View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositorio da Producao Cientifica e Intelectual da Unicamp

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Uma Ferramenta de Programação Visual Para Previsão e Reconhecimento de Padrões

Joaquim José Fantin Pereira

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica, sob orientação do Prof. Dr. Takaaki Ohishi.

Este exemplar corresponde à redação da Dissertação de Mestrado defendida e aprovada pela comissão julgadora em 06/07/2007

Banca Examinadora

Paulo Sérgio Franco Barbosa, Prof. Dr. . . . . FEC/Unicamp
Ricardo Ribeiro Gudwin, Prof. Dr. . . . . DCA/FEEC/Unicamp
Rosângela Ballini, Prof. Dra. . . . . .IE/Unicamp
Takaaki Ohishi, Prof. Dr. . . . . DENSIS/FEEC/Unicamp

Campinas - SP

Julho/2007

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

P414f	Pereira, Joaquim José Fantin Uma ferramenta de programação visual para previsão e reconhecimento de padrões / Joaquim José Fantin Pereira. -- Campinas, SP: [s.n.], 2007. Orientador: Takaaki Ohishi Dissertação (Mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação. 1. Processo decisório. 2. Programação visual (Computação). 3. Reconhecimento de padrões. 4. Previsão estatística. 5. Engenharia – Modelos. I. Ohishi, Takaaki. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Título em Inglês: A visual programming tool for forecasting and pattern recognition
Palavras-chave em Inglês: Decision making, Visual programming language, Pattern recognition, Load forescating, Systems modeling
Área de concentração: Energia Elétrica
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: Paulo Sérgio Franco Barbosa, Ricardo Ribeiro Gudwin, Rosângela Ballini
Data da defesa: 06/07/2007
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidato: Joaquim José Fantin Pereira

Data da Defesa: 6 de julho de 2007

Título da Tese: "Uma Ferramenta de Programação Visual para Previsão e Reconhecimento de Padrões"

Prof. Dr. Takaaki Ohishi (Presidente): ____________________

Profa. Dra. Rosângela Ballini: ____________________

Prof. Dr. Paulo Sérgio Franco Barbosa: ____________________

Prof. Dr. Ricardo Ribeiro Gudwin: ____________________

Resumo

A tomada de decisão, em qualquer setor e nos mais diversos níveis, é um processo cada vez mais complexo, principalmente em função do nível de incerteza em relação ao futuro. Neste contexto, a disponibilidade de previsões torna-se um fator importante para uma decisão mais eficaz. As ferramentas de reconhecimento de padrões, por sua vez, são importantes em muitas áreas, tais como nas determinações de comportamentos típicos e em sistemas de controle. Nessa conjuntura, a proposta deste trabalho consistiu em explorar a criação e o uso de uma linguagem de programação visual, denominada Linguagem VisualPREV, de modo a facilitar a concepção e a execução dos modelos de previsão e classificação. Nesta Linguagem, blocos visuais colocados num diagrama (interface visual computacional) representam conceitos envolvidos num processo de modelagem do problema. O modelo pode então ser configurado, executado e armazenado para acesso futuro. Embora essa escolha implique uma perda de vantagens exclusivas da programação em código tradicional, como a maior flexibilidade para programação genérica, por exemplo, a linguagem diminui sensivelmente o tempo de criação dos modelos específicos para tratamento de dados em previsão de séries temporais e reconhecimento de padrões. Em algumas aplicações com dados relevantes, a linguagem foi avaliada com critérios baseados em métricas de usabilidade e os resultados foram discutidos ao longo do trabalho.

Palavras-chave: Processo decisório, Programação visual (Computação), Reconhecimento de padrões, Previsão estatística, Engenharia – Modelos.

Abstract

Decision making, in any area and in many different levels, is a process with growing complexity, mainly if you consider the level of uncertainty related to the future. In this context, the possibility of forecasting plays a major role in an efficient decision. On the other hand, pattern recognition tools are important in many areas, like fitting typical behaviors and in control systems, as well. In this context, we propose a visual programming language, called VisualPREV Language, intended to make easier the conception and execution of forecasting and pattern recognition models. Within this language, visual blocks that can be put into a diagram (computational visual interface) represent concepts involved when modeling the processes. These models can be configured, executed and stored for future access. Although these approach implies losing exclusive advantages of traditional programming (like flexibility of generic programming, for example), VisualPREV decreases considerably the amount of time needed for creating specific models for forecasting and pattern recognition. In few applications with relevant data, the language was evaluated based on usability metrics, and the results were discussed throughout the text.

Keywords: Decision Making, Visual Programming Language, Pattern Recognition, Load Forecasting, Systems Modeling.

Agradecimentos

Agradeço primeiramente a Deus, por me dar força e saúde para vencer mais uma jornada e pela benção de ter colocado e continuar colocando em minha vida oportunidades e, principalmente, pessoas tão especiais. Entre elas, uma família maravilhosa, os melhores professores e todos aqueles que só eu sei o quanto são e foram importantes nesta vitória.

Aos meus pais, Joaquim e Vanessa, e à minha irmã, Bruna, por serem o alicerce, a força e a inspiração para que eu continue sempre, e sem os quais a realização deste trabalho não teria sido possível. Tenho orgulho de tê-los em minha vida e esta vitória também é de vocês.

Ao professor Takaaki por ter acreditado na proposta deste trabalho, bem como por ter contribuido ativamente para a realização do mesmo.

À Renata Monezzi, arquiteta e artista plástica, pelo projeto visual integrado da VisualPREV.

Aos eternos amigos da 'República Buraco': especialmente Carioca, Chico Migranha, Irmãos Metralha, Juan, Jundiaí, Marmota, Mimosa, Musgo, Palmitão, Piauí, Roman, Theco, Vó, Wando e a todos os outros que passaram por aquela pacata e tradicional casa de família!

À Igreja do Nazareno Central, templo onde fiz muitos amigos e por onde passei uma parte muito especial de minha vida. Um abraço especial nos colegas do coral Kairós.

À querida equipe de natação da UNICAMP, a USS Reloaded, um grupo muito especial de amigos.

Aos companheiros da Faculdade de Engenharia Elétrica colegas do curso, em especial àqueles do COSE, com os quais passei a maior parte de meu tempo, seja trabalhando, conversando, comendo, nos divertindo ou praticando esportes (COSE-Esportes).

Aos participantes dos testes de usabilidade, pela imensa colaboração e disposição para ajudar em minha pesquisa.

A FAPESP pelo suporte financeiro através da bolsa de estudos.

Finalmente, a todos aqueles – amigos, professores, funcionários – que estiveram envolvidos, direta ou indiretamente, na execução deste trabalho e cujos nomes, que são muitos, eu tenha me esquecido de enumerar.

Dedico este trabalho à toda a minha querida família,
especialmente aos meus pais *Joaquim* e *Vanessa*
e à minha irmã *Bruna*.

“A coisa mais grandiosa que a alma humana faz neste mundo, é ver alguma coisa...
e ver claramente significa poesia, profecia e religião, juntas.”
John Ruskin, Pintores Modernos

“...e qual a utilidade de um livro, pensou Alice, que não possua figuras....”
Lewis Carroll, Alice no País das Maravilhas

Sumário

1. Introdução..........15
1.1 Objetivo e Motivação..........16
1.1.1 Previsão de Carga..........16
1.1.2 Classificação..........17
1.1.3 Reconhecimento de Regime de Escoamento..........18
1.2 Organização do Trabalho..........19

2. Programação Visual..........21
2.1 Introdução..........21
2.2 Definição..........23
2.2.1 Ambientes de Programação Visual e Linguagens de Programação Visual..........25
2.2.2 Nomenclatura Alternativa..........26
2.2.3 Nomenclatura Adotada..........28
2.3 Histórico..........28
2.4 Estratégias em Programação Visual..........30
2.5 Classificação das Linguagens de Programação Visual..........31
2.6 Programação Visual e Abstração..........34
2.7 Especificação de VPLs..........36
2.8 Programação Visual e Aspectos Cognitivos..........39
2.9 Prós e Contras..........43
2.9.1 Evidências Empíricas..........45

3. A linguagem VisualPREV e seu Ambiente de Desenvolvimento..........47
3.1 Introdução e Histórico..........47
3.2 A VisualPREV: Ambiente e Linguagem..........48
3.3 Especificação..........49
3.3.1 Visão Geral do Ambiente da VisualPREV..........51
3.3.2 Vocabulário da Linguagem: *Tokens* Visuais e Formulários associados..........53
3.3.2.1 *Tokens* ou Blocos de Entrada de Dados..........53

3.3.2.2 *Tokens* ou Blocos de Ferramentas..55
3.3.2.3 *Tokens* ou Blocos de Armazenamento dos Resultados....................62
3.3.3 Fluxo de Dados..63
3.3.4 Validação Sintática e Semântica..65
3.3.5 *Framework* Computacional...66
3.4 VisualPREV *versus* abordagem tradicional..67
3.5 Perspectivas Futuras de Implementação...67

4. Duas Aplicações da Linguagem VisualPREV..69
4.1 Exemplo 1: Previsão de Séries Temporais...69
4.1.1 Primeira Fase: Montagem dos Conjuntos de Dados.........................69
4.1.2 Segunda Fase: Construção do Modelo...71
4.1.3 Terceira Fase: Análise dos Resultados...72
4.2 Exemplo 2: Classificação de Padrões com Decodificação dos valores classificados..74
4.2.1 Primeira Fase: Montagem dos Conjuntos de Dados.........................74
4.2.2 Segunda Fase: Construção do Modelo...77
4.2.3 Terceira Fase: Análise dos Resultados...78

5. Avaliação da VisualPREV..81
5.1 Testes de Usabilidade...81
5.1.1 Método..82
5.1.2 Procedimento...82
5.1.3 Resultados...83

6. Conclusões..87

Apêndice A – Análise de Alguns Exemplos de VPLs.................................89
Apêndice B – Evidências Empíricas Contra e a Favor das VPLs..............101
Apêndice C – Instruções Sobre os Experimentos para os Participantes......107
Apêndice D – Questionário Pós-Experimento...111
Referências Bibliográficas...115

1. Introdução

A tomada de decisão, em qualquer setor e nos mais diversos níveis, é um processo cada vez mais complexo, principalmente em função do nível de incerteza em relação ao futuro. Neste contexto, a disponibilidade de previsões torna-se um fator importante para uma decisão mais eficaz. As ferramentas de reconhecimento de padrões, por sua vez, são importantes em muitas áreas, tais como nas determinações de comportamentos típicos e em sistemas de controle. No primeiro grupo de aplicações, podem ser consideradas as decisões em planejamento da produção, na definição de contratos de curto, médio e longo prazos, no planejamento estratégico e na operação do sistema. No segundo grupo, pode-se considerar a determinação de padrões típicos, como, por exemplo, a identificação de padrões de consumo de energia elétrica e o agrupamento de elementos com características similares. Uma importante aplicação de reconhecimento de padrões é na área de controle de processos, como, por exemplo, no controle de escoamento de fluidos. Nestes sistemas são importantes a identificação do padrão de escoamento e adequar a operação para a situação dada.

Na área de previsões, como também na de reconhecimento de padrões, muitos modelos matemáticos têm sido desenvolvidos e utilizados, e, em muitos setores, estas ferramentas estão definitivamente incorporadas nos seus processos de tomada de decisão. Em termos de técnicas, as mais utilizadas têm sido as baseadas em modelos estatísticos, regressão, *box-jenkins* e inteligência artificial (nesta última, destacam-se as Redes Neurais Artificiais (RNA) e as Redes Neuro-*Fuzzy* (RNF)). É uma área em que há muito desenvolvimento e novas técnicas estão sempre sendo propostas. A maioria destas técnicas oferece muitas alternativas em termos de estruturas e parâmetros, e combinação entre diferentes técnicas, o que possibilita investigar múltiplos aspectos. Por outro lado, esta grande flexibilidade implica em testar um número muito grande de alternativas, mas que, em geral, devido a limitações de tempo, não são exaustivamente exploradas. Com a multiplicidade de metodologias, os modelos híbridos, nos quais diferentes técnicas são utilizadas, estão sendo muito pesquisados e, em muitos casos, com excelentes resultados. Nestes sistemas híbridos, as possíveis alternativas são, em geral, muito variadas, de modo que no desenvolvimento destes modelos requer-se também um número muito grande de testes. Em todos estes estudos, a disponibilidade de um sistema que possa facilmente configurar diferentes alternativas, bem como a integração de diferentes modelos,

seria de grande utilidade, pois possibilitaria pesquisar, em menor espaço de tempo, um maior número de alternativas.

1.1 Objetivo e Motivação

O objetivo desde projeto de pesquisa foi desenvolver uma Linguagem de Programação Visual para Previsão de Séries Temporais e Reconhecimento de Padrões (VisualPREV). Esta linguagem possui um ambiente computacional que facilita a definição, configuração e aplicação de modelos de previsão de séries temporais e de reconhecimento de padrões. Neste projeto, serão enfocadas a previsão de carga de energia elétrica (Kadowaki et al., 2002), o agrupamento de consumidores de energia elétrica (barramento) com padrões de consumo similares (Menezes at al., 2004), e a determinação de padrões de escoamento bifásico. Neste último tópico, pode-se determinar o regime de escoamento em uma tubulação pelo qual passa líquido e gás. Outras aplicações podem ser desenvolvidas, pois o ambiente da VisualPREV é flexível, tanto do ponto de vista da incorporação de novas técnicas, como também de aplicações.

A linguagem VisualPREV apresenta facilidades de modelagem tanto em termos de técnicas, como também de estrutura e combinação de modelos. Também apresenta facilidades para configurar um dado modelo, seja de previsão de séries temporais ou de reconhecimento de padrões. Em cada caso, há a possibilidade de selecionar a técnica a ser empregada, desde a sua estrutura e até a sua aplicação. Para dar uma idéia mais detalhada serão aqui considerados três possíveis casos de modelagem, os quais serão brevemente apresentados a seguir.

1.1.1 Previsão de Carga

A previsão da demanda de energia elétrica é importante para diversos fins. Por exemplo, nas decisões de longo prazo, quando são definidos os contratos de comercialização e as decisões sobre a expansão do sistema (construção de novas fontes geradoras e de novas linhas de transmissão e distribuição). Já no horizonte de médio prazo, as previsões são importantes para o controle de fluxo de caixa (faturamento) e para a definição de contratos comerciais; já no curto prazo, as previsões são importantes, principalmente, para as decisões relativas à operação do sistema. Em cada caso requerem-se modelos específicos, tanto em

termos de técnica, como também em termos de estrutura e aplicação. Neste projeto, pretende-se enfocar a previsão de carga de curto prazo, cujo objetivo principal é subsidiar as decisões relativas à operação do sistema, mais especificamente na definição da programação da operação diária (Soares at al., 2003) e na análise de segurança da operação de um sistema de energia elétrica.

Tradicionalmente, quando um modelo é desenvolvido, a sua programação computacional é, em geral, especializada para tratar um dado conjunto de entradas disponíveis em uma dada base de dados. Quando há a necessidade de se testar o desempenho de outro modelo, esta modificação no programa computacional pode levar um tempo considerável, que, em muitos casos, pode atingir meses. Com a VisualPREV, estas alterações podem ser facilmente implementadas e testadas, o que contribui muito nos testes comparativos de modelos.

Uma outra importante funcionalidade do sistema de modelagem é a capacidade de "conectar" diferentes modelos, no qual as saídas de um ou mais modelos previamente definidos poderão ser incluídas como entradas de outros modelos. Esta funcionalidade possibilita o desenvolvimento de um dado modelo em módulos, e isto trás vantagens em termos de implementação, manutenção e alteração, e possibilita construir modelos complexos a partir de módulos simples.

1.1.2 Classificação

O reconhecimento de padrões é importante em diversas áreas, e, em particular, em um sistema de energia elétrica, o conhecimento dos padrões típicos de consumo é muito utilizado, por exemplo, na área de operação de curto prazo (operação diária), pois o perfil do consumo diário influi na definição do programa de operação, no nível de reserva, e no nível de intercâmbios. Recentemente, Salgado (Salgado, 2004) tratou do problema de previsão de carga por barramento. Além das diversas metodologias apresentadas em seu trabalho, foi utilizada uma metodologia de classificação ("clustering") baseada em RNA de Kohonen visando agrupar barramentos com perfis de consumo diário similares. Por meio desta metodologia, a maioria das barras foi classificada em três grupos, um tipicamente industrial, com o consumo diário apresentando poucas variações ao longo do dia; outro grupo tipicamente comercial, com o maior consumo durante o período comercial; e um grupo com um perfil de consumo diário

típico, com reduzida demanda na madrugada e um grande consumo no período de ponta (das 17:00h às 22:00h).

Outros padrões poderiam ser pesquisados, como, por exemplo, identificar o perfil de consumo anual. A classificação das barras com perfis similares pode ser muito útil no processo de previsão de carga por barramento, pois possibilita fazer uma previsão da demanda por barramento de forma agregada, diminuindo assim o número de previsões.

Em termos de metodologia, uma das abordagens para o reconhecimento de padrões é através de técnicas de classificação. Para isso a VisualPREV também pode ser utilizada para implementar um modelo de reconhecimento de padrões.

1.1.3 Reconhecimento de Regime de Escoamento

A crescente complexidade operacional dos atuais sistemas requer um eficiente sistema de controle e, nesta área, o reconhecimento de padrões pode ser de muita utilidade. Em particular, em sistemas de transporte de fluidos através de dutos, muito comuns nas indústrias químicas e petrolíferas, o conhecimento do regime de escoamento é de grande importância, pois nestes dutos passam, em geral, líquidos e gases, e, dependendo da proporção entre estes dois fluidos, têm-se diferentes regimes de escoamento, como mostrado na Figura 1. Para a operação adequada destes sistemas deve-se ajustá-las para cada tipo de regime. Como estes regimes podem variar, é interessante a existência de sistemas que automaticamente obtenham dados sobre o escoamento atual e detectem alterações nestes regimes. Através destes dados também se podem estimar as vazões dos fluidos. O Laboratório de Fenômenos Multifásicos da Faculdade de Engenharia Mecânica da UNICAMP já vem desenvolvendo há muitos anos uma sonda capacitiva capaz de traduzir em sinais elétricos representativos (série temporal) o regime de escoamento. A próxima etapa é desenvolver um sistema que automaticamente identifica o padrão de escoamento a partir deste sinal. A identificação deste regime de escoamento é uma das aplicações desejadas da VisualPREV neste trabalho.

Em termos metodológicos, esta identificação pode ser realizada através da determinação da Função de Densidade de Probabilidade (*PDF – Probability Density Function*), que é típica para cada tipo de escoamento. Estas *PDF*s serão classificadas (agrupadas) determinando os padrões típicos de regime de escoamento. Através destes padrões, pode-se facilmente identificar o tipo de escoamento referente a um dado sinal.

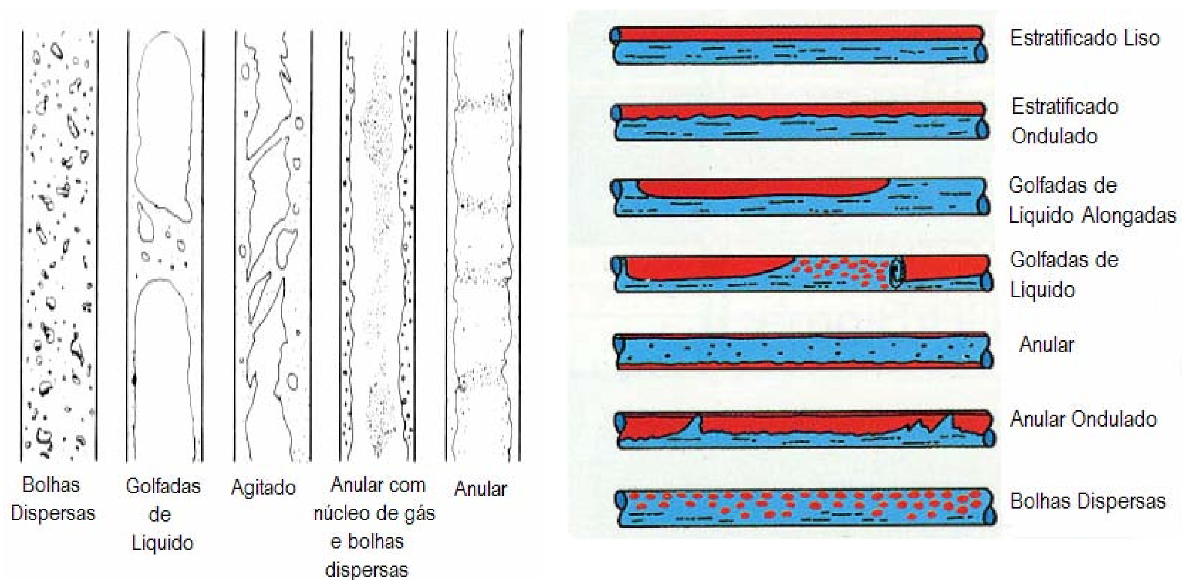


Fig. 1. Padrões de escoamento bifásico gás-líquido. (a) padrões encontrados no escoamento vertical ascendente. (b) padrões encontrados no escoamento horizontal.

1.2 Organização do Trabalho

Os capítulos descrevem em detalhes todos os resultados obtidos na busca do atendimento dos objetivos genéricos enunciados na seção anterior. O capítulo 1 inclui motivação, objetivos e organização do texto. O capítulo 2 apresenta um panorama das Linguagens de Programação Visual, explorando o surgimento, definição, estratégias, classificação, tendências, especificação, prós e contras e outros assuntos correlatos. O capítulo 3 expõe a Linguagem VisualPREV e seu ambiente de desenvolvimento, seu histórico e suas características. O capítulo 4 exibe uma aplicação exemplo da Linguagem VisualPREV. O capítulo 5 mostra os resultados da avaliação da VisualPREV. O capítulo 6 apresenta as conclusões do trabalho.

Há também um conjunto de apêndices que contêm informação complementar relacionada a essa pesquisa. O apêndice A descreve algumas linguagens visuais existentes, explicando-as de acordo com os conceitos apresentados neste trabalho. O apêndice B mostra trabalhos com evidências empíricas contra e a favor das VPLs. O apêndice C apresenta

instruções sobre os experimentos para os participantes. O apêndice D exibe o questionário pós-experimento.

Por último, há a seção de referências bibliográficas que lista todos os trabalhos que foram utilizados para esta pesquisa.

2. Programação Visual

2.1 Introdução

No princípio, toda a comunicação homem-máquina ocorria somente devido a razões técnicas e era restrita a uma dimensão. Comandos ao sistema operacional e programas eram todos cadeias lineares de caracteres. Por muitos anos, estas cadeias de caracteres eram tecladas via console do operador ou quebradas em segmentos e inscritas em cartões perfurados.

Em 1975, a dissertação histórica de David Canfield (Smith, 1975) assinalou o despontar de uma nova era na programação na qual o poder crescente dos engenhos computacionais e suas capacidades gráficas tornaram possível tentar utilizar a tela bidimensional mais do que como um mero cenário de fundo de suporte à cadeia linear de caracteres. Ambientes de programação, ao invés de linguagens, tornaram-se o foco da atenção. O termo *ambiente de programação* se refere à linguagem de programação no contexto de um conjunto integrado de ferramentas de suporte de *software* e *hardware* apropriado, especialmente dispositivos de entrada/saída. Os dois novos estilos de interação homem-computador podem coletivamente (e de alguma maneira livremente) ser chamados de não-textuais. Em *ambientes visuais*, elementos gráficos desempenham um papel proeminente com relação ao texto, pelo menos em algumas fases do trabalho do usuário; em *ambientes icônicos*, o usuário interage com a máquina apontando, manipulando e justapondo pequenas imagens.

Na década de 80, houve um acúmulo de evidências convincentes de que ambientes visuais e icônicos poderiam provar-se altamente benéficos, tanto para usuários de computador em geral quanto para programadores em particular. Isto ocorre porque a habilidade do computador em representar de uma maneira visual aspectos normalmente abstratos e efêmeros do processo de computação, tais como recursão, concorrência e evolução das estruturas de dados ao longo do tempo, pode ter um impacto notável e positivo na produtividade dos programadores e em seu grau de satisfação com o ambiente de trabalho. Ambientes icônicos, em particular, são intrigantes porque eles carregam consigo a promessa de nos permitir técnicas como "manipulação direta" de objetos (ou pelo menos de suas fachadas visíveis) para compor programas.

Apesar de todo o progresso que foi conseguido desde que o primeiro compilador FORTRAN foi escrito nos anos 50, é inegável o fato de que nós permanecemos incapazes de vencer o abismo que continua a separar a *maneira que nós, seres-humanos, concebemos as soluções dos problemas* da *maneira que nós devemos agora programá-los para nossos computadores*. Verdadeiramente, nossos computadores tornam-se mais poderosos ano a ano, e nosso conhecimento de como os utilizar avança também. Mas nossas expectativas e demandas crescem ainda mais rapidamente. O resultado é que, hoje em dia, quase meio século desde o surgimento da programação, o problema central da ciência de computação é ainda encontrar uma *maneira pela qual nós seremos capazes de fazer estas máquinas fazerem o que nós realmente queremos que elas façam*. Isto é verdade se o termo *programação* for interpretado num sentido estrito e convencional ou num aspecto mais amplo, como um esforço que englobe todos os aspectos da interação homem-máquina, incluindo atividades tão diversas como dialogar com o sistema operacional, resolver um sistema de equações diferenciais, implementar um vídeo game, editar e formatar um documento, enviar e-mail, ou desenhar projetos de chips VLSI, apenas para enumerar alguns.

De acordo com Glinert (Glinert, 1990), existe uma diferença entre como concebemos as soluções de programação hoje e como deveríamos concebê-las, conforme Tabela 1. "Este esboço da maneira pela qual nós gostaríamos de trabalhar naturalmente faz surgir o conceito de *ambiente de programação* – isto é, linguagem de programação no contexto de um conjunto integrado de ferramentas de suporte e *hardware* apropriados, especialmente dispositivos de entrada e saída. Questões relacionadas aos gráficos, paradigmas alternativos de programação e inteligência artificial, do lado do *software*, e múltiplos dispositivos de entrada e saída, do lado do *hardware*, são apenas poucos dentre os que precisam ser estudados se tais ambientes se tornarão realidade. Concentraremos nossa atenção na idéia revolucionária de usar gráficos para complementar (em alguns casos, até mesmo substituir) o texto no ambiente de programação, na crença de que o estudo das linguagens textuais atingiu o ponto em que apenas avanços evolucionários menores devem ser esperados" (Glinert, 1990).

Embora alguns dos avanços descritos por Glinert em 1990 (e.g. combinação de meios gráficos e textuais) tenham se consolidado nos anos seguintes, ainda parecemos estar distantes de sua visão quanto às tarefas ideais de programação. Entretanto, avanços teóricos e práticos surgiram desde o princípio desta discussão, e serão abordados neste capítulo.

Tabela 1. Diferenças entre como concebemos as soluções de programação hoje e como deveríamos concebê-las, apontadas por Glinert (Glinert, 1990)

Tarefa de Programação Atual	**Tarefa de Programação Ideal**
1. Selecionar *um* paradigma apropriado para o algoritmo à mão, juntamente com *uma* de suas linguagens associadas.	1. Selecionar *um conjunto* de paradigmas, textuais e/ou gráficos mais apropriados, adequados para expressar os aspectos variados do algoritmo à mão.
2. Selecionar *símbolos* (usualmente cadeias de letras e dígitos) para as estruturas de dados e variáveis necessárias.	2. Representar as estruturas de dados e as variáveis necessárias por uma *combinação de meios gráficos e textuais*.
3. Codificar o algoritmo como uma *cadeia textual (linear)*, e alimentá-lo como entrada ao computador via *editor de texto*, com o auxílio de um *teclado e monitor* (e muitas vezes um *mouse* para facilitar poucas e simples funções relacionadas à tarefa de programação).	3. Compor o programa com *assistência inteligente* do computador, usando uma *variedade de ferramentas e dispositivos de entrada e saída*, e *desenhando* o algoritmo como uma *figura multidimensional logicamente estruturada*.
4. Executar a tarefa e *esperar* pelos resultados. Se os resultados parecem incorretos, *você* tenta descobrir onde estão os erros do programa; então *você* modifica o programa e repete este passo.	4. *Assistir* o código *sendo executado* e ver os resultados serem gerados, pelo menos na fase de encontrar e remover erros no desenvolvimento do programa, de tal forma que, caso a saída não seja a esperada, nós na verdade seremos capazes de ver onde e quando o erro ocorreu.

2.2 Definição

De acordo com Burnett (Burnett, 1999), *programação visual* é a programação em que mais de uma dimensão é usada para transmitir semântica. Exemplos de tais dimensões adicionais são o uso de objetos multidimensionais, o uso de relações espaciais ou o uso da dimensão "tempo" para especificar relações semânticas do tipo "antes-e-depois". Cada objeto

ou relação multidimensional potencialmente significativa é um "*token*" (assim como em linguagens de programação textuais tradicionais cada palavra é um "*token*") e a coleção de um ou mais destes "*tokens*" é uma *expressão visual*. Exemplos de expressões visuais usadas em programação visual incluem diagramas, esquemas à mão-livre, ícones ou demonstrações de ações realizadas por objetos gráficos. Quando a sintaxe de uma linguagem de programação (semanticamente significativa) inclui expressões visuais, a linguagem de programação é uma *linguagem de programação visual* (*visual programming language* - VPL).

Embora as linguagens de programação textuais tradicionais freqüentemente incorporem dispositivos de sintaxe bidimensionais de uma maneira limitada – uma dimensão x para transmitir uma cadeia linear permitida na linguagem, e uma dimensão y permitindo espaçamento em linhas opcionais como um dispositivo de documentação ou para semântica limitada (como em "continuação da linha anterior") – apenas uma destas dimensões transmite semântica, e a segunda dimensão foi limitada a uma noção de "teletipo" das relações espaciais com a intenção de ser expressa numa gramática de cadeia unidimensional. Portanto, multidimensionalidade é a diferença essencial entre VPLs e linguagens estritamente textuais.

De acordo com Myers, programação visual refere-se a qualquer sistema que permita ao usuário especificar um programa em duas ou mais dimensões (Myers, 1990). Linguagens textuais convencionais não são consideradas bidimensionais uma vez que os compiladores e interpretadores as processam como uma longa cadeia unidimensional de caracteres.

Golin e Reiss transmitem a idéia de que uma linguagem visual manipula informação visual ou suporta interação visual, ou ainda permite programar com expressões visuais (Golin e Reiss, 1990). A última pode ser tomada também como sendo a definição de uma linguagem de programação visual. Ambientes de programação visual provêm elementos gráficos ou icônicos que podem ser manipulados pelo usuário de uma maneira interativa de acordo com alguma gramática especial de programação.

De acordo com McIntyre e Glinert, programação visual é comumente definida como o uso de expressões visuais (como gráficos, desenhos, animações e ícones) no processo de programação (McIntyre e Glinert, 1990). Essas expressões podem ser usadas em ambientes de programação com interfaces gráficas para programação de linguagens textuais tradicionais; elas podem ser usadas para formar a sintaxe de novas linguagens de programação visual conduzindo à novos paradigmas tais como a programação por demonstração; ou podem ser usadas em apresentações gráficas do comportamento ou estrutura do programa.

Sobretudo, os autores concordam, em suas definições, que o aspecto diferenciador marcante e decisivo entre as linguagens textuais convencionais e as linguagens de programação visual é a multidimensionalidade.

2.2.1 Ambientes de Programação Visual e Linguagens de Programação Visual

Sistemas como o Visual Basic e Visual C++ não são linguagens de programação visual porque ainda existe código baseado em texto sob elas. Enquanto uma linguagem de programação visual deve ter uma versão baseada em texto que a acompanhe, ou um suporte localizado abaixo, ela nunca deve disponibilizar esta versão em texto ao usuário. Isto quer dizer, um programador nunca deve ter acesso ao código baseado em texto, mas, ao invés disso, deve criar e controlar todas as funções pictoricamente.

De acordo com Burnett, quando expressões visuais são usadas num ambiente de programação como um atalho editor para gerar código que pode ou não ter uma sintaxe diferente daquela usada para editar o código, o ambiente é chamado de *ambiente de programação visual* (*visual programming environment* - VPE) (Burnett, 1999). Ambientes de programação visual para linguagens textuais tradicionais provêm um nível intermediário entre VPLs e as largamente conhecidas linguagens textuais. Em contraste com alguns anos atrás, quando ambientes de programação em linha de comando estritamente textuais eram a norma, as VPEs para linguagens tradicionais de hoje são o tipo predominante de ambientes de programação comerciais. VPEs comerciais para linguagens tradicionais são focadas em programadores profissionais; estes programadores usam as linguagens textuais que eles já conhecem, mas são auxiliados por técnicas de interface gráfica de usuário e acessibilidade à informação que abordagens visuais podem adicionar. VPEs para linguagens tradicionais servem como um condutor para transferir os avanços em pesquisa de VPL em prática através da aplicação destas novas idéias às linguagens tradicionais já familiares aos programadores, portanto proporcionando uma migração gradual de técnicas de programação textual para técnicas mais visuais. VPEs têm se concentrado mais em fazer o projeto da interface ao usuário mais fácil, o que também é um problema importante que precisa ser resolvido.

Programação visual é encontrada em ambas VPLs e VPEs. Comercialmente, a programação visual é mais comumente encontrada nas VPEs, que servem como um condutor eficaz para alguns dos ganhos feitos da pesquisa em VPLs a serem rapidamente transferidos em prática industrial. Disto, conclui-se que VPEs são um subconjunto de VPLs quando desconsideramos o acesso (normalmente permitido) ao código pelos VPEs. Entretanto, é importante frisar que VPLs são normalmente integradas em seus próprios ambientes personalizados.

2.2.2 Nomenclatura Alternativa

Com o intuito de evitar a confusão gerada pelos termos VPL e VPE e, por conseguinte, a falsa impressão de que VPEs são de fato VPLs, e também considerando a multiplicidade de significados possíveis do termo "programação visual", foram encontradas na literatura duas referências de propostas alternativas para designar as VPLs.

Glinert prefere chamar VPLs de "ambientes não-textuais" (Glinert, 1990). Entretanto, Burnett opta por não excluir conceitualmente a utilização do texto na definição de VPLs ao afirmar que "um mal-entendido comum é que o objetivo da pesquisa em programação visual em geral e VPLs em particular é o de eliminar o texto completamente (Burnett, 1999). Isto é uma falácia – na verdade, a maior parte das VPLs incluem texto de alguma forma, num contexto multidimensional". Para o contexto do presente trabalho, manteremos o termo "ambientes não textuais" quando o texto fizer referência a Glinert.

Lakin sugere "gráfico executável" em vez de linguagens de programação visual, pois considera este último termo inadequado, argumentando que significa tanto uma linguagem de programação que podemos ver, o que é trivial, quanto uma linguagem usada para programar o comportamento de coisas visuais, o que é limitante (Lakin, 1986). "Gráfico executável" expressa uma orientação diferente em direção ao domínio do problema: os gráficos podem ser executados.

Lyon cunhou o termo "hiperprogramação" que ele considera que resume melhor as capacidades e o suporte provido pelas linguagens de programação visual, e argumenta esta adoção tanto com base teórica quanto prática (Lyon at al., 1993). Os argumentos teóricos estão relacionados à grande expressividade e intuição das representações diagramáticas de relações

complexas. Os argumentos práticos estão relacionados à disponibilidade de poder computacional suficiente para suportar a captura e processamento de diagramas visuais. Especificamente, são necessários:

Tabela 2. Argumentos práticos relacionados à disponibilidade de poder computacional para suportar a captura e processamento de diagramas visuais, (Lyon at al., 1993)

Característica	**Justificativa**
Velocidade de Processamento	Permitir "tempo-real"
Gráfico de alta-resolução	Representar notações diagramáticas complexas
Entrada de Mouse	Criar notações diagramáticas complexas
Display baseado em Janelas	Para dividir os diagramas resultantes em tamanho gerenciável

Este último ponto é fundamental, já que dividir grandes programas para torná-los mais gerenciáveis é positivo, mas cria dificuldades de navegação. Estes tipos de problemas de navegação foram resolvidos, para documentos "ordinários", pelos sistemas de hipertexto. Agora, documentos de hipertexto "ordinários" são tediosos para serem criados porque adicionar todos os *hyperlinks* demanda tempo, mas não existe este problema com os programas, porque é fácil para o sistema de suporte de entrada gerar *hyperlinks* automaticamente, sob demanda. Tanto quanto prover aos programadores técnicas de navegação simples e consistentes, os *hyperlinks* podem ser usados para automaticamente atualizar informação compartilhada entre as visões do programa.

VPLs já conseguem fazer divisões baseadas em múltiplas janelas, com *hyperlinks* entre as janelas conectando os itens compartilhados de informação. Chamar de linguagens de hiperprogramação refletirá esta situação, e poderia reduzir a sugestão (inerente ao nome linguagens de programação visuais) de que estas linguagens deveriam evitar completamente o texto.

2.2.3 Nomenclatura Adotada

Neste trabalho, termos como VPL, Linguagem Visual de Programação, Gráficos Executáveis, Ambientes Não Textuais e Programação Icônica serão tratados como termos de significado equivalente.

2.3 Histórico

Os primeiros trabalhos em programação visual eram em duas direções: abordagens visuais para linguagens de programação tradicionais (como os "*flowcharts*" executáveis), e novas abordagens visuais para programação que se desviavam significativamente das abordagens tradicionais (como a programação a partir da demonstração das ações desejadas na tela). Muitos destes sistemas primordiais tiveram vantagens que pareciam instigantes e intuitivas quando demonstrados como "programas-brinquedo", mas encontraram problemas difíceis quando tentativas foram feitas para estendê-los para programas de tamanho mais realista. Estes problemas conduziram a um desencantamento inicial com a programação visual, levando muitos a acreditar que a programação visual era inerentemente inadequada para "trabalho real" – que era apenas para exercício acadêmico.

Para vencer estes problemas, os pesquisadores em programação visual começaram a desenvolver maneiras de usar a programação visual apenas para certas partes selecionadas do desenvolvimento de *software*, conseqüentemente aumentando o *número* de projetos nos quais a programação visual poderia auxiliar. Nesta abordagem, técnicas diretamente visuais foram amplamente incorporadas em ambientes de programação que suportam programação de linguagens textuais, para substituir o trabalhoso processo de especificação do *layout* da GUI (*Graphical User Interface* ou Interface Gráfica do Usuário), para suportar formulários eletrônicos de diagramas de engenharia de *software* para criar e/ou visualizar relações dentre as estruturas de dados, e para visualmente combinar unidades programadas em texto para construir novos programas. VPEs de sucesso comercial logo surgiram; dentre os primeiros exemplos estão o Visual Basic da Microsoft (para Basic) e o Visual Works da ParcPlace Systems (para Smalltalk). Um outro grupo de VPEs comerciais, focados primariamente em programação modular, são as

ferramentas de Engenharia de *Software* Assistida por Computador (do inglês Computer-Aided Software Engineering - CASE) que suportam especificação visual (por exemplo, usando diagramas) de relações entre os módulos do programa, culminando em geração automática de código-fonte.

Outros pesquisadores de programação visual tomaram uma abordagem diferente – eles trabalharam para aumentar os tipos de projetos adequados para a programação visual através do desenvolvimento de sistemas de programação visual de domínio específico. Sob esta estratégia, a adição de cada novo domínio suportado aumentou o número de projetos que podiam ser visualmente programados. Um benefício adicional que se seguiu foi aumentar a acessibilidade – usuários finais eram algumas vezes capazes de usar estes novos sistemas. Os desenvolvedores de VPLs e VPEs de domínio específico descobriram que proporcionando maneiras de escrever programas para um problema de domínio particular eliminava muitas das desvantagens encontradas nas abordagens iniciais, porque eles criaram suporte para trabalhar diretamente no estilo de comunicação do domínio do problema em particular – usando artefatos visuais (ícones e menus) que refletiam necessidades em particular, diagramas de solução de problemas e vocabulário específico àquele domínio – e nunca forçaram os usuários a abandonar aquele estilo de comunicação. Esta abordagem rapidamente produziu um número de sucessos tanto na pesquisa quanto no mercado. Hoje existem VPEs e VPLs comerciais disponíveis em muitos domínios; exemplos incluem programação laboratorial de aquisição de dados (LabVIEW, da National Instruments), programação de visualizações científicas (AVS, da Sistemas Visuais Avançados), programação de comportamento do telefone e do correio-de-voz (PhonePro, da Cypress Research), programação de simulações gráficas e jogos (Cocoa, da Stagecoach Software), programação de simulações matemáticas (MATLAB, da The MathWorks), programação de planilhas eletrônicas (Excel, da Microsoft Corporation). Um número de geradores de agentes de *software* está sendo incluído na computação pessoal também, permitindo "macros" que auxiliam com tarefas repetitivas a serem inferidas das manipulações realizadas pelo usuário final (como em Chimera, por exemplo). Nesta conjuntura, VPLs comerciais com as características necessárias para a programação de propósito geral surgiram e estão sendo utilizadas para produzir pacotes de *software* comerciais; um exemplo é o Prograph CPX, da Pictorius International.

2.4 Estratégias em Programação Visual

Um mal-entendido comum é que o objetivo da pesquisa em programação visual em geral e VPLs em particular é o de eliminar o texto completamente. Isto é uma falácia – na verdade, a maior parte das VPLs incluem texto de alguma forma, num contexto multidimensional. Em vez disso, o objetivo geral das VPLs é alcançar melhorias no projeto de linguagens de programação. A oportunidade de conseguir isto vem do simples fato de que VPLs têm menos restrições sintáticas na maneira pela qual um programa pode ser expresso (pelo ser humano ou pelo computador), e isto incentiva uma certa liberdade em explorar mecanismos de programação que não foram tentados antes porque eles não foram possíveis de serem implementados no passado.

Os objetivos específicos mais comuns procurados com a pesquisa em VPL têm sido (1) tornar a programação mais acessível para algum público em particular; (2) melhorar a capacidade de correção com a qual as pessoas realizam tarefas de programação; e/ou (3) aumentar a velocidade com a qual as pessoas realizam as tarefas de programação. Para atingir estes objetivos, Burnett lista quatro estratégias comuns usadas em VPLs (Burnett, 1999):

Concretness (concretude): *Concretness* é o oposto de abstração, e significa expressar alguns aspectos de um programa usando instâncias particulares. Um exemplo é permitir ao programador especificar alguns aspectos da semântica num objeto ou valor específico e um outro exemplo é ter o sistema automaticamente mostrando os efeitos de alguma porção do programa num objeto ou valor específico.

Directness (de maneira direta): *Directness* no contexto da manipulação direta é usualmente descrita como um "sentimento de quando se está manipulando o objeto" (Shneiderman, 1983). De um ponto de vista cognitivo, *directness* em computação significa uma distância pequena entre o objetivo e as ações requeridas do usuário para atingir este objetivo (Green e Petre, 1996), (Hutchins at al., 1986), (Nardi, 1993). Dado a concretude numa VPL, um exemplo de *directness* seria permitir ao programador manipular um objeto ou valor específico diretamente para especificar a semântica ao invés de descrever esta semântica textualmente.

Explicitness (explicitude): Algum aspecto da semântica é explícito no ambiente de programação se ele é diretamente mostrado (textualmente ou visualmente), sem a necessidade de que o programador o infira. Um exemplo de explicitude em VPL seria ao sistema

explicitamente descrever relações de fluxo de dados (informação de um pedaço de programa) através do desenho de arestas direcionadas dentre as variáveis relacionadas.

Immediate Visual Feedback (retorno visual imediato): No contexto da programação visual, retorno visual imediato se refere a mostrar automaticamente os efeitos da edição do programa. Tanimoto cunhou o termo *liveness* (termo não traduzido), que categoriza a imediatez do resultado semântico que é automaticamente provido durante o processo de edição do programa (Tanimoto, 1990). Tanimoto descreveu 4 níveis de *liveness*. No nível 1, nenhuma semântica é encarregada ao computador, e por esta razão nenhum retorno sobre o programa é proporcionado ao programador. Um exemplo de nível 1 é um diagrama entidade-relacionamento utilizado para documentação. No nível 2, o programador pode obter retorno semântico sobre uma porção do programa, mas isso não é proporcionado automaticamente. Os compiladores suportam o nível 2 de *liveness* minimamente, e os interpretadores fazem mais porque eles não estão restritos aos valores finais de saída do programa. No nível 3, o retorno semântico incremental é automaticamente proporcionado independentemente se o programador realiza uma edição incremental de programa, e todos os valores presentes na tela que são afetados são automaticamente re-exibidos. Isto garante a consistência no estado de exibição e do estado do sistema se o único causador de mudanças no estado do sistema é a edição do programa. A função de recálculo automático das planilhas eletrônicas suporta *liveness* nível 3. No nível 4, o sistema responde a edições do programa tal como o nível 3, e a outros eventos tais como marcação do relógio do sistema ou cliques de mouse ao longo do tempo, garantindo que todos os dados exibidos refletem acuradamente o estado corrente do sistema enquanto as computações continuam a evoluir.

As quatro estratégias previamente descritas podem ser avaliadas quando aplicadas a alguns exemplos de VPLs no Apêndice A.

2.5 Classificação das Linguagens de Programação Visual

Com o surgimento e a evolução das VPLs, muitas propostas distintas surgiram, o que gerou um esforço da comunidade em categorizá-las.

Marriot e Meyer afirmam que "houve muito pouca atenção em desenvolver uma hierarquia sistemática e compreensível de linguagens visuais baseada em suas propriedades

formais" (Marriot e Meyer, 1997). Golin e Reiss, por exemplo, dizem tão-somente que "linguagens de programação visuais podem ser classificadas, de acordo com o tipo e a extensão das expressões visuais utilizadas, em linguagens baseadas em ícones, linguagens baseadas em formulários e linguagens baseadas em diagramas" (Golin e Reiss, 1990). Idealmente, uma hierarquia taxonômica para linguagens visuais deveria ser geral o suficiente para incluir todos os tipos de formalismos de linguagens visuais. Isto, contudo, parece difícil de ser obtido se considerarmos as diferenças substanciais entre estes formalismos:

- Web and Array Grammars (Rosenfeld, 1976)
- Shape Grammars (Gips, 1974)
- Positional Grammars (Costagliola et al., 1993)
- Relation Grammars (Ferruci et al., 1994)
- Unification Grammars (Wittenburg e Weitzmann, 1990), (Wittenburg et al., 1991), (Wittenburg, 1993), (Wittenburg, 1998)
- Attributed Multiset Grammars (Golin e Reiss, 1989), (Golin, 1990)
- Constraint Multiset Grammars (Marriot, 1994)
- Graph Grammars (Courcelle, 1990), (Rekers e Schürr, 1997)
- Algebraic Approaches (Üsküdarli, 1994)
- Logic-Based Approaches (Helm e Marriot, 1991), (Meyer, 1992), (Haarslev, 1993)

Além do tipo de gramática, Marriot e Meyer listam uma série de questões complementares cujas respostas especificam uma linguagem:

1. Quais são os objetos atômicos (básicos) para se trabalhar? (Alguns formalismos usam apenas linhas e pontos como objetos atômicos, enquanto outros permitem símbolos complexos quase arbitrários, o que significa que os limites de reconhecimento entre o léxico e o sintático não são fixos).

2. Estes objetos são sem atributos ou eles podem ter atributos, sub-objetos etc.? (Muitos formalismos não usam atributos de maneira definitiva, enquanto outros permitem atributos complexos arbitrários ou às vezes até mesmo estruturas aninhadas de objetos).

3. Se os atributos são permitidos, que tipo de computação sobre estes atributos é permitida? Às vezes nenhum tipo de computação é permitida, enquanto outros formalismos permitem o uso de qualquer função para calcular os valores.

4. Como as relações espaciais são estabelecidas? Em muitos formalismos elas são explicitamente dadas como relações abstratas não interpretadas, enquanto outros métodos usam relações implícitas que são expressas como condições nos valores de atributos.

5. As relações espaciais são interpretadas? Uma interpretação de uma relação espacial pode ser tanto simbólica/aritmética (em termos de geometria subjacente) ou as relações podem ser completamente não interpretadas. Interpretar relações espaciais permite-nos expressar axiomas gerais de geometria (por exemplo, transitividade de inclusão espacial) e então enriquecer a expressividade do *framework*.

6. Qual é a estrutura matemática geral usada para representar uma expressão visual? Em muitas abordagens, os objetos gráficos são agregados como conjuntos arbitrários de subconjuntos, enquanto outros métodos usam algum "fixador" do espaço (normalmente estruturas do tipo grade) nas quais os objetos têm que ser arranjados.

As respostas a essas perguntas influenciam profundamente o poder expressivo e a complexidade computacional do método de especificação resultante.

Costagliola propôs um sistema próprio de classificação, baseado em duas classes principais: baseadas em geometria e baseadas em conexões (Costagliola et al., 2002). Marriot e Meyer usaram a abordagem CMG (Constraint Multset Grammars) (Marriot, 1994) – seção 2.7 para maiores explicações - para derivar uma taxonomia semelhante à de Chomsky para VPLs (Marriot e Meyer, 1997). Para mostrar que a generalidade da taxonomia não é dependente em suas raízes no CMG, eles também mostraram como muitos dos outros formalismos podem ser mapeados para CMGs.

Burnett e Baker sugerem um sistema (Tabela 3) para a organização de publicações científicas, mas que também pode ser aplicado diretamente as VPLs (Burnett e Baker, 1994).

Tabela 3. O Sistema de Classificação para Linguagens de Programação Visual – extraído de (Burnett e Baker, 1994)

<table>
<tr>
<td>VPL – Linguagens de Programação Visual
VPL-I – Ambientes e Ferramentas para VPLs
VPL-II – Classificações das Linguagens
A. Paradigmas
1. Linguagens concorrentes
2. Linguagens baseadas em restrições
3. Linguagens orientadas a fluxo-de-dados
4. Linguagens baseadas em formulários e planilhas
5. Linguagens funcionais
6. Linguagens Imperativas
7. Linguagens lógicas
8. Linguagens multiparadigmas
9. Linguagens orientadas a objetos
10. Linguagens de programação por demonstração
11. Linguagens baseadas em regras
B. Representações visuais
1. Linguagens Diagramáticas
2. Linguagens Icônicas
3. Linguagens Baseadas em Seqüências Pictóricas Estáticas
VPL-III – Características das Linguagens
A. Abstração
1. Abstração de dados
2. Abstração de rotinas
B. Controle de fluxo
C. Estruturas e tipos de dados
D. Documentação
E. Tratamento de eventos
F. Tratamento de exceções</td>
<td>VPL-IV – Tópicos em Implementação da Linguagem
A. Abordagens computacionais
B. Eficiência
C. Parsing
D. Tradutores (interpretadores e compiladores)
VPL-V – Propósito da Linguagem
A. Linguagens de propósito geral
B. Linguagens de bancos de dados
C. Linguagens de processamento de imagens
D. Linguagens de visualização científica
E. Linguagens de geração de interface de usuário
VPL-VI – Teoria das VPLs
A. Definição formal das VPLs
B. Teoria dos ícones
C. Tópicos em projeto de linguagem
1. Tópicos em cognição e projeto de interface com o usuário (e.g., estudos de usabilidade, percepção gráfica)
2. Uso efetivo do estado real da tela de computador
3. Liveness
4. Escopo
5. Checagem de tipos e teoria de tipos
6. Tópicos em representação visual (e.g. representação estática, animação)</td>
</tr>
</table>

2.6 Programação Visual e Abstração

Um dos desafios na pesquisa em programação visual é crescer em escala a ponto de suportar programas muito grandes. Esse é um assunto mais importante para as VPLs que para

as linguagens textuais tradicionais (embora certamente possa ser dito que exista em ambas) por razões relacionadas à representação, projeto de linguagem e implementação, e por ser uma área relativamente nova. Por exemplo, alguns dos mecanismos visuais usados para atingir características tais como explicitude podem ocupar um grande desafio de espaço, tornando difícil manter o contexto. Também, é difícil aplicar de uma maneira direta técnicas desenvolvidas para linguagens tradicionais, porque fazer isso freqüentemente resulta numa re-introdução das muitas complexidades que as VPLs vêm tentando remover ou simplificar.

Desenvolvimentos recentes na área de abstração têm sido particularmente importantes para a escalabilidade das VPLs. Os dois tipos mais amplamente suportados de abstração, tanto num contexto visual quanto textual, são a abstração de rotinas e a abstração de dados. Em particular, a abstração de rotinas tem se mostrado capaz de ser suportada por uma variedade de VPLs. Um atributo-chave para suportar a abstração de rotinas numa VPL tem sido a consistência com o restante da programação numa mesma VPL. Soluções representativas (ver Apêndice A) incluem permitir ao programador selecionar, nomear e iconizar uma seção de um grafo de fluxo de dados (Apêndice A - Figura A.4), o que adiciona um nó representando o tal subgrafo a uma biblioteca de nós de funções numa linguagem de fluxo de dados; fazer diferentes planilhas eletrônicas (Apêndice A - Figura A.2), que podem ser generalizadas automaticamente para permitir rotinas definidas pelo usuário numa linguagem baseada em formulário; e gravar e generalizar uma seqüência de manipulações diretas (Apêndice A - Figura A.1) numa linguagem por demonstração.

A abstração de dados tem sido implantada de forma mais lenta nas VPLs, principalmente porque é às vezes difícil achar uma maneira de manter as características de *concretness* e retorno visual imediato enquanto se adiciona suporte para idéias centrais para a abstração de dados tais como generalização e encapsulamento de informações. Mesmo assim, o suporte para a abstração de dados surgiu em algumas VPLs. Por exemplo, em Forms/3 (ver Apêndice A), um novo tipo de dados é definido por meio de uma planilha eletrônica, com as células ordinárias definindo operações e métodos e com duas células distintas que permitem a composição de objetos complexos a partir de objetos mais simples e a definição de como um objeto deveria aparecer na tela. Em Cocoa (ver Apêndice A), a aparência de cada personagem é pintada usando-se um editor gráfico e cada demonstração de uma nova regra pertence ao tipo de personagem sendo manipulado, proporcionando de maneira robusta a funcionalidade de

uma operação ou método. Ambos Forms/3 e Cocoa também suportam formas limitadas de herança.

2.7 Especificação de VPLs

A unidimensionalidade de linguagens textuais tradicionais significa que existe apenas uma única relação possível entre símbolos numa sentença, "próximo a". Assim, numa frase como "Linguagem visual é importante", o símbolo (ou a palavra) "Linguagem" está *próximo ao* símbolo "visual", que por sua vez está *próximo ao* símbolo "é", a mesma relação valendo para "importante", logo a seguir. Pelo fato de haver somente um tipo de relação possível, a parte da especificação de uma linguagem que indica a relação entre os símbolos pode ser omitida. É o que acontece, por exemplo, com a notação Backus-Naur (BNF – trata-se de uma maneira formal de se descrever linguagens formais). Portanto, ao se descrever uma linguagem textual em BNF (Backus–Naur form), é necessário especificar apenas os símbolos na linguagem, não a relação "próximo a" (que acontece quando um símbolo é escrito próximo à outro na gramática; esta relação é implícita na notação). Uma especificação BNF é um conjunto de regras de derivação escritas como:

<símbolo> ::= <expressão com símbolos>

onde <símbolo> é um não-terminal (que não finaliza a produção), e a expressão consiste de uma seqüência de símbolos e/ou seqüências separadas por uma barra vertical, '|', indicando uma escolha, uma possível substituição para o símbolo à esquerda. Símbolos que nunca aparecem à esquerda são terminais (que finalizam uma produção). Como um exemplo, considere esta BNF para um endereço postal dos EUA, composta pelas seguintes quatro regras:

(1) <endereço postal> ::= <nome> <endereço> <código-postal>

(2) <nome> ::= <nome> <sobrenome> <EOL> | <nome>

(3) <endereço> ::= [<apt>] <número> <endereço> <EOL>

(4) <código-postal> ::= <nome-de-cidade> "," <código-postal> <EOL>

Isto pode ser traduzido como:

(1) Um endereço postal consiste de um nome, seguido por um endereço, seguido por um código-postal.

(2) Um nome consiste de: um nome seguido de um sobrenome seguido de um marcador de fim-de-linha (<EOL>), ou um nome (esta regra ilustra o uso da recursão nas BNFs, cobrindo o caso de pessoas que usam múltiplos nomes e/ou sobrenomes).

(3) Um endereço consiste de um especificador de apartamento opcional, seguido de um número de casa, seguido de um nome de rua, seguido de um marcador de fim-de-linha (<EOL>).

(4) Um código-postal consiste de um nome-de-cidade, seguido por uma vírgula, seguido por um código-postal, seguido por um marcador de fim-de-linha (<EOL>).

(Note que o especificador de apartamento, por exemplo, não está especificado à esquerda. Se necessário, ele deve ser descrito usando regras BNF adicionais ou deixado como abstração se puder ser considerado irrelevante para o propósito. Isto vale para todos os símbolos não especificados à esquerda).

Podemos observar que a BNF é adequada para descrevermos linguagens textuais unidimensionais. Contudo, a multidimensionalidade das VPLs significa que muitas relações são possíveis, tais como "sobrepõe-se a", "toca" e "à esquerda de", e não há definição universalmente aceita de exatamente quando tais relações devem ser usadas, ou até mesmo quantas delas podem ser aplicadas entre os mesmos símbolos. Por causa disso, as relações entre os símbolos não podem ser deixadas implícitas, e mecanismos tradicionais tais como BNF para a especificação de linguagens textuais não podem ser usados sem modificação para a especificação de VPLs.

Muitos formalismos diferentes para a especificação de linguagens visuais têm sido investigados. Formalismos do tipo gramaticais variam de abordagens como gramática da web e gramática de vetores a formalismos recentes como gramática posicional, gramática relacional, gramática de unificação, gramática *attributed multiset*, e muitos tipos de gramáticas de grafos. Existem também formalismos de tipo não-gramatical. Uma abordagem gramatical é *constraint multiset grammars* (CMGs) (Marriot e Meyer, 1997). CMGs foram formalmente introduzidos por Helm (Helm et al., 1991) e têm sido utilizada para uma variedade de tarefas complexas tais

como *parsing* de diagramas de estados e de equações matemáticas. Um tratamento formal preciso é dado por Marriot (Marriot, 1994) e serão apresentadas apenas noções básicas nesta seção. Um exemplo de uma produção da CMG retirada da especificação de um diagrama de estado é:

TR:transição ::= A:aresta, T: texto
onde existe R: estado, S: estado onde
T.ponto-médio perto_de A.ponto-médio.
R.raio = distância (A.ponto-inicial,R.ponto-médio).
S.raio = distância (A.ponto-final, S.ponto-médio)
E TR.de = R.nome, TR.para = S.nome, TR.rótulo = T.*string*

Esta produção define que uma transição num diagrama de estado consiste de uma aresta A apontando de algum estado R para algum outro estado S e sendo rotulada por um objeto de texto T. Como saber se o estado R e o estado S estão conectados pela aresta? As condições geométricas que a aresta A deve atender para que seja considerada como conector os dois estados (R e S) são determinadas como expressões aritméticas usando, nessa produção, os componentes *distância* e *perto_de*. Finalmente, a última linha de produção define os atributos desse objeto *transição* TR definido.

***Parsing* Visual**

A eficiência de compiladores para linguagens textuais de programação depende dos métodos de *parsing* empregados, e também de técnicas de programação tradicionais tais como a implementação da tabela de símbolos. Na presença de programas gráficos, contudo, as questões quanto à eficiência são provavelmente diferentes. Por exemplo, o *parsing* convencional é eliminado uma vez que a hierarquia estrutural do programa é, com freqüência, explicitamente representada no computador como resultado do processo de edição interativa. Na verdade, o ato de compilar um programa pode ser tomado em um significado diferente no contexto icônico. Normalmente, entende-se que a frase *compilar um programa* significa a produção de uma representação alternativa mais rápida que a versão interpretada. No contexto de programas textuais, isto pode envolver otimização pela substituição de subprogramas mais eficientes, e, no

caso da programação gráfica, isso pode significar a simplificação, estilização ou eliminação de animações associadas ao programa (Glinert, 1990).

2.8 Programação Visual e Aspectos Cognitivos

De acordo com Glinert, "para atingir o objetivo de ambientes não-textuais amigáveis, um novo tipo de integração de *hardware* e *software* é necessário, o qual amarre projeto gráfico, processamento de imagem, bases de dados pictóricas, técnicas de programação e animação de *hardware* em conjunto" (Glinert, 1990). Portanto, existem três categorias principais de tópicos que devem ser abordados, se quisermos criar bons ambientes não textuais.

1. Modelar o entendimento das pessoas sobre as idéias de programação, e também identificar estas qualidades que fazem os sistemas de programação icônica fáceis de aprender.

2. A representação visual de objetos de computação e o projeto de linguagens visuais para interação homem-computador.

3. Problemas de implementação que podem limitar a eficiência de sistemas visuais em áreas tais como detalhamento (granularidade) e características gráficas do monitor do computador, espaço necessário para armazenar versões animadas dos programas e velocidade de ferramentas gráficas.

Tais tópicos serão discutidos ao longo do texto.

Facilidade de aprender em ambientes de programação:

- Compreensibilidade do Sistema: Os resultados de Mayer indicam que novatos aprendem programação de computadores muito mais facilmente quando modelos físicos ou mecânicos para a computação são sugeridos do que quando tais analogias não estão presentes (Mayer, 1981). Isto sustenta a noção de que uma linguagem gráfica de programação, para ser fácil de aprender, deveria trazer tais modelos físicos ou mecânicos à mente. Na verdade, as maneiras pelas quais mecanismos físicos dão origem a modelos mentais não é trivial (De Kleer, 1980). Sheil apresenta uma explicação de porque o impacto desta adoção tem sido menor do que o esperado (Sheil, 1981). De acordo com Malone, o que torna os jogos de computador divertidos é um casamento entre uma metáfora física aliada à estrutura do jogo (Malone, 1980).

São os projetistas de computador que estão agora mais preocupados com este tópico, embora tenha havido alguns estudos significativos de cientistas behavioristas (Pane e Myers, 2002), (Pane, 2002), (Smith, 1975).

- Incorporar Camadas de Aprendizagem: Piaget (Hockenbury e Hockenbury, 2003) observou um padrão de vários estágios, ou níveis de aquisição, no desenvolvimento intelectual de uma criança. É plausível afirmar que, com exposição adequada a vários jogos de programação de computadores que estiverem organizados numa seqüência didática, as crianças poderiam vir a entender conceitos de programação de computadores ainda com pouca idade. Pane e Myers realizaram um estudo com a utilização de um programa especialmente projetado para crianças (Pane e Myers, 2002). Neste estudo, ele conclui que há evidências de que a usabilidade de programas pode ser melhorada a partir de técnicas visuais de busca, agregação e visibilidade de dados (Pane, 2002). Para ele, a tarefa de programar é uma atividade notoriamente difícil de ser realizada, e algumas destas dificuldades podem ser atribuídas à interface com o usuário ao invés de outros fatores, pois, historicamente, os projetistas de ferramentas e linguagens de programação nunca enfatizaram a usabilidade. Neste contexto, um ambiente de programação "neopiagetiano" incorporaria *camadas conceituais* adequadas, projetadas para permitir aos iniciantes tornarem-se efetivamente programadores através de um processo direto de aprendizagem.

Linguagens Visuais para a Computação:

- Metáforas para a computação: Para o novato, o maior problema com a computação é que ela normalmente parece muito abstrata. Para um modelo visual ser imediatamente absorvido, ele deve basear-se em alguma analogia com algum sistema ou experiência que já é bem entendida. Criar tal modelo requer a escolha de um fenômeno análogo, escolher quais aspectos do fenômeno devem fazer parte do modelo, e, finalmente, estabelecer todas as correspondências conceituais necessárias entre o modelo e o fenômeno escolhido.

- Partes de uma linguagem visual de programação: Ao se projetar um sistema de programação gráfica, o objetivo maior é ter um sistema integrado cujos símbolos pictóricos tenham significados consistentes com a natureza da programação. Um paradigma baseado em

encanamentos poderia ser estendido para lidar com uma abordagem de fluxo de dados no processamento da informação.

- Fluxo de controle e de dados: A descrição de controle de fluxo para programas tem sido realizada para diagramas de fluxo. Ilustrar as possíveis seqüências de execução de instrução dessa maneira parece ser natural. A analogia entre uma máquina de Von Neumann (arquitetura atual dos computadores pessoais) executando suas instruções, e um objeto inanimado percorrendo um sistema de rodovias, avenidas e ruas, é capaz de relatar a execução invisível de um programa para um tipo de processo para qual os seres humanos desenvolveram intuição desde a idade em que aprenderam a engatinhar.

- Operações que são descritas por ícones: Ícones deveriam visualmente sugerir o que eles denotam e devem apresentar analogias plausíveis em termos de compreensão e facilidade de serem lembrados. Isso deve acontecer numa maneira similar aos hieróglifos egípcios que envolviam apenas uma semelhança próxima à forma do objeto que inspirou o desenho, para trazer o conceito denotado à mente no contexto normal do uso de um símbolo. Resolução e complexidade de animação são pontos fundamentais neste tópico.

- Exibição gráfica de dados: A exibição de dados geométricos é inerentemente gráfica, porquanto há gráficos em barra, em pizza e muitos outros métodos avaliados (McClearly Jr., 1983). Dados textuais criam um outro problema, mas a formatação, o uso de janelas e de figuras estilizadas de documentos podem ajudar. Embora muitas técnicas existam para a exibição gráfica de dados, cada novo problema traz suas próprias peculiaridades. Este é um aspecto dos ambientes de programação que não se pode ignorar. Problemas que demandam atenção em particular são: (1) a exibição de números inteiros e reais usando idéias tais como caixas, réguas e analogias tais como o nível d'água num recipiente; (2) o uso de contadores e gráficos para os dados numéricos; (3) o uso de mapas para dados bidimensionais; (4) o uso de vetores e árvores para bases de dados; (5) o uso de coleções de figuras diminutas de vários objetos para representar conjuntos.

Interação Homem-Computador

Um outro componente de um sistema de programação gráfica é a interface ou os meios pelos quais os usuários controlam as partes dos programas que eles vêem, editam, mudam o modo de exibição, e assim por diante. Uma interface adaptativa é aquela que, inicialmente,

apresenta um sistema que é *fácil de aprender* e, quando o usuário vai ganhando familiaridade com ele, o sistema se altera para tornar-se mais *fácil de usar*. Para exemplificar os possíveis pontos a se ponderar numa decisão de projeto, há o exemplo do "menu". Sistemas baseados em menu são geralmente fáceis de aprender, porque o usuário vê as opções disponíveis listadas explicitamente. Por um lado, um menu não deve apresentar muitas opções simultaneamente, senão se tornará confuso. Por outro lado menus curtos podem resultar em interação burocrática uma vez que um simples comando pode requerer muitas seleções de menu.

O tópico de interação homem-máquina tem avançado muito nos últimos anos (Shneiderman, 1998). Foge ao escopo desta dissertação aprofundar-se nesse tema, visto que é um universo vasto por si só e não constitui foco do trabalho; tratar-se-á principalmente dos aspectos cognitivos deste tema.

Considerando que os objetivos das VPLs estão relacionados à melhorar a habilidade humana em programar, é importante considerar o que é conhecido sobre tópicos cognitivos relevantes à programação. Muito desta informação foi coletada no campo da psicologia cognitiva, e o psicólogo Thomas Green e seus colegas tornaram muitas destas descobertas disponíveis aos não-psicólogos a partir das "Dimensões Cognitivas" (Green e Petre, 1996), um conjunto de termos descrevendo a estrutura dos componentes de uma linguagem de programação tal como eles se relacionam aos tópicos cognitivos em programação.

A Tabela 4 lista as dimensões com suas respectivas descrições. A relação de cada dimensão a um número de estudos empíricos e princípios psicológicos é dada no estudo de Green e Petre, mas os autores também cuidadosamente apontam as falhas neste conjunto de evidências. Em suas palavras, "o esqueleto das dimensões cognitivas consiste num número pequeno de termos que foram escolhidos para serem de fácil compreensão para não-especialistas, enquanto não deixam de capturar uma quantidade significativa de psicologia e de interface homem-máquina na programação" (Green e Petre, 1996).

Uma aplicação concreta das dimensões cognitivas é a representação de projetos de *benchmark* (Yang et al., 1997), um conjunto de medidas quantificáveis que pode ser feito numa representação estática de uma VPL.

Tabela 4. As dimensões cognitivas

Dimensão	**Questões pertinentes**
Gradiente de abstração	Quais são os níveis máximos e mínimos de abstração? Os fragmentos podem ser encapsulados?
Consistência	Quando alguma parte da linguagem foi aprendida, quanto do restante pode ser inferido?
Difusão	Quantas entidades gráficas ou simbólicas são necessárias para expressar um significado?
Propensão a erro	O projeto da notação induz a "erros sem atenção"?
Operações mentais difíceis	Existem lugares onde o usuário precisa apelar às anotações à mão para não perder de vista o que está acontecendo?
Dependências escondidas	Toda dependência é explicitamente indicada nas duas direções? A indicação é perceptual ou apenas simbólica?
Compromisso prematuro	Os programadores têm que tomar decisões antes que eles tenham a informação de que precisam
Avaliação progressiva	Um programa parcialmente feito pode ser executado para obter um retorno do tipo "como estou indo"?
Expressividade organizacional	O programador pode compreender como cada componente de um programa se relaciona com o conjunto total?
Notação secundária	Os programadores podem usar desenhos, cores e outras pistas para transmitir significado adicional, além da semântica oficial da linguagem?
Viscosidade	Quanto de esforço é necessário para se realizar uma única mudança?
Visibilidade	Todas as partes do código são simultaneamente visíveis (supondo um visor grande o suficiente) ou é pelos menos possível comparar duas partes do código lado-a-lado à vontade? Se o código estiver disperso, é pelo menos possível saber em que ordem lê-lo?

2.9 Prós e Contras

Muitas bandeiras estão tremulando ao redor da estratégia de projeto de Linguagens Visuais. A mais conhecida – 'Uma imagem vale mil palavras' – apela para a sinteticidade e universalidade. Mas seguir bandeiras pode conduzir à falhas. Por exemplo, alguns psicólogos sugerem que as imagens são intrinsecamente ambíguas e que, por esta razão, até mesmo ícones bem estabelecidos são apenas entendidos adequadamente por membros que compartilham de uma mesma cultura. Então, quais 'mil palavras' fazem leitores diferentes associarem com uma

imagem em particular? E dois comunicantes de diferentes culturas podem esperar univocamente entenderem-se mutuamente por imagens visuais? (Mussio, 1993).

Deste ponto de vista, parece que a teoria das linguagens visuais ainda está despontando. Alguns marcos iniciais foram estabelecidos (Golin, 1991), mas ainda não existe definição universalmente aceita de uma linguagem visual, especialmente quando se fala da metodologia de atribuir significado a uma imagem.

Meyer acredita que a programação puramente visual é importante por uma série de razões (Meyer, 1995):

1. Pode fazer da programação uma atividade mais fácil e menos sujeita a erros;

2. Pode auxiliar programadores ou outros a apreciar mais a programação, a aprender a criar padrões complexos;

3. Pode permitir que alguns não programadores criem seus próprios programas modestos para completar suas tarefas; tanto quanto *Unix pipes* e redirecionamento permitem aos não programadores utilizarem os comandos do *Unix* de maneira muito flexível.

4. Necessidade de motivação estética não pode ser ignorada uma vez que ela aciona a imaginação e estimula o intelecto, permitindo potencializar o visual humano e permitir a ele se conectar a um sentido visual concernente à atividade de programação;

5. Razões pedagógicas.

Mas também aponta problemas. "O problema real é como acessar qualquer e todos os dados de que precisamos. Até mesmo algoritmos simples provam serem muito difíceis se há muitas variáveis. Outro é o problema quando se tem muito *software*(escalabilidade)"

Na verdade, algumas pessoas respeitadas não vêem nada de agradável na representação visual de *software*. Num artigo famoso, (Brooks Jr, 1987) diz que:

"Um tópico favorito de dissertações em Engenharia de *Software* é programação gráfica ou visual – a aplicação de gráficos de computadores no projeto de *software*...nada nem um pouco convincente, muito menos excitante, surgiu de tais esforços. E estou convencido de que nada surgirá".

O'Brien adverte "...seja precavido quanto ao que diz a programação visual. Desenhar linhas entre objetos tornou-se desconcertantemente estilo *web*. A programação puramente visual ainda não é e talvez nunca será viável (O'Brien, 1993)".

A diferença entre linguagens tradicionais baseadas em texto e linguagens visuais se concentra no uso de figuras e diagramas, em vez de cadeias de palavras, para expressar

significado. A vantagem primária potencial da utilização da representação visual em vez da representação textual é a possibilidade da imediatez do entendimento do ponto de vista de um observador não treinado. É bem sabido, por um número grande, senão a maioria, das pessoas, que os filmes são capazes de prover uma experiência mais imediata do que os livros correspondentes (sobre os quais aqueles são baseados).

Contudo, para manter a vantagem da imediatez e da obviedade, uma linguagem visual de computação deve ser apropriadamente projetada. Além disto, passos devem ser tomados para vencer um problema em potencial com linguagens visuais, que é a ineficiência na representação. A necessidade de explicitude visual pode, num primeiro momento, parecer inconsistente com a necessidade de concisão. Deve haver representações alternativas de objetos computacionais em diferentes níveis de explicitude.

2.9.1 Evidências Empíricas

As duas últimas décadas têm testemunhado a emergência de uma comunidade de pesquisa em programação visual cujos esforços produziram muitas VPLs e sistemas de visualização. Entretanto, há poucos estudos empíricos corroborando as decisões de projeto nestes sistemas de visualização e VPLs. Surgem conseqüências desta falta de evidências. Por exemplo, críticos e uma parte da comunidade de ciência de computação estão mais inclinados a desmerecer a programação visual como um campo de pesquisa sem futuro. Além disto, os pesquisadores de programação visual são forçados a tomar decisões de projeto sem dados que os orientem. Considerando a situação não comprovada da programação visual, estudos empíricos poderiam contribuir com estes esforços. O problema da falta de garantias de evidência empírica tem sido considerado o maior problema em aberto na pesquisa em programação visual. O problema da falta de evidências também afeta a questão da escalabilidade das VPLs (Burnett et al., 1995).

O trabalho em direção ao uso de técnicas de programação visual para melhorar a corretude e/ou velocidade nas tarefas de programação focou primariamente em três áreas: compreensão do programa, criação do programa e procura por e correção de erros no programa. Destas três áreas, a maior parte dos estudos empíricos foi feita nos efeitos da VPL na compreensão do programa. Veja detalhes no Apêndice B (Whitley, 1997). Os resultados

destes trabalhos têm sido mistos, reportando descobertas para alguns tipos de programas ou públicos-alvo nas quais VPLs ou notações visuais são conectadas com grande compreensão, e outros nas quais linguagens estritamente textuais e/ou notações têm sido conectadas com grande compreensão.

Têm existido poucos estudos empíricos na criação de programas até agora, mas estes estudos têm produzido resultados mais consistentes que os estudos de compreensão. A maior parte tem demonstrado as abordagens visuais sobrepujando abordagens textuais tradicionais neste aspecto (Baroth e Hartsough, 1995), (Burnett e Gottfried, 1998), (Modugno et al., 1996), (Pandley e Burnett, 1993).

Por último, os efeitos da programação visual são os menos estudados de todos na procura por e correção de erros de programa. Estes estudos não encontraram melhorias estatisticamente significativas para todos os aspectos estudados, mas, para os aspectos nos quais significância estatística foi encontrada, as abordagens visuais, incluindo retorno imediato, foram consideradas superiores às abordagens estáticas não orientadas ao retorno imediato, na maioria dos casos (Cook et al., 1997), (Green e Petre, 1996).

3. A Linguagem VisualPREV e seu Ambiente de Desenvolvimento

3.1 Introdução e Histórico

A motivação de se criar a linguagem VisualPREV começou a partir de dificuldades descritas no Capítulo 1. A experiência no desenvolvimento de modelos de previsão de séries temporais mostra que um aspecto crítico é a necessidade de testar diferentes modelagens e técnicas. Isto exige em geral um maior tempo de desenvolvimento e acaba por limitar o escopo e a quantidade de testes. A disponibilidade de um sistema com flexibilidade para facilitar a configuração e execução de diferentes modelos, certamente será de grande utilidade para o desenvolvimento de modelos de previsão de séries temporais.

Primeiramente, foi esboçado um Sistema de Modelagem para a Previsão de Séries Temporais e o Reconhecimento de Padrões (SMPR). Quando passou-se a priorizar os aspectos de flexibilidade e portabilidade, e evolução natural foi a criação da VisualPREV, linguagem de programação visual que será apresentada nas próximas seções. Em suma, a linguagem VisualPREV possui um ambiente de modelagem visando aplicações em previsão de séries temporais e reconhecimento de padrões, baseado em recursos da área de Programação Visual. Com este ambiente pretende-se criar facilidades para a configuração de modelos, utilizando diferentes técnicas, tais como redes neurais, redes neuro-*fuzzy*, modelos estatísticos, modelos baseados em *Box-Jenkins*, etc. Atualmente, estes modelos são, em geral, implementados em linguagem procedural, requerendo um longo processo de desenvolvimento.

A partir da criação de um documento de visão do problema, iniciou-se a tarefa de discutir o escopo e alvo deste projeto de *software*. Também foram realizados diversos testes no intuito de estabelecer quais seriam as tecnologias mais apropriadas de programação, e uma conclusão importante foi a de que a linguagem Java é estratégica para o funcionamento do sistema pelo fato de ser multiplataforma e orientada a objetos.

O projeto foi realizado partindo-se do usuário: quem é ele(a)? Como vê o problema? Como lida atualmente com o problema? Como seria uma ferramenta "ideal" para se lidar com estes modelos?

3.2 VisualPREV: Ambiente e Linguagem

A literatura e as experiências em previsão de séries temporais e reconhecimento de padrões têm mostrado que não há uma única técnica que seja a mais adequada para todos os casos. Embora as redes neurais tenham sido muito exploradas nos últimos anos, nem sempre são a melhor ferramenta. Em muitas aplicações, os tradicionais modelos estatísticos tais como regressão linear e modelos auto-regressivos têm apresentado desempenhos superiores. Infelizmente, a determinação da melhor alternativa é quase sempre baseada na tentativa e erro, exigindo quase sempre o teste de um número muito grande de diferentes técnicas, configurações e arquiteturas. Usualmente, a implementação destes modelos em linguagem procedural pode exigir um longo tempo de implementação.

Já há vários pacotes que podem ser utilizados para a previsão de séries temporais, tanto os baseados em modelos estatísticos, como é o caso do Statistical Analisys System (SAS) e outros pacotes similares, e os baseados em redes neurais e redes neuro-*fuzzy*, como é o caso do MATLAB. Estes programas, embora eficientes para as aplicações previstas, apresentam dificuldades de integrar diferentes modelos. Uma outra dificuldade é em relação à utilização desses pacotes, os quais requerem um conhecimento mais aprofundado nessas técnicas.

O objetivo da VisualPREV é possuir um ambiente em que seja fácil utilizar diferentes modelos sem a necessidade de muito treinamento. Por exemplo, para um determinado usuário de uma companhia de energia elétrica que queira estimar quanto será o pico de consumo de energia elétrica em um dado dia através de um modelo de redes neurais *multilayer perceptron* (abreviadas como MLP a partir deste ponto do texto), o mesmo poderia fazê-lo sabendo muito pouco sobre redes neurais. Na linguagem VisualPREV, o usuário precisaria apenas especificar quais seriam os conjuntos de treinamento, validação e teste, e configurar a rede neural. Todo o processo de treinamento, validação, teste e apresentação dos resultados seriam automaticamente realizados pelo sistema.

O ambiente da VisualPREV apresenta uma tela de modelagem, na qual é possível especificar os dados a serem utilizados num dado modelo, as ferramentas (técnicas) e a apresentação dos resultados. A Figura 2 mostra um exemplo de edição de um modelo de aplicação de uma RNA para previsão de séries temporais. Neste exemplo, são especificados os dados (blocos verdes), representando os conjuntos de treinamento, validação e teste; a ferramenta (bloco cinza), que no caso é uma RNA; e é especificado um arquivo de saída dos

resultados (bloco azul). Aqui entende-se como modelo a definição do conjunto de entradas, as técnicas ou ferramentas a serem utilizadas e o formato de saída dos resultados. Por técnica entende-se a metodologia a ser utilizada pelo modelo, tais como RNA MLP, RNA Kohonen, modelos auto-regressivos, modelos baseados em regressão linear, etc.

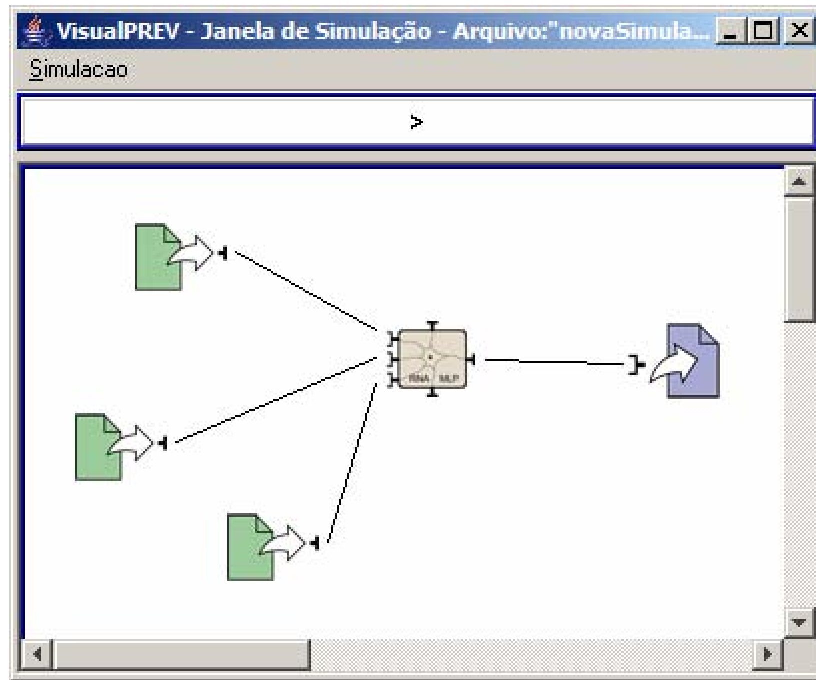


Figura 2. Tela de edição de um modelo de previsão de séries temporais.

3.3 Especificação

A idéia de linguagens de domínio específico existe desde que as primeiras linguagens de programação foram projetadas (Lohr, 2001); na verdade, este fato provavelmente contribui com a proliferação primordial das linguagens de programação. Muitos anos depois, a idéia que guia o desenvolvimento das linguagens permanece próxima àquela que emergiu com as primeiras linguagens: uma abstração aprimorada do problema permite a rápida criação e manutenção de uma aplicação complexa. A *linguagem visual de domínio específico* (do inglês *domain-specific visual language* - DSVL) é um tipo especial de linguagem de domínio específico que provê

uma interface gráfica de programação. As vantagens de uma interface visual são muitas, mas, em particular, uma DSVL permite a algum especialista de domínio e também um usuário de computador capacitado (que não seja um desenvolvedor de computador) a usar a linguagem visual como uma ferramenta de desenvolvimento de aplicação para o domínio. Como tal, a DSVL é tradicionalmente uma linguagem restritiva que consiste apenas de conceitos do domínio, e também de regras bem-definidas que restringem os programas do domínio criados. Uma linguagem restritiva provê ao usuário da linguagem apenas aquelas associações e objetos que são relevantes no domínio e permite suas associações apenas quando aquela associação tem um significado bem-definido (Sprinkle, 2004).

Anos recentes têm mostrado um crescimento explosivo da computação para usuários finais. De fato, em 2005 havia 55 milhões de programadores do tipo usuário-final apenas nos Estados Unidos e um número estimado de 2,75 milhões de programadores profissionais (Boehm et al., 2000). Exemplos reais de ambientes de programação destinados a usuários-finais incluem construtores de simulações educacionais, sistemas de criação de websites, sistemas de criação multimídia, sistemas de filtragem de e-mails por regra, sistemas CAD, sistemas de visualização científica e planilhas eletrônicas. Todos estes sistemas fazem uso de pelo menos algumas técnicas de programação visual no intuito de dar suporte a seus usuários, tais como sintaxe gráfica, "arrastar e colar" para especificar aparências ou resultados desejados, programação por demonstração, e/ou retorno visual imediato sobre a semântica do programa (Ruthruff et al., 2005).

A Linguagem VisualPREV tem como domínio a área de previsão de séries temporais e reconhecimento de padrões, sendo, portanto, uma linguagem de domínio específico destinada à usuários-finais. A Tabela 5 resume as características gerais da Linguagem VisualPREV, e também frente a alguns dos conceitos explorados no Capítulo 2.

Tabela 5. Características da linguagem VisualPREV

Característica	**Atributos**
Domínio	Previsão de Séries Temporais e Reconhecimento de Padrões
Público-Alvo	Pesquisadores que desejam criar e testar soluções no domínio descrito
Plataforma	Multiplataforma (Java®)
Licença	Copyright
Abstração de Rotinas	Prevista, mas não implementada por falta de tempo hábil.
Abstração de Dados	Não suporta e sem previsão de suporte. Irrelevante para o alcance da linguagem no domínio visado
Paradigma de Linguagem	Orientada a Fluxo de Dados (mesmo de Simulink® - Matlab®)
Tipo de fluxo de dados	Matriz Bidimensional
Vocabulário	6 *Tokens* Visuais (Modelo inicial)

3.3.1 Visão Geral do Ambiente da VisualPREV

Para a utilização da VisualPREV é necessário inicialmente definir o modelo, incluindo nele a definição da base de dados a ser utilizada, a ferramenta (técnica) e a saída dos resultados. Os modelos não precisam ser recriados com freqüência; modelos editados anteriormente e armazenados podem ser reutilizados e também reeditados.

A Figura 3 apresenta as janelas do ambiente da VisualPREV. Pode-se observar a existência de três janelas no aplicativo. A menor à esquerda é a Janela Principal, sendo a primeira janela a aparecer quando o programa é iniciado. Ela contém o vocabulário de *tokens* básicos (ou palavras) utilizados na tarefa de programação. A janela maior superior, denominada Janela de Edição e Simulação, é para a configuração de um novo modelo, ou para a alteração de um modelo já existente e para a execução do modelo; e a janela inferior, denominada Janela de Mensagem, é para apresentar as mensagens de resposta do programa. As três principais atividades da linguagem VisualPREV são a configuração do modelo (ou alteração de um modelo já existente), a execução do modelo e a visualização dos resultados.

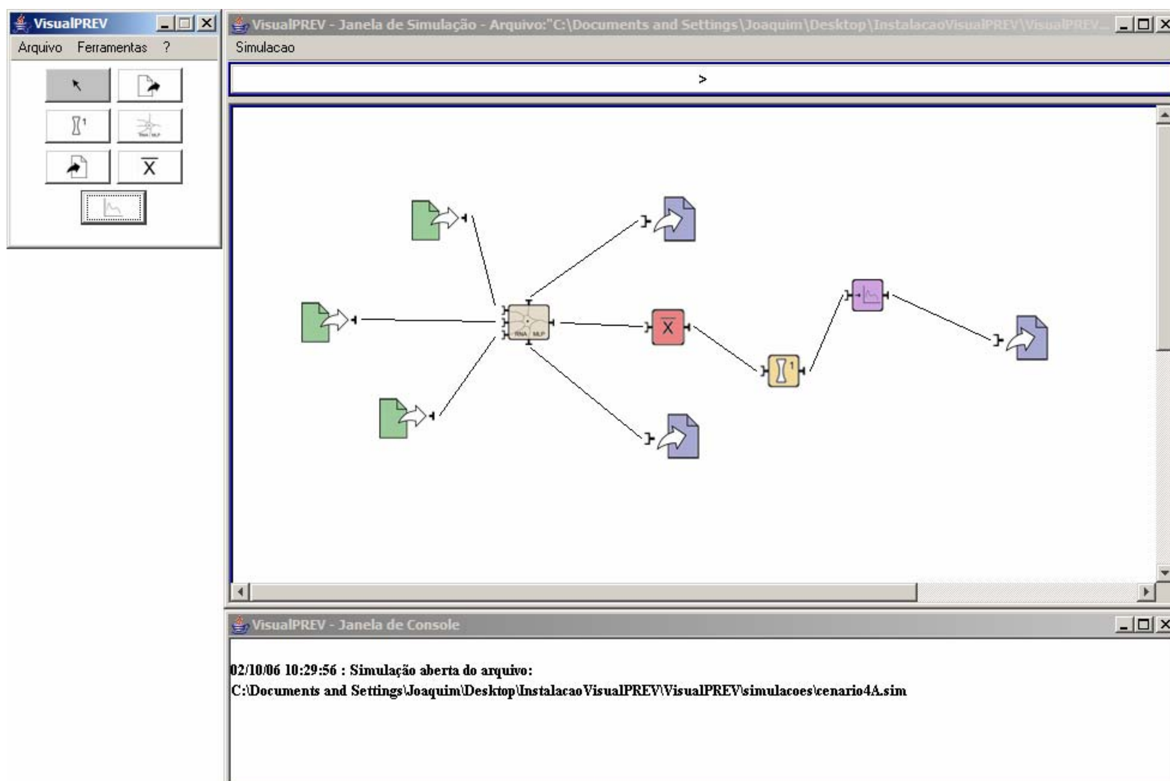


Figura 3. Visão geral do ambiente da VisualPREV

A partir da Janela Principal, pode-se construir um novo modelo. Quando é solicitada a criação de um novo modelo obtém-se uma janela de edição em branco. Para a edição do modelo, basta escolher, um-a-um, os blocos na Janela Principal, colocá-los na Janela de Edição e configurá-los (clicando sobre cada um deles). Os *tokens* podem ser interligados com os botões do mouse. A partir da Janela Principal é possível também abrir um modelo já existente. A Figura 3 apresenta um modelo já configurado e armazenado em arquivo, fato que pode ser verificado pela mensagem contida na janela de mensagem.

Para executar o modelo que acabou de ser criado, basta clicar no botão play da Janela de Simulação. A Janela de Mensagem apresenta os dados sobre os resultados obtidos. Detalhes desse processo serão apresentados ao longo desse capítulo.

3.3.2 Vocabulário da Linguagem: *Tokens* Visuais e Formulários Associados

A definição de uma linguagem de programação visual começa pela definição de suas palavras visuais básicas, i.e., de seus *tokens*. Esta seção visa listar os *tokens* presentes na linguagem em seus diferentes estados possíveis. Estes *tokens* possuem formulários associados que permitem ao usuário especificar os atributos dos *tokens* a partir do preenchimento destes formulários.

3.3.2.1 *Tokens* ou Blocos de Entrada de Dados

Bloco de Entrada de Dados (Treinamento, Teste e Validação)

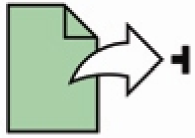


Figura 4. Representação do bloco de entrada de dados

O Bloco de Entrada de Dados (Figura 4) é o *token* responsável pela inserção dos dados num modelo; isso significa que eles devem estar sempre presentes pelo fato de serem a origem de todos os dados que percorrerão o modelo. Ao se clicar no bloco, obtém-se seu formulário associado (Figura 5) que possibilita sua configuração. Para o que foi até então desenvolvido, o que é relevante para esse bloco é um caminho de arquivo origem dado pelo usuário; basta clicar no botão 'Escolha o Arquivo de Dados' que um diálogo de escolha de arquivos aparecerá. Após a escolha, o caminho do arquivo escolhido aparecerá no bloco, e seu conteúdo será visualizado também. Esse arquivo conterá dados de Treinamento, Teste ou Validação, dependendo da necessidade do usuário.

Os dados em um sistema de previsão de séries temporais e de reconhecimento de padrões são de grande importância, uma vez que a qualidade dos resultados obtidos através dos modelos são fortemente influenciados pela qualidade dos dados. Vários fatores

influenciam a qualidade de uma base de dados, entre os quais pode-se destacar erros na coleta e armazenamento dos dados, dados *outliers*, e alteração importantes em fatores exógenos importantes para a previsão de séries temporais. Em relação ao primeiro fator, a experiência com dados reais tem mostrado que os sistemas de coleta e armazenamento de dados introduzem em geral muitos erros nas bases de dados, de modo que é importante um sistema de tratamento de dados para a detecção de dados incorretos e atípicos. Em relação aos dados atípicos, grande parte dos sistemas apresentam comportamentos atípicos, como por exemplo o consumo de energia em um feriado nacional, em eventos especiais tais como final de Copa do Mundo. Se estes comportamentos atípicos não forem identificados e eliminados dos conjuntos de treinamento, certamente o resultado da previsão será influenciado por estes dados atípicos. Em relação às variáveis exógenas, estas também podem influir significativamente sobre o comportamento do sistema. Por exemplo, no caso da energia elétrica, tivemos um racionamento durante os anos de 2001 e 2002, de modo que a série temporal que representa o consumo de energia elétrica apresenta um perfil de consumo diferente neste período.

Portanto, em relação ao sistema de entrada de dados, pretende-se futuramente implementar um sistema de validação e de seleção de dados para a montagem dos dados de entrada. Na versão atual, os dados já estão armazenados em arquivos para treinamento, validação e teste.

Não há nenhuma facilidade para capturar dos dados na VisualPREV, atualmente. A coleta de dados em planilhas, embora seja a parte mais sensível e importante do processo, ainda não está totalmente resolvida. Portanto, podemos concluir que o bloco de entrada de dados é limitado para o que se propõe, visto que a parte da escolha dos dados é delegada a um software auxiliar fora do ambiente de desenvolvimento da VisualPREV. Um bloco de aquisição de dados mais amigável está na lista de possíveis trabalhos futuros.

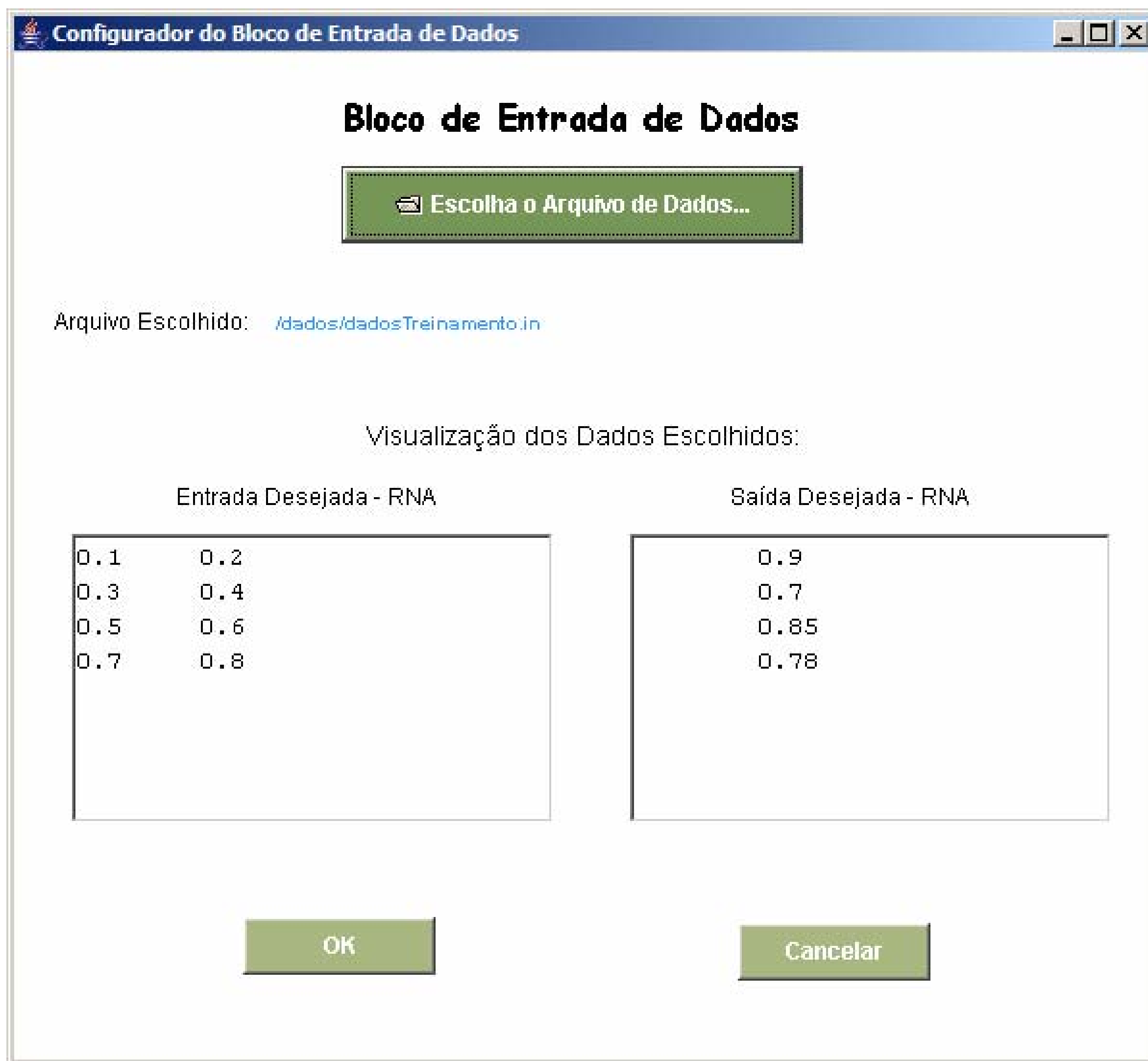


Figura 5. Formulário associado ao bloco de entrada de dados

3.3.2.2 *Tokens* ou Blocos de Ferramentas

Apresenta-se a seguir um conjunto de ferramentas, que englobam desde funções simples de normalização e cálculo de média de uma dada série, até ferramentas mais complexas como é o caso das metodologias baseadas em redes neurais. As ferramentas a seguir apresentadas constituem um conjunto inicial de ferramentas. Está prevista a implementação de

uma funcionalidade do tipo *plug-and-play*, através da qual novas ferramentas podem facilmente ser incluídas na biblioteca de ferramentas e de *tokens* da Janela Principal.

Bloco Normalizador

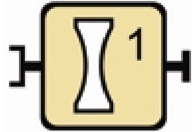


Figura 6. Representação do bloco de componente normalizador

O Bloco Normalizador (Figura 6) é o *token* responsável pela normalização dos dados que entram nele; isso significa que o bloco recebe os dados como uma matriz bidimensional, normalizando os mesmos a partir do valor máximo encontrado na mesma. Ao se clicar no bloco, obtém-se seu formulário associado (Figura 7) que possibilita sua configuração. Para o que foi até então desenvolvido, o que é relevante para esse bloco é a opção de normalização; no caso, há uma única opção implementada (normalização pelo valor máximo).

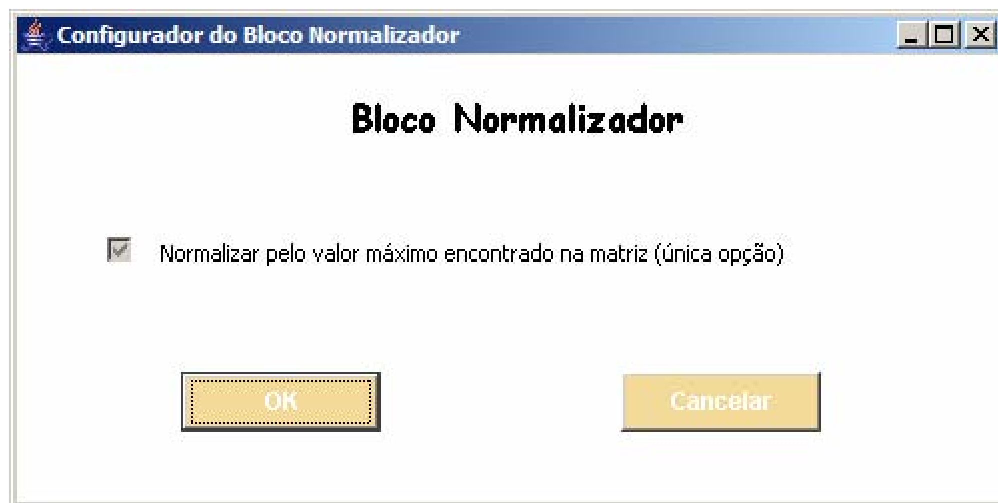


Figura 7. Formulário associado ao bloco normalizador

Bloco de Rede Neural - MLP

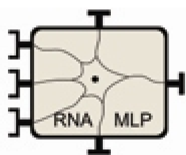


Figura 8. Representação do bloco de Rede Neural tipo MLP

O Bloco Rede Neural-MLP (Figura 8) é o *token* responsável por representar uma rede neural artificial (RNA) com arquitetura MLP, com suas 3 entradas possíveis (conjuntos de treinamento – entrada superior, teste – entrada intermediária e validação – entrada inferior) e suas 3 saídas possíveis (pesos dos neurônios – saída inferior, erros associados a previsão ou classificação – saída superior e valores previstos/classificados – saída à direita). Ao se clicar no bloco, obtém-se seu formulário associado (Figura 9) que possibilita sua configuração. Para o que foi até então desenvolvido, o que é relevante para esse bloco é a opção de tipo de RNA (no caso, há uma única opção implementada – *classic back-propagation*), número de neurônios, taxa de aprendizagem e termo de momento. A RNA recebe os dados de entrada e processa os mesmos, a partir do modelo de previsão de séries temporais, fornecendo os valores resultantes na saída.

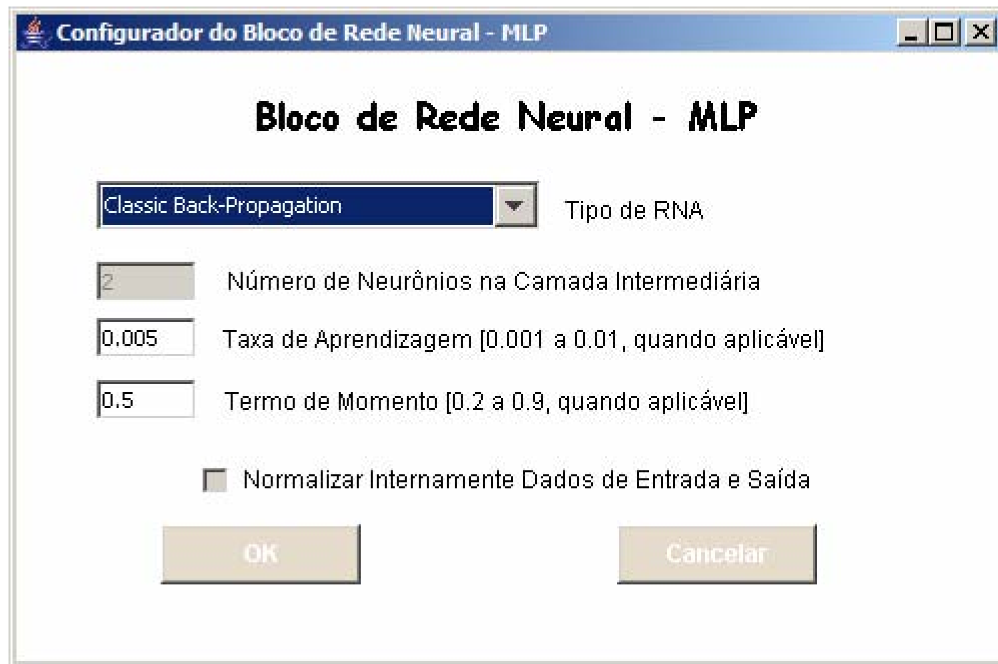


Figura 9. Formulário associado ao bloco de componente RNA

É importante destacar que outras metodologias baseadas em RNA, explorando diferentes arquiteturas, e outras aplicações como em redes não supervisionadas para fins de classificação podem ser introduzidas tanto para a previsão de séries temporais quanto para a classificação de padrões. Outras metodologias baseadas em redes neuro-*fuzzy*, estatísticos, etc... poderão também ser introduzidas no sistema, facilitando a integração entre diferentes técnicas.

Bloco Calcula Média

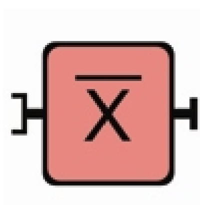


Figura 10. Representação do bloco de cálculo da média

O Bloco Calcula Média (Figura 10) é o *token* responsável por calcular a média dos dados que entram nele; isso significa que o bloco recebe os dados como uma matriz bidimensional, calculando o valor médio de cada linha, o que resulta numa única coluna. Ao se clicar no bloco, obtém-se seu formulário associado (Figura 11) que possibilita sua configuração. Para o que foi até então desenvolvido, o que é relevante para esse bloco é a opção de calcular a média a partir de cada linha; no caso, essa é a única opção implementada.

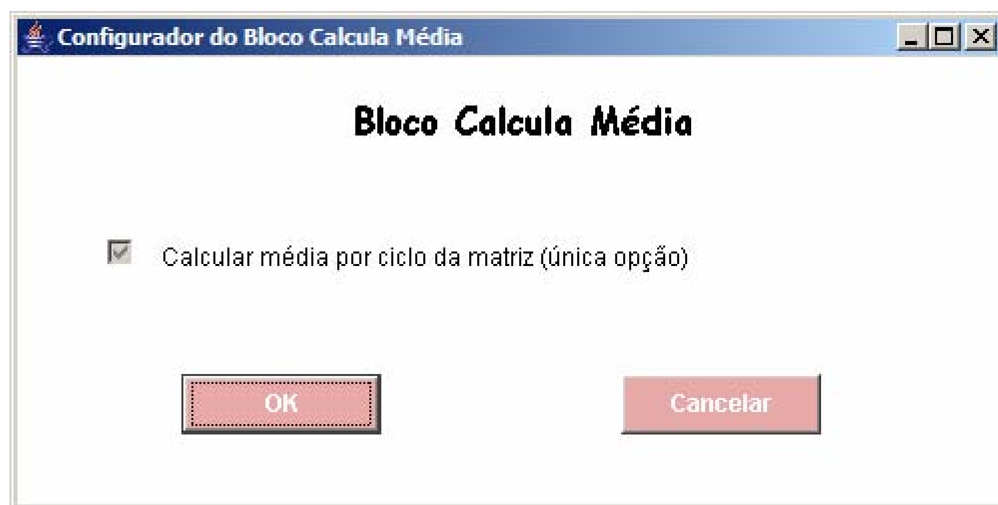


Figura 11. Formulário associado ao bloco de cálculo da média

Bloco Identificação de Classes (Decodificador)

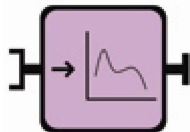


Figura 12. Representação do bloco de identificação de classes

O Bloco Identificação de Classes (Figura 12) funciona como uma espécie de 'tradutor' (com as classes pré-definidas em sua base interna de informações) e é normalmente utilizado

na saída da Rede Neural visando decodificar a classificação gerada por esta (Rede Neural como classificador supervisionado simples). Ele determina, para cada linha correspondente ao vetor de saída da Rede Neural, a classe à qual mais se aproxima (distância euclideana mínima no espaço n-dimensional). Caso haja dúvida (mais de um resultado com a mesma distância euclideana), a primeira encontrada é definida como a classe-padrão.

Ao se clicar no bloco, obtém-se seu formulário associado (Figura 13) que possibilita sua configuração. Para o que foi até então desenvolvido, o que é relevante para esse bloco é a escolha de um arquivo que contenha a base de dados que alimentará o mesmo com as classes pré-definidas; basta clicar no botão 'Escolha o Arquivo de Classes' que um diálogo de escolha de arquivos aparecerá. Após a escolha, o caminho do arquivo escolhido aparecerá no bloco, e seu conteúdo será visualizado também. O bloco realiza um teste de erros (quadrados mínimos) para detectar a qual valor da base o valor de entrada se aproxima mais. A partir de então, obtém-se o resultado da classe a que essa entrada pertence.

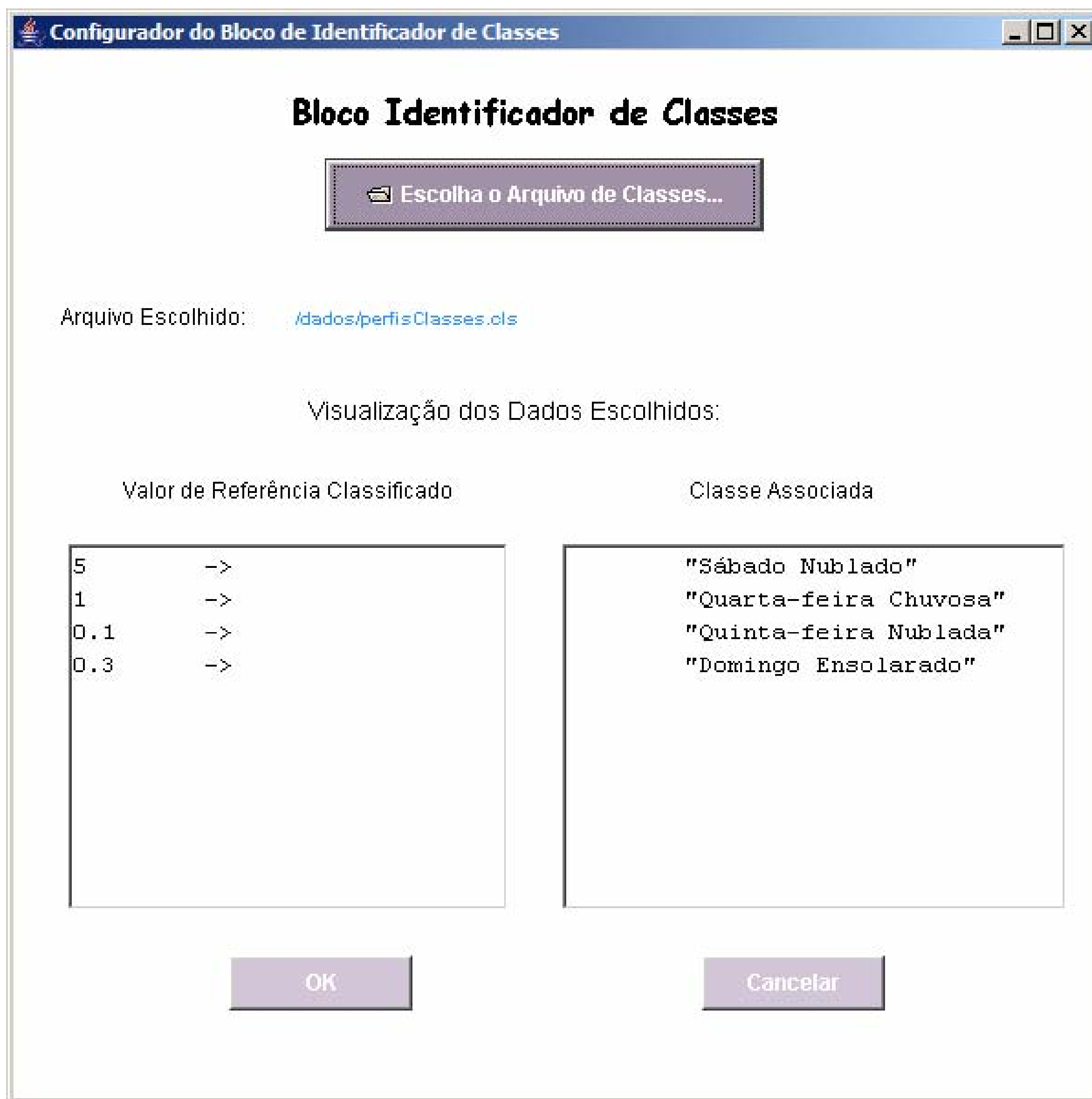


Figura 13. Formulário associado ao bloco de identicação de classes

Este bloco retorna, em sua saída, os valores de saída da RNA mais as classes a que estes valores pertencem, armazenando toda esta informação num arquivo de saída.

3.3.2.3 *Tokens* ou Blocos de Armazenamento dos Resultados

Bloco de Armazenagem de Dados

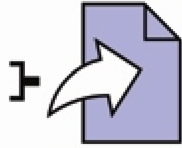


Figura 14. Representação do bloco de armazenagem de dados

O Bloco de Armazenagem de Dados (Figura 14) é o *token* responsável por guardar os dados resultantes de uma simulação; estes blocos, embora opcionais para que o modelo seja executado, são fundamentais para a verificação dos resultados. Eles armazenam qualquer informação numérica que estiver no conector de saída de outro bloco. Ao se clicar no bloco, obtém-se seu formulário associado (Figura 15) que possibilita sua configuração. Para o que foi até então desenvolvido, o que é relevante para esse bloco é um caminho de arquivo destino dado pelo usuário, que pode já existir ou ser criado arbitrariamente; basta clicar no botão 'Escolha o Arquivo de Armazenamento' que um diálogo de escolha de arquivos aparecerá. Após a escolha, o caminho do arquivo escolhido aparecerá no bloco. Esse arquivo conterá os dados resultantes da simulação. No caso das simulações de classificação, as classes serão armazenadas neste bloco como números naturais; cada número corresponde à ordem em que a classe aparece no arquivo de classes (.cls).

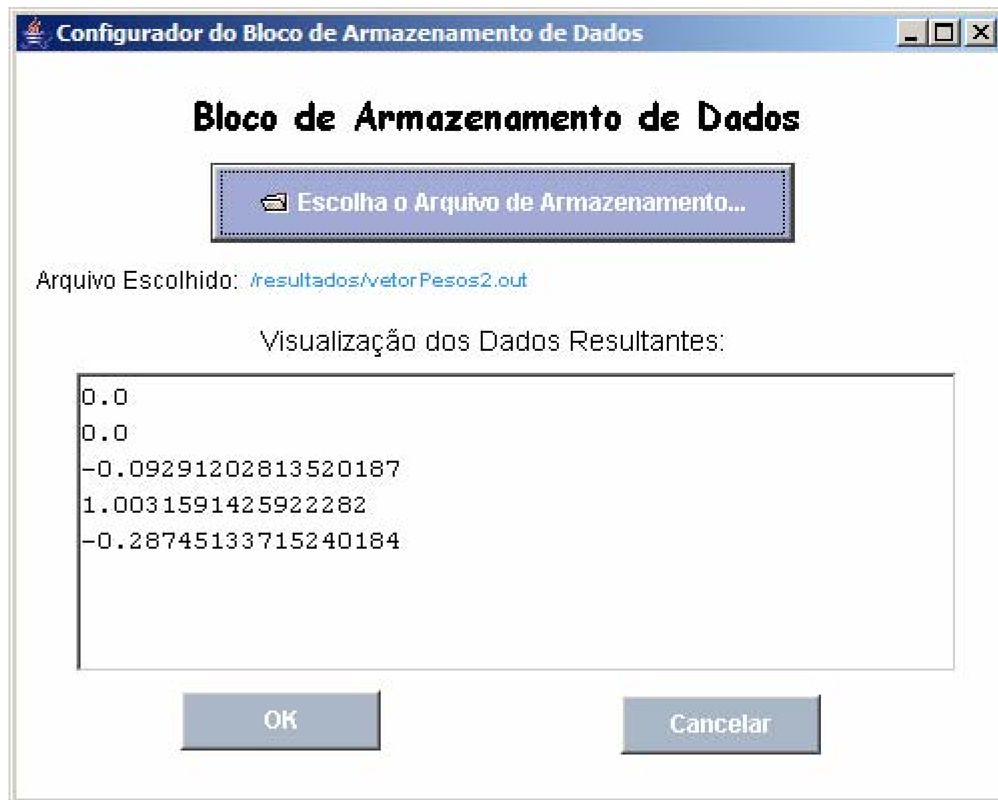


Figura 15. Formulário associado ao bloco de armazenagem de dados

3.3.3 Fluxo de Dados

Considere o exemplo de modelo da Figura 16:

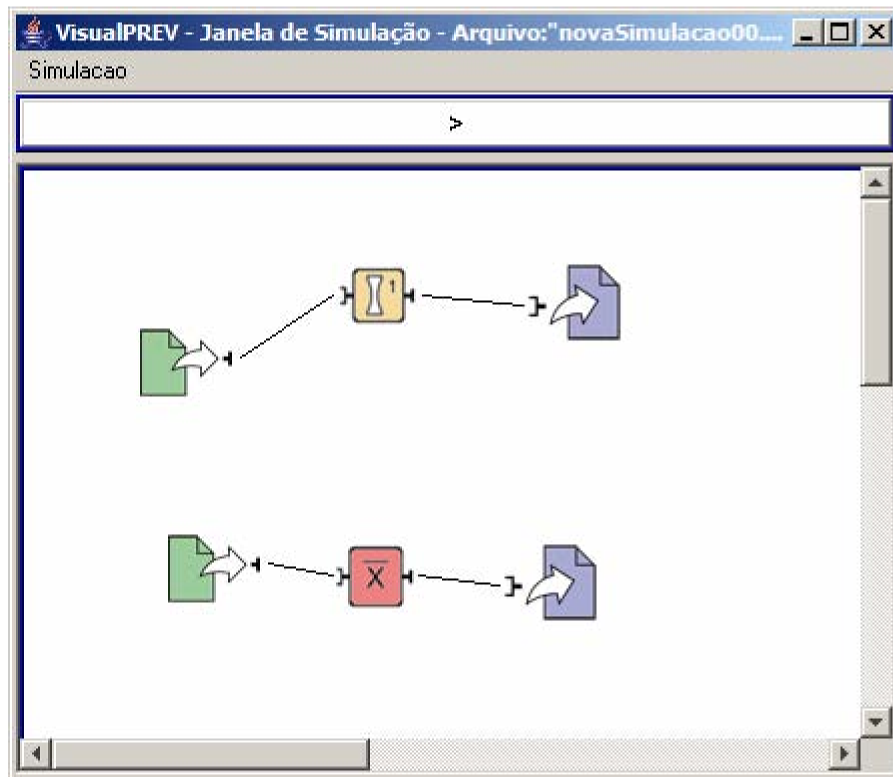


Figura 16. Modelo de Normalização e Modelo de Cálculo de Média, numa mesma janela.

De acordo com este exemplo, pode-se verificar a existência de dois modelos numa mesma janela (determinados pelas conexões dos blocos). Na superior, dados entram na simulação e passam por um processo de normalização a partir do valor máximo encontrado neles, e são posteriormente armazenados. Na inferior, há uma extração do valor médio de cada uma das linhas do arquivo de entrada, de forma que o arquivo resultante apresente apenas uma única coluna, que é então armazenada. Esse processo de extração da média foi adequado ao problema de previsão de carga.

A idéia de "conexão" e do direcionamento do fluxo de dados foi concentrada nos blocos, em detrimento dos conectores. Observa-se que a saída de um bloco corresponde a um conector do tipo "macho", enquanto a entrada de outro corresponde um conector do tipo "fêmea"; é possível que haja mais de um conector ligado à saída de um bloco. Portanto, os blocos são conectados com arestas direcionadas que interligam os blocos de origem e destino. Elas são as responsáveis por direcionar e conduzir os dados pelo modelo.

3.3.4 Validação Sintática e Semântica

A validação sintática da VisualPREV é dada pelas seguintes características:

- Existência de *Tokens* ou Palavras corretas: isto é garantido pelo funcionamento intrínseco da linguagem, pois somente *tokens* (blocos) existentes podem ser acrescidos aos modelos;

- Ligação apropriada entre os *Tokens*: esta etapa é garantida pela existência de um sistema de supervisão das conexões apropriadas, de maneira que só as conexões possíveis podem ser realizadas ao longo do processo de modelagem, i.e., quando o usuário tenta ligar dois blocos, o sistema realiza uma sistemática de avaliação quanto a esta possibilidade (está se ligando uma entrada a uma saída? Não se está tentando realizar uma ligação já existente? Depois de validada, ao usuário somente é permitida a criação de uma ligação direcional (origem-destino), pois o editor autoriza uma única direção da aresta para atender aos requisitos do sistema (origem da aresta corresponde à saída de um bloco e destino da aresta corresponde à entrada de outro bloco);

A validação semântica da VisualPREV, ainda não plenamente funcional, é dada pelas seguintes características:

- Considerando que os formulários associados aos blocos devem ser preenchidos corretamente, o bloco inicialmente colocado no modelo (em vermelho) demonstra ter aceitado os parâmetros preenchidos no formulário (pelo fato de eles terem sido preenchidos e estarem corretos) mudando a sua cor de vermelho para a sua cor característica (cada bloco possui um desenho específico que auxilia em sua identificação);

- Ao acionarmos o botão *play* (botão para executar o modelo existente na "palheta de blocos e funções"), há ainda uma série adicional de testes para se verificar se todas as exigências foram atendidas. Um teste adicional que se pretende implementar é o de blocos solteiros/faltantes: em caso de haver blocos separados (não conectados) ou blocos faltantes (blocos com entradas não atendidas e formulários associados indevidamente preenchidos), o editor emite avisos no primeiro caso e alertas de erro e impossibilidade de execução no segundo.

3.3.5 *Framework* Computacional

No desenvolvimento do *software*, um *framework* ou arcabouço é uma estrutura de suporte definida em que um outro projeto de *software* pode ser organizado e desenvolvido. Um *framework* pode incluir programas de suporte, bibliotecas de código, linguagens de *script* e outros *softwares* para ajudar a desenvolver e unir diferentes componentes de um projeto de *software*. *Frameworks* são projetados com a intenção de facilitar o desenvolvimento de *software*, habilitando projetistas e programadores a gastarem mais tempo determinando as exigências do *software* do que com detalhes de baixo nível do sistema. Um *framework* se diferencia de uma simples biblioteca, pois esta se concentra apenas em oferecer implementação de funcionalidades, sem definir a reutilização de uma solução de arquitetura.

Com o intuito de facilitar a organização e a manutenção do código da VisualPREV, foi criado um *framework* contendo um conjunto de classes de Java. As classes se subdividem em dois pacotes (*packages*): núcleo (*kernel*) e interface. O pacote de interface está voltado ao diálogo do sistema com o usuário/programador, tais como janelas do programa (principal, simulação e mensagem) e janelas de configuração (de cada um dos blocos funcionais). O pacote de núcleo inclui as classes que efetivamente executarão os modelos, tais como gerenciador de conexões, compilador visual e RNA.

A estrutura de instalação do programa implementa a existência de diversas pastas, cujas funções são descritas na Tabela 6.

Tabela 6. Pastas de sistema do ambiente da VisualPREV

Pasta	Conteúdo
/dados	Arquivos de entrada de dados preparados para a criação e execução dos modelos
/lib	Bibliotecas da linguagem VisualPREV
/log	Arquivos de log do sistema (erros de compilação e execução dos modelos)
/resultados	Arquivos de saída de dados a serem armazenados após a execução do modelo
/simulações	Arquivos de modelos que se deseja armazenar/recuperar

O ambiente de desenvolvimento Eclipse® foi utilizado para todas as tarefas relativas à atividade de programação, incluindo a geração dos arquivos executáveis ".jar". Com relação à concepção artística, os desenhos dos blocos foram criados pela arquiteta e artista gráfica Renata Monezzi, que se inspirou em símbolos matemáticos e formas que transmitissem as idéias pretendidas sobre a essência das funcionalidades de cada um dos blocos.

A VisualPREV possui um ambiente que pode facilmente receber blocos novos, bastando para isso a criação de um conjunto de classes específica que atenda às especificações do ambiente, além das figuras e interfaces correspondentes. Existe, portanto, um padrão de criação de blocos para adição ao ambiente.

3.4 VisualPREV *versus* abordagem tradicional

Para o presente trabalho considera-se 'abordagem tradicional' como sendo a criação de *softwares* específicos para a solução de problemas. Duas formas de se comparar duas linguagens são: (1) desempenho computacional (velocidade de resolução do problema) e (2) desempenho humano (velocidade de criação dos modelos). Não foram feitos testes comparativos em relação a nenhum dos dois itens. O trabalho focou tão somente na análise da usabilidade da interface.

3.5 Perspectivas Futuras de Implementação

Havia-se previsto a criação de blocos adicionais de tratamento estatístico de dados (mediana, variância, desvio padrão, filtragem por parâmetro), de entrada de dados (extração de Banco de Dados), de componentes (Redes Neuro-*Fuzzy*, Kohonen, modelos híbridos, outros modelos em IA), de clusterização de dados (*Fuzzy* C-Means), de combinação (agregadores e *ensembles*, ajuste de perfil de curvas) e de exibição de dados (*displays*, geradores de gráficos). Não houve tempo hábil de se implementar tais blocos.

Além dessas propostas, encapsulamento é uma funcionalidade desejável no caso em que os modelos tendem a crescer muito, quando o número de *tokens* começa a ficar significativo numa mesma tela de modelagem, podendo gerar confusão para o usuário. A proposta é a de se criar um *token* de encapsulamento, um bloco adicional dentro do qual todo

um conjunto de blocos possa ser inserido; esse *token* de encapsulamento representaria todo o conjunto de blocos que o contém, e ações sobre esse *token* afetariam o conjunto de blocos internos a ele. A especificação dessa funcionalidade não foi completamente obtida e, portanto, segue como perspectiva futura. Outro objetivo é o de atingir um número “crítico” de blocos que permita a maximizar a integração de modelos.

4. Duas Aplicações da Linguagem VisualPREV

Neste capítulo serão apresentadas duas aplicações da Linguagem VisualPREV em um problema de previsão de carga.

4.1 Exemplo 1: Previsão de Séries Temporais

4.1.1 Primeira Fase: Montagem dos Conjuntos de Dados

Tabela 7. Exemplo de dados históricos normalizados. Destacados somente os dados relevantes para o exemplo.

	Seg	Ter	Qua	Qui	Sex	Sáb	Dom	Seg	Ter	Qua	Qui	Sex	Sáb	Dom
Dia/Mês ->	**1/jan**	**2/jan**	**3/jan**	**4/jan**	**5/jan**	**6/jan**	**7/jan**	**8/jan**	**9/jan**	**10/jan**	**11/jan**	**12/jan**	**13/jan**	**14/jan**
1h														
2h														
3h														
4h														
5h														
6h														
7h														
8h														
9h														
10h														
11h														
12h														
13h														
14h														
15h														
16h														
17h														
18h														
19h			0.23	0.25	0.23					0.26	0.24	0.25		
20h			0.29	0.32	0.30					0.31	0.30	0.31		
21h		0.40	0.37	0.35	0.39				0.38	0.34	0.41	0.36		
22h														
23h														
24h														

	Seg	Ter	Qua	Qui	Sex	Sáb	Dom	Seg	Ter	Qua	Qui	Sex	Sáb	Dom
Dia/Mês ->	**15/jan**	**16/jan**	**17/jan**	**18/jan**	**19/jan**	**20/jan**	**21/jan**	**22/jan**	**23/jan**	**24/jan**	**25/jan**	**26/jan**	**27/jan**	**28/jan**
1h														
2h														
3h														
4h														
5h														
6h														
7h														
8h														
9h														
10h														
11h														
12h														
13h														
14h														
15h														
16h														
17h														
18h														
19h			0.26	0.22	0.24									
20h			0.33	0.29	0.32									
21h		0.39	0.39	0.40	0.41									
22h														
23h														
24h														

Imaginem que se queira fazer a previsão do consumo de energia elétrica no intervalo de tempo das 21h de uma sexta-feira. Supõe-se que se dispõe dos dados históricos representativos do consumo horário de energia elétrica, conforme ilustrado na Tabela 7 (dados já normalizados).

Para testar um modelo de previsão de séries temporais baseado em uma RNA, o usuário deve selecionar os dados da série histórica para montar os conjuntos de dados para treinamento, validação e teste. Na montagem dos dados deve-se levar em conta a configuração da RNA a ser utilizada. Ou seja, para a previsão da carga às 21h é necessário definir quais variáveis (entradas) devem ser consideradas. No caso do consumo de energia elétrica, usualmente o nível de consumo de energia elétrica em um dado intervalo de tempo é fortemente influenciado pelo nível de consumo nos intervalos anteriores e do nível de consumo no mesmo horário no dia anterior. Portanto, para o modelo de previsão do consumo de energia elétrica no dia d durante o intervalo de tempo t, c_d^t, poderia ser adotado como o conjunto de variáveis de entrada o valor do consumo de energia observados nos dois intervalos anteriores, c_d^{t-1} e c_d^{t-2}, e o consumo no intervalo t do dia anterior, c_{d-1}^t. A variável de saída seria o consumo c_d^t. Baseado neste conjunto de entradas e saídas são montados os conjuntos de treinamento, validação e de teste, como exemplificado na Tabela 8.

Tabela 8. Sub-conjuntos selecionados dos dados históricos normalizados.

	Entrada 1	Entrada 2	Entrada 3	Saída desejada
Conjunto de	0.23	0.30	0.35	0.39
Treinamento	0.25	0.32	0.37	0.35
1 a 7 de Jan	0.23	0.29	0.40	0.37

	Entrada 1	Entrada 2	Entrada 3	Saída desejada
Conjunto de	0.25	0.31	0.41	0.36
Teste	0.24	0.30	0.34	0.41
8 a 14 de Jan	0.26	0.31	0.38	0.34

	Entrada 1	Entrada 2	Entrada 3	Saída desejada
Conjunto de	0.24	0.32	0.40	0.41
Validação	0.22	0.29	0.39	0.40
15 a 21 de Jan	0.26	0.33	0.39	0.39

A correspondência desses sub-conjuntos com os dados históricos pode ser rastreada a partir das cores.

4.1.2 Segunda Fase: Construção do Modelo

Ao solicitar a criação de um novo modelo, o ambiente da VisualPREV disponibiliza uma janela de edição em branco. Passa-se então a inserir cada um dos blocos, interligando-os de maneira conveniente e configurando-os, valendo-se das janelas de configuração individual de cada um dos blocos.

A Figura 17 apresenta um modelo para a previsão definida no item anterior, utilizando uma RNA. O Bloco de Entrada superior corresponde aos dados de treinamento; o do meio corresponde aos dados de teste, e o inferior correspondem aos dados de validação. Ao se pressionar o botão PLAY na janela de simulação, o bloco RNA recebe os dados de entrada e processa-os e retorna os resultados. Todos os blocos estão devidamente configurados uma vez que se apresentam no estado 'colorido' na janela de edição.

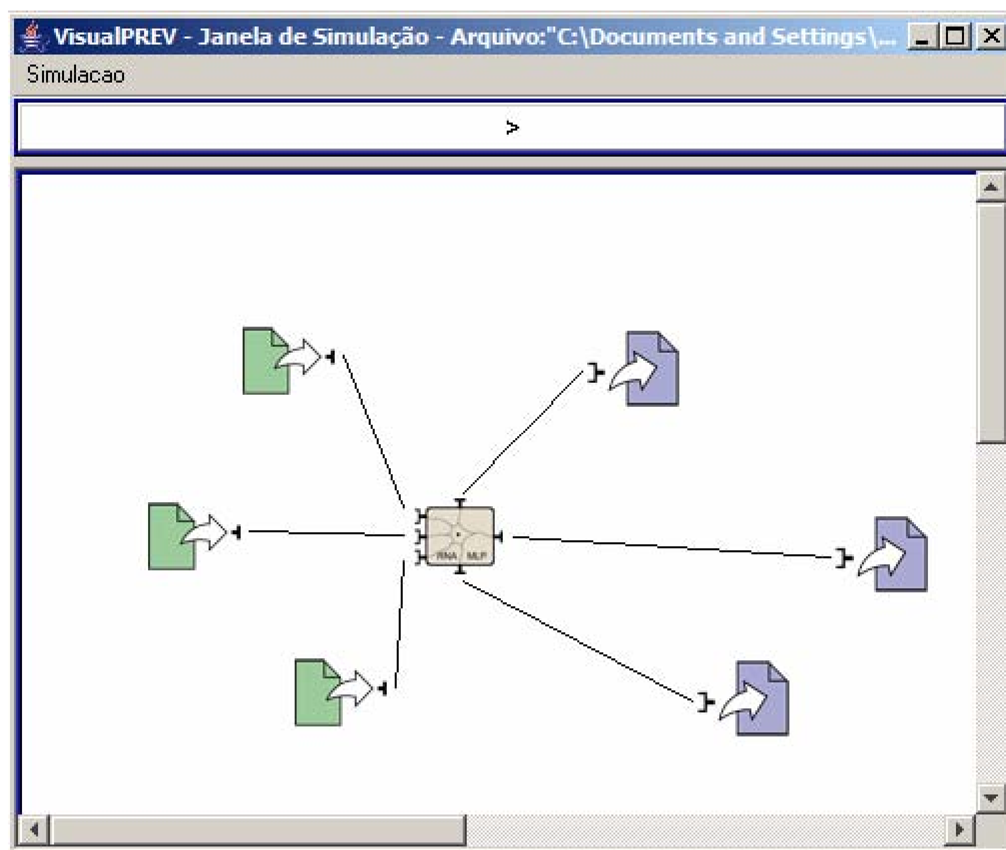


Figura 17. Um exemplo de modelo, para previsão de séries temporais, baseado em RNA.

A Figura 17 mostra o modelo já configurado. Vale lembrar que o Bloco Rede Neural-MLP (Figura 8) é o *token* responsável por representar uma rede neural artificial (RNA), alimentada pelos conjuntos de treinamento, teste e de validação. Os resultados da simulação são armazenados em três saídas, as quais contém os pesos dos neurônios, os valores previstos e os erros associados à previsão.

4.1.3 Terceira Fase: Análise dos Resultados

Ao clicarmos no botão PLAY, aparece na janela de mensagens um aviso explicitando o sucesso/insucesso da execução do modelo. Nesse caso, o modelo executou com sucesso, e os arquivos de saída podem ser analisados de duas maneiras: clicando-se sobre os blocos ou obtendo-se os arquivos de saída no diretório '/resultados' do programa. Escolheu-se a segunda forma para o exemplo.

No caso deste modelo, pode-se ter informações sobre os pesos, as previsões e os erros, conforme as Figuras 18, 19 e 20.

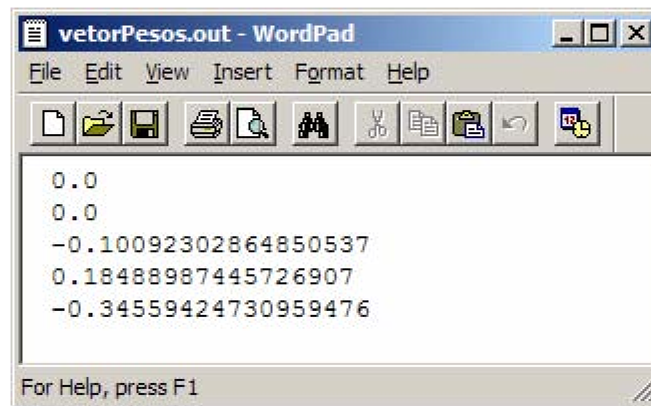


Figura 18. Arquivo de saída contendo os pesos dos neurônios após treinamento.

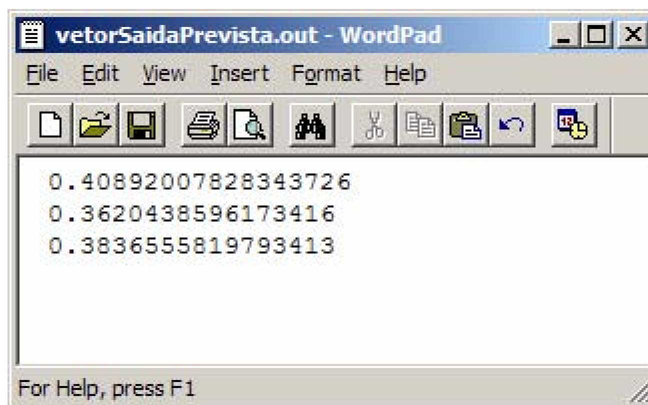


Figura 19. Arquivo de saída contendo as saídas previstas normalizadas.

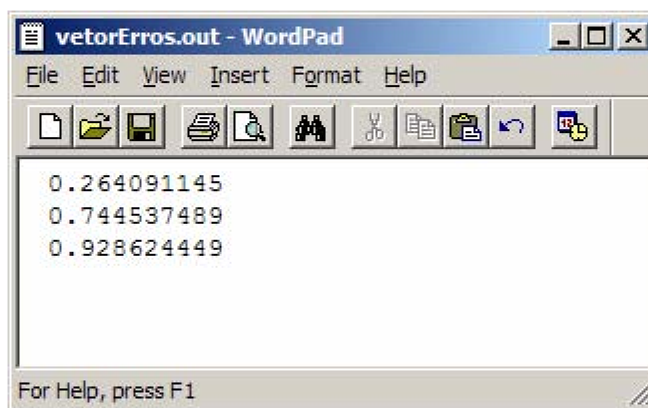


Figura 20. Arquivo de saída contendo erros associados aos valores previstos.

Podemos então comparar os resultados previstos com os dados do histórico, avaliando-se assim a previsão que acaba de ser realizada, conforme Tabela 9.

Tabela 9. Comparação entre os valores previstos e os valores desejados (valores normalizados)

	Entrada 1	Entrada 2	Entrada 3	Saída desejada	Saída Prevista
Conjunto de	0.24	0.32	0.40	0.41	0.4089200782834372
Validação	0.22	0.29	0.39	0.40	0.3970438596173416
15 a 21 de Jan	0.26	0.33	0.39	0.39	0.3936555819793413

4.2 Exemplo 2: Classificação de Padrões com Decodificação dos valores classificados

4.2.1 Primeira Fase: Montagem dos Conjuntos de Dados

A principal diferença entre a utilização da RNA para previsão e para classificação é que, neste segundo caso, faz-se necessária a utilização de um bloco adicional que traduza ou decodifique os valores numéricos gerados pela RNA na etapa de validação. Para este teste, foi utilizado outro histórico de dados, com exceção de que os dados foram escolhidos de maneira distinta do outro exemplo, visto que o propósito dessa tarefa é o de classificação.

Tabela 10. Exemplo de dados históricos normalizados. Destacados somente os dados relevantes para o exemplo.

	Seg	Ter	Qua	Qui	Sex	Sáb	Dom	Seg	Ter	Qua	Qui	Sex	Sáb	Dom
Dia/Mês ->	**1/jan**	**2/jan**	**3/jan**	**4/jan**	**5/jan**	**6/jan**	**7/jan**	**8/jan**	**9/jan**	**10/jan**	**11/jan**	**12/jan**	**13/jan**	**14/jan**
1h														
2h														
3h														
4h														
5h														
6h														
7h														
8h														
9h														
10h														
11h														
12h														
13h														
14h														
15h														
16h														
17h														
18h														
19h			0.21	0.11						0.21	0.11			
20h			0.2	0.1						0.22	0.09			
21h			0.2	0.12						0.2	0.12			
22h														
23h														
24h														

	Seg	Ter	Qua	Qui	Sex	Sáb	Dom	Seg	Ter	Qua	Qui	Sex	Sáb	Dom
Dia/Mês ->	**15/jan**	**16/jan**	**17/jan**	**18/jan**	**19/jan**	**20/jan**	**21/jan**	**22/jan**	**23/jan**	**24/jan**	**25/jan**	**26/jan**	**27/jan**	**28/jan**
1h														
2h														
3h														
4h														
5h														
6h														
7h														
8h														
9h														
10h														
11h														
12h														
13h														
14h														
15h														
16h														
17h														
18h														
19h			0.22	0.10						0.19	0.09			
20h			0.19	0.11						0.22	0.09			
21h			0.18	0.08						0.21	0.10			
22h														
23h														
24h														

Suponha ainda que se queira responder à seguinte pergunta: como classificar dados de entrada em quatro classes distintas (primeira, segunda, terceira e quarta semanas de janeiro)? Para conseguir testar um modelo simples de classificação contendo apenas uma RNA, deve-se selecionar os dados da série histórica baseando-se numa heurística própria em que assume que dados associados a horários próximos apresentam comportamentos semelhantes. Portanto, os horários das 19h, 20h e das 21h devem ser utilizados no modelo.

Assim procedendo, o usuário programador gera três sub-conjuntos a partir da série histórica, conforme Tabela 11. Observe que os valores de saída desejada foram criados pelo próprio usuário, que os definiu como sendo os valores representativos para a classe (Tabela 12).

Tabela 11. Sub-conjuntos selecionados dos dados históricos normalizados.

	Entrada 1	Entrada 2	Saída desejada
Conjunto de	0.21	0.11	0.2
Treinamento	0.21	0.11	0.4
19hs	0.22	0.10	0.6
	0.19	0.09	0.8

	Entrada 1	Entrada 2	Saída desejada
Conjunto de	0.2	0.1	0.2
Teste	0.22	0.09	0.4
20hs	0.19	0.11	0.6
	0.22	0.09	0.8

	Entrada 1	Entrada 2	Saída desejada
Conjunto de	0.2	0.12	0.2
Validação	0.2	0.12	0.4
21hs	0.18	0.08	0.6
	0.21	0.10	0.8

Observe que as classes associadas a estes valores são definidas numericamente no intervalo [0-1], conforme Tabela 12.

Tabela 12. Classes numéricas e seus valores associados

Classe Numérica	Classe Nominal
0.2	Primeira semana
0.4	Segunda Semana
0.6	Terceira Semana
0.8	Quarta-Semana

4.2.1 Segunda Fase: Construção do Modelo

A Tabela 12 alimentará o arquivo de classes (.cls) que fará a tradução / decodificação dos valores numéricos das classes geradas, gerando uma lista de números que correspondem ao valor posicional da classe no arquivo de classes.

Na Figura 21, pode-se observar o modelo já configurado. Vale lembrar que o Bloco Rede Neural-MLP (Figura 8) é o *token* responsável por representar uma rede neural artificial (RNA), com suas 3 entradas possíveis (conjuntos de treinamento – entrada superior, teste – entrada intermediária e validação – entrada inferior) e suas 3 saídas possíveis (pesos dos neurônios – saída inferior, erros associados a previsão ou classificação – saída superior e valores previstos/classificados – saída à direita), conforme descrito anteriormente.

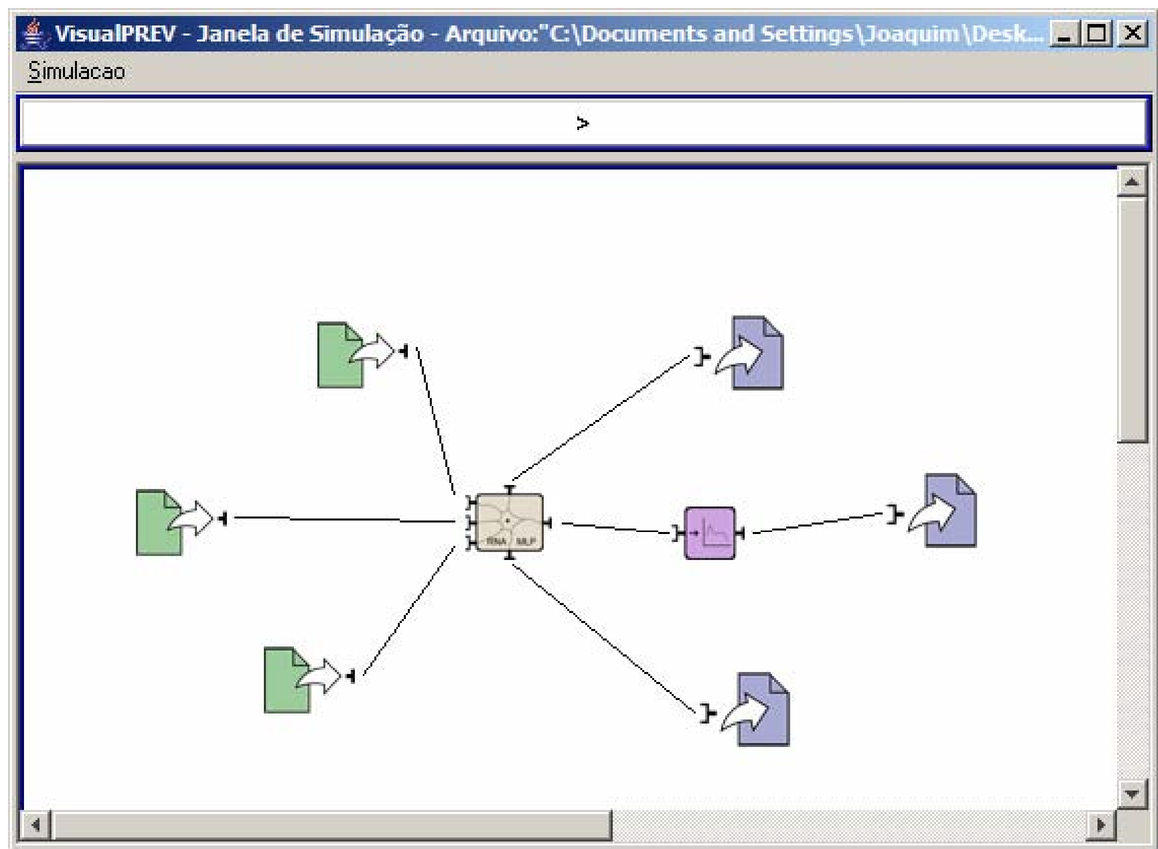


Figura 21. Um exemplo de modelo baseado em classificação de padrões com decodificação

4.2.3 Terceira Fase: Análise dos Resultados

Ao clicarmos no botão PLAY, aparece na janela de log um aviso explicitando o sucesso/insucesso da execução do modelo. Nesse caso, o modelo executou com sucesso, e os arquivos de saída podem ser analisados de duas maneiras: clicando-se sobre os blocos ou obtendo-se os arquivos de saída no diretório '/resultados' do programa. Escolheu-se a segunda forma para o exemplo.

No caso deste modelo, além das informações de erros e pesos obtidas de acordo com o exemplo da seção 4.1, pode-se obter informações das classes identificadas conforme a Figura 22.

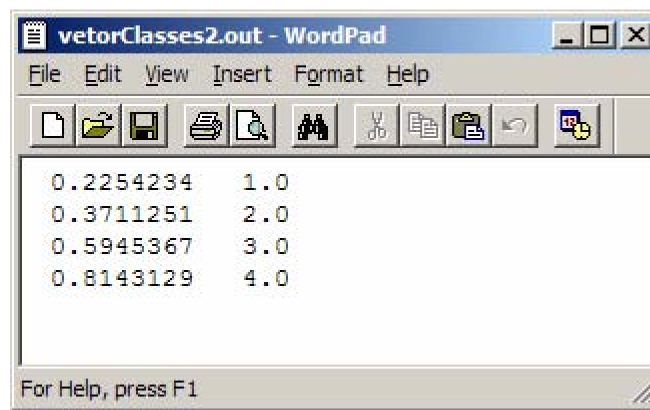


Figura 22. Arquivo de saída contendo as entradas classificadas.

Ao clicarmos no bloco de identificação de classes, obtemos os valores definidos das classes. A Figura 23 mostra a janela de configuração do bloco:

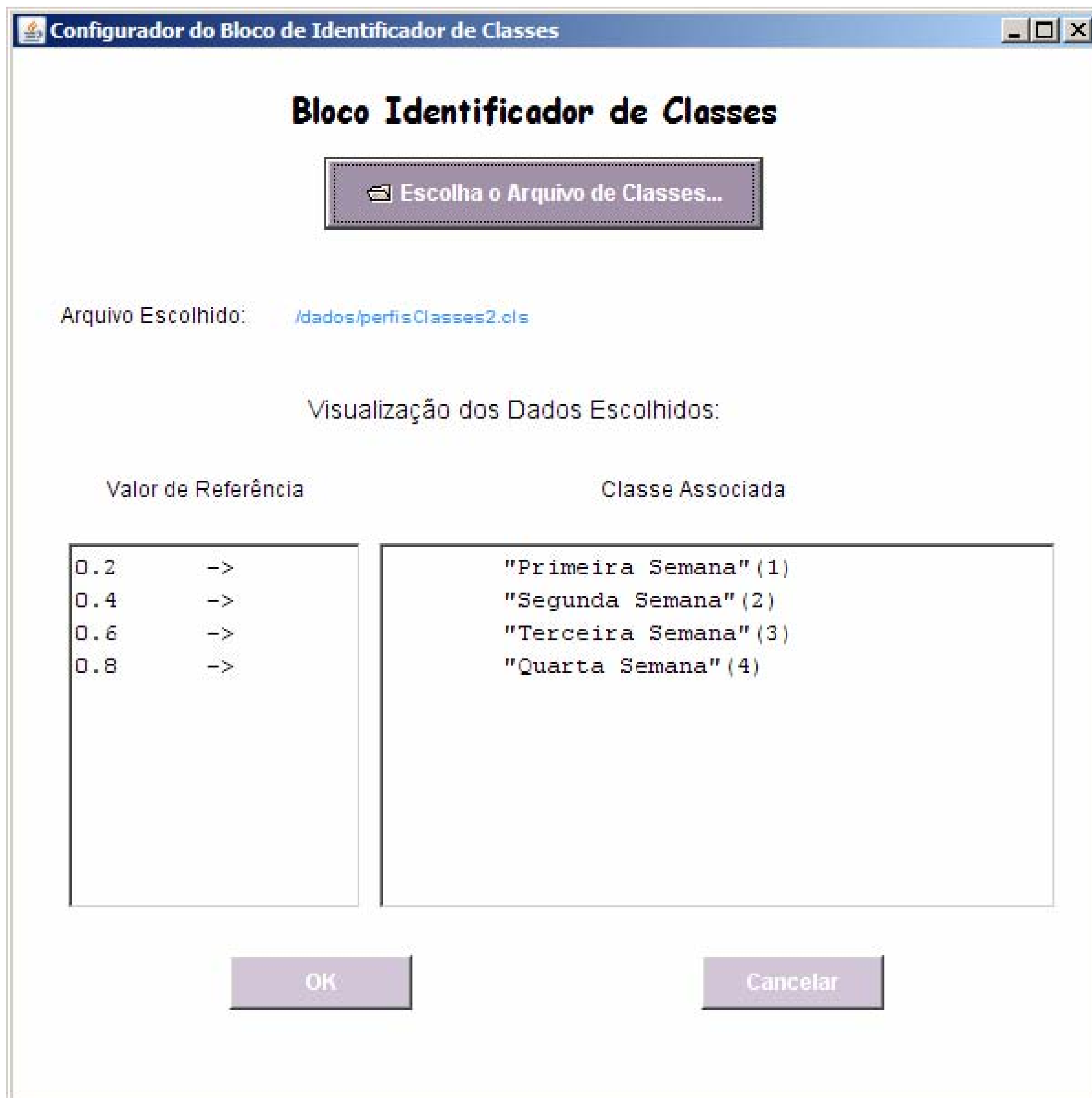


Figura 23. Arquivo de classes para referência

Comparando-se o arquivo de classes e os resultados, pode-se observar que os valores de validação foram corretamente classificados.

5. Avaliação da VisualPREV

5.1 Testes de Usabilidade

Por ser a proposta deste projeto a de uma linguagem visual, é mais adequado, num primeiro momento, concentrar a avaliação da mesma na usabilidade do sistema e não em métricas de desempenho computacional. O foco do trabalho é o de se testar a possibilidade do uso de figuras e ícones para facilitar a implementação de modelos de previsão de séries temporais e reconhecimento de padrões, o que conduz a uma métrica de avaliação baseada no usuário. Foram utilizados questionários aplicados a alguns usuários, para descrever o resultado da pesquisa.

Os testes de usabilidade foram baseados no trabalho de avaliação de usabilidade realizado em (Junior, 2005), tanto em organização quanto em aplicação. Visando validar a factibilidade da proposta de uma linguagem visual de programação para previsão de séries temporais e reconhecimento de padrões, desenvolvemos o protótipo VisualPREV e estabelecemos os principais requisitos para prover uma interação mais intuitiva, útil e amigável. No entanto, para ser considerada prática, uma linguagem visual deve provar sua efetividade tanto em suas funcionalidades como também em sua facilidade de uso. O propósito do teste de usabilidade proposto é avaliar o grau de complexidade da linguagem visual proposta, em termos de quão fácil e rapidamente um usuário pode atingir as tarefas propostas, com precisão, agilidade, facilidade e compreensão. O teste de usabilidade também fornece informações úteis sobre a forma como os usuários interagem com a linguagem, o que pode ser utilizado em posteriores melhorias da linguagem visual. A seguir será descrito o teste de usabilidade da linguagem do protótipo, a VisualPREV, utilizando dados hipotéticos de um sistema elétrico, adaptados de (Salgado et al., 2006).

5.1.1 Método

Participantes

A avaliação contou com 8 participantes (pós-graduandos), 5 homens e 3 mulheres, todos recrutados na Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, no Departamento de Engenharia de Sistemas. Alguns dos participantes não possuíam conhecimentos sobre a previsão de carga. Antes do procedimento dos testes propriamente ditos, os participantes eram orientados sobre o funcionamento e o comportamento do sistema.

Aparato Computacional (*Hardware* e *Software*)

Foi utilizada no experimento a versão mais atual da VisualPREV, que permite a criação e execução de um modelo de previsão de carga simplificado e a execução de um modelo de classificação de carga simplificado. O *software* foi executado em um computador com processador Pentium IV de 2.4 GHz, 1 GB de RAM, monitor de 17 polegadas, utilizando o mouse e o teclado para a interação. O sistema operacional utilizado foi o Windows XP e a máquina virtual Java versão 1.5_08.

5.1.2 Procedimento

Fase 1: Tutorial e instruções. Como nenhum dos participantes possuía conhecimento prévio do *software*, foi necessário um rápido treinamento, de cerca de 30 minutos, para demonstrar a linguagem e suas funcionalidades. Em seguida, os participantes eram instruídos sobre o propósito do teste e sobre os procedimentos a serem executados. A cada participante eram designadas as mesmas tarefas.

Fase 2: Execução das tarefas. As tarefas basicamente consistiam em se criar modelos, utilizando o ambiente da VisualPREV, que comportassem:

- Transferência de dados entre arquivos (simples e simultânea);
- Operações sobre dados: Extração do valor médio e Normalização pelo valor máximo;

• Execução de Modelo de Previsão de Séries Temporais contendo uma Rede Neural Artificial (convergente, convergente mais lenta e não convergente);

• Execução de Modelo de Classificação de Padrões com Decodificação dos valores classificados;

Durante a execução de cada uma destas tarefas, para todos os cenários propostos, foi efetuada uma avaliação quantitativa do sistema, levando-se em conta aspectos de velocidade – quanto tempo o participante levou para terminar a tarefa – e precisão – quão correta era a solução de cada tarefa. Após cada tarefa, foi requisitado que o usuário respondesse um questionário avaliando o sistema qualitativamente, levando em conta aspectos subjetivos. A reprodução em detalhes das tarefas entregues aos usuários encontra-se no Apêndice C.

Fase 3: Questionário pós-experimento. Terminados os procedimentos do experimento, os participantes deveriam responder um pequeno questionário para avaliar, de forma subjetiva, o grau de satisfação e a eficiência da linguagem na resolução dos problemas. As respostas eram devidamente associadas a uma escala que poderia variar entre valores de 1 (um) a 5 (cinco). A íntegra do questionário é reproduzida no Apêndice D.

5.1.3 Resultados

A partir dos testes, dois tipos de resultados puderam ser obtidos: quantitativos, onde foram aferidos os tempos gastos por cada participante para terminar cada tarefa e os qualitativos, que demonstravam, por meio de uma escala e também de opiniões subjetivas por escrito, o quão eficiente e útil se mostrou a linguagem visual de programação proposta.

Os resultados quantitativos (Figuras 24 e 25) serviram para provar que, apesar de os tempos terem se mantido dentro de uma média, o nível de experiência dos usuários com relação à manipulação de sistemas computacionais faz diferença. A partir da comparação dos tempos gastos nas tarefas com os perfis dos usuários, foi possível perceber que usuários que já tinham conhecimento mais avançado de programação, sentiram-se mais confortáveis com a linguagem e gastaram menos tempo. Outro fator que afetou os resultados foi o nível de experiência: alguns participantes possuíam menos contato com os modelos de previsão de séries temporais e com a programação em geral do que outros, o que retardou o entendimento e a execução de algumas tarefas.

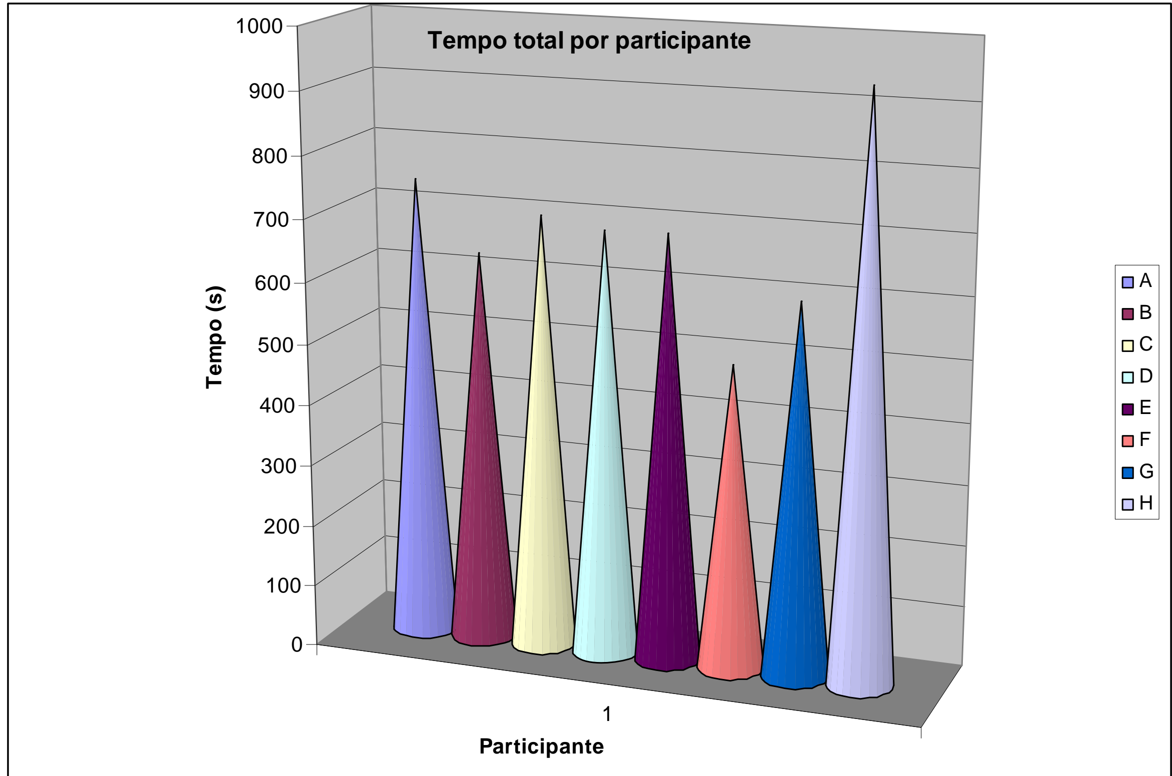


Figura 24. Tempo total por participante.

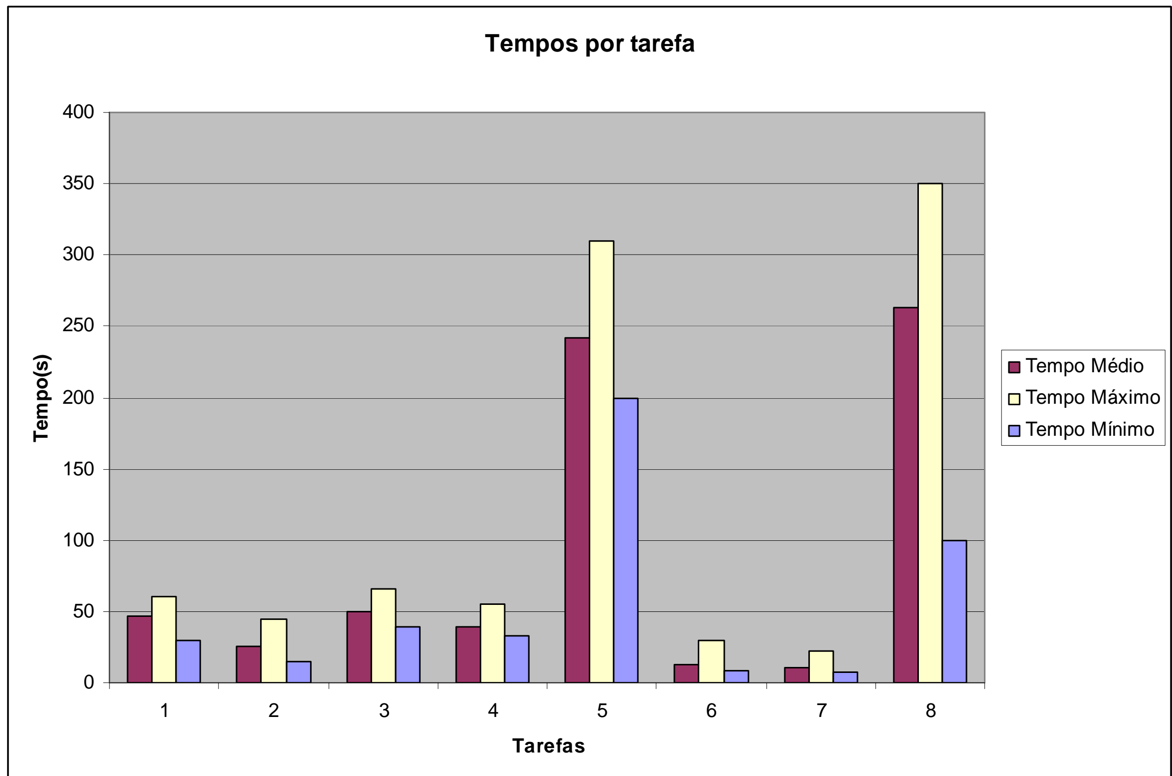


Figura 25. Tempos por tarefa.

Observando a Figura 24, pode-se notar que os participantes H e F foram os que mais se afastaram dos tempos médios de teste. O participante F possui contato prévio com paradigmas de programação orientados a fluxo de dados, provável razão pela qual teve facilidade na realização dos testes. O participante H era o que apresentava menos experiência com computação como um todo de todos os participantes, provável razão pela qual sua atividade demandou maior tempo.

De acordo com a Figura 25, as tarefas 5 e 8 demandaram maior tempo dos participantes por que nelas é necessário conhecimento para a realização das mesmas. A tarefa 5 representa a utilização do componente Rede Neural – MLP. Já a tarefa 8 representa contato com um modelo de classificação.

Para que os tempos apresentados representassem com maior fidelidade uma aferição da usabilidade da linguagem, seria necessário que os usuários tivessem um maior tempo de aprendizado das funcionalidades disponíveis no VisualPREV. Todos os participantes leram as explicações sobre o sistema, que acompanhavam o roteiro dos testes. Devido a restrições de tempo, a primeira explanação sobre o protótipo era dada apenas antes da execução do teste propriamente dito – ou seja, o participante acabaria de ouvir instruções sobre como utilizar a linguagem e, em seguida, sem tempo para reter totalmente as informações que lhe foram dadas, teria seu desempenho aferido no primeiro contato com o sistema. O ideal seria que houvesse um treinamento prévio para o aprendizado dos usuários, para que só então fossem medidos os aspectos de agilidade no uso da linguagem.

A maior contribuição dos testes sem dúvidas resultou dos aspectos qualitativos dos resultados, dando maior apoio aos aspectos quantitativos. Terminados os procedimentos do experimento, os participantes responderam a um questionário (ver Apêndice D) para avaliar, de forma subjetiva, o grau de satisfação e a eficiência da interface na resolução dos problemas. Denomina-se avaliação qualitativa ao questionário pós-experimento supra citado; estes resultados estão resumidos na Figura 26. Muitas observações e sugestões surgiram nesta etapa, contribuindo para o possível aperfeiçoamento de versões futuras do protótipo. Estas contribuições foram provenientes não apenas de sugestões explicitamente citadas pelos participantes, mas também da observação do comportamento e das heurísticas exibidas pelos usuários na resolução das tarefas. Muitos recursos que pareciam de óbvia utilização não foram usados por alguns usuários, assim como outros recursos se mostraram muito mais úteis do que o esperado – caso da janela de mensagem, por exemplo.

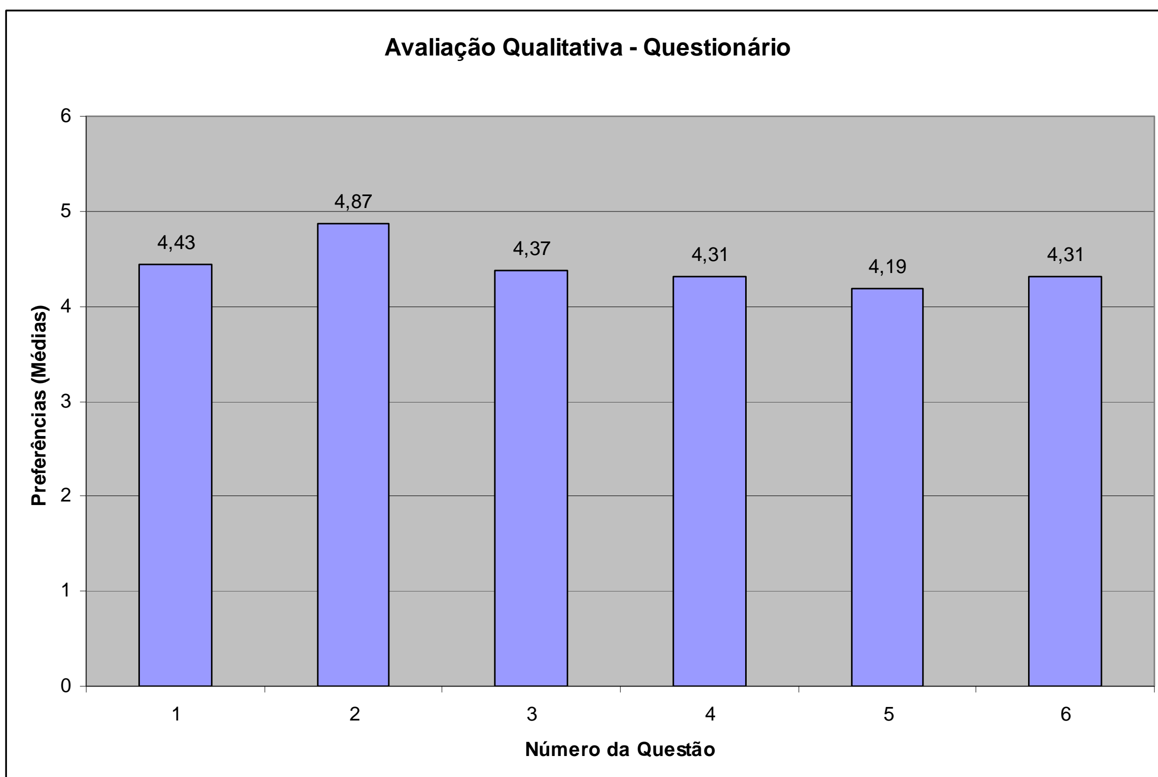


Figura 26: Avaliação qualitativa

Por ter fornecido tantos retornos positivos ao desenvolvimento do sistema, percebemos, ainda que tardiamente, que o ideal seriam várias execuções deste teste de usabilidade, em estágios anteriores do ciclo evolutivo da VisualPREV.

6. Conclusões

A tomada de decisão, em qualquer setor e nos mais diversos níveis, é um processo cada vez mais complexo, principalmente em função do nível de incerteza em relação ao futuro. Os reponsáveis por tais decisões têm que tomá-las de maneira precisa, muitas vezes em curtos espaços de tempo. Neste contexto, a disponibilidade de previsões torna-se um fator importante para uma decisão mais eficaz. As ferramentas de reconhecimento de padrões, por sua vez, são importantes em muitas áreas, tais como nas determinações de comportamentos típicos e em sistemas de controle.

Basendo-se no fato de que linguagens visuais podem desempenhar um papel importante para a programação de modelos para usuários finais (não programadores) e também para aumentar a facilidade de criação de modelos por programadores experientes, foi proposta uma linguagem visual de programação de uso específico na área de previsão de séries temporais e reconhecimento de padrões. Essa linguagem apresenta a possibilidade de armazenar e reutilizar parcial ou totalmente modelos previamente criados.

Uma das contribuições deste trabalho no contexto da previsão de séries temporais e reconhecimento de padrões foi a de traduzir conceitos dessa área de pesquisa em blocos visuais que podem ser interligados para a criação de modelos. Apesar de o número de blocos ser limitado nesse protótipo, prevê-se a implementação futura de outros blocos para a realização de outros modelos não abordados nesse trabalho. O paradigma da linguagem, baseado em fluxo de dados, é adequado para a tradução visual de uma grande variedade de modelos existentes. Portanto, a proposta não foi a de um novo paradigma de programação, mas antes sim a de um conjunto de conceitos que tentem resolver de maneira prática, para um usuário final, o problema de previsão de séries temporais e reconhecimento de padrões. A contribuição da pesquisa para a comunidade foi essencialmente de caráter tecnológico, pois não houve a criação de um novo paradigma de programação, mas sim uma nova aplicação de um paradigma já existente.

Por uma decisão inicial de projeto, baseada na tradição do laboratório COSE/DENSIS de implementar suas próprias soluções, considerou-se implementar essa linguagem sem a utilização de ambientes pré-definidos (como Simulink®, por exemplo). A principal crítica a aborgadem adotada (implementação completa) é que, utilizando-se ambientes pré-definidos,

poderia ter se criado um protótipo em menos tempo e com menor custo para somente então se avaliar as potencialidades do mesmo. Por outro lado, com a escolha feita, adquiriu-se know-how básico para conceber linguagens visuais de programação orientadas a fluxo de dados, além de obter-se flexibilidade de solução dada a especificidade do problema abordado.

Não utilizar ambientes pré-definidos foi antes uma decisão de infra-estrutura tecnológica e não de paradigma de programação. Todas as funcionalidades e características da linguagem poderiam ter sido criadas e/ou emuladas em ambiente Simulink/MATLAB.

Uma característica peculiar é que a Linguagem VisualPREV possui um "corte conceitual" em alto nível, não permitindo que os programadores acessem os códigos contidos em seus blocos. Isso siginifica que ela possui conceitos (blocos básicos em alto nível – Rede Neural, Identificador de Classes etc.) que não podem ser reduzidos em unidades menores de programação. A principal desvantagem disto é que isso limita o trabalho de programadores que desejem alterar algumas características da linguagem. A principal vantagem é que isso impede que um usuário final menos experiente cometa erros. Vale lembrar que a linguagem é de uso específico, proposta apenas para previsão de séries temporais e reconhecimento de padrões.

Consideráveis limitações da linguagem podem ser relacionadas principalmente a uma estimativa inadequada do custo de criação de uma VPL. O resultado disso foi que apenas 6 (das dezenas de blocos cujas especificações de modelos foram levantadas com os usuários) puderam ser efetivamente criados e testados; portanto, muitas das funcionalidades descritas no Capítulo 1 não puderam ser implementadas.

Como trabalho futuro, sugere-se a criação de mais blocos para a criação de outros modelos, além de a compilação das sugestões de livre opinião dados pelos participantes do teste da VisualPREV. O protótipo da VisualPREV, apesar de ainda não poder ser considerado um sistema completo e robusto, serviu para avaliar a usabilidade da linguagem visual proposta.
Devido à falta de dados do sistema elétrico brasileiro, utilizamos os dados de casos de teste adaptados disponíveis em (Salgado et al., 2006). Apesar dos testes de avaliação terem sido efetuados com base em dados hipotéticos, os resultados obtidos nos levam a concluir que a proposta é promissora. Espera-se que, no futuro, possa-se avaliar o desempenho do sistema com dados reais do sistema elétrico brasileiro.

APÊNDICE A

Análise de Alguns Exemplos de Linguagens de Programação Visual (Extraído e Adaptado de (Burnett, 1999). Neste apêndice, são discutidos cinco exemplos de VPLs e demonstradas muitas maneiras pelas quais as estratégias descritas têm sido empregadas.

Programação Visual Imperativa por Demonstração

Chimera (Kurlander, 1993) é um exemplo da maneira mais comum em que a programação imperativa é suportada em VPLs, normalmente tendo o programador demonstrado as ações desejadas. No caso de Chimera, o "programador" é um usuário-final: portanto, Chimera é um exemplo de um VPL que tem por objetivo melhorar a acessibilidade de se programar certos tipos de tarefas.

O domínio do Chimera é edição gráfica. Conforme um usuário final trabalha numa cena gráfica, ele ou ela podem achar que tarefas repetitivas de edição surgem, e pode indicar que uma seqüência de manipulações que acabaram de ser realizadas numa cena podem ser generalizadas e tratadas como macro. Isto é possível porque o histórico das ações do usuário é descrito usando uma metáfora baseada em história em quadrinhos (comic strips metaphor) (veja a Figura A.1), e o usuário pode selecionar painéis do histórico, indicando quais dos objetos deveriam ser vistos como parâmetros-exemplo, (graficamente) editar as ações descritas em qualquer dos painéis desejados, e finalmente salvar a seqüência de painéis editados como uma macro. Chimera usa inferência para determinar a versão generalizada da macro; o uso da inferência é comum em linguagens "por demonstração", e seu sucesso depende de domínios de problemas limitados tais como o do Chimera. Contudo, existem também um número de linguagens "por demonstração" que não usam inferência, uma das quais é Cocoa (discutida mais à frente neste apêndice).

Chimera está no nível de *liveness* 3; isto é, ela é capaz de prover resultado visual imediato sobre os efeitos da edição do programa. Uma vez que estes efeitos são aplicados em termos de seus efeitos nos objetos existentes no programa, isto é um exemplo de concretude (concretness). Directness em Chimera é usada, na maneira em que a semântica do programa é especificada, a partir da manipulação direta de objetos para demonstrar os resultados

desejados. Combinações similares de retorno visual mediato, concretness e directness estão presentes na maioria das VPLs por demonstração.

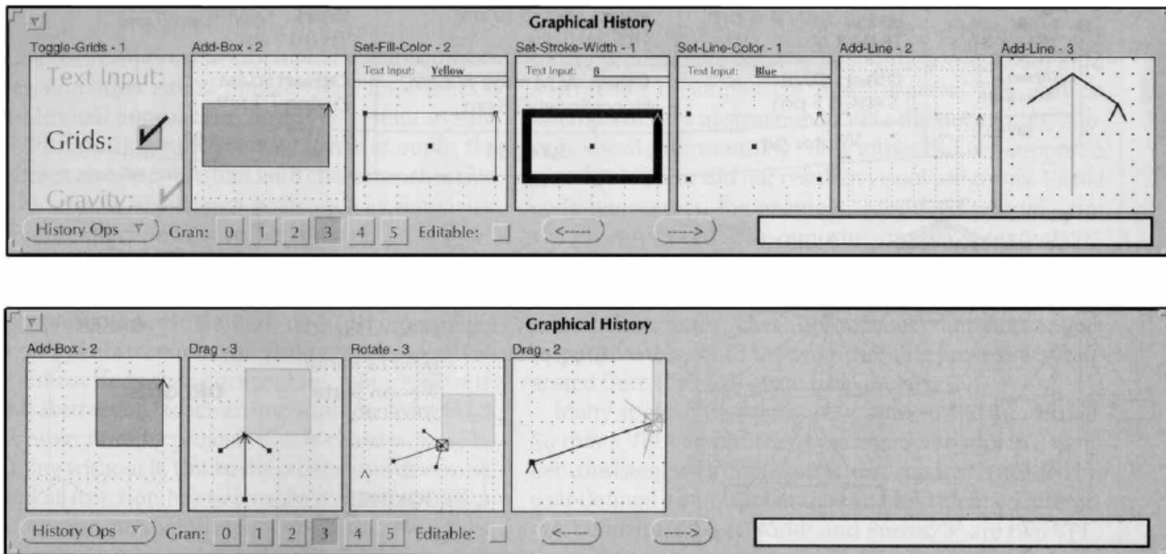


Figura A.1: Programação por demonstração no Chimera. Neste exemplo, o usuário desenhou uma caixa com uma seta apontando a ela (tal como num diagrama de grafo), e esta demonstração está descrita após ter sido realizada numa série de painéis inteligentemente filtrados. Este conjunto de demonstrações podem ser generalizado numa macro para ser usado na criação de outros nós no grafo de maneira semi-automática.

Programação Visual Baseada em Formulários e Planilhas

Forms/3 (Burnett e Gottfried, 1998) é um exemplo de uma VPL que segue o paradigma baseado em formulário. Neste paradigma, um programador programa a partir da criação de um formulário e da especificação de seu conteúdo (atributos e valores dos atributos). Este paradigma é mais comumente visto em planilhas eletrônicas comerciais, nas quais o formulário é em formato de grade, e seu conteúdo é especificado a partir das fórmulas das células.

Os programas Forms/3 incluem formulários (planilhas) com células, mas as células não estão bloqueadas (amarradas a) uma grade. O programador cria um programa a partir da manipulação direta para posicionar as células nos formulários, e define uma fórmula para cada célula usando uma combinação flexível de apontamento e digitação. As fórmulas se combinam

numa rede de restrições (de via única), e o sistema continuamente garante que todos os valores exibidos à tela satisfaçam estas restrições.

Forms/3 é uma linguagem "*Turing-complete*". O objetivo é melhorar o uso de conceitos de uma planilha ordinária para suportar a funcionalidade avançada necessária para uma programação completa em todos os aspectos. Portanto, ela suporta características tais como gráficos, animações e recursão, mas sem recorrer a macros de modificação de estado ou ligações para linguagens de programação tradicionais. Por exemplo, Forms/3 suporta uma rica e extensível coleção de tipos por permitir que atributos de um tipo sejam definidos por fórmulas e que uma instância de um tipo seja o valor de uma célula, que pode ser referenciada tanto quanto uma outra célula. Na Figura A.2, uma instância de um tipo "caixa" está sendo modificada a partir de um rascunho gráfico; esta especificação pode ser alterada, se necessário, rascunhando-se a caixa por manipulação direta. Retorno visual imediato no nível 4 é proporcionado em cada caso. *Concretness* está presente no fato de que a caixa resultante é imediatamente vista quando fórmulas em número suficiente foram fornecidas para fazer isso possível; directness está presente na manipulação direta de mecanismos para especificar uma caixa porque se pode demonstrar a especificação diretamente na caixa.

O público alvo para Forms/3 são "futuros" programadores – aqueles cujo trabalho será criar aplicativos, mas cujo treinamento não foi enfatizado nas linguagens de programação tradicionais atuais. Um dos objetivos de Forms/3 é reduzir o número e a complexidade dos mecanismos requeridos para se programar aplicativos, com a esperança de que a maior facilidade de uso por parte dos programadores resultará, diferente do que tem sido característica das linguagens tradicionais, num aumento de correctness e /ou velocidade de programação. Em estudos empíricos, programadores demonstraram maiores correctness e velocidade tanto na criação quanto no *debugging* do programa quando usaram técnicas de Forms/3, diferentemente de quando usaram uma variedade de técnicas alternativas (Burnett e Gottfried, 1998), (Cook et al., 1997), (Pandley e Burnett, 1993).

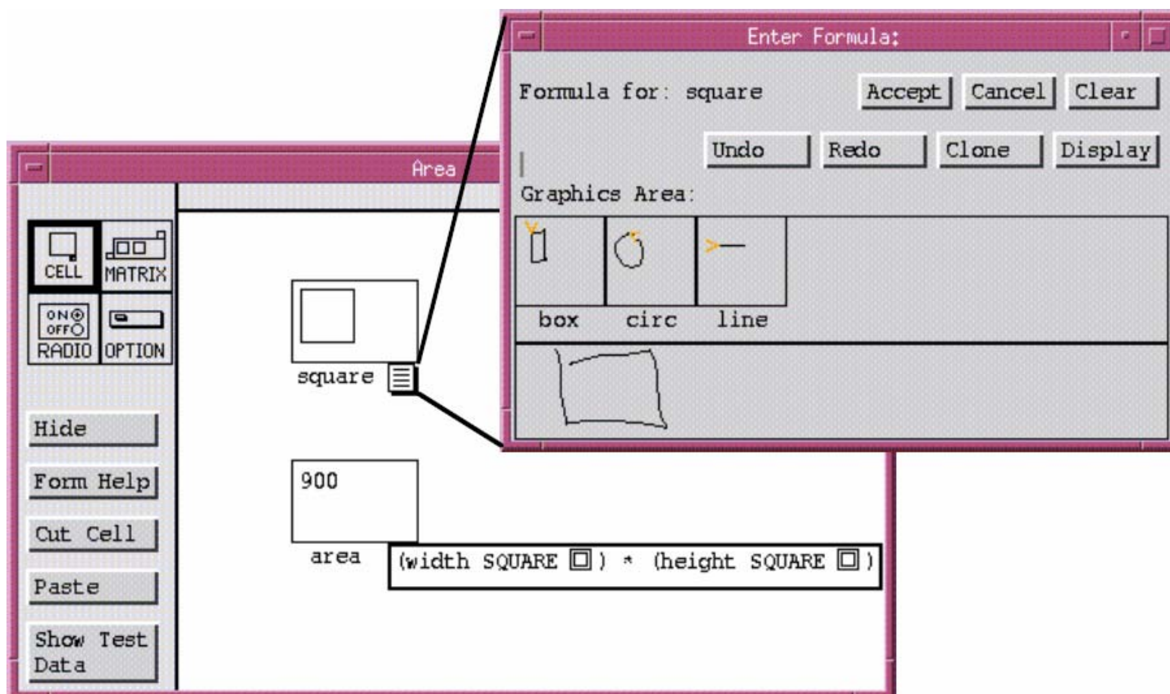


Figura A.2: Determinando a área de um quadrado usando células do tipo planilha e fórmulas em Forms/3. Tipos gráficos são suportados como valores de primeira classe, e o programador pode entrar a fórmula do quadrado da célula tanto rascunhando uma caixa quadrada quanto digitando especificações textuais (e.g., "Box 30 30").

Programação Visual Baseada em Regras

Cocoa (Smith et al., 1994) (formalmente conhecida por KidSim) é uma VPL baseada em regras em que o programador especifica as regras demonstrando uma pós-condição e uma pré-condição. Veja a Figura A.3. Os programadores-alvo são crianças, e o domínio do problema é especificar simulações gráficas e jogos. Cocoa é uma linguagem "*Turing-complete*", mas suas características ainda não foram projetadas para tornar acessível às crianças a habilidade de programar suas próprias simulações.

A maneira pela qual a concretness e directness são vistos em Cocoa é bem similar à Chimera, uma vez que ambas usam "por demonstração" como a maneira pela qual a semântica é especificada. O nível de *liveness* está entre 2 e 3. Não é nível 3 para alguns tipos de mudanças no programa (e.g., adição de novas regras) que não afetam a exibição corrente das variáveis até que a criança requisite que o programa comece a rodar, mas para outros tipos de mudanças no

programa (e.g., mudança na aparência de um objeto), as mudanças são automaticamente propagadas e exibidas imediatamente.

Ao listar propriedades comuns aos sistemas baseados em regras, Hayes-Roth incluiu a habilidade para explicar o seu comportamento (Hayes-Roth, 1985). Em Cocoa, uma criança pode abrir (selecionando ou clicando duas vezes) qualquer personagem que participe da simulação, e uma janela contendo as regras governando o comportamento dos usuários é exibida, como na Figura A.3. Em cada ciclo de execução, as regras de cada personagem são consideradas de cima para baixo na lista de personagens. Os indicadores próximos a cada regra são sem prioridade (em cinza) para uma regra sendo considerada. Então, se a regra de casamento falha, o indicador próximo a cada regra se torna vermelha; se o casamento de padrões é bem-sucedido, a regra dispara, e o indicador próximo a se torna verde. Tendo a regra disparado para um personagem, o personagem muda, e nenhuma regra a mais para aquele personagem será avaliada até o próximo ciclo.

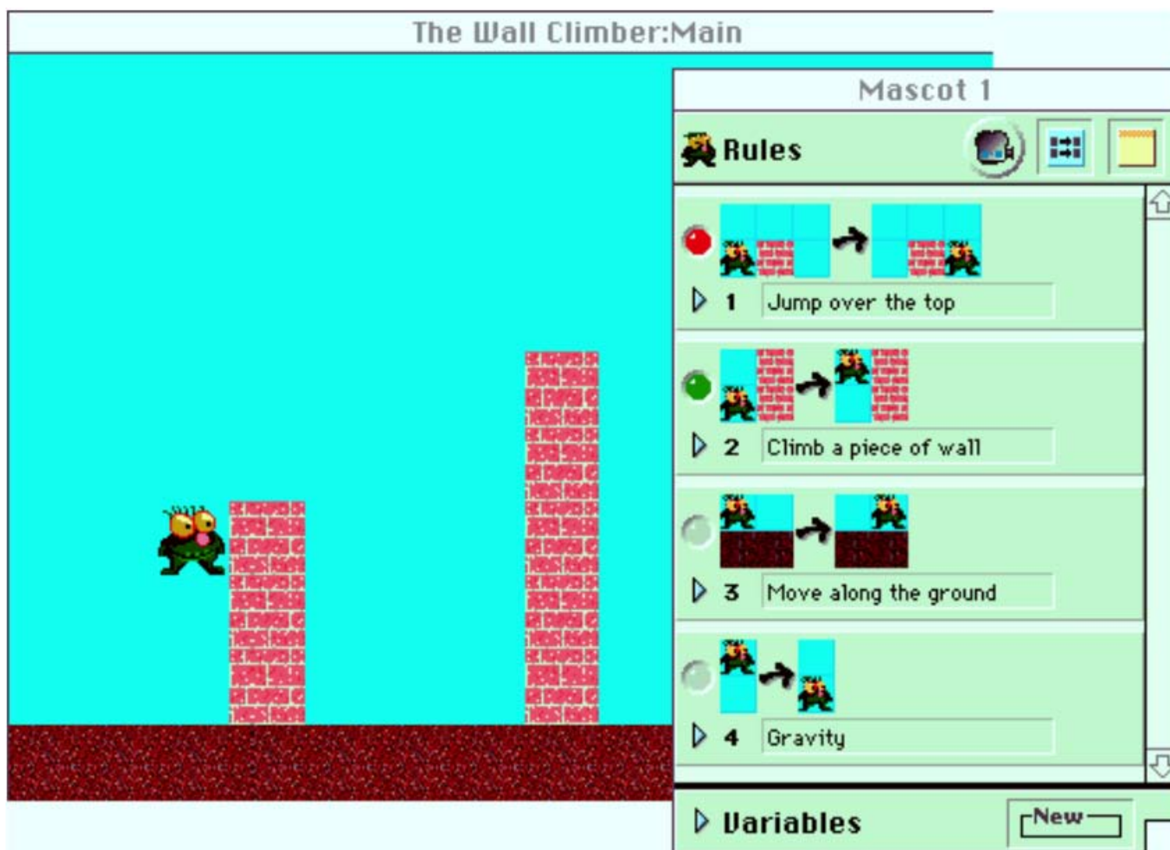


Figura A.3: Um escalador de muros de Cocoa (O escalador de muros: janela principal) está seguindo as regras (janela do Mascote 1) que foram demonstradas para ele.

Cada regra é mostrada com a pré-condição gráfica à esquerda da aresta (arrow) e a pós-condição gráfica à direita da aresta (arrow). O escalador de muros acabou de seguir a regra 2, que o posiciona numa posição adequada para seguir a regra 1 seguinte.

Programação Visual Orientada a Fluxo de Dados (1)

Prograph é uma VPL orientada a fluxo e dados dirigida a programadores profissionais. O paradigma de fluxo de dados é a abordagem para programação visual usada mais largamente na indústria. Prograph exemplifica seu uso para programação em todos os níveis, dos detalhes de mais baixo nível que podem ser agrupados em rotinas e objetos (veja a Figura A.4), para a composição de rotinas e objetos. O paradigma orientado a fluxo de dados é também comumente usado por VPLs de domínio específico para a composição de componentes de baixo nível que tem que ser escritos de alguma outra maneira; por exemplo, sistemas de visualização científica e sistemas de simulação freqüentemente fazem uso pesado de programação visual orientada a fluxo de dados.

Prograph proporciona robusto suporte de "*debugging*" fazendo uso extensivo de técnicas de visualização dinâmica. O nível de liveness é 2 para os valores de dados – o programador tem que requerer explicitamente a exibição de um valor cada vez que ele/ela quiser vê-lo. No entanto, a atividade da pilha de execução e a ordem pelas quais os nós podem disparar podem ser vistas ao longo da execução, e, se o programador mudar um bit de dados ou o código-fonte do meio da execução, a janela de pilha e as visões correlatas automaticamente se ajustam para proceder daquele ponto considerando a nova versão, e este aspecto está no nível de liveness 3.

Uma maneira na qual o paradigma orientado a fluxo de dados se distingue de outros paradigmas é através da explicitude (a partir do tratamento explícito dos arcos no grafo) sobre as relações de fluxo de dados no programa. Desde que muitas linguagens orientadas a fluxo de dados governam até mesmo o controle de fluxo por fluxo de dados, estes arcos também são suficientes para refletir o controle de fluxo explicitamente numa linguagem puramente orientada a fluxo de dados.

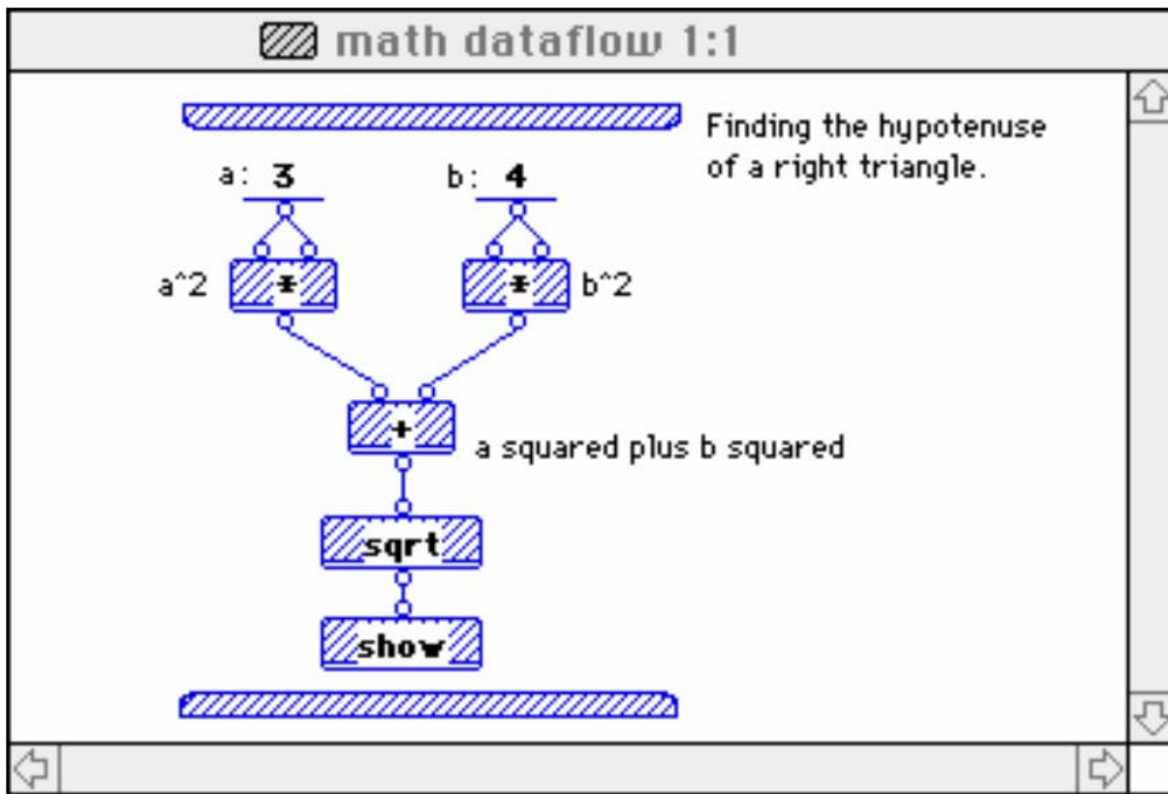


Figura A.4: A programação orientada a fluxo de dados em Prograph. Aqui, o programador está usando as operações de baixo-nível (primitivas) para encontrar a hipotenusa do triângulo-retângulo. Prograph permite ao programador nomear e compor tais gráficos de baixo nível em gráficos de nível mais alto que podem ser compostos em gráficos de nível mais alto ainda, e assim por diante.

Programação Visual Orientada a Fluxo de Dados (2)

A linguagem de programação usada em LabVIEW (Figura A.5), chamada 'G', é uma linguagem orientada a fluxo de dados. Oposta a linguagens tradicionais, a linguagem orientada a fluxo de dados não é determinada pela execução seqüencial de instruções. Ao invés disso, a execução se inicia quando todos as entradas de dados tornam-se disponíveis para um nó. Uma vez que este pode ser o caso simultaneamente para múltiplos nós, a linguagem 'G' é inerentemente capaz de execução paralela. Multiprocessamento e Multi-escalonamento em *hardware* é automaticamente explorado no escalonador interno de construção. Este multiplexa

os processos do Sistema Operacional sobre os nós que estão prontos para a execução. Programadores com experiência em outras linguagens de programação geralmente dizem que LabVIEW é propenso a condições de corrida. Na realidade, isto provém de um mal entendimento do paradigma orientado à fluxo de dados. O 'fluxo de dados' mencionado anteriormente (que pode ser 'forçado' tipicamente ligando entradas e saídas de erro de sub-vi's) na verdade previne isto completamente. Desta forma, a seqüência de execução quando apropriadamente conectada de acordo com o paradigma de fluxo de dados é equivalente à execução seqüencial de linguagens tais como C, Visual Basic ou Phyton.

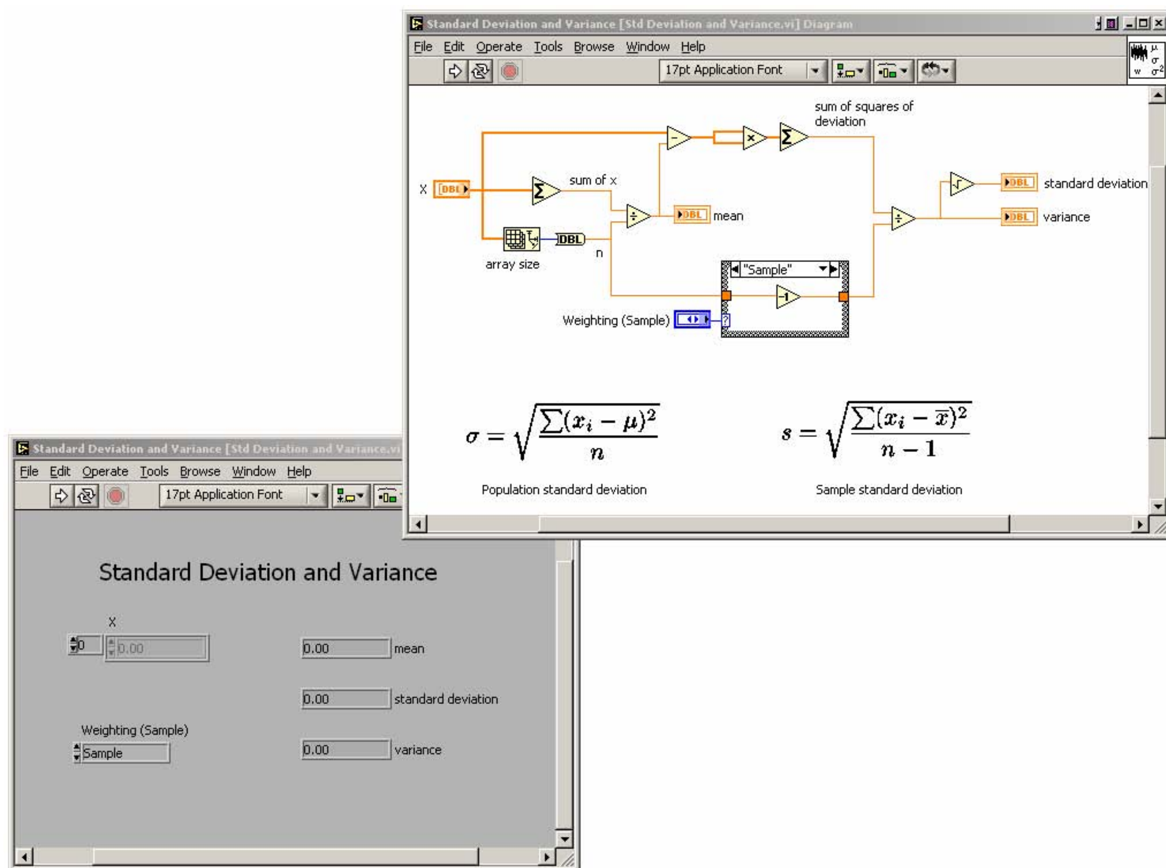


Figura A.5: Tela de um programa simples em LabVIEW que calcula a media e o desvio padrão de um vetor de entrada, mostrando o painel frontal e o diagrama de blocos.

A principal vantagem do paradigma orientado à fluxo de dados é que ele pode ser representado com uma metáfora gráfica: os nós são os ícones gráficos e o fluxo de dados percorre fios de um nó ao seguinte. Em essência, o fio é a variável. Devido à extensão de um

diagrama se encaixar perfeitamente em uma única tela, a representação gráfica em 2D faz um uso melhor da largura de banda do sistema visual humano. A geração de sub-vi's pode também auxiliar grandemente a reduzir o tamanho do código total na tela. Na realidade, muitos acreditam que a limitação da resolução da tela na verdade é fator encorajador na produção de sub-vi's, algo que conduz, em última instância, a um código melhor organizado. Há pesquisas de opinião que foram realizadas sobre o LabVIEW que mostram que, no geral, a linguagem melhora o desempenho da atividade de programação (Whitley e Blackwell, 2001).

Programação Visual Orientada a Fluxo de Dados (3)

Simulink® (Figuras A-6 e A-7), desenvolvida pela The MathWorks, é uma ferramenta para modelagem, simulação e análise de sistemas dinâmicos multidomínio. Sua interface primária é uma ferramenta de diagramação em blocos e um conjunto personalizável de bibliotecas de blocos, extensível para aplicações específicas.

Ela permite o projeto, simulação, implementação e controle de teste de processamento de sinais, comunicação e outros sistemas variantes no tempo. Produtos adicionais estendem essa funcionalidade com ferramentas específicas para geração de código, implementação algorítimica, teste e verificação. Simulink® é integrado com MATLAB®, provendo acesso imediato a um grande conjunto de ferramentas para desenvolvimento de algoritmos, visualização de dados, análise de dados e computação numérica.

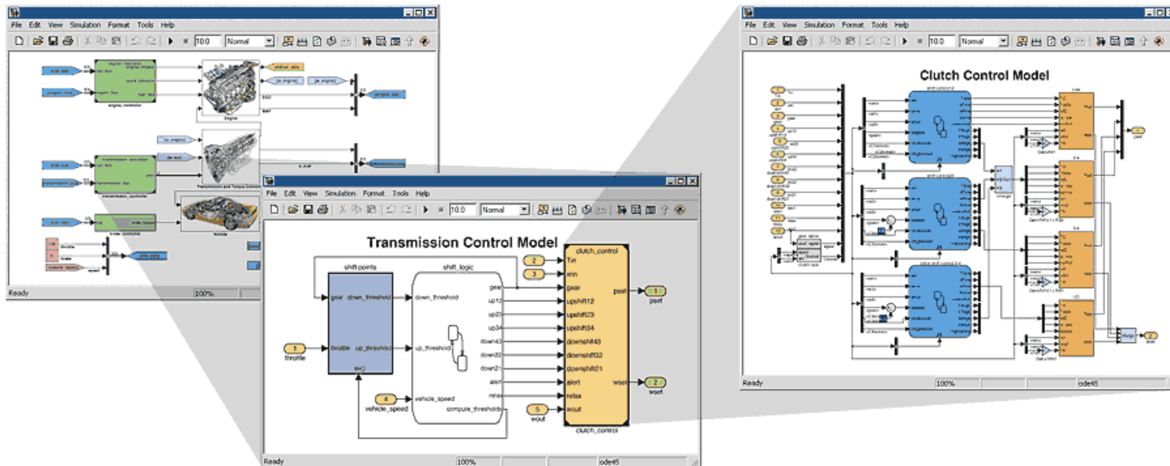


Figura A.6: Telas de modelagens-exemplo efetuadas com o Simulink.

Características principais:

1. Bibliotecas extensíveis de blocos pré-definidos;
2. Edição gráfica interativa para criar e gerir diagramas de blocos intuitivos;
3. Habilidade de gerenciar projetos complexos pela segmentação de modelos em hierarquias de components de projeto;
4. Explorador de Modelo, uma ferramenta para navegar, criar, configurar e procurar todos os sinais, parâmetros e propriedades do seu modelo;
5. Habilidade de conversar com outros programas de simulação e código escrito à mão, incluindo algoritmos MATLAB;
6. Opção de executar simulações (interativamente ou em "batch") em passo fixo ou variável de sistemas variantes no tempo;
7. Funções para interativamente definir entradas e visualizar saídas para avaliar o comportamento do modelo;
8. Analizador gráfico de erros para examinar os resultados das simulações e diagnosticar comportamentos inesperados em seu projeto;
9. Acesso integral ao MATLAB para analisar e visualizar dados, desenvolver interfaces gráficas com o usuário e criar modelos de dados e parâmetros;
10. Ferramentas de análise de modelo e de diagnóstico para garantir a consistência do modelo e identificar erros do mesmo.

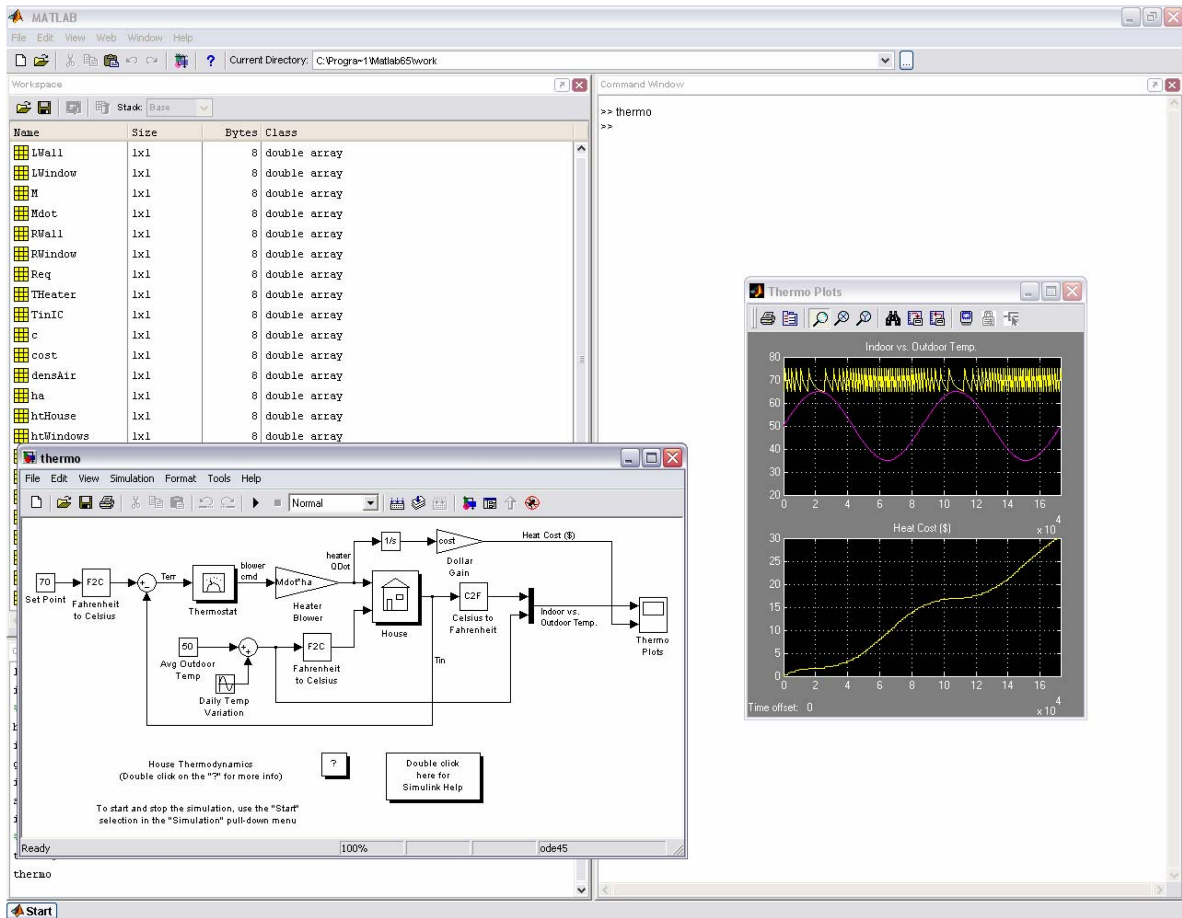


Figura A.7: Simulink executando uma simulação de um sistema termostato controlador de calor.

APÊNDICE B

Sumário das evidências descobertas em estudos empíricos sobre VPLs. (Extraído e adaptado de Whitley (Whitley, 1997))

Sumário de evidências a favor:

As pessoas, em situações de projeto e de solução de problemas, atuam melhor quando a informação está apresentada de uma maneira consistente e organizada. Além disto, as representações mais eficientes tendem a ser aquelas que tornam a informação explícita. Estas duas linhas-guia se aplicam a todas as notações, incluindo as textuais. Os estudos realizados neste trabalho indicam que, comparadas a notações textuais, as notações visuais podem proporcionar uma organização melhor e podem tornar a informação explícita. Mais além, expressões visuais apropriadamente utilizadas resultam em benefícios de desempenho mensuráveis. Muitos estudos mostraram que os expressões visuais superaram o texto ou em tempo ou em corretude (*correctness*), às vezes em ambos.

As notações não são provavelmente boas num sentido absoluto. Tal como Green escreve, 'Uma notação não é absolutamente boa, portanto, mas somente em relação a certas tarefas' (Green, 1989). Esta situação é observável no estudo de McGuinness de uma notação em forma de árvore contra uma notação em forma de matriz (McGuiness, 1986). As matrizes usadas neste estudo superaram as árvores em algumas situações, embora as árvores no estudo de Vessey e Weber consistentemente superaram as tabelas de decisão (i.e. matrizes) (Vessey e Weber, 1984). Nesta comparação em particular entre estes dois estudos, uma possibilidade que explica quando uma matriz foi útil é a sua característica de acesso sistemático. No estudo de McGuinness, as matrizes foram mais rápidas aos usuários que as árvores quando o conjunto de questões permitiu aos indivíduos procurarem na matriz de uma maneira consistente, linha-a-linha e coluna-a-coluna, que progrediu de um elemento ao elemento fisicamente adjacente. Este padrão de acesso sistemático estava faltando nas tabelas de decisão de Vessey e Weber; também, em partes do estudo de McGuinness, o padrão sistemático de elementos adjacentes podia ser equiparado às árvores. Nestes casos, as matrizes falharam em superar as árvores.

É importante notar que estes estudos lidaram amplamente com a representação não-espacial de conceitos (e.g. as instruções de cozinhar no estudo de (Scanlan, 1989)). Estes estudos contam com a especulação de que expressões visuais são benéficas apenas quando representando objetos que tem existência concreta no mundo real. Esta idéia pode surgir da observação de que usos bem-sucedidos de expressões visuais normalmente lidam com objetos espaciais concretos. Um exemplo disto é o uso bem-sucedido de diagramas para ajudar estudantes a resolverem problemas de física. Larkin e Simon fornecem exemplos disso (Larkin e Simon, 1987). Um outro exemplo é o Visual Basic; nele, as expressões visuais são usadas para construir GUIs que são essencialmente objetos espaciais. A idéia poderia também surgir de discussões tais como a de Raymond na qual ele especula qual a programação geral lida com informação discreta (e não analógica) e portanto não é adequada a visualização útil (Raymond, 1991).

Por um lado, o benefício do uso de representações visuais cresce quando o tamanho e a complexidade dos problemas cresce. Esta tendência foi observada em quatro estudos: estudo de edição de Day (Day, 1988), estudo de *flowchart* de Polich e Schwartz (Polich e Schwartz, 1974), estudo de McGuinness e estudo de *flowchart* de Scanlan. Então, VPLs podem possivelmente desempenhar um papel importante na programação tradicional, onde os problemas são usualmente maiores que os problemas usados em experimentos controlados. Por outro lado, expressões visuais podem fornecer desempenho melhor até mesmo em problemas de escala pequena. Este efeito foi notado em quatro dos estudos sumarizados: estudo de edição de Day, estudo de *flowchart* de Scanlan., estudo de *flowchart* de (Cunnif e Taylor, 1987) e estudo de Forms/3 de (Pandley e Burnett, 1993). Portanto, VPLs podem desempenhar um papel importante na programação ao usuário final, onde os problemas são usualmente menores que os encontrados por programadores profissionais.

Baroth e Harsough descrevem um estilo de programação participatória na qual os usuários finais e os programadores trabalham bem, juntos (Baroth e Harsough, 1995). Se Baroth e Hartsough estiverem corretos, então uma vantagem adicional das VPLs concentra-se na sua acessibilidade a certas classes de não programadores. As questões de se, porquê e quando as VPLs são inteligíveis aos usuários finais são certamente bons direcionadores para pesquisa.

Sumário de evidências contra:

A eficácia de uma notação depende da tarefa a ser realizada. Ambos os estudos de Green e Petre (Green e Petre, 1992) e de Moher (Moher et al., 1993) sustentam a hipótese do princípio de acoplamento-desacoplamento. Embora não sendo um resultado forte, o estudo de (Wright e Reid, 1973) também reforça a importância de considerar os efeitos da notação sobre um conjunto de tarefas. A questão para a pesquisa em VPL se torna: "dado o espectro de informação requerido na atividade de programação, uma VPL pode destacar quantidade suficiente de informação importante para ser de benefício prático?".

O estudo observacional de Hendry e Green (Hendry e Green, 1994) sobre planilhas também reforça a hipótese de acoplamento-desacoplamento. O sucesso de mercado da planilha eletrônica torna claro que elas são úteis; ainda, mesmo assim, a planilha não facilita todas as tarefas cognitivas. Além disto, Hendry e Green nos lembram que os projetos de linguagens estão situados num espaço de projetos complexo e que as decisões feitas durante o processo de projeto podem impedir certas tarefas cognitivas. Estudos empíricos podem auxiliar em tais situações atuando no sentido de melhorias.

Alguns membros da comunidade VPL descartam os estudos de *flowchart* como irrelevantes às VPLs atuais, sendo seu argumento que a maior parte das VPLs atuais não são baseadas em *flowchart*. Contudo, os estudos de *flowchart* não deveriam ser completamente ignorados. Sua aplicabilidade óbvia é para as VPLs que usam representações visuais para representar controle de fluxo. Portanto, os estudos de *flowchart* têm uma influência direta em VPLs baseadas em *flowchart*s, tais como Pict (Glinert et al., 1990) e FPL (Cunnif e Taylor, 1987). Além disto, os estudos de *flowchart* podem também ser ligados a VPLs que, embora não baseadas em *flowchart*, são projetadas para ilustrar o fluxo de controle de um programa; esta categoria inclui as notações estruturadas em árvore (Aoyama et al., 1989), (McHenry, 1990) e projetos mais exóticos tais como VIPR (Citrin, 1997). A implicação dos estudos de *flowchart* é que o controle de fluxo sozinho não parece comprometer uma parte grande o suficiente do que tornar a programação uma tarefa difícil.

Um fato notável do estudo de *flowchart* de Curtis (Curtis, 1989) é a tarefa importante que diferenças individuais desempenharam no experimento. Em seus resultados, as diferenças individuais contabilizavam tipicamente um terço ou metade da variabilidade vista na variáveis dependentes. Tal como mencionado acima, o impacto das diferenças individuais é um tema

recorrente em estudos empíricos em programação e poderia fazer os efeitos das VPLs difíceis de serem descritos. Este tópico também conduz-nos a lembrar da distinção entre significância estatística e benefício realizado (i.e., magnitude do impacto). Mais importante do que se considerar se uma VPL tem efeito estatisticamente significativo é a questão de se o tamanho do efeito é grande o suficiente para ser de interesse prático.

No estudo de Ramsey, o uso de técnicas de poupar espaço nos levanta a discussão quanto ao "Limite de Deutsch" (Ramsey et al., 1983). O termo Deutsch se refere à densidade relativamente baixa das notações visuais – impossibilidade empírica de se ter mais que 50 primitivas visuais na tela ao mesmo tempo – quando comparadas com as notações textuais. Isto tem sido um tópico reconhecido há tempos pela comunidade de programação visual, que alimenta a especulação de que VPLs falharão em serem práticas. Esta questão envolve não apenas como os efeitos visuais afetam a natureza do texto usado em conjunção com as expressões visuais, mas também se as VPLs permitem um número suficiente de elementos visuais numa tela ao mesmo tempo.

Notação secundária é um tópico tanto para notações textuais quanto para notações visuais. No caso do texto, o espectro das técnicas de notação é relativamente limitado devido à natureza simbólica do texto e a seu layout serial básico. Em notações visuais, a notação secundária pode ser potencialmente muito enriquecida. Se as afirmações de Green e Petre estiverem corretas, usuários novatos não irão dominar imediatamente a notação secundária das VPLs. Isso permanece para sabermos se os usuários novatos tem problemas equivalentes ou superiores com a notação secundária em programas visuais tanto quanto em programas textuais. Este tópico poderia ser especialmente problemático para VPLs que visam o usuário final.

Algumas das dificuldades nos estudos em VPLs surgiram de problemas de projeto experimental. A identificação das variáveis independentes, escolha cuidadosa se uma variável validamente dependente e a noção clara da tarefa do experimento são também importantes para o sucesso do estudo. Ainda, atenção particular deveria ser tomada a essas decisões sobre uma área ainda pouco compreendida. No estudos de VPL, a maior parte das variáveis dependentes tem sido o tempo e a corretude; Scanlan pensa que a compreensão do tempo terá o efeito mais importante nas notações visuais.

Os estudos de animação de algoritmos, com um grupo, não puderam trazer nenhum benefício às animações para se aprender algoritmos. Isto poderia verdadeiramente refletir o

fato de que animações não oferecem benefícios sobre as técnicas de ensino. Em contraste, Gurka e Citrin pensam que esta conclusão é prematura (Gurka e Citrin, 1996). Eles apresentam uma meta-análise de estudos de animação existentes que enfatiza a possibilidade de que estudos anteriores deixaram de obter os efeitos das animações. Por exemplo, tanto no estudo Statsko (Statsko et al., 1993) quanto em Lawrence (Lawrence, 1994), as medidas de efetividade eram testes de compreensão com pontuação. Talvez os efeitos da animação possam ser vistos num estudo que tenha examinado o tempo necessário para se aprender um algoritmo.

APÊNDICE C

Instruções Sobre os Experimentos para os Participantes

Leia atentamente a descrição dos cenários e em seguida efetue as tarefas propostas.

Cenário 1 – Transferência de dados entre arquivos. Pede-se:

A – Realizar a transferência de dados de um arquivo fonte "dadosTreinamento.in" para um arquivo, a ser criado, "dadosDestino.out".

B – Realizar a transferência de dados de um arquivo fonte "dadosValidacao.in" para "dadosDestinoA.out" e, simultaneamente, a transferência de dados de um arquivo fonte "dadosTreinamento.in" para "dadosDestinoB.out".

Cenário 2 – Operações obre dados: Extração do valor médio e Normalização pelo valor máximo. Pede-se:

A – Configurar um modelo de transferência de dados entre arquivos (de "dadosTreinamento.in" para "valorMedioDadosTreinamento.out"), contendo um bloco de cálculo de valor médio entre eles. Executar o modelo e verificar o resultado.

B – Configurar um modelo de transferência de dados entre arquivos (de "dadosTeste.in" para "dadosNormalizadosDadosTeste.out"), contendo um bloco de normalização entre eles. Executar o modelo e verificar o resultado.

Cenário 3 – Execução de Modelo de Previsão de Séries Temporais contendo uma Rede Neural Artificial. Pede-se:

A – Configurar um modelo de RNA, contendo, os conjuntos de dados de entrada padrão do sistema (“dadosTreinamento.in”, “dadosTeste.in” e “dadosValidacao.in”) e os conjuntos de dados de saída (“vetorPesos.out”, “vetorSaidaPrevista.out” e “vetorErros.out”). Configure a RNA com os seguintes parâmetros: (Tipo de RNA: Classic Back-Propagation); (Número de Neurônios na camada intermediária: 2); (Taxa de Aprendizagem: 0.9); (Termo de Momento: 0.2).

B – Configurar um modelo de RNA, contendo os mesmos arquivos de entrada e de saída. Configure a RNA com os seguintes parâmetros: (Tipo de RNA: Classic Back-Propagation); (Número de Neurônios na camada intermediária: 2); (Taxa de Aprendizagem: 0.1); (Termo de Momento: 0.2). Observe que o modelo demorará muito mais tempo para convergir. Observe que essa configuração demora aproximadamente 4 vezes mais tempo para rodar que a anterior, levando-se em conta a diferença na taxa de aprendizagem.

C – Configurar um modelo de RNA, contendo, os conjuntos de dados de entrada padrão do sistema (“dadosTreinamento1.in”, “dadosTeste1.in” e “dadosValidacao1.in”) e os conjuntos de dados de saída (“vetorPesos1.out”, “vetorSaidaPrevista1.out” e “vetorErros1.out”). Configure a RNA com os seguintes parâmetros: (Tipo de RNA: Classic Back-Propagation); (Número de Neurônios na camada intermediária: 2); (Taxa de Aprendizagem: 0.5); (Termo de Momento: 0.2). Se o modelo não convergir (ficar com erro oscilante ou crescente por muito tempo), apenas tecle CTRL+C para cancelar a execução do mesmo.

Cenário 4 – Execução de Modelo de Classificação de Padrões com Decodificação dos valores classificados

A – Configurar um modelo de previsão de séries temporais com identificação de padrões, utilizando, na saída de previsão da RNA, o bloco de identificação de padrões. Executar o modelo e verificar o resultado. Configurar um modelo de RNA, contendo, os conjuntos de dados de entrada padrão do sistema (“dadosTreinamento.in”, “dadosTeste.in” e “dadosValidacao.in”) e os conjuntos de dados de saída (“vetorPesos2.out” e

“vetorErros2.out”). Configure a RNA com os seguintes parâmetros: (Tipo de RNA: Classic Back-Propagation); (Número de Neurônios na camada intermediária: 2); (Taxa de Aprendizagem: 0.5); (Termo de Momento: 0.2). Na saída da RNA, acrescente um bloco de identificação de classes (“perfisClasses.cls”) e um bloco de saída (“vetorSaidaPrevista2.out”). Observe as classes decodificadas.

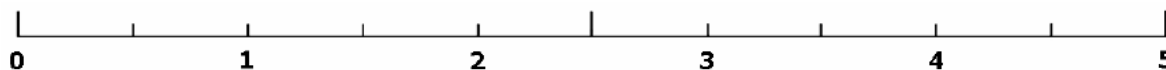
Durante a execução de cada uma das tarefas, para todos os cenários propostos, será efetuada uma avaliação quantitativa do sistema, levando-se em conta aspectos de velocidade – quanto tempo o participante levou para terminar a tarefa – e precisão – quão correta era a solução de cada tarefa. Após cada tarefa, será requisitado que o usuário responda um questionário, que avaliará o sistema qualitativamente, levando em conta aspectos subjetivos por parte do usuário.

APÊNDICE D

Questionário Pós-Experimento

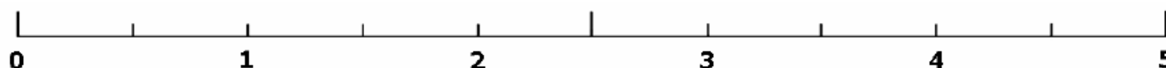
Terminados os procedimentos do experimento, os participantes devem responder o questionário abaixo para avaliar, de forma subjetiva, o grau de satisfação e a eficiência da interface na resolução dos problemas. Leia atentamente as questões e faça um ***círculo*** na escala de avaliação disponibilizada para cada pergunta. As respostas serão devidamente associadas a uma escala que poderia variar entre valores de 1 (um) a 5 (cinco), de acordo com o nível que você experimentou em cada uma das tarefas que acabou de executar. Além disso, caso haja necessidade, escreva comentários adicionais (sugestões, reclamações, opiniões etc.) no espaço destinado em cada questão.

1 – Foi fácil modelar a transferência de dados entre arquivos nos 2 (dois) casos propostos?



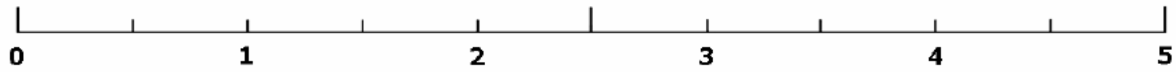
Comentários adicionais:

2 – Foi fácil modelar as operações de Extração do valor médio e Normalização sobre os dados?



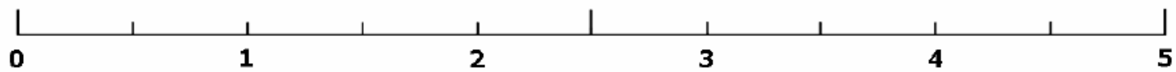
Comentários adicionais:

3 – Foi fácil modelar as simulações de previsão de séries temporais com a RNA?



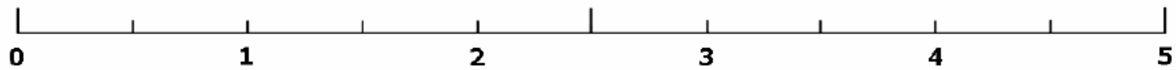
Comentários adicionais:

4 – Foi fácil modelar a simulação de de classificação de padrões com decodificação dos valores classificados?



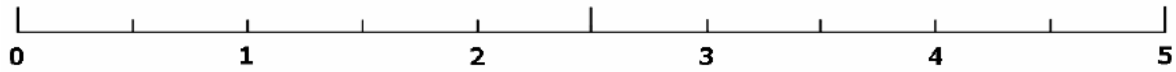
Comentários adicionais:

5 – A apresentação das funcionalidades da VisualPREV são condizentes com sua expectativa?



Comentários adicionais:

6 – Você se sentiu inseguro ou desencorajado durante a realização das tarefas?



Comentários adicionais:

7 – Escreva aqui suas sugestões, críticas ou opiniões para a melhoria da VisualPREV.

TERMO DE CONSENTIMENTO DO PARTICIPANTE

NOME COMPLETO:____

DEPARTAMENTO:____

POSIÇÃO (Circule uma): Graduação Pós-Graduação Docente

Os testes aqui conduzidos serão utilizados para a avaliação da usabilidade do trabalho de pesquisa desenvolvido na Faculdade de Engenharia Elétrica da UNICAMP para a tese de mestrado de Joaquim José Fantin Pereira, orientado pelo Prof. Dr. Takaaki Ohishi. Reconheço que minha participação neste teste é voluntária e que poderei me ausentar a qualquer momento caso desejar.

ASSINATURA:_______________________

Campinas, ____ de Outubro de 2006.

REFERÊNCIAS BIBLIOGRÁFICAS

Aoyama, M., Miyamoto, K., Murakami, N., Nagano, H. e Oki, Y. (1989) *"Design Specification in Japan: Tree-Structured Charts"*. IEEE Software **6**(3), 31-37.

Baroth, E. e Hartsough, C. (1995) *"Visual Programming in the Real World"*. Em (M. Burnet, A. Goldberg, T. Lewis, editores), Visual Object-Oriented Programming: Concepts and Environments, Prentice-Hall, Englewood Cliffs, NJ; Manning Publications, Greenwich, Connecticut; e IEEE, Los Alamitos, California.

Boehm, B. W., Abts, C., Brown, A. W. e Chulani, S. (2000) *"Software Cost Estimation with COCOMO II"*. Prentice-Hall, PTR, Upper Sadle River, Nj.

Burnett, M. M. e Baker, M. J. (1994) *"A Classification System for Visual Programming Languages"*. Journal of Visual Languages and Computing, 5, 287-300.

Burnett, M., Baker, M., Bohus, P., Carlson, P., Yang, S. e van Zee, P. (1995) *"The Scaling Up Problem for Visual Programming Languages"*. IEEE Computer, 28, 45-54.

Burnett, M. e Gottfried, H. (1998) *"Graphical Definitions: Expanding Spreadsheet Languages trough Direct Manupulation and Gestures"*. ACM Transactions on Computer-Human Interaction **5**(1).

Burnett, M. M. (1999) *"Visual Programming"*. Encyclopedia of Electrical and Eletronics Engineering. (John G Webster, ed.), John Wiley & Sons Inc., New York.

Citrin, W., Doherty M. e Zorn, B. (1997) *"Formal Semantics of Control in a Completely Visual Programming Language"*. Proceedings IEEE Symposium on Visual Languages (VL'94), 208-215.

Cook, C., Burnett, M. e Boom, D. (1997) *"A Bug's Eye View of Immediate Visual Feedback in Direct-Manipulation Programming Systems"*. Empirical Studies of Programmers: Seventh Workshop, Alexandria, Virginia, 20-41.

Costagliola, G., Orefice, S., Polese, G., Tortora, G., Tucci, M. (1993) *"Automatic Parser Generation for Pictorial Languages"*. IEEE Symposium on Visual Languages, 306-313.

Costagliola, G. Adelucia, S. Orefice e G. Polese. (2002) *"A Classification Framework to Support the Design of Visual Languages"*. Journal of Visual Languages and Computing, 13, 573-600.

Courcelle, B. (1990) *"Graph Rewriting: an Algebraic and Logic Approach"*. Handbook of Theoretical Computer Science (J. van Leeuwen, editor), Elsevier, Amsterdan, Holanda.

Cunnif, N. e Taylor, R. P. (1987) *"Graphical vs. Textual Representation: an Empirical Study of Novice's Program Comprehension"*. Empirical Studies of Programmers: Second Workshop, 114-131.

Curtis, B., Sheppard, S. B., Kruesi-Bailey, E., Bailey, J. e Boehm-Davis, D. A. (1989) *"Experimental Evaluation of Software Documentation Formats"*. Journal of Systems and Software, 9, 167-207.

Day, R. S. (1988) *"Alternative Representations"*. The Psychology of Learning and Motivation, Vol. 22 (G. H. Bower, ed.), Academic Press, New York, 261-305.

De Kleer, J. e Brown, J. S. (1980) *"Mental Models of Physical Mechanisms"*. Xerox, PARC Cognitive and Instructional Sciences, Palo Alto, California.

Ferruci, F., Tortora, G., Tucci, M., Vitiello, G. (1994) *"A Predictive Parser for Visual Languages Specified by Relation Grammars"*. IEEE Symposium on Visual Languages, 245-252.

Gips, J. E. (1974) *"Shape Grammars and their Uses"*. Ph.D. thesis, Stanford University.

Glinert, E. P. (1990) *"Principles of Visual Programming Systems"*. Capítulo 3 ("Nontextual Programming Environments"), Prentice-Hall International, 144-230.

Glinert, E. P., Kopache, M. E. e McIntyre, D. W. (1990) *"Exploring the General-Purpose Visual Alternative"*. Journal of Visual Languages and Computing, 1, 3-39.

Golin, E. J. e Reiss, S. P. (1989) *"The Specification of Visual Language Syntax"*. IEEE Symposium on Visual Languages, 118-125.

Golin, E. J. e Reiss, S. P. (1990) *"The Specification of Visual Language Syntax"*. Journal of Visual Languages and Computing, Número 2, Volume 1, 141-157.

Golin, E. J. (1990) *"A Method for the Specification and Parsing of Visual Languages"*. Brown University.

Golin, E. (editor). (1991) *"Special Issue on Theory of Visual Languages"*. Journal of Visual Languages and Computing, **2**(4).

Green, T.R. (1989) *"Cognitive Dimensions of Notations"*. People and Computers V, British Computer Society, HCI'89, 443-460.

Green, T. R. G. e Petre, M. (1992) "*When Visual Programs are Harder to Read than Textual Programs"*. Proceedings 6th European Conference on Cognitive Ergonomics (ECCE 6), 167-180.

Green, T. e Petre, M. (1996) *"Usability Analysis of Visual Programming Environments: a 'Cognitive Dimensions' Framework"*. Journal of Visual Languages and Computing, **7**(2), 131-174.

Gurka, J. S. e Citrin, W. (1996) *"Testing Effectiveness of Algorithm Animation"*. Proceedings IEEE Symposium on Visual Languages (VL'96), 182-189.

Haarslev, V. (1993) *"Formal Semantics of Visual Languages Using Spatial Reasoning"*. IEEE Symposium on Visual Languages, 156-163.

Hayes-Roth, F. (1985) *"Rule-Based Systems"*. Comunications of the ACM **28**(9), 921-932.

Helm, R. e Marriot, K. (1991) *"A Declarative Specification and Semantics for Visual Languages"*. Journal of Visual Languages and Computing, 2, 311-331.

Helm, R., Marriot, K. e Odersky, M. (1991) *"Bulding Visual Language Parsers"*. ACM Conf. Human Factors in Computing, 105-112.

Hendry, D. G. e Green, T. R. G. (1994) *"Creating, Comprehending and Explaining Spreadsheets: a Cognitive Interpretation of What Discretionary Users Think of Spreadsheet Model"*. International Journal of Human-Computer Studies, 40, 1033-1065.

Hockenbury, D. H. e Hockenbury, S. H. (2003) *"Descobrindo a Psicologia"*. Primeira Edição Brasileira, Editora Malone.

Hutchins, E., Hollan, J. e Normann, D. (1986) *"Direct Manipulation Interfaces"*. Em D. Norman e S. Draper (editores), "User Centered System Design: New Perspectives on Human-Computer Interaction", Lawrence Erlbaum Assoc., Hillsdale, NJ, 87-124.

Junior, C. F. S. (2005) *"Um Modelo de Interação Gráfica para Suporte ao Pré-despacho de Sistemas de Energia Elétrica",* Dissertação de Mestrado, FEEC/Unicamp, Campinas.

Kadowaki, M., Ohishi, T., Soares, S., Ballini, R. (2002). *"Short-term load forecasting using a neurofuzzy network model"*, XIV CBA, Natal, RN, 365-370.

Kurlander, D. (1993) *"Chimera: Edição Gráfica baseada em Exemplo"*. D. Norman e S. Draper, editores, Watch what I do: Programming by Demonstration, MIT Press, Cambridge, Mass.

Lakin, F. (1986) *"Spatial Parsing for Visual Languages"*. Visual Languages, (editores: K. Ghang, T. Ichikawa e P. Ligomenides), Plenum Press, Nova York, 35-85.

Larkin, J. H. e Simon, H. A. (1987) *"Why a Diagram is (sometimes) worth ten thousands words",* Cognitive Science, 11, 65-99.

Lawrence, A., Badre, A. e Statsko, J. (1994) *"Empirically Evaluating the Use of Animations to Teach Algorithms"*. Proceedings IEEE Symposium on visual Languages (VL'94), 48-54.

Lohr, S. (2001) *"The Programmers who Created the Software Revolution - Go To"*. Basic Books, New York.

Lyon, P., Simmons, C. e Apperley, M. (1993) *"Hyperpascal: A Visual Language to Model Idea Space"*. Proceedings of the 13th New Zealand Computer Society Conference, New Zealand, 492-508.

Malone, T. W. (1980) *"What Makes Things Fun to Learn? A Study of Intrinsically Motivating Computer Games"*. Ph.D. Thesis, Dept. of Psychology, Stanford University (Xerox PARC), Palo Alto, Calif.

Marriot, K. (1994) *"Constraint Multiset Grammars"*. IEEE Symposium on Visual Languages, 118-125.

Marriot, K. e Meyer, B. (1997) *"On the Classification of Visual Languages by Grammar Hierarchies"*. Journal of Visual Languages and Computing, 8, 375-402.

Mayer, R. E. (1981) *"The Psychology of How Novices Learn Computer Programming"*. ACM Computing Surveys, **13**(1), 121-141.

McClearly Jr., C. F. (1983) *"An Effective Graphic 'Vocabulary"*, IEEE Computer Graphics and Applications, **3**(2), 46-53.

McGuiness, C. (1986) *"Problem Representation: the Effects of Spatial Arrays"*. Memory and Cognition, 14, 270-280.

McHenry, W. K. (1990) *"R-Technology: a Soviet Visual Programming Environment"*. Journal of Visual Languages and Computing, 1, 199-212.

McIntyre, D. W. e Glinert, E. P. (1990) *"Visual Tools for Generating Iconic Programming Environments"*. IEEE Workshop on Visual Languages, Seattle, W. A., 162-168.

Meyer, B. (1992) *"Pictures Depicting Pictures"*. IEEE Symposium on Visual Languages, 41-47.

Meyer, R. M. (1995) *"www-cs.canisius.edu/~meyer/"*. Canisius College, Buffalo, NY, Assoc. Professor of Computer Science.

Modugno, F., Corbett, A. e Meyers, B. (1996) *"Evaluating Program Representation in a Demonstrational Visual Shell"*. Empirical Studies for Programmers: Sixth Workshop, Alexandria, Virginia, 131-146.

Moher, T. G., Mak D. C., Blumenthal, B. e Leventhal, L. M. (1993) *"Comparing the Comprehensibility of Textual and Graphical Programs: the Case of Petri Nets"*. Empirical Studies of Programmer: 5th Workshop. 137-161.

Mussio, P. (1993) *"Representation problems in Visual language Design"*. Journal of Visual Languages and Computing, 4, 325-326.

Myers, B. A., (1990) *"Taxonomies of Visual Programming and Program Visualization"*. Journal of Visual Languages and Computing, Número 1, Volume 1, 97-123.

Nardi, B. (1993) *"A Small Matter of Programming: Perspectives on End User Computing"*. MIT Press, Cambridge, Mass.

O'Brien. L. (1993) *"Issues of Programming"*. Computer Language, Volume 10, Número 1, 45-52.

Pandley, R. e Burnett, M. (1993) *"Is it easier to write matrix manipulation programs visually or textually? An Empirical study"*. IEEE Symposium on Visual Languages, Bergen, Norway, 344-351.

Pane, J.F., e Myers, B.A. (2002) *"The Impact of Human-Centered Features on the Usability of a Programming System for Children"*, CHI 2002 Extended Abstracts: Conference on Human Factors in Computing Systems, Minneapolis, MN: ACM Press, 20-25, 684-685.

Pane, J.F. (2002) *"A Programming System for Children that is Designed for Usability"*. Ph.D. Thesis, Carnegie Mellon University, Computer Science Department, CMU-CS-02-127, Pittsburgh, PA.

Polich, J. M. e Schwartz, S. H. (1974) *"The Effect of Problem Size on Representation in Deductive Problem Solving: the Matrix"*. Memory & Cognition 2, 683-686.

Ramsey, H. R., Atwood, M. E. e Van Doren, J. R.. (1983) *"Flowcharts versus Program Design Languages: an Experimental Comparison"*. Communications for the ACM 26, 445-449.

Raymond, D. R. (1991) *"Characterizing Visual Languages"*. Proceedings IEEE Workshop on Visual Languages (VL'91), 176-182.

Rekers, J. e Schürr, A. (1997) *"Defining and Parsing Visual Languages with Layered Graph Grammars"*. Journal of Visual Languages and Computing, 8, 27-55.

Rosenfeld, A. (1976) *"Array and Web Grammars: an Overview"*. Em: *Automata, Languages and Development* (A. Lindenmayer & G. Rosenberg, editores). North-Holland, Amsterdan, 517-529.

Ruthruff, J. R., Prabhakararao, S., Reichwein, J., Cook, J., Creswick, E. e Burnett, M. (2005) *"Interactive, Visual Fault Localization Support for End-Users Programmes"*. Journal of Visual Languages and Computing, 16, 3-40.

Salgado, R. M., Ohishi, T., Ballini, R. (2004). *Clustering Bus Load Curves*, Power Systems Exposition and Conference PSCE – IEEE, New York - USA.

Salgado, R. M. (2004) *"Um Modelo de Previsão de Carga por Barramento",* Dissertação de Mestrado, FEEC/Unicamp, Campinas.

Salgado, R. M., Pereira, J. J. F., Ohishi, T., Ballini, R., Lima, C. A. M. and Von Zuben, F. J. (2006) *"A Hybrid Ensemble Model Applied to the Short-Term Load Forecasting Problem",* 2006 International Joint Conference on Neural Networks Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada.

Scanlan, D. A. (1989) *"Structured Flowcharts Outperform Pseudocode: an Experimental Comparison".* IEEE Software, 6, 28-36.

Sheil, B. A. (1981) *"The Psychological Study of Programming".* ACM Computing Surveys, **13**(1), Março, 101-120.

Shneiderman, B. (1983) *"Direct Manipulation: a step beyond programming languages".* Computer 16(8), 57-69.

Shneiderman, B. (1998) *"Designing the User Interface: strategies for effective human-computer interaction",* Addison Wesley Longman, Inc.

Smith, D.C. (1975) *"PYGMALION: A Creative Programming Environment".* Ph.D. Thesis, Dept. of Computer Science, Technical Report STAN-CS-75-499 (Palo Alto, Calif.: Stanford University).

Smith, D., Cypher, A. e Spohrer, J. (1994) *"KidSim: Programming Agents without a Programming Language".* Communications of the ACM **37**(7), 54-67.

Soares, S., Ohishi, T., Cicogna, M., Arce, A. (2003) *"Dynamic Dispatch of Hydro Generating Units"*, IEEE Bolonha PowerTech.

Sprinkle, J. e Karsai, G. (2004) *"A Domain-Specific Language for Domain Model Evolution".* Journal of Visual Languages and Computing, 15, 291-307.

Statsko, J. T., Badre, A. M. e Lewis, C. (1993) *"Do Algorithm Animations Assist Learning? An Empirical Study and Analysis"*. Proceedings INTERCHI'93 Conference on Human Factors in Computing Systems, 61-66.

Tanimoto, S. (1990) *"VIVA: a Visual Language for Image Processing"*. Journal of Visual Languages and Computing **2**(2), 127-139.

Üsküdarli, S. (1994) *"Generating Visual Editors for Formally Specified Languages"*. IEEE Symposium on Visual Languages, 278-287.

Vessey, I. e Weber, R. (1984) *"Research on Structured Programming: an Empiricist's Evaluation"*. IEEE Transactions on Software Engineering, SE-**10**(4). 397-407.

Whitley, K. (1997) *"Visual Programming Languages and the Empirical Evidence For and Against"*. Journal of Visual Languages and Computing, **8**(1), 109-142.

Whitley, K.N. e Blackwell, A. F. (2001) "Visual Programming in the Wild: A Survey of LabVIEW Programmers". Journal of Visual Languages and Computing, pp. 435-472, 2001.

Wittenburg, K. e Weitzmann, L. (1990) *"Visual Grammars and Incremental Parsing for Interface Languages"*. IEEE Symposium on Visual Languages, 111-118.

Wittenburg, K., Weitzman, L. e Talley, J. (1991) *"Unification-based Grammars and Tabular Parsing for Graphical Languages"*. Journal of Visual Languages and Computing, 2, 347-370.

Wittenburg, K. (1993) *"Adventures in Multidimensional Parsing: cycles and Disorders"*. International Workshop on Parsing Technologies, 333-348.

Wittenburg, K. (1998) *"Predictive Parsing for Unordered Relational Languages"*. Recent Advances in Parsing Languages.

Wright, P. e Reid, F. (1973) *"Some Alternatives to Prose for Expressing the Outcomes of Complex Contingencies"*. Journal of Applied Psychology, 57, 160-166.

Yang, S., Burnett, M., DeKoven, E. e Zloof, M. (1997) *"Representation design benchmarks: a design-time aid for VPL navigable static representations"*. Journal of Visual Languages and Computing **8**(5/6), 563-599.