais de programas e de matrizes para ressaltar a forma de interação entre o usuário e o APP.

A exposição da máquina de execução do ambiente permitiu que o autor destacasse os principais elementos que controlam e viabilizam a execução de um programa na máquina virtual de fluxo de dados do APP.

O exemplo utilizado para verificar o modelo proposto revelou possuir um alto nível de abstração e uma estrutura visual próxima à estratégia adotada para o algoritmo implementado. Estas duas características são fundamentais para os propósitos de aplicação do ProVisual.

Capítulo 6
Conclusões

Neste capítulo, o autor apresenta as conclusões e as contribuições deste trabalho de pesquisa, bem como sugere trabalhos futuros.

6.1 Conclusões

Durante mais de dez anos de experiência profissional com o desenvolvimento de software científico na Embrapa Informática Agropecuária, o autor deste trabalho de pesquisa constatou a existência de importantes dificuldades, apresentadas pelos desenvolvedores de software, de interpretação de requisitos recebidos dos pesquisadores e especialistas dos domínios de aplicação. Em adição, o autor também observou que estes pesquisadores não possuíam o conhecimento de linguagens de programação que lhes permitisse o desenvolvimento de módulos estatísticos requeridos pela instituição.

O desenvolvimento da programação textual vem sendo influenciado, desde as primeiras linguagens, pela arquitetura do hardware, dificultando sua utilização por usuários que não pertençam a área da computação.

A tarefa de desenvolvimento de software tornou-se mais amigável com o advento da programação visual, pois as representações visuais encontram-se mais próximas das soluções dos problemas dos usuários, quando comparadas às representações textuais.

O problema considerado neste trabalho de pesquisa foi o de aumentar a eficiência de implementação de modelos estatísticos e matemáticos por pesquisadores com pouca experiência em programação.

A solução escolhida para resolver com sucesso o problema acima foi a criação de uma arquitetura visual de desenvolvimento de software para a modelagem de métodos estatísticos e matemáticos que permita aumentar a eficiência da implementação de algoritmos em computador por pesquisadores com pouca experiência em programação.

Na investigação realizada sobre linguagens visuais, constatou-se sua inadequação para soluções genéricas. Todavia, relatos recentes mostram que linguagens visuais, quando bem projetadas, possuem inúmeras vantagens sobre as textuais, tais como: acesso aleatório à informação, maior capacidade de transferência de informação, expressividade da linguagem e descrição de um problema de forma concreta ou abstrata. As linguagens

visuais relacionadas a domínios específicos tiveram maior aceitação por parte dos usuários finais.

Este trabalho de pesquisa propôs e desenvolveu um Modelo para Programação Visual de Matrizes – ProVisual, baseado nos paradigmas de planilhas eletrônicas e de fluxo de dados.

A planilha eletrônica foi utilizada por permitir sua adaptação para representar e manipular uma matriz sem a necessidade de recorrer a uma notação textual. O autor considera que a planilha adotada no ProVisual é uma adequada representação visual de uma matriz, que tende a facilitar seu entendimento e sua manipulação mesmo por usuários não treinados em programação.

A planilha do ProViSual possui um conjunto de características que a torna crucial para o modelo. Neste sentido, a planilha permite a criação de visões sobre uma matriz a partir de partições e de padrões de iteração. A partição de uma matriz associada ao padrão de iteração possibilita a utilização de uma planilha como um robusto construtor de iteração. Desta forma, a planilha pode representar um processo iterativo complexo em um nível mais alto de abstração, facilitando a implementação e o entendimento de algoritmos matriciais.

A técnica de programação por exemplos, utilizada na planilha eletrônica do ProVisual, facilita a tarefa de implementação de algoritmos matriciais sem a necessidade, em muitos casos, de se recorrer a construtores usuais de uma linguagem de programação. Esta técnica é uma generalização do processo de cópia de fórmulas em planilhas, que inclui o controle de parada do processo de indução das fórmulas e uma estratégia para a geração aninhada de fórmulas. Isto permite a construção de algoritmos mais complexos, que incorporam processos iterativos em seu comportamento.

O ProVisual incorpora a técnica de visualização transitória dos fluxos de dados entre as fórmulas de uma planilha, permitindo a diminuição de possíveis efeitos colaterais decorrentes da existência de relações escondidas entre as fórmulas. Esta técnica mostra os fluxos de dados tanto entre as células que afetam a fórmula quanto entre as células que são por ela afetadas.

A linguagem de composição de planilhas é baseada em fluxo de dados. O fluxo de dados foi adotado por explicitar as dependências entre os diversos dados (matrizes) de um programa e por ser adequado a aplicações do domínio da matemática aplicada [Evangelista, 2001 e 2001a]. Em adição, a dinâmica de funcionamento do fluxo de dados permite a execução paralela de um programa. Apesar desta execução paralela não ser o objetivo principal do trabalho, esta possibilidade pode ser explorada em situações que exijam um grande volume de cálculos matemáticos.

A linguagem de composição do ProVisual fornece suporte gráfico para os construtores de expressão simples, repetição, decisão, fusão e documentação. Ademais, permite o controle da sincronização de processos e viabiliza a abstração funcional a partir da definição de procedimentos (globais e locais) e de casos de execução.

A linguagem de composição admite três formas para compor dois pictogramas: interseção, composição e conexão. As diferentes formas de composição, associadas ao conceito de abstração funcional, fornecem meios para tornar um programa ProVisual mais conciso.

O ProVisual foi descrito formalmente segundo a lógica espacial entre regiões, pois a relação topológica mais importante para este trabalho de pesquisa é a conectividade. Uma das vantagens deste tipo de abordagem para a especificação de uma linguagem visual é a utilização do mesmo formalismo para a sintaxe e a semântica dos diagramas da linguagem.

O autor demonstrou também a viabilidade do modelo proposto com o desenvolvimento de um protótipo que implementa os conceitos utilizados na definição do ProVisual.

O autor deste trabalho de pesquisa acredita que o ProVisual representa uma solução inovadora para o problema de desenvolvimento de softwares baseados em matrizes por usuários com pouca experiência em programação.

6.2 Contribuições

As principais contribuições deste trabalho de pesquisa foram classificadas como: contribuições gerais e contribuições específicas.

6.2.1 Contribuições Gerais

As principais contribuições gerais deste trabalho de pesquisa são:

Modelagem - O ProVisual pode propiciar a modelagem de um software estatístico em diferentes graus de abstração, principalmente porque permite a aproximação da representação visual de um problema à forma de obtenção de sua solução. Desta forma, ele pode ser utilizado para diminuir dificuldades de conexão entre as exigências dos domínios das aplicações, as necessidades dos pesquisadores e a forma de desenvolvimento dos softwares estatísticos;

Prototipação – O ProVisual permite o mapeamento mais efetivo entre os requisitos do algoritmo e a implementação. Neste sentido, ele pode ser usado para a prototipação rápida de módulos estatísticos, principalmente se os recursos de abstração e de programação por demosntração forem utilizados; e

Aplicabilidade – O ProVisual foi desenvolvido especificamente para o domínio da matemática aplicada. Entretanto, a planilha eletrônica proposta pode ser utilizada para enriquecer a semântica de outras linguagens visuais baseadas em fluxo de dados, pertencentes a outros domínios, pois permite o encapsulamento de comportamentos complexos e viabiliza a manipulação direta de dados tabulares. Assim sendo, esta planilha pode ser utilizada com sucesso em outras linguagens de programação visual, pois as aplicações que se utilizam de dados tabulares extrapolam o domínio de aplicação considerado neste trabalho de pesquisa.

6.2.2 Contribuições Específicas

As principais contribuições específicas deste trabalho de pesquisa são:

- desenvolvimento da primeira linguagem de programação visual híbrida (ProVisual);
- aumento da eficiência na implementação de algoritmos matriciais;

- redução dos problemas relacionados à limitação do espaço físico do monitor; e
- simplificação do entendimento de programas já implementados.

O ProVisual pode ser considerado como a primeira linguagem de programação visual híbrida que se baseia simultaneamente nos paradigmas de fluxo de dados e de planilhas eletrônicas. O fluxo de dados e a planilha são paradigmas utilizados em várias propostas de linguagens visuais; porém, não existem propostas que utilizem simultaneamente os dois paradigmas na mesma linguagem de programação visual.

A técnica de programação por exemplos utilizada na planilha eletrônica do ProVisual permite uma maior eficiência na implementação de algoritmos matriciais expressos por processos indutivos. Isto se deve à simplicidade de implementação de tais processos indutivos no ProVisual, que equivale apenas à definição dos dois primeiros valores de um processo iterativo. Uma planilha assim programada pode ser encapsulada e reutilizada em outros programas.

O ProVisual adota um conjunto de soluções que permite a resolução de programas de maior porte e a redução de problemas relacionados ao espaço físico do monitor. O modelo inclui recursos para a abstração procedimental, para a redução da necessidade de recursões e iterações e para a aplicação de uma operação a todos os elementos de uma matriz. A planilha do modelo ajuda a reduzir problemas de escalabilidade e de limitação do espaço físico do monitor, uma vez que ela, se adequadamente utilizada, pode encapsular um comportamento mais complexo que substitua recursões e iterações, tornando o programa gerado mais conciso e, ao mesmo tempo, mais legível. Ademais, a linguagem de composição do ProVisual admite a composição de interseção de dois pictogramas. A composição por interseção equivale à utilização de um processo iterativo a ser aplicado a cada um dos dados de uma matriz.

A visualização estrutural da interdependência global da planilha eletrônica permite uma espécie de fotografia da estrutura de relações existentes na planilha independente de sua dimensão. Neste caso, células interdependentes são agrupadas e representadas por uma

única cor. Esta técnica simplifica o entendimento da forma de estruturação da planilha por um usuário que não tenha participado de sua montagem original.

6.3 Sugestões para Trabalhos Futuros

Com o intuito de aprimorar o modelo e a implementação do ProVisual, sugerem-se os seguintes trabalhos futuros:

- Aumento do grau de paralelismo do ProVisual - A execução do ProVisual é inerentemente paralela, uma vez que se baseia no fluxo de dados. Todavia, o autor adotou estratégias que reduzem o grau de paralelismo obtido com a aplicação do modelo: a utilização de texto para representar uma expressão matemática mais complexa e a adoção e planilhas para implementação de funções, que visam uma melhor legibilidade e uma maior produtividade no desenvolvimento. Deste modo, o autor sugere a realização de trabalhos de evolução do modelo para permitir a expressão interna da fórmula textual como uma rede de fluxo de dados. Ele sugere ainda a realização de estudos sobre a obtenção de paralelismo em planilhas;
- Evolução da máquina virtual de fluxo de dados - A atual máquina de execução do protótipo do ProVisual simula o comportamento de um computador baseado em fluxo de dados. Entretanto, esta máquina executa os nós de forma seqüencial. O autor sugere o desenvolvimento de trabalhos de evolução da máquina de execução do ProVisual com o objetivo de aproveitar o paralelismo permitido pelo fluxo de dados. Um outro aspecto que merece atenção na evolução da máquina virtual é o tratamento de exceções;
- Exploração de paralelismo na WEB - Uma forma de viabilizar a execução em paralelo é a utilização de computadores conectados à WEB. O autor sugere estudos para dotar o ProVisual de mecanismos que permitam a utilização de processadores distribuídos na WEB. Os estudos não devem se limitar ao desenvolvimento da máquina de execução, mas devem também incluir formas de interação e visualização do fluxo de informações entre vários computadores. A exploração da WEB exige a re-implementação do protótipo do ProVisual na linguagem JAVA,

permitindo que ele possa ser executado em diferentes plataformas de hardware e software. Cabe lembrar que o editor visual de programas, consumidor da maior parte do tempo de desenvolvimento, é baseado no OpenGL, que possui suporte para a linguagem JAVA;

- Inclusão de suporte para matrizes esparsas - A implementação do protótipo do ProVisual não contempla a manipulação de matrizes esparsas. Desta forma, o autor sugere estudos de evolução do protótipo para suportar este tipo de matriz. Este suporte deve incluir formas de visualização da estrutura de distribuição dos valores desta matriz, uma vez que as matrizes esparsas são geralmente de grandes dimensões. Uma possível solução para a visualização da estrutura de uma matriz esparsa pode ser a adaptação da estratégia adotada pelo ProVisual para mostrar a interdependência estrutural de todo o fluxo de dados de uma matriz;
- Generalização do ProVisual para aplicação em outros domínios - Uma planilha pode ser aplicada a vários domínios de aplicação. Neste sentido, o autor sugere trabalhos que estudem formas de aumentar a gama de situações em que o modelo proposto possa ser aplicado;
- Biblioteca de Funções Matriciais – É necessário que se incorpore ao ProVisual uma biblioteca de funções matriciais, visando a construção de programas matriciais de forma mais direta e rápida;
- Interoperabilidade com outros Softwares Científicos – O ProVisual pode ser evoluído para permitir a sua interoperabilidade com outros programas ou dados gerados por eles. Desta forma, pode-se investigar a adoção de um padrão para a troca de informação entre o ProVisual e outras aplicações. Esta evolução acarretará um substancial incremento da aplicabilidade do ProVisual, ao mesmo tempo que facilitará a utilização de outros softwares que operarem com o ProVisual;
- Evolução da interação com o protótipo do ProVisual - Os ícones escolhidos pelo autor para expressar os elementos sintáticos do ProVisual precisam de uma representação mais próxima ao domínio de aplicação. Ademais, a forma de interação do usuário com os editores de programas e de matrizes precisa ser aprimorada de modo a torná-la mais ergonômica. Por fim, as saídas gráficas dos

resultados obtidos pela aplicação de um programa ProVisual precisam ser desenvolvidas;

- Métricas – É importante que possa medir a eficiência do ProVisual frente a outras soluções aplicáveis ao mesmo domínio. Neste sentido, o autor sugere trabalhos que estudem métricas que possam ser aplicadas a uma linguagem de programação visual com as características do ProVisual; e
- Incorporação de novos operadores de iteração - A utilização de padrões de iterações nas planilhas reduz a necessidade de iterações e de recursões. Todavia, para facilitar a utilização do ProVisual, o autor sugere trabalhos que estudem e incorporem outros padrões de iteração para uma matriz e para a linguagem de composição. Estes padrões devem ser visuais e permitir uma melhor legibilidade do programa produzido.

Bibliografia:

[Ambler, 1987] Ambler, A., *Forms: Expanding the visualness of sheet languages.* Proceedings 1987, Workshop on visual language, Tryck-Center, Linkoping, Sweden, 105-117.

[Ambler, 1989] Ambler, A., Burnett, M., *Influence of Visual Technology on the Evolution of Language Environments,* Computer, vol. 22, n. 10, pp. 9-22, Outubro, 1989.

[Ambler, 1990] Ambler, A. L., Burnett, M. M., *Visual Forms of Iteration that Preserve Single Assignment*, Journal of Visual Languages and Computing, Vol. 1, 1990, pp. 159-181, Junho, 1990.

[Ambler, 1997] Ambler, A., Green, T., Kimura, T., Repenning, A., Smedley, T., *Visual Programming Challenge Sumary*, *Proceedings of VL´97*, Capri, Itália, Setembro, 1997.

[Auguston, 1997] Auguston, M., Delgado, A., *Iterative Constructs in Visual Data Flow Language,* Proceedings of VL´97, Capri, Itália, IEEE Computer Society Press, 1997, pp.152-159.

[Baroth, 1995] Baroth, E., Hartsough, C., *Visual programming in the real world*, Visual Object-Oriented Programming: Concepts and Environments, Capítulo 2, pp. 21-42, Manning Publications Co., Greenwich, CT, 1995.

[Bertin, 1981] Bertin, J. *Graphics and Graphic Information Processin*g, Berlim, Alemanha, Walter de Gruyter, 1981.

[Blackwell, 1998] Blackwell, A. F., Whitley, K., Good J., Petre M., *Cognitive Factors in Programming with Diagrams,* Proceedings of Thinking with Diagrams '98, Aberystwyth, Inglaterra, pp. 22-23, 1998, Agosto, 1998.

[Borland, 2000] Borland Software Corporation. <http://www.borland.com/delphi>. Acesso 10/04/2000.

[Borning, 1981] Borning A, *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation,* ACM - Transactions on Programming Languages and Systems, vol. 3. No. 4, pp. 353-387, Outubro, 1981.

[Brown, 1994] Brown, T., Kimura, T., *Completeness of a Visual Computation Model,* Software-Concep ts and Tools, vol. 15, pp. 34-48, 1994.

[Burnett, 1994] Burnett, M., Ambler, A., *Interactive Visual Data Abstraction in a Declarative Visual Programming Language*, Journal of Visual Languages and Computing, pp. 29-60, Março, 1994.

[Burnett, 1995] Burnett, M., Baker, M., Bohus, C., Carlston, P., Yang , S., Van Zee, P., *Scaling Up Visual Programming Languages*, Computer, Volume 28, Número 3, pp. 44-54, Março, 1995.

[Burnett, 1999] Burnett, M., *Visual Programming*, Encyclopedia of Electrical and Eletronics Engineering, John Wiley & Sons Inc., New York, 1999.

[Clarke, 1981] Clarke, B. L., *A Calculus of individuals based on connection.* Notre Dame Journal of Formal Logic, Vol. 23, pp. 204-218, Março, 1981.

[Cox, 1989] Cox, P.T., Giles, F.R., Pietrzykowski, T. *Prograph: a step towards liberating programming from textual conditioning'*. 1989 IEEE Workshop on Visual languages, pp. 150-6.

[Cui, 1993] Cui, Z., Cohn, A. G.,Randell, D.,*Qualitative and Topological Relationships*, Springer Verlag, pp. 290-315, 1993.

[Davis, 1982] Davis, A. L., Keller, R. M., *Data Flow Programs Graphs*, IEEE Computer, vol.15, pp. 175-182, 1982.

[Erwig, 1995] Erwig, M., Meyer, B., *Heterogeneous Visual Languages – Integrating Visual and Textual Programmings*, Proceeding of the 11th International IEEE workshop on Visual Languages, Darmstadt, Germany, Setembro, 1995.

[Evangelista, 2001] EVANGELISTA, S. R. M, Daltrini, B. M., *Um Modelo para Programação Visual de Matrizes aplicado à Biologia Computacional*, 10º Congresso da Associação Latinoamericana de Biomatemática, Campinas-SP, Out/2001.

[Evangelista, 2001a] EVANGELISTA, S. R. M, *Um Modelo para Programação Visual de Matrizes*, EMBRAPA – Informática Agropecuária, Relatório Técnico, Campinas-SP, Nov/2001.

[Excel, 2000] <http://www.microsoft.com>. Acesso 02/02/2001.

[Fu, 1973] Fu, K. S., Bhargava, B. L., *Tree Systems for Syntactic Pattern Recognition.* IEEE Transactions on Computing, Vol 22, 1973.

[Glinert, 1984] Glinert, E. P., Tanimoto, S. L., *Pict: An Interactive Graphical Programming Environment*, Computer, Vol. 17, No. 11, pp. 7-25, Novembro, 1984.

[Glinert, 1990a] Glinert, E. P., *Visual Programming Environments: Applications and Issues.* IEEE Computer Society Press, 1990.

[Glinert, 1990b] Glinert, E. P., *Visual Programming Environments: Paradigms and Systems.* IEEE Computer Society Press, 1990.

[Gooday, 1996] Gooday, J. M., Cohn A., G., *Visual Language syntax and semantics: A spatial logic approach.* International Workshop on Theory of Visual Language, Gubbio, Itália, 1996.

[Gould, 1984] Gould, L., Finzer, W., *Programming by Rehearsal,* Byte, Volume 9, Número 6, pp. 187-210, junho, 1984.

[Graf, 1995] Graf, H. W., Neurohr, S., *Constraint-based layout in visual program design*. Proceedings of the 1995 IEEE Symposium on Visual Languages (VL '95), pp. 116-117, Darmstadt, Setembro, 1995.

[Green,1996] Green, T., Petre, M., *Usability Analysis of Visual Programming Environments: A Cognitive Dimensions' Framework*, Journal of Visual Languages and Computing, Volume 7, Número 2, pp. 131-174, Junho, 1996.

[Harel, 1988] Harel, D., *On Visual Formalisms,* Communications of ACM, Volume 31, Número 5, pp. 514-530, 1988.

[Hendry, 1994] Hendry, D. G., Green, R. G., *Creating, comprehending, and explaining spreadsheets: a cognitive interpretation of what discretionary users think of spreadsheet model,* International Journal of Human-Computer Studies, Vol. 40, pp. 1033-1065, 1994.

[Hendry, 1995] Hendry, D. G., *Display-based problems in spreadsheets: a critical incident and a design remedy*, in Proceedings of the 1995 IEEE Visual Language (VL 95), pp.284-290, Darmstadt, Setembro ,1995.

[Hikmet, 1991] Hikmet S., Santos L., *Graphical representation of logic programs and their behavior.* 1991, IEEE Workshop on Visual Languages, 25-31, Kobe, Japão, Outubro 1991.

[Hils, 1992] Hils, D., *Visual Languages and Computing Survey: Data Flow Visual Programming Languages* , Journal of Visual Languages and Computing, Vol. 3, pp. 69-101, 1992.

[Hirakawa, 1988] Hirakawa M, *A Framework for Construction of Icon System.* Proceedings of the 1988 IEEE Workshop on Visual Languages, Pittsburgh Pennsylvania, pp. 45-51, Outubro, 1988.

[Igarashi, 1998] Igarashi, T., Mackinlay, D., *Fluid Visualization of Spreadsheet Structures*, Proceedings of the 1998 IEEE Symposium on Visual Languages, Halifax, Nova Scotia, Setembro, 1998.

[Ingalls, 1988] Ingalls D, *Fabrick: A Visual Programming Environment. Proceedings ACM OOPSLA*, pp. 176-190, 1988.

[Iverson, 1973] Iverson, K. E., Falkoff, A., *The Design of APL*, IBM Journal of Research and Development, Volume 17, Número 5, pp. 324-334, Julho, 1973.

[Jaffar, 1987] Jaffar, J., Lassez, J., *Constraint Logic Programming*, Proceedings of the Fourteenth ACM Principles of Programming Languages, Munich, pp. 111-119, Janeiro, 1987.

[Kajler, 1995] Kajler, N., Soiffer, N., *Survey of User Interfaces for Computer Algebra Systems*, Journal of Symbolic Computation, Vol. 11, 33p, 1995.

[Kimura, 1990] Kimura T, Choi J, Mack J, *Show and Tell: A visual programming language.* Visual Computing Environments, IEEE Computer Society Press, Washington, D.C., 397-404, 1990.

[Kimura, 1993] Kimura, T. D., *Hyperflow: A Uniform Visual Language for Different Levels of Programming,* Proceedings of the 1993 ACM Conference on Computer Science [CSC 93], 209-214, Indianapolis, USA, Fevereiro, 1993.

[Leopold, 1996] Leopold, J., Ambler, A., *A User Interface for the Visualization and Manipulation of Arrays,* Proceedings of the 1996 IEEE Symposium on Visual Languages [VL '96] , pp. 54-55, Boulder, Colorado, USA, Setembro,1996.

[Marriott, 1998] Marriott, K., Meyer, B., *Visual Language Theory*, Springer-Verlag, New York, 381p., 1998.

[Matwin, 1985] Matwin S, Pietrzykowski T, *Prograph: A Preliminary Report.* Computer Language, Volume 10, Número 2, pp. 91-126, 1985.

[Milosko, 1984] Milosko, J., Kotov, V. E., Algorithms, Software and Hardware of Parallel Computers, Springer-Verlag-NY, pp. 323-349, 1984.

[Myers, 1986] Myers, B., *Visual Programming, Programming By Example and Program Visualization: A Taxonomy*, Proceedings of the 1986 Computer Human Interaction [CHI 86] - Human Factors in Computing Systems, pp. 59-66, Boston, Massachusetts, USA, Abril, 1996.

[Nardi, 1993] Nardi, Bonnie A. *A small matter of programming: perspectives on end user computing*. The MIT Press. 1993.

[Nickerson 1994] Nickerson, J., *Visual programming: Limits of graphical representation*. In Proc. 1994 IEEE Symposium on Visual Languages, pp 178-179, Outubro 1994.

[OpenGL, 2000] OpenGL. <http://www.opengl.org/users/index.html>. Acesso 10/04/2000.

[Pandley, 1993] Pandley K, Burnett M., *Is it easier to write matrix manipulation programs visually or textually?* Estudo Empírico, IEEE Symposium on Visual Languages, 344-351, Bergen. Norway, 1993.

[Parnas, 1978] Parnas, D., Bartussek W., *Using assertions about traces to write abstract specifications for software modules*, Proceedings of the 2nd Conference on European Cooperation in Informatics, Springer-Verlag, 1978, pp. 111--130.

[Raeder, 1985] Raeder, G. A Survey of Current Graphical Programming Techniques, IEEE Computer, No. 8, Agosto, 1988, pp. 11-25, Agosto, 1988.

[Randell, 1992] Randell, D., Cui, Z., Cohn, A., *A Spatial Logic based on Regions and Connection,* 3 rd Int. Conf. on Knowledge Representation and Reasoning, Morgan Kaufman, 1992.

[Reiss, 1985] Reiss, S. P., *Pecan: Program Development Systems that Support Multiple Views*, IEEE Trans. Software Engineering, vol. 11, No. 3, pp. 276-285, Março, 1985.

[Shu, 1988a] Shu, N. C., *Visual Programming*. Van Nostrand Reinhold Company Limited, 115 Fifth Avenue, New York, 1988.

[Shu, 1988b] Shu. N. C., *Principles of Visual Programming,* Systems Shi-Kuo Chang, Van Nostrand Reinhold, New York, NY., Prentice Hall, Englewood Cliffs, NJ, 1988.

[Shür, 1997] Shür, A., "BDL - *A Nondeterministic Data Flow Programming Language with BackTracking*", Proceedings of the 1997 IEEE Symposium of Visual Language [VL 97], IEEE Computer Society Press, Capri, Itália, Setembro, 1997.

[Smith, 1994] Smith, D., *KidSim: Programming Agents Without a Programming Language*, Communications of the ACM, Vol. 37, No. 7, Julho, 1994.

[Shneiderman, 1983] Shneiderman, B., *Direct manipulation: a step beyond programming language*. Computer, Volume 16, Número 80, pp. 57-69, agosto, 1983.

[Stella, 2001] High Performance Systems, Inc. <http://www.hps-inc.com>. Acesso 23/06/2001.

[Tripp, 1988] Tripp, L., *A Survey of Graphical Notations for Program Design: An Update*, ACM SIGSOFT Software Engineering Note, Vol. 13, No. 4, pp. 39-44. Abril, 1988.

[Vose, 1986] Vose, G., Williams, G., *LabView: Laboratory Virtual Instrument Engineering Workbench,* Byte, Volume 11, Número 9, pp. 84-92, Setembro, 1986.

[Yeung, 1988] Yeung, R., *MPL - A Graphical Programming Environment for Matrix - Processing Based on Logic and Constraints*, in IEEE Workshop of Visual Languages, pp. 137- 143. IEEE Computer Society Press, Outubro,1988.

[Wang, 1993] Wang, D., Lee, J. R., *Visual reasonin,: its formal semantics and applications. Journal of Visual Languages and Computing, Abril, 1993.*

[Whitley, 2000] Whitley, K. N., *Empirical Research Of Visual Programming Languages: An Experiment Testing the Comprehensibility of Labview*, Tese de Doutorado, Graduate School of Vanderbilt University, Nashville, Tennessee, USA, Maio, 2000.