Universidade Federal de Santa Catarina
Programa de Pós-Graduação em Ciência da Computação

Uma Perspectiva Multiparadigmática para Implementação de Modelos Conexionistas

Dissertação submetida à Universidade Federal de Santa Catarina para a obtenção do grau de mestre em Ciência da Computação

Antonio Carlos Mariani

Florianópolis, abril de 1995

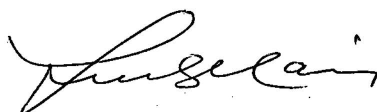
Uma Perspectiva Multiparadigmática para Implementação de Modelos Conexionistas

Antonio Carlos Mariani

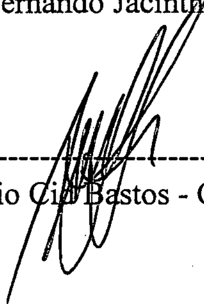
Esta dissertação foi julgada para obtenção do título de

Mestre em Ciência da Computação

e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação



Prof. Dr. Luiz Fernando Jacintho Maia - Orientador

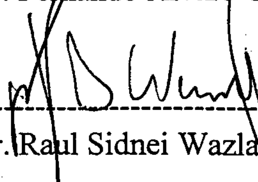


Prof. Dr. Rogério Cid Bastos - Coordenador do Curso

Banca examinadora

Prof. Dr. Luiz Fernando Jacintho Maia - Presidente

Prof. Dr. Fernando Álvaro Ostuni Gauthier



Prof. Dr. Raul Sidnei Wazlawick

SUMÁRIO

1 - INTRODUÇÃO 1
1.1 - **As Diferentes Perspectivas de Programação** 1
1.2 - **A Integração de Perspectivas** 3
1.3 - **Este Trabalho** 3

2 - PROGRAMAÇÃO POR RESTRIÇÕES 5
2.1 - **Deficiências de Expressividade das Linguagens Imperativas** 5
2.2 - **A Alternativa Oferecida pela Programação por Restrições** 6
2.3 - **Grafo de Restrições** 6
2.4 - **A Satisfação de Restrições** 8
2.5 - **Modelos de Restrições** 9
2.5.1 - Perturbação 9
2.5.2 - Refinamento 9
2.6 - **Opções de Escolha na Re-satisfação de Restrições** 9
2.6.1 - Hierarquia de Restrições 10
2.6.2 - Anotações Somente-para-leitura 10
2.6.3 - Anotações Somente-para-escrita 11
2.7 - **Atribuição versus Igualdade** 11
2.8 - **Sequenciamento versus Invariância** 12
2.9 - **Restrições Envolvendo Tempo** 13
2.10 - **Técnicas para Satisfação de Restrições** 13
2.10.1 - Propagação Local de Estados Conhecidos 14
2.10.2 - Relaxação 16
2.10.3 - Propagação de Graus de Liberdade 16
2.10.4 - Visões Redundantes 16
2.10.5 - Transformações de Grafos (reescrita de termos) 17
2.10.6 - Outras Técnicas 18
2.11 - **Considerações** 18

3 - PROGRAMAÇÃO ORIENTADA A OBJETOS 20
3.1 - **O Modelo de Objetos** 20

3.1.1 - Os Objetos 20
3.1.2 - A Comunicação entre os Objetos 21
3.2 - **As Noções de Classe e de Instância** 21
3.3 - **Hierarquias de Classes** 22
3.3.1 - Hierarquia de Generalização/Especialização 22
3.3.2 - Hierarquia de Agregação/Decomposição 23
3.4 - **Tipos e a Programação Orientada a Objetos** 23
3.4.1 - A Noção de Tipos 23
3.4.2 - Tipagem de Linguagens 24
3.4.3 - A Verificação de Tipos 24
3.4.4 - Polimorfismo 25
3.4.5 - Hierarquia de Tipos versus Hierarquia de Implementação 25
3.5 - **Mensagens versus Métodos** 26
3.5.1 - Ligação Estática versus Ligação Dinâmica 26
3.5.2 - Disparo Simples versus Disparo Múltiplo (multi-métodos) 27
3.6 - **Identidade versus Igualdade de Objetos** 28
3.7 - **Considerações** 29

4 - A INTEGRAÇÃO DA PROGRAMAÇÃO ORIENTADA A OBJETOS E DA PROGRAMAÇÃO POR RESTRIÇÕES **30**

4.1 - **Motivação para a Integração** 30
4.2 - **Características Desejáveis nesta Integração** 31
4.2.1 - Ampliação da Flexibilidade e Expressividade 31
4.2.2 - Suporte ao Estilo Orientado a Objetos 31
4.2.3 - Manutenção do Estilo de Programação Orientado a Objetos 32
4.3 - **Alguns Obstáculos para a Integração** 32
4.3.1 - Os modelos de Execução e de Armazenamento de Dados 33
4.3.2 - Domínio Computacional da Linguagem 33
4.3.3 - Restrição sobre o Todo e/ou sobre as Partes 33
4.3.4 - Definição de Novas Restrições 34
4.3.5 - Atribuição Destrutiva versus Satisfação de Restrição 34
4.4 - **Os Diferentes Tipos de Restrições** 35
4.4.1 - Restrições sobre valores 35
4.4.2 - Restrições de identidade 36
4.4.3 - Restrições de classe 36
4.4.4 - Restrições sobre Estruturas 37

4.5 - A Aplicabilidade das Técnicas Clássicas de Satisfação de Restrições 37
4.5.1 - Uso exclusivo de propagação local 37
4.5.2 - Restrições sobre folhas (objetos primitivos) 38
4.5.3 - Adição de Novos Resolvedores de Restrições 38
4.5.4 - Reescrita de grafo 38
4.6 - A abordagem de constructores 39
4.6.1 - Os construtores 39
4.6.2 - A vidas das Restrições 40
4.6.3 - A Máquina K 41
4.7 - Considerações 42

5 - INTEGRAÇÃO COM A PROGRAMAÇÃO EM LÓGICA **43**
5.1 - A Programação em Lógica 43
5.2 - A Programação em Lógica com Restrições (CLP) 44
5.3 - As Relações em Leda 44
5.4 - Definindo Relações 46
5.5 - O Estabelecimento de Relações 47
5.6 - A Consulta a Relações 47
5.7 - Considerações 48

6 - REDES NEURAIS **49**
6.1 - Redes Neurais versus Arquiteturas Tradicionais 49
6.2 - Aplicações Típicas 49
6.3 - O Funcionamento das Redes Neurais 50
6.3.1 - As Formas de Aprendizagem 50
6.3.2 - O Sincronismo 51
6.4 - O Modelo Back-propagation 51
6.4.1 - Uma Unidade Típida 51
6.4.2 - A Topologia 52
6.4.3 - A Propagação de Valores de Ativação 53
6.4.4 - A Retro-Propagação (correção de erros) 53
6.5 - A Implementação de Redes Neurais 54
6.5.1 - Sistemas Simuladores 54
6.5.2 - "Frameworks" 55
6.5.3 - Os Elementos Básicos 55
6.6 - A Neurociência Computacional 56

6.7 - **Considerações** 56

7 - UMA IMPLEMENTAÇÃO DO MODELO BACK-PROPAGATION DE REDES NEURAIS **57**

7.1 - **A Linguagem Hipotética** 57

7.1.1 - Características da Linguagem 57

7.1.2 - Biblioteca de classes primitivas 58

7.1.3 - Algumas Construções Sintáticas 58

7.2 - **As Classes** 60

7.2.1 - Estrutura de Classes para Implementação de Redes Neurais 60

7.2.2 - As Classes para Implementação de Redes Back-propagation 61

7.3 - **As Funções/Procedimentos** 63

7.4 - **As Relações** 65

7.5 - **As Restrições** 66

7.5.1 - Uso do contexto para definir o fluxo de dados 66

7.5.2 - Anotação de não-dependência 67

7.6 - **Os Construtores** 67

7.7 - **Considerações** 69

8 - DIAGRAMAS PERT **70**

8.1 - **O Problema do Caminho Crítico** 70

8.2 - **A Estruturação do Problema** 71

8.3 - **A Construção do Diagrama PERT** 72

8.4 - **A Modificação do Diagrama PERT** 77

9 - CONSIDERAÇÕES FINAIS **79**

9.1 - **Semântica da Linguagem Hipotética** 79

9.2 - **Ambiente para Experimentação de Redes Neurais** 79

9.3 - **Concorrência, Paralelismo e Distribuição** 79

10 - BIBLIOGRAFIA **80**

ÍNDICE DE FIGURAS

Figura 2.1 - **Grafo de conversão de temperatura (C° <=> F°)** 7

Figura 2.2 - **A subdivisão de uma restrição** 7

Figura 2.3 - **Hipergrafo de conversão de temperatura (C° <=> F°)** 7

Figura 2.4 - **A definição do grafo de conversão de temperatura (C° <=> F° <=> K°)** 8

Figura 2.5 - **As regras para um nodo +** 14

Figura 2.6 - **As regras para um nodo *** 14

Figura 2.7 - **Um grafo de restrições contendo ciclo** 15

Figura 2.8 - **Grafo de conversão de temperaturas com $F = 32$** 17

Figura 2.9 - **Grafo de conversão de temperaturas após a transformação** 18

Figura 3.1 - **Parte da taxonomia animal** 22

Figura 3.2 - **Agregação de partes de um carro** 23

Figura 4.1 - **Definição das classes *Círculo* e *Quadrado*** 32

Figura 4.2 - **Construtores para a classe *Point*** 39

Figura 4.3 - **Um programa envolvendo restrições** 40

Figura 4.4 - **Representação do grafo de restrições** 42

Figura 5.1 - **A classe *Set*.** 45

Figura 5.2 - **A função *unify*** 45

Figura 5.3 - **A consulta lógica em Leda** 46

Figura 5.4 - **Duas relações clássicas entre pessoas** 46

Figura 5.5 - **A definição de relações** 47

Figura 6.1 - **Um neuronodo típico** 51

Figura 6.2 - **Uma rede back-propagation** 52

Figura 7.1 - **Biblioteca de classes primitivas** 58

Figura 7.2 - **Estrutura para implementação de redes neurais** 61

Figura 7.3 - **Descrição das classes para implementação de rede back-propagation** 62

Figura 7.4 - **Funções sobre classes primitivas** 63

Figura 7.5 - **Conexão completa entre camadas de neuronodos** 64

Figura 7.6 - **Procedimentos de montagem de uma rede back-propagation** 64

Figura 7.7 - **Procedimentos de reconhecimento e treinamento** 65

Figura 7.8 - **Descrição das relações** 65

Figura 7.9 - **Constrututores sobre a classe *Neuronodo*** 67

Figura 7.10 - Constrututores sobre a classe *BackPropagation* 68
Figura 8.1 - Um diagrama PERT 70
Figura 8.2 - A classe *Evento* 71
Figura 8.3 - A relação de *atividade* 71
Figura 8.4 - As funções *max* e *min* sobre a classe *Collection* 72
Figura 8.5 - Procedimento de montagem do Diagrama PERT 73
Figura 8.6 - A instanciação dos eventos *e1* e *e2* 73
Figura 8.7 - O estabelecimento da atividade entre os eventos *e1* e *e2* 74
Figura 8.8 - Após a satisfação da restrição [E] 74
Figura 8.9 - Após a satisfação da restrição [C] 74
Figura 8.10 - Após a satisfação da restrição [B] sobre o evento *e2*. 75
Figura 8.11 - Após a satisfação da restrição [F]. 75
Figura 8.12 - Após a satisfação de todas as restrições 75
Figura 8.13 - A do evento *e3* e da atividade de *e1* para *e3*. 76
Figura 8.14 - A do evento *e4* e da atividade de *e2* e *e3* para *e4*. 77

RESUMO

A programação orientada a objetos oferece uma filosofia de programação que propicia uma melhor apreensão da realidade, além de prover um mecanismo bastante eficiente para extensão do domínio computacional da linguagem. A programação por restrições, por sua vez, introduz mecanismos de estabelecimento de dependências entre objetos e de satisfação automática de restrições. Por último, os mecanismos apresentados pela programação em lógica possibilitam o estabelecimento e a manutenção de relações ordinárias entre objetos, outras que não as relações de dependência definidas pela restricões.

São discutidas neste trabalho algumas perspectivas de integração destes diferentes estilos de programação. Em particular é apresentada uma construção denominada de "relation" que introduz a noção de estado numa relação e que possibilita o estabelecimento de restrições vinculadas às relações. Esta construção é particularmente apropriada para a implementação de estruturas que tomam a forma geral de grafos valorados e que caracterizam-se pelo fluxo de valores que dependam ou que modifiquem os pesos das conexões destes grafos, a exemplo das redes neurais. O uso da construção "relation" é exemplificado pela implementação do modelo back-propagation de redes neurais e pelo cálculo de caminho crítico num diagrama PERT.

ABSTRACT

Object-Oriented Programming offers a programming model that provides better ways to capture aspects of the "real world". It also provides eficient mechanisms to further extend a programming language's computing domain. On the other hand, constraint-programming introduces mechanisms that promote the automatic establishment of dependencies between objects, with automatic constraint satisfaction. At last, the mechanisms available in logic-programming models promote the establishment and maintainance of ordinary relations between objects other than the dependency ones defined by the constraints.

In this work we discuss some integration perspectives for these three different programming styles. We introduce the "relation" construct, which itself introduces a notion of state for a relation, promoting the establishment of constraints for the relations. This construct is particularly useful to implement structures that can be viewed as a valued graph, which main characteristics is the flow of values that depend on/modify connection weights of these graphs (like neural networks). The use of the "relation" is demonstrated by the implementation of the back-propagation model of neural networks and by the calculus of the critical path in a PERT diagram.

1 - INTRODUÇÃO

Vários autores apontam para o fato de que nenhum dos modelos de computação inseridos nas diferentes perspectivas de programação individualmente representam completamente a forma como o ser humano elabora a solução para problemas. É certo que há equivalência entre praticamente todas as diferentes perspectivas de programação no que tange àquilo que pode ser efetivamente computado (conjectura de Church). Contudo, isto não deve ser confundido com o poder de expressão e com a aplicabilidade de cada uma delas. Deste ponto de vista algumas das perspectivas são significativamente distintas de outras.

Por outro lado, as várias técnicas de desenvolvimento de software normalmente buscam decompor recursivamente uma aplicação em elementos mais simples até um nível em que estes elementos possam ser expressos diretamente em construções de uma linguagem de programação. É sabido, contudo, que estes vários elementos que compõem uma aplicação não são necessariamente homogêneos no que se refere à melhor forma de representá-los. Certas partes de uma aplicação podem ser melhor descritas num estilo declarativo, por exemplo, enquanto que outras ficam melhor representadas num estilo imperativo. A uniformização de visão de uma aplicação em torno de uma única perspectiva em geral conduz a complexidades absolutamente indesejáveis e/ou desnecessárias.

Num terceiro ponto de vista, há que se notar a estreita vinculação entre o pensamento e o veículo do pensamento. Conforme mencionado por Takahashi [Takahashi 90] a assertiva básica é a de que a linguagem condiciona decisivamente o pensamento, e vice-versa, e nenhuma precede a outra. Isto significa que linguagem e pensamento se moldam mutuamente de sorte que a linguagem deveria ser tão expressiva e abrangente quanto possível.

As considerações acima tem, em maior ou menor grau, conduzido vários pesquisadores a buscar formas de reunir diferentes pespectivas em torno de uma mesma estrutura. Assim o programador não precisaria necessariamente ser forçado a resolver todos os problemas em um estilo único. Ao contrário, ele pode sentir-se livre para escolher aquele que melhor atende às necessidades da tarefa que ele está modelando.

1.1 - As Diferentes Perspectivas de Programação

São descritos a seguir as características básicas de diversas perspectivas de programação.

- A modelo de computação inserido na perspectiva **Imperativa** de programação é bastante próximo do modelo real de execução dos computadores. A computação é vista como uma tarefa (uma seqüência de passos) que é executada por uma unidade de processamento, a qual manipula e modifica uma memória. O ponto culminante desta perspectiva foi alcançado na década de 70 com a apresentação da programação estruturada que introduziu construções para controle de fluxo (tais como <*if-then-else*> e <*while*>) e que baniu o comando <*go to*>.

- Na programação **Orientada a Objetos** o mundo computacional é visto como uma composição de objetos que se comunicam por troca de mensagens. Cada objeto tem sua memória

privada e um comportamento (um conjunto de ações) os quais são descritos por classes[1] que estão estão organizadas em hierarquias. Estas hierarquias introduzem a noção de herança de características e conduzem a um estilo recursivo, incremental e evolutivo de programação. Apesar de manter características imperativas, a orientação a objetos apresenta um modelo de computação mais próximo da realidade e introduz a efetiva oportunidade de reuso de artefatos de software.

- Divergindo da programação imperativa, na programação **Funcional** os valores/estruturas são tratados como entidades simples que após criados não são mais modificados. O modelo de computação é baseado na aplicação de funções a valores, as quais conduzem a novos valores que são independentes dos originais e que, em geral, são tomados como entrada para novas aplicações de funções. Assim, os valores não tem a noção de estado que vai sendo modificado ao longo do tempo. Outra característica da programação funcional é a de ver as funções como entidades de primeira ordem, de forma que elas podem ser atribuídas a identificadores, passadas como argumentos ou retornadas como resultado da aplicação de uma função.

- A programação em **Lógica** também diverge da abordagem imperativa ao apresentar um estido declarativo de programação. Por basear-se na prova proposicional de teoremas, a programação em lógica consiste de três partes: um conjunto de axiomas (os fatos), um conjunto de regras de inferência, e as questões ou perguntas. O modelo de execução consiste em utilizar um mecanismo de busca (em geral busca em profundidade na árvore de derivação) para mostrar que partindo dos fatos é possível derivar a resposta por meio das regras de inferência. Nesta perspectiva, o programador não precisa especificar como a pergunta deva ser respondida. Ele preocupa-se apenas em estabelecer os fatos, as regras de inferência e as perguntas.

- A exemplo da programação em lógica, a programação por **Restrições** é mais declarativa do que imperativa. O programador especifica um conjunto de restrições que devem ser mantidas durante a execução de um programa. Um sistema de satisfação de restrições se encarrega de garantir que estas restrições sejam satifeitas.

- A programação **Orientada ao Acesso** considera a inclusão de efeitos colaterais (*Demons*) associados às variáveis, de sorte que certas ações são disparadas quando estas variáveis são manipuladas.

- Na programação **Visual** os programas são especificados por meio de algum recurso visual (diagramas, ícones, figuras, etc), ou seja, outro que não a forma textual pura. Esta perspectiva normalmente inclue a manipulação direta de representações visuais para objetos computacionais via algum dispositivo de apontamento (a exemplo do "mouse").

- Os sistemas de programação **Distribuída** consideram a existência de múltiplos processadores autônomos (inclusive no que tange à memória) que cooperam entre si por envio de mensagens via um substrato de comunicação. A distinção básica entre a programação distribuída

[1] Uma classe pode ser vista como o agrupamento de objetos que possuem uma estrutura e um comportamento idêntico. O mecanismo de classes é uma das características fundamentais de Smalltalk, a primeira linguagem a popularizar a orientação a objetos. Trabalhos mais recentes, contudo, relegam a noção de classes em favor da noção de protótipos.

e a programação seqüencial clássica está na forma como ela trata o paralelismo, a comunicação e as falhas parciais [Bal 89].

• Na programação por **Eventos** a computação é vista como uma seqüência de eventos de interação. Cada evento é composto por um sinal ("prompt") do sistema, uma entrada (comando) do usuário, uma ação a ser executada e um fluxo de controle que determina a próxima iteração.

1.2 - A Integração de Perspectivas

A integração de diferentes perspectivas de desenvolvimento de software é alvo de diversos estudos. Referências para algumas integrações são indicadas a seguir.

• Programação orientada a objetos e programação por restrições: [Wilk 91], [Freeman-Benson 90], [Horn 92], [Freeman-Benson 92], [Lopez 93], [Lopez 94a] e [Lopez 94b].

• Programação em Lógica e programação por restrições: [Borning 92] e [Lassez 87].

• Programação visual e programação por restrições: [Myers 90].

• Programação distribuída e programação por restrições: [Kahn 90].

• Programação distribuída e programação orientada a objetos: [Marchini 94].

• Programação orientada a objetos, programação funcional, programação em lógica e programação por restrições: [Budd 93].

1.3 - Este Trabalho

O presente trabalho concentra-se basicamente no relacionamento de três das perspectivas descritas em 1.1: a programação por restrições, a programação orientada a objetos e a programação em lógica. O objetivo é oferecer uma estrutura que seja particularmente adequada para a representação da classe de problemas cujo modelo computacional caracteriza-se por apresentar uma arquitetura na forma geral de grafos valorados e pelo fluxo de valores que dependam ou que modifiquem os pesos das conexões destes grafos.

A motivação básica para a busca desta estrutura é a de suprir uma deficiência observada nos problemas acima caracterizados, a exemplo das redes neuronais. Normalmente as redes neurais são apresentadas nos textos pertinentes na forma de equações que procuram descrever os relacionamentos e restrições (um estilo declarativo) entre os componentes, enquanto que a implementação efetiva é, em geral, baseada num perspectiva puramente imperativa.

Para se chegar à estrutura proposta, inicialmente, no capítulo 2, são apresentadas as noções e mecanismos básicos da programação por restrições. Isto inclui algumas das técnicas comumentemente utilizadas em sistemas de satisfação de restrições. No capítulo 3 é feita uma

incursão pelo mundo da programação orientada a objetos. Além dos fundamentos e dos mecanismos mais comuns desta perspectiva, são rapidamente apresentadas algumas derivações do modelo clássico, a exemplo da tipagem opcional e incremental e o mecanismo de multi-métodos.

A integração da programação por restrições e da programação orientada a objetos é discutida no capítulo 4. São apresentadas algumas características e obstáculos desta integração e, em particular, é apresentado o mecanismo de contrutores inserido na linguagem Kaleidoscope [Lopez 94a]. O capítulo 5 discute a integração da programação em lógica com a programação orientada a objetos. Em especial são apresentadas a abordagem de integração adotada na linguagem Leda [Budd 93] e uma construção sintática denomida "*relation*" que introduz a noção de estado e possibilita o estabelecimento de restrições numa relação.

As características básicas das redes neurais são apresentadas no capítulo 6. Maior atenção é dada ao modelo back-propagation pois no capítulo 7 é discutida sua implementação numa perspectiva multiparadigmática. As características das três perspectivas de programação vistas nos capítulos anteriores são reunidas em torno de uma linguagem hipotética sobre a qual é apresentada uma implementação para o modelo back-propagation de redes neurais. Por último, o capítulo 8 apresenta a implementação de um digrama PERT utilizando a linguagem hipotética. Além de mostrar a aplicabilidade da linguagem hipotética em outra classe de aplicacões, este capítulo detalha o modelo dinâmico subjacente às construções da linguagem hipotética.

2 - PROGRAMAÇÃO POR RESTRIÇÕES

2.1 - Deficiências de Expressividade das Linguagens Imperativas

Grande parte das atuais linguagens de programação (como Pascal, C e Modula-2) implementam o paradigma imperativo de programação. Neste paradigma um problema é modelado pela construção de um algoritmo específico, uma seqüência de passos (comandos), cuja execução conduz a uma solução para o problema.

O uso de programação algorítmica para a solução de problemas diferentes, porém relacionados, exige que o programador antecipe-os e inclua pontos de decisão explícitos nos algorítmos. Um exemplo significativo deste tipo de situação é a equação:

$$C := (F - 32) * 5/9 \quad \text{[equação 2.1]}$$

que, numa sintaxe similar à linguagem Pascal, computa a temperatura Celsius (C) equivalente à temperatura Farenhait (F). Se por outro lado a intenção fosse computar F a partir de C, uma nova equação deveria ser escrita:

$$F := C * 9/5 + 32 \quad \text{[equação 2.2]}$$

com a inclusão de um ponto de decisão (classicamente a estrutura de seleção <se ... então ... senão ...>) no programa para permitir a escolha de um ou outra situação.

Agora, se a aplicação também inclui transformações para a escala Kelvin (K), a conversão de temperatura exigiria a inclusão de novas equações (e correspondentes pontos de decisão) tais como:

$$K := C + 273 \quad \text{[equação 2.3]}$$

$$C := K - 273 \quad \text{[equação 2.4]}$$

$$K := 290.78 + 5/9 * F \quad \text{[equação 2.5]}$$

$$F := 523.4 + 9/5 * K \quad \text{[equação 2.6]}$$

Na medida em que novas variáveis forem sendo adicionadas a este programa, o número de declarações cresce exponencialmente.

Dada esta deficiência de expressão das linguagens imperativas clássicas, a atividade de programação tende a ser tediosa, requerendo um esforço que desencoraja muitos potenciais usuários. Perspectivas recentes de desenvolvimento de software, a exemplo da programação orientada a objetos (ver capítulo 3), tentam reduzir este esforço ao prover mecanismos (classes, herança, polimorfismo, etc) que melhor estruturam o domínio do problema e que propiciam a reusabilidade efetiva de artefatos de software. Estes mecanismos introduzem vários níveis de abstração, permitindo que o programador fixe sua atenção nas partes significativas da aplicação ao invés de tratar de detalhes todo o tempo. Contudo, eles respondem apenas parcialmente às necessidades observáveis quando da efetiva construção de programas pois mesmo as linguagens imperativas orientadas a objetos tem um nível de expressividade inferior ao que se pode desejar.

2.2 - A Alternativa Oferecida pela Programação por Restrições

Divergindo da programação imperativa, a programação por restrições é uma tarefa declarativa. O programador estabelece um conjunto de relações entre objetos , ficando a cargo de um sistema de satisfação de restrições encontrar uma solução que satisfaça estas relações.

Numa linguagem de restrições, a equação 2.1 de transformação de graus Celsius para graus Farenhait pode ser escrita como:

$$C = (F - 32) * 5/9 \quad \text{[equação 2.7]}$$

ou

$$C * 1.8 = F - 32 \quad \text{[equação 2.8]}$$

ou, ainda,

$$C * 1.8 + 32 = F \quad \text{[equação 2.9]}$$

que estabelece uma relação entre C e F que possibilita o cômputo destes valores pelo mesmo programa em ambos os sentidos (bidirecional): oferecido um valor para C o valor de F pode ser computado e vice-versa.

As transformações de temperatura envolvendo graus Kelvin pode ser obtida pela simples adição da relação:

$$K = C + 273 \quad \text{[equação 2.10]}$$

a qualquer uma das alternativas de programa descritas pelas equações 2.7, 2.8 ou 2.9. Sem requerer qualquer declaração adicional, programa composto pelas duas equações é suficiente para permitir a conversão de temperatura entre as três escalas. Esta habilidade de resolver muitos problemas diferentes com o mesmo programa, mesmo que eles não tenham sido antecipados, é a vantagem chave das linguagens de programação por restrições.

2.3 - Grafo de Restrições

As restrições estabelecidas por um programa por restrições podem ser representadas por meio de um grafo de dependências. Mas dependendo da forma como uma restrição é vista, o grafo de dependências pode apresentar diferentes formas.

Uma possível representação seria considerar os operandos e operadores de uma restrição como nodos do grafo, sendo as arestas utilizadas para indicar as dependências entre eles. Na notação adotada em [Leller 88] os nodos quadrados representam variáveis enquanto que os nodo arredondados representam operadores. Além disto, os argumentos de um operador aparecem à esquerda do nodo, enquanto que o resultado aparece à direita. Nesta visão, a restrição expressa na equação 2.9 tomaria a forma do grafo apresentado na figura 2.1.

Não é muito usual ter-se dependência entre nodos de um grafo atrelada às suas posições na representação gráfica. Contudo, situação similar ocorre com a noção de árvores binárias quando há referências aos ramos da direita e da esquerda.

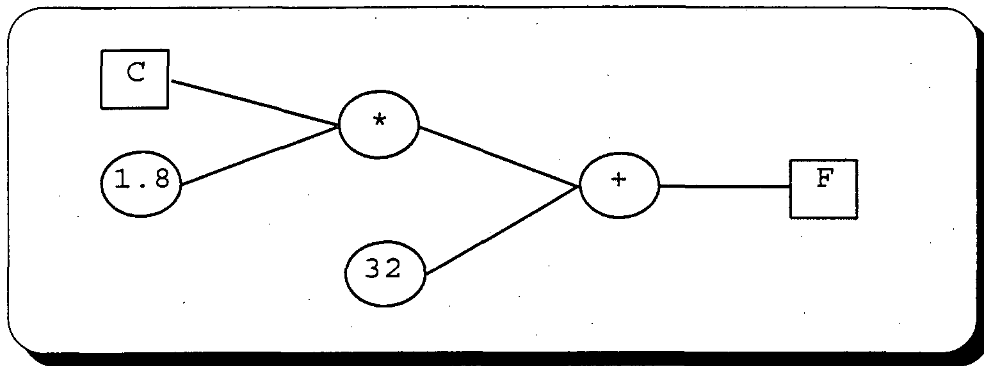


Figura 2.1 - Grafo De Conversão De Temperatura (C° <=> F°)

A que se notar, contudo, que em realidade restrições como a apresentada na equação 2.9 são compostas por sub-restrições, figura 2.2, as quais normalmente são internamente representadas nos sistemas de satisfação por asserções na forma pré-fixada.

$$c * 1.8 + 32 = f \Rightarrow \left\{ \begin{array}{c} *(c, 1.8, tmp1) \\ +(tmp1, 32, tmp2) \\ = (tmp2, f) \end{array} \right\}$$

Figura 2.2 - A Subdivisão De Uma Restrição

Esta nova visão de uma restrição conduz a uma representação na forma de um hipergrafo no qual as variáveis tornam-se nodos[2] e as restrições definem as hiper-arestas.

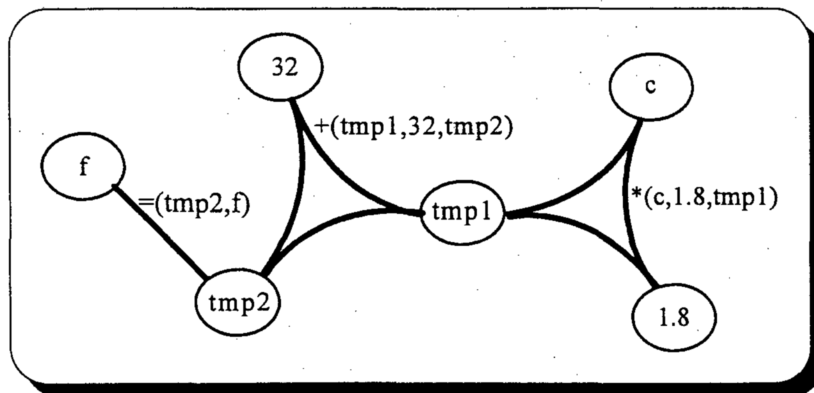


Figura 2.3 - Hipergrafo De Conversão De Temperatura (C° <=> F°)

Numa visão mais geral, um programa escrito numa linguagem de restrições pode ser representado por um grafo de restrições *G*={*V*, *C*, *D*} (adaptado de [Freeman-Benson 92]), onde C é o conjunto de restrições, *V* é o conjunto das variáveis envolvidas, e D é o domínio das variáveis. Assim, o grafo de restrições que representa o problema de transformação de temperaturas toma a forma apresentada na figura 2.4.

[2] Na figura 2.3 as constantes assumiram um "status" de variável apenas para efeito de exemplo.

$$G = \left\{ \{c,f,k\}, \left\{ \begin{array}{c} c * 1.8 + 32 = f \\ c = k - 273.13 \end{array} \right\}, \Re \right\}$$

Figura 2.4 - A Definição Do Grafo De Conversão De Temperatura (C° <=> F° <=> K°)

Uma avaliação *A:V->D* é uma função que mapeia as variáveis do grafos de restrições para elementos do domínio *D*. Para o grafo *G* da figura 2.3, uma possível avaliação é:

$$A = \{(c \rightarrow 0.0), (f \rightarrow 32.0), (k \rightarrow 273.13)\}$$

Por sua vez, uma solução *S* para um grafo de restrições é o conjunto das possivelmente infinitas avaliações que satisfazem as restrições do grafo. Para o grafo *G* da figura 2.3, a solução é:

$$S = \left\{ \begin{array}{c} (c \rightarrow 0.0), (f \rightarrow 32.0), (k \rightarrow 273, 13) \\ (c \rightarrow -40.0), (f \rightarrow -40.0), (k \rightarrow 233.13) \\ (c \rightarrow 100.0), (f \rightarrow 212.0), (\rightarrow 373.13) \\ \bullet\bullet\bullet \end{array} \right\}$$

2.4 - A Satisfação de Restrições

Em linguagems de programação por restrições, as restrições são asserções declarativas que estabelecem relações entre elementos do domínio computacional da linguagem. Estas relações em geral devem ser mantidas durante toda a vida do programa. Por serem declarativas, as restrições enfatizam a relação entre os objetos mais do que os passos necessários para mantê-las.

Dado o alto grau de expressividade das linguagens de programação por restrições, estabelecer as relações (restrições) entre elementos computacionais é uma atividade bem menos enfadonha e bem mais simples do que a construção de um programa equivalente em uma linguagem imperativa. Satisfazer estas relações, contudo, nem sempre é uma tarefa trivial. Isto contrasta significativamente com as linguagem imperativas convencionais nas quais um compilador facilmente "satisfaz" um algoritmo corretamente especificado.

Na programação por restrições satisfazer restrições significa encontrar valores para os objetos de forma a tornar verdadeiras as relações estabelecidas pelas restrições. Os sistemas de satisfação de restrições, contudo, não pretendem ser sistemas gerais de resolução de problemas. As implementações de sistemas de satisfação de restrições normalmente preocupam-se mais em resolver rápida e eficientemente problemas pequenos e mais triviais, deixando a cargo de programas de propósito especiais encontrar solução para casos particulares e/ou mais complexos.

2.5 - Modelos de Restrições

Quanto ao modelo implementado, as linguagens e sistemas de programação baseados em restrições podem ser divididos em duas abordagens distintas: o modelo de perturbação e o modelo de refinamento. Em ambos os casos, as restrições restringem os valores que podem ser assumidos pelas variáveis.

2.5.1 - Perturbação

O modelo de perturbação pressupõe que no início de um ciclo de execução as variáveis denotam valores que satisfazem as restrições. O usuário, ou alguma influência externa, perturba o sistema por modificação de uma ou mais variáveis, cabendo ao sistema de satisfação de restrições ajustar os valores das variáveis de forma a re-satisfazer as restrições.

Este modelo é o mais frequentemente utilizado pelos sistemas baseados em restrição. À excessão de sistemas que tratam apenas de restrições não circulares e unidirecionais, contudo, o modelo de perturbação frequentemente não é claro no tocante a quais variáveis serão alteradas ao serem satisfeitas as restrições. Uma variedade de heurísticas tem sido empregada nos sistemas conhecidos, mas nenhuma delas é inteiramente satisfatória. Além disso, é por vezes difícil especificar declarativamente quais soluções são preferidas, assim como alterar estas preferências, já que estas heurísticas estão ocultas no código procedural do sistemas de satisfação de restrições. Soluções para este problema incluem o uso de anotações especiais e de hierarquias de restrições, as quais são descritas em 2.6.

2.5.2 - Refinamento

No modelo de refinamento inicialmente as variáveis não estão sujeitas a nenhuma restrição. As restrições que vão sendo posteriormente adicionadas vão progressivamente refinando os valores permitidos para as variáveis. Bertrand [Leler 88] e as versões de 90 e 91 de Kaleidoscope [Freman-Benson 90] [Freeman-Benson 92] são linguagens que utilizam o modelo de refinamento.

2.6 - Opções de Escolha na Re-satisfação de Restrições

Quando o sistema de satisfaçãode restrições é baseado no modelo de perturbação geralmente há várias formas de alterar o estado corrente (valores das variáveis) de forma a que as restrições sejam satisfeitas. Numa restrição simples como:

$$A = B + C$$

ao se alterar o valor da variável B, uma das seguintes alternativas poderia ser escolhida de forma a resatisfazer a restrição: alterar somente A; alterar somente C; alterar A e C; desfazer a modificação de B.

Duas formas para reduzir as opções de escolha são o uso de hierarquias de restrições e o uso de anotações especiais.

2.6.1 - Hierarquia de Restrições

Em geral, um programa escrito numa linguagem de restrições é composto de várias restrições inter-relacionadas. É responsabilidade do sistema de satisfação de restrições aplicar a técnica adequada para mantê-las todas satisfeitas.

Muitos sistemas de restrições utilizam um mecanismo de hierarquização das restrições como forma de prover um meio conveniente de estabelecer quais restrições devem ser efetivamente satisfeitas. Este mecanismo apresenta uma gama variada de usos, tendo sido originalmente proposto para resolver o problema de como declarativamente especificar o que modificar quando ocorrer uma perturbação no sistema de restrições [Borning 92].

Em uma hierarquia de restrições a cada restrição é atribuído um peso (poder) numa escala que pode conter um número arbitrário de níveis. Este níveis podem estar em ordem parcial ou total. Nesta ordenação, as restrições mais fortes dominam completamente as mais fracas, de forma que uma dada restrição podem não ser satisfeita em função de haver outra restrição de maior peso que de alguma forma a contradiga.

Na linguagem Kaleidoscope, por exemplo, as restrições marcadas como "required" devem ser válidas para todas as soluções. Outras restrições preferencias ("strong" e "weak") são satisfeitas apenas se for possível. A não satisfação das restrições preferenciais, contudo, não implica em sinalização de condição de erro.

2.6.2 - Anotações Somente-para-leitura

Uma segunda alternativas para reduzir as opções de escolha é o uso de marcações de *somente-para-leitura,* ou seja, explicitamente indicar quais variáveis não podem ser modificadas, estando seus valores disponíveis apenas para consulta. Em Kaleidoscope, por exemplo, isto é feita pela inclusão do símbolo "?" após o nome de uma várivel.

Apesar do mecanismo de hierarquização de restrições ser mais geral, as anotações de somente-para-leitura podem ser úteis em situações como a de se ter uma restrição que relacione um ponto com a posição do "mouse" durante uma movimentação. Visivelmente a posição do "mouse" é que determina o valor do ponto, e não vice-versa. Outras aplicações envolvem restrições que descrevem alterações sobre o tempo, em particular as restrições que relacionam um estado antigo e um novo. Neste caso o estado passado deve ser marcado como somente-para-leitura para evitar que o futuro altere o passado.

Abordagens como a de "Constrains pattern" [Horn 92] não suportam uma hierarquização geral de restrições, mas permitem uma hierarquia de dois níveis por meio da especificação de restrições que devem permanecer fixas durante resatisfação de restrições. Num programa como:

```
aRectangle: {
        top, left, bottom, right: aNumber;
        width: { right - left } 'number;
        height: { bottom - top } 'number;
        moveBy delta'point!: {
                width fixed;
                heigh fixed;
                center = previous center + delta; };
} 'rectangle;
```

o uso da palavra *fixed* indica que os valores de *width* e de *heigh* não devem ser modificados quando da execução da ação *moveBy*, correspondente a uma forma de anotação de somente-para-leitura

As restrições unidirecionais podem ser vistas como um caso especial no qual apenas uma das variáveis não é declarada como somente-para-leitura.

2.6.3 - Anotações Somente-para-escrita

Um segundo tipo de anotação é o de somente-para-escrita, ou seja, explicitamente indicar que um valor flua para esta variável e não o inverso. Considere, por exemplo as restrições abaixo escritas em Kaleidoscope:

$$required: X! = Y$$
$$strong: \quad X = 4$$
$$weak: \quad Y = 3$$

Apesar da restrição X = 4 ser mais forte do que a restrição Y = 3, o fluxo de informações é permitido apenas de Y para X em função da restrição X! = Y, já que X está anotado como *somente-para-escrita*. Portanto o conjunto de soluções para este caso é: {(X –> 3), (Y –> 3)}.

2.7 - Atribuição versus Igualdade

As linguagens algorítmicas normalmente introduzem operadores diferenciados para tratamento da igualdade e da atribuição. Em Pascal, por exemplo, o símbolo = correspondente ao operador binário relacional que retorna verdade ou falso, dependendo se os argumentos são iguais ou não. Já o símbolo := é utilizado para indicar a atribuição.

Nas linguagens de restrições puras, contudo, o operador de atribuição não é necessário pois o próprio sistema de satisfação de restrições é responsável por buscar e "atribuir" valores às variáveis de forma a tornar verdadeiras as relações de igualdade. Deste ponto de vistas, as equações que se seguem são sintaticamente distintas, porém são semanticamente equivalentes:

$$5 = X$$

$$X + 1 = 6$$

$$3 * X = X + 10$$

Nos três casos o sistema de satisfação de restrições calculará e "atribuirá" o valor 5 para X de forma a tornar verdadeira as equações.

O que se observa é que as linguagens de restrições se caracterizam pela grande expressividade em função do tratamento que elas dispensam ao operador relacional de igualdade; um programa pode ser escrito de várias formas, cabendo ao programador escolher aquela que ele julgue mais conveniente. Ao mesmo tempo, este tratamento da igualdade associado a inexistência de um operador de atribuição torna mais natural um programa escrito numa linguagem de restrições, fato que permite uma melhor compreensão de um programa mesmo por pessoas leigas em programação.

2.8 - Sequenciamento versus Invariância

Quando da implementação de programa, um programador normalmente precisa especificar dois tipos de relações[Freeman-Benson 90]:

a) relações de *longa-vida* entre objetos para definir informações, consistência e estrutura interna de uma aplicação: um motor é parte de automóvel; esta cadeia de caracteres é uma representação impressa daquele número inteiro;

b) relações de *sequenciamento* entre estados de um programa e entre objetos nestes estados: quando o botão do "mouse" for pressionado, trazer a janela para a frente; a posição de um automóvel é computada a partir da posição prévia mais a velocidade corrente vezes o tempo.

Linguagens imperativas tradicionais somente provêm relações de sequenciamento, forçando o programador a encarregar-se de garantir que todas as relações de consistências sejam mantidas após cada atribuição. Linguagens mais atuais, a exemplo de EIffel, incluem mecanismos de asserções e invariantes que permitem checar as relações de consistência, mas elas não sintetizam código para mantê-las.

Por outro lado, muitas linguagens de restrições não incluem a noção de estado ou de sequenciamento. Aquelas que o fazem utilizam mecanismos similares aos descritos em 2.9, os quais oferecem uma semântica imperativa em maior ou menor grau, dependendo do mecanismo.

2.9 - Restrições Envolvendo Tempo

Na programação imperativa o tempo é avançado a cada execução de uma instrução atômica (da linguagem de máquina, por exemplo). Na programação por restrições, por outro lado, as asserções usualmente são independentes do tempo. O estabelecimento de uma restrição do tipo x = 0 significa que x *sempre* deve ser igual a zero.

Muitos problemas, contudo, não são independentes do tempo. Em sistemas de animação, por exemplo, a movimentação de objetos em geral é dada em função do tempo. Há, também, problemas que não estão diretamente relacionados com o tempo, mas que devem manipular restrições que podem ser modificadas por uma ou outra razão. Se somente o valor de um objeto for modificado, e não a topologia do grafo de restrição, então um tipo particular de propagação (ver 2.10.1) pode ser utilizada: a retração.

A retração consiste de propagar alterações pelo grafo de restrições. Para fazer isto, inicialmente o valor antigo de um objeto é marcado como desconhecido, fato que pode causar a retração de outros valores dependentes. Em seguida os novos valores são propagados.

Outras abordagens, a exemplo de ThingLab II [Maloney et al. 89], introduzem uma semântica imperativa informal ao prover uma variável de tempo somente para consulta. Já linguagens como Kaleidoscope [Freeman-Benson 90], procuram integrar o paradigma declarativo de restrições com o paradigma imperativo (orientação a objetos neste caso). A chave para esta integração é a definição de uma semântica que combina as duas.

Em Kaleidoscope-90 cada variável guarda uma série (histórico) de valores. Cada um deles representa o valor de uma variável em um instante diferente, com valores subsequentes representando instantes subsequentes. O conteúdo de uma variável num dado instante *t* é o resultado das restrições que existem neste instante. O tempo é virtual e representado por números inteiros positivos e, similarmente à realidade, o passado está disponível apenas para consulta.

2.10 - Técnicas para Satisfação de Restrições

De forma similar a outras técnicas de solução de problemas, a satisfação de restrições normalmente é compostas de duas partes: um conjunto de regras para solução de problema (de propósito geral ou mais específicas para uma aplicação) e um mecanismo de controle (que controla como e quando as regras são aplicadas).

No caso específico dos sistemas de satisfação de restrições, contudo, em geral não há distinção significativa entre o mecanismo de controle e as regras para solução de problemas. Esta é uma das razões da dificuldade de se construir e/ou modificar os sistemas de satisfação de restrições.

Os tópicos seguintes procuram examinar alguns dos mecanismos de controle que tem sido empregados em sistemas de satisfação de restrições.

2.10.1 - Propagação Local de Estados Conhecidos

A forma mais simples e mais comum de implementar um sistema de satisfação de restrições é conhecida por *propagação local de estados conhecidos*. A propagação local parte de algum conjunto conhecido de valores e determina algum outro valor por meio da satisfação de alguma restrição simples. A repetição desta operação faz com que os valores conhecidos sejam propagados através dos arcos do grafo de dependências.

Uma restrição como:

$$p + q = r$$

pode ser representada pelo grafo de dependência da figura 2.5. Nesta figura são também apresentadas as regras utilizadas para satisfazer esta restrição. A escolha de uma ou de outra regra depende de quais são os valores conhecidos. Dados os valores de q e r, por exemplo, a regra disparada será "p <– r-q", que indica como se pode calcular p a partir de q e r.

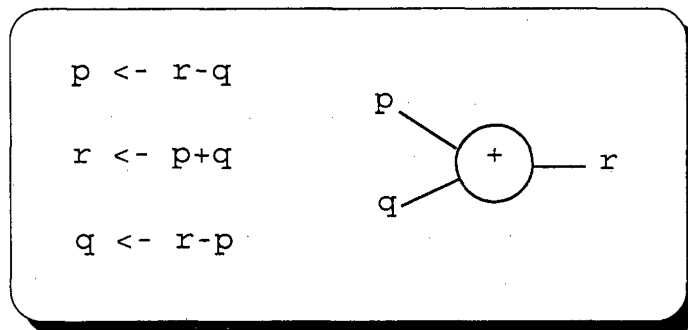


Figura 2.5 - As Regras Para Um Nodo +

Para um nodo do grafo que represente a operação de multiplicação:

$$r = p * q$$

as regras são um pouco mais complicadas pois elas devem ser capazes de tratar algumas situações particulares, tais como: se p ou q tiverem valor zero, então r terá valor zero independentemente do outro valor (q ou p respectivamente); se for conhecido o valor de r e o valor de p ou o valor de q for zero, então nada se pode afirmar sobre o valor da outra variável (q ou p respectivamente).

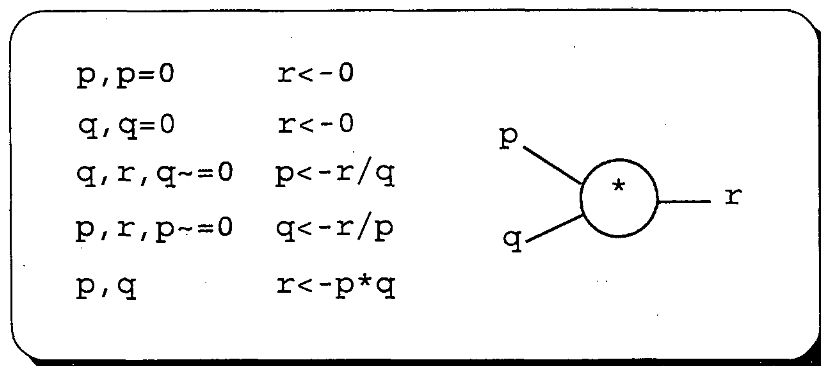


Figura 2.6 - As Regras Para Um Nodo *

A primeira regra da figura 2.6, por exemplo, diz que se o valor de *p* é conhecido e ele é igual a zero, então *r* também será zero, independentemente do valor de *q*.

Além de ser simples, a técnica de propagação local possibilita ao sistema de satisfação de restrições manter registro de qual regra foi dispara em cada nodo. Esta informação pode ser muito útil em casos de depuração de resultados, e mesmo para justificar (um histórico) a obtenção de uma resposta particular.

Contudo, a propagação local apresenta uma grande desvantagem. As regras para satisfação de restrições são locais a cada nodo e envolvem somente informações contidas nos arcos a ele conectados. Esta característica a torna incapaz de resolver restrições cíclicas (tais como equações simultâneas), impossibilitando a solução de vários problemas. Considere, por exemplo, as restrições:

$$A + T = B$$
$$B + T = C$$

que restringem o valor B a ser a média entre A e C, e o valor T a ser a metade da diferença entre C e A.

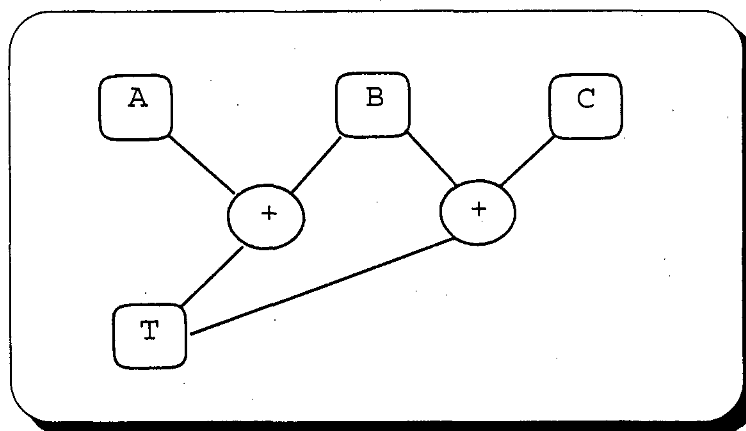


Figura 2.7 - Um Grafo De Restrições Contendo Ciclo

O grafo de restrições para este programa, figura 2.7, contém um ciclo. Se valores são fornecidos para A e para C, digamos 1 e 11, as equações passam a ser:

$$1 + T = B$$
$$B + T = 11$$

de forma que a propagação local seria incapaz de resolver o sistema pois nenhuma das equações pode individualmente ser resolvida.

A solução de restrições por meio de propagação de valores normalmente envolve duas fases: a seleção das restrições a serem usadas e a execução destas restrições. Por questões de simplicidade alguns sistemas nao fazerem distinção clara entre estas duas fases. Contudo, uma das vantagens da separação de fases é que uma escolha de restrições (a exemplo dos "planos" inseridos em ThingLab II [Maloney 89]) possibilitam o reuso da solução até que a topologia do grafo mude, resultando em melhor velocidade de execução.

2.10.2 - Relaxação

Outra técnica para solução de restrições é a clássica aproximação numérica iterativa, conhecida por relaxação. Partindo de estimativas iniciais para os valores desconhecidos, novas estimativas vão sendo calculadas. O processo se repete até que o erro (diferença entre a estimativa anterior e o novo valor calculado) seja de alguma forma minimizado, geralmente convergindo para zero. Caso não haja convergência a relaxação falha. Contudo, mesmo que não haja convergência, esta técnica pode ser utilizada para se obter solucões aproximadas para alguns problemas.

Dadas as suas características, a técnica de relaxação consegue tratar dependências cíclicas. Contudo ela só é aplicada a valores numéricos contínuos. Exclue-se, desta forma, o tratamento de outros tipos de objetos, a exemplo valores lógicos (verdade e falso) e de valores inteiros.

Outra desvantagem desta técnica é sua lentidão. Mesmo sendo diretamente aplicável como mecanismo de satisfação de restrições, em geral a relaxação só é usada após a propagação local ter sido tentada e falhar. Uma alternativa frequentemente utilizada para aumentar a eficiência da relaxação é usá-la em combinação com a propagação local para reduzir o número de objetos que precisam ser relaxados em cada iteração.

2.10.3 - Propagação de Graus de Liberdade

A técnica de relaxação não considera a estrutura global do grafo de restrições, de forma que por vezes são relaxadas mais variáveis do que o realmente necessário. Para minimizar este problema, em vez de procurar por objetos que são conhecidos e propagar seus valores, a técnica de propagação de graus de liberdade procura objetos com um número de restrições suficientemente pequeno que possibilite a alteração do valor deste objeto, satisfazendo assim suas restrições. Tipicamente uma variável tem suficiente grau de liberdade se ela tem somente uma restrição a ela vinculada.

Quando uma parte do grafo com suficientes graus de liberdade é encontrada, ela é removida do grafo juntamente com as restrições a ela aplicadas. Esta poda de braços conduz a grafos mais simples (em particular a grafos que contém ciclos) de forma que a relaxação é feita sobre um número bem menor de variáveis. Os valores destas variáveis são então propagados para os braços podados.

2.10.4 - Visões Redundantes

Um grafo que contém ciclos em geral não pode ser resolvido por propagação local. Contudo este tipo de grafo poderia ser resolvido se alguma das restrições fosse substituida por uma restrição equivalente. É o caso, por exemplo de substituir restrições como:

$$X + X = 40$$

por:

$$X * 2 = 40$$

Outra alternativa é a inclusão de restrições redundantes. Considerando o problema apresentado em 2.10.1 de achar a média entre os valores A e C. A solução poderia ser obtida por propagação local, sem a necessidade de relaxação, se fosse incluida pelo menos uma das restrições redundantes apresentadas a seguir:

$$B = (A + C)/2$$
$$A + T * 2 = C$$

Visões redundantes podem inclusive ser usadas para auxiliar a solução problemas que a técnica de relaxação não é capaz de resolver. As visões redundantes permitem que o usuário auxilie o sistema de satisfação de restrições a resolver certos problemas. Em alguns casos é possível que o próprio sistema de satisfação de restrições encontre partes do grafo que ele não é capaz de resolver e tente transformar estes subgrafos em grafos que possam ser resolvidos.

2.10.5 - Transformações de Grafos (reescrita de termos)

A técnica de transformação de grafos consiste de utilizar regras de reescrita para transformar subgrafos de restrições em outros grafos possivelmente mais simples. Um exemplo de regra de rescrita é:

$$V + V \Leftrightarrow 2 * V$$

que pode ser utilizada para estabelecer a visão redundante apresentada no exemplo do ítem anterior. Em realidade as regras de reescritas são esquemas de regras pois a variável V pode ser substituída por qualquer expressão.

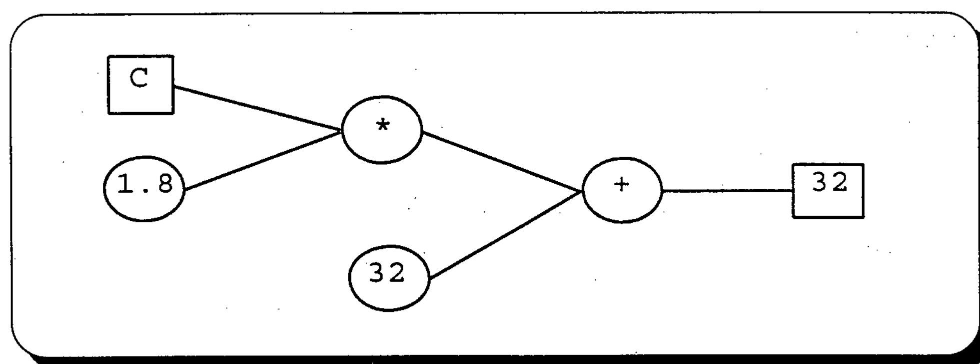


Figura 2.8 - Grafo De Conversão De Temperaturas Com $F = 32$

A transformação de grafos pode substituir completamente a propagação local. Para tal basta proceder à substituição dos operadores cujos arcos contém constantes pela constante equivalente. Por exemplo, no caso do programa de conversão de temperatura da figura 2.1, se for atribuido o valor 32 a F o grafo toma a forma apresentada na figura 2.8. Nele é possível substituir-se o operador + pela constante equivalente (zero, neste caso), resultando num novo grafo conforme figura 2.9. Este grafo pode, então, ser transformado para uma simples constante zero, que é a resposta para o problema.

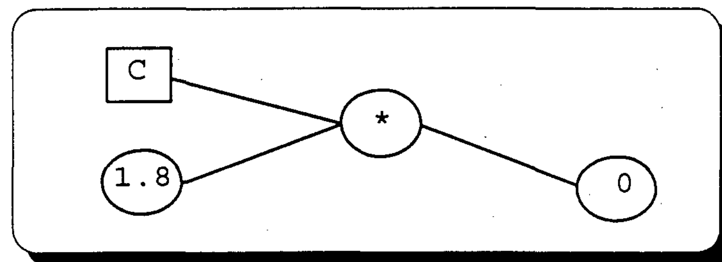


Figura 2.9 - Grafo De Conversão De Temperaturas Após A Transformação

Apesar de ser mais poderosa do que a propagação local, a técnica de transformação de grafos só é capaz de resolver ciclos simples (do tipo X + X). Ciclos mais complexos, particularmente aqueles formados por equações simultâneas, não são resolvidos por transformação de grafo pois estes ciclos tipicamente não são locais a um nodo em particular.

2.10.6 - Outras Técnicas

Algumas outras técnicas para satisfação de restrição são apresentadas em [Leler88], a exemplo da técnicas de solução de equações utilizadas em sistemas de computação simbólica. Contudo, conforme notado por Leler, estas técnicas em geral são muito lentas para serem consideradas como mecanismos gerais de satisfação de restrições.

Outra deficiência deste tipo de técnica é sua característica destrutiva, similar às técnicas de transformação de grafos. Uma vez que o grafo tenha sido transformado, o antigo é perdido a não ser que ele seja explicitamente salvo. Isto torna difícil usar o mesmo grafo repetidamente para valores diferentes.

2.11 - Considerações

A exemplo do que acontece com praticamente todas as áreas da Ciência da Computação, os sistemas e linguagens de programação por restrições estão gradativamente evoluindo. Os trabalhos iniciais desenvolvidos na década de 60, a exemplo do sistema Sketchpad de Ivan Sutherland, eram bastante específicos, sendo difícil, portanto, adaptá-los para outros contextos.

Trabalhos posteriores, como o laboratório de simulação Thinglab, já apresentam características que os tornam aplicáveis em contextos diferentes para a solução de uma classe limitada de problemas. O uso destes sistemas, contudo, requeria que o programador desçesse a nível da linguagem de implementação, neste caso Smalltalk, para definir proceduralmente novos objetos e restrições.

Outra grande deficiência de praticamente todos os ambientes de programação por restrições disponíveis é de manipular um número fixo de tipos de dados. Normalmente eles se limitam a tratar alguns tipos primitivos (caracteristicamente números inteiros, números reais, valores lógicos e, em alguns casos, pontos). Neste sentido, um dos trabalhos mais significativo (considerando ambientes que trabalham puramente com a programação por restrições) foi o

desenvolvimento da linguagem Bertrand [Leler 88]. Este sistema inclue uma forma de Tipos Abstratos de Dados, que o enriquecem no sentido de proporcionar um mecanismo de adição de objetos e restrições definidos pelo usuário.

Recentemente as pesquisas na área de programação por restrições tem dirigido sua atenção para a integração da programação por restrições com outros paradigmas, a exemplo da programação em lógica e da programação orientada a objetos. Esta integração é discutida nos capítulo 4 e 5.

3 - PROGRAMAÇÃO ORIENTADA A OBJETOS

A Programação Orientada a Objetos (POO) tem se tornado a coqueluche desta década. Apesar de algumas das idéias imbutidas nesta perspectiva de desenvolvimento de software terem suas origens nos anos 60 e 70, somente no final dos anos 80 é que ela passou a ser largamente empregada na produção de software.

Um grande número de programadores utilizam uma das várias de linguagens orientadas a objetos (LOO) disponíveis, a exemplo de Smalltalk, C++, Eiffel e outras. Contudo, e de forma diferente do que acontece com a programação em lógica e com a programação funcional, a programação orientada a objetos (POO) carece de uma teoria formal e de uma estrutura conceitual bem estabelecida. Em geral ela é expressa mais em termos filosóficos, nem sempre havendo concenso sobre o que é exatamente programação orientada a objetos.

Os tópicos seguintes procuram caracterizar programação orientada a objetos por apresentação de várias noções a ela ligadas. Estas noções estão implementadas numa ou noutra das linguagens orientadas a objetos mais conhecidas.

3.1 - O Modelo de Objetos

O modelo de objetos inserido nas linguagens orientadas a objetos caracteriza-se por ver o domínio de aplicação como uma composição de objetos que se comunicam através de troca de mensagens.

Este estilo de programação (por vezes chamado de programação por simulação, por personificação ou programação antropomórfica) procura refletir objetos físicos ou conceituais de algum domínio de aplicação em objetos de um domínio computacional. Ao propiciar um percepção mais natural do mundo real ele torna mais fácil a tarefa de modelar esta realidade e de compreender o que está descrito nas linhas de um programa.

3.1.1 - Os Objetos

Um objeto, no contexto do modelo de objetos, pode ser visto como um ente composto de uma memória privada (uma estrutura que reflete seu estado em cada instante) e um comportamento (um conjunto de ações que define sua interface de comunicação com o mundo externo). Assim, sob certo sentido, cada objeto pode ser visto como um pequeno computador ou, numa analogia mais otimista, como um ente autônomo e auto-suficiente.

Sob um ponto de mais pragmático, os objetos encapsulam estado e ações em torno de uma construção elementar, cuja noção é herdaça de Tipos Abstratos de Dados - TAD. Uma consequência imediata deste encapsulamento é a possibilidade de separação entre a interface externa (o ponto de vista do usuário do objeto) e representação (o ponto de vista do implementador). Desta forma, detalhes de implementação não precisam ser conhecidos pelo usuário do objeto, havendo um ocultamento intencional de informações.

A conjugação destes fatores, encapsulamento e ocultamento de informações, tem implicações diretas em problemas clássicos relacionados com o desenvolvimento de software. Em particular, a noção de objetos oferece uma oportunidade impar de facilitar a manutenção de programas e de exercitar o efetivo reuso de componentes de sofware.

3.1.2 - A Comunicação entre os Objetos

O mecanismo básico de comunicação entre os objetos é a troca de mensagens entre eles. Na visão classicamente implementada nas linguagens orientadas a objetos, quando ocorre uma troca de mensagens, o objeto que a envia susta temporariamente sua atividade interna e aguarda uma resposta. Esta "hibernação" só é rompida quando ele recebe um retorno em atenção à sua solicitação.

Por sua vez, o objeto que recebe a mensagem seleciona e executa uma das ações que compõem a sua interface de modo a atender convenientemente à demanda. Na execução desta ação pode ocorrer mudanças no estado interno deste objeto, envio de mensagens a outros objetos, criação de novos objetos, ou, o que é mais tradicional, todas estas atividades concomitantemente.

Enquanto aguardam a recepção de uma mensagem ou a resposta a uma demanda, os objetos ficam inativos. Este comportamento, característico de Tipos Abstratos de Dados, determina um modelo no qual os objetos são passivos, com um desenvolvimento estritamente sequencial de ações[3].

3.2 - As Noções de Classe e de Instância

O modelo de objetos descrito em 3.1, corresponde a uma visão operacional da POO. Ele pressupõe que o domínio de aplicação já tenha sido adequadamente compreendido e de alguma forma descrito. Isto implica na identificação dos fenômenos (e suas propriedades) pertinentes ao domínio em questão e na construção de um modelo conceitual que pode, então, ser representado por uma linguagem.

A classificação, uma das operações básicas para modelagem conceitual de domínios [Mattos 89] [Takahashi 90], é elemento chave na construção de modelos conceituais. Tal operação consiste de observar os diversos fenômenos (objetos) e categorizá-los (agrupá-los) segundo certas propriedades comuns a estes fenômenos. Assim, a descrição de propriedades de objetos com estrutura e comportamento idênticos pode ser feita de uma só vez, de forma concisa, gerando uma *classe* de objetos[4].

Inversamente à classificação, a operação de instanciação possibilita a obtenção objetos que satisfaçam às propridades especificadas pela classe. Cada um destes objetos é dito, então, ser

[3] Uma alternativa a este modelo considera a existência de objetos ativos. Nele, um objeto está em permanente atividade interna, sendo interrompido quando do recebimento de uma mensagem.

[4] Numa visão distinta ao modelo de classes, algumas linguagens implementam um modelo de objetos baseado na noção de protótipos. Em Cecil [Chambers 92], por exemplo, objetos auto-suficientes implementam abstrações de dados e objetos herdam propriedades diretamente de outros objetos por compartilhamento de código

uma *instância* desta classe. Neste contexto classes podem ser vistas como gabaritos (matrizes) a partir dos quais são gerados (instanciados) objetos particulares.

3.3 - Hierarquias de Classes

Além das operação de classificação descrita em 3.2, outras duas operações básicas para modelagem conceitual de domínio são utilizadas no modelo de objetos: a generalização (e sua operação inversa, a especialização) e a agregação (e sua operação inversa, a decomposição). Diferentes relacionamentos entre classes surgem da aplicação destas operações, os quais conduzem a duas hierarquias ortogonais de classes: de generalização/especialização e de agregação/decomposição.

3.3.1 - Hierarquia de Generalização/Especialização

A árvore apresentada na fig 3.1 mostra parte da taxonomia animal. Nesta árvore quanto mais próximo do topo, mais geral é o conceito apresentado. Inversamente, quanto mais próximo das folhas da árvore, mais específico é o conceito. Nesta hierarquia a relação estabelecida entre as classes é conhecida como relação *é_um* (peixe *é_um* animal, primata *é_um* mamífero, etc).

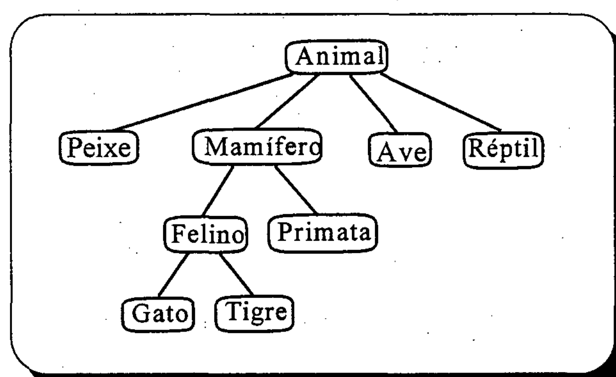


Figura 3.1 - Parte Da Taxonomia Animal

A aplicação da operação de generalização a um conjunto de classes que possuam algumas propriedades em comum, conduz à criação de uma nova classe (uma *super-classe)* mais genérica que fatora tais propriedades. Por sua vez, a operação de especialização possibilita a obtenção de uma nova classe (uma *sub-classe*) por adição de atributos que descrevam apenas as propriedades que a tornam diferente de sua super-classe.

A hierarquia de especialização/generalização inclue um número razoável de classes que não são instanciáveis, ou seja, que puramente cumprem um papel de estruturação. São as *classes abstratas*. Considerando o exemplo da figura 3.1, observa-se que na natureza há animais como gatos e tigres. Contudo, conceitos como felinos e mamíferos são puramente estruturais uma vez que não há objetos reais que sejam instâncias destas classes.

O processo de construção da hierarquia de generalização/especialização pressupõe que as super-classes descrevam aquelas propriedades que são comuns a todas as suas sub-classes, de forma que instâncias destas sub-classes *herdam* todas as propriedades descritas na sua super-classes, recursivamente[5].

3.3.2 - Hierarquia de Agregação/Decomposição

Uma perspectiva diferente de tratar objetos é vê-los como composições de outros objetos. Assim, podemos tomar instâncias de uma classe como sendo o resultado da agregação de instâncias de outras classes. Neste caso a relação entre as classes é conhecida como *é_parte_de* (motor *é_parte_do* carro, bloco *é_parte_do* motor, etc).

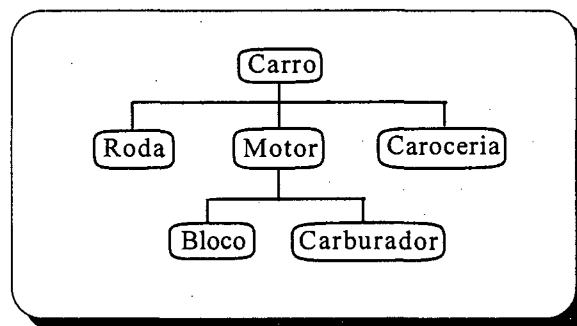


Figura 3.2 - Agregação De Partes De Um Carro

Pela composição sucessiva de partes obtem-se uma hierarquia de agregação/decomposição estruturada em forma de árvore na qual podem ser visualizados níveis de agregação. Cada nível correspondente a uma aplicação indutiva da operação de agregação que tomam um conjunto de elementos simples (atômicos) como base para construção de elementos compostos.

3.4 - Tipos e a Programação Orientada a Objetos

3.4.1 - A Noção de Tipos

A noção de tipos, em sua visão mais ampla, "visa organizar tanto o universo de discurso quanto a linguagem, pela aglutinação de elementos deste universo e de expressões da linguagem em classes (tipos, categorias) de entidades afins" [Costa 90]. Deste ponto de vista, a noção de tipo por vezes se confunde com a noção de classe.

[5] Uma derivação ao modelo de *herança simples* pressupõe a possibilidade de instâncias de uma classe herdarem propriedades de mais de uma super-classe direta: a *herança múltipla*. Neste caso a hierarquia de generalização/especialização passa a tomar a forma mais geral de um grafo dirigido acíclico. O mecanismo de herança múltipla é semanticamente mais rico que o de herança simples uma vez que permite que instâncias de uma classe compartilhem propriedades de várias (super) classes. Contudo, ele requer alguns cuidados especiais, como o de incorporar critérios para resolver certas ambiguidades decorrentes da herança de uma mesma propriedade de mais de uma superclasse.

Mas os tipos são também empregados como forma de impor restrições às construções de uma linguagem, seja ela matemática ou computacional, com o intuito de auxiliar e/ou garantir a correção de aplicações. Como notado por Danforth e Tomlinson [ver Passerino 90], em programação os tipos podem ser utilizados como meio de caracterizar os valores que dinamicamente aparecerão no curso de uma computação. Assim, os tipos servem como indicação do significado de uma expressão (uma semântica de aproximação) ao mesmo tempo que representam restrições que descrevem propriedades que tais valores devem ter. Sob certo sentido, então, as informações de tipo servem como uma forma de especificação parcial de um programa.

3.4.2 - Tipagem de Linguagens

Uma linguagem que tenha a ela associada um sistema de tipos é dita ser uma linguagem tipada. Por sistema de tipos de uma linguagem deve-se entender a maquinaria formal pela qual se associam tipos à expressões desta linguagem.

Em algumas linguagens orientadas a objetos tais como C++ e Eiffel, os tipos de variáveis e símbolos de função são explicitamente definidos. Outras linguagens são implicitamente tipada, ou seja, o sistema de inferência de tipos é capaz de deduzir do contexto os tipos de expressões com pouca ou nenhuma informação de tipo fornecida explicitamente.

Independentemente de ser explícita ou implicitamente tipada, se todas as expressões de uma linguagem possuirem algum tipo, então tal linguagem é dita ser fortemente tipada. Se tais tipos puderem ser determinados por meio de uma análise estática do programa, então esta linguagem é dita ser estaticamente tipada. Desta forma é possível ver que toda linguagem estaticamente tipada é também fortemente tipada; a relação inversa, contudo, nem sempre é verdadeira.

Num outro extremo, linguagens como Smalltalk não incorporam explicitamente a noção de tipos. Os tipos de cada variável ou expressão só podem ser determinados em tempo de execução.

3.4.3 - A Verificação de Tipos

A verificação de tipos pode ser vista como a implementação dos sistemas de tipos e, dependendo da estratégia escolhida, pode ser estática (em tempo de compilação) ou dinâmica (durante a execução). Sua função é a de checar se o tipo de um construção unifica (casa) com o tipo esperado pelo contexto.

A verificação estática de tipos é útil em muitas situações pois facilita a detecção prévia (em tempo de compilação) de erros de tipagem, garantindo a consistência dos tipos. Ela também proporciona grande eficiência em tempo de execução e força uma disciplina de programação que, em geral, conduz a programas mais estruturados e fáceis de ler. Mas seu uso pode conduzir a uma perda de flexibilidade e de poder de expressão da linguagem por restringir prematuramente o comportamento de objetos àquele associado a um tipo particular.

A posição advogada por vários pesquisadores é a de utilizar tipagem estática tanto quanto possível e tipagem dinâmica quando necessário. A observância de uma ou ambas as técnicas conduz à tipagem forte, com consequente ausência de erros de tipagem em tempo de execução. A idéia é que todas as expressões de uma linguagem sejam consistentes quanto ao tipo, apesar de os tipos poderem ser estaticamente desconhecidos.

Uma alternativa distinta considera a utilização de um sistema de checagem de tipos opcional e incremental. Linguagens como Cecil [Chambers 92] suportam um sistema estático de tipos, mas com declaração opcional. Elas provêm suporte para uma mistura de programação exploratória (característica de linguagens não tipadas, a exemplo de Smaltalk) e de programação voltada para a produção de sistemas (característica normalmente atribuida às linguagens fortemente tipadas).

3.4.4 - Polimorfismo

O termo polimorfismo, literalmente falando, significa muitas formas e é frequentemente empregado para adjetivar substantivos. Assim diz-se, por exemplo, que uma função é polimórfica quando esta função aceita argumentos de mais de um tipo. A base do polimorfismo está em permitir que as expressões de tipo possuam variáveis de tipo, cujo valor depende de cada contexto.

As formas de polimorfismo podem, em princípio, ser divididas em dois grupos [Passerino 90]: o polimorfismo universal e o ad-hoc. No polimorfimso universal as funções trabalham uniformemente sobre um conjunto possivelmente infinito de tipos, os quais apresentam uma certa estrutura comum. Segundo a forma de obtenção da uniformidade este tipo de polimorfismo divide-se em: paramétrico (a uniformidade é obtida por um parâmetro de tipo que denota o tipo do argumento) e o de inclusão (é pressuposta a existência de inclusão de classes de forma que um objeto pode pertencer a várias classes distintas). O polimorfismo de inclusão é a base para modelagem da herança.

No polimorfismo ad-hoc as funções operam sobre um conjunto finitos de tipos diferentes e potencialmente não relacionados. Comporta-se como um pequeno conjunto de funções monomórficas. A forma mais comum deste tipo de polimorfismo é a sobrecarga de operadores, no qual um mesmo nome denota diferente funções sendo que o contexto é que determina qual função é denotada por uma instância particular.

3.4.5 - Hierarquia de Tipos versus Hierarquia de Implementação

Em praticamente todas as linguagens orientadas a objetos comerciais não há distinção entre a hierarquia de tipos (conceitual) e a hierarquia de implementação no que se refere à herança. Uma subclasse herda tanto interface quanto implementação de sua super-classe.

O propósito dos dois tipos de heranças são distintos. A herança conceitual preocupa-se em especializar o comportamento enquanto que a herança de implementação visa reuso de

código. Assim, a existência de uma hierarquia única para as duas herança pode causar algumas anomalias estruturais.

Um exemplo clássico deste tipo de anomalia pode ser visto na hierarquia de classes de Smalltalk-80: a classe *Dictionary* é subclasse de *Set*, com consequente compartilhamento de implementação, apesar de a abstração para *Dictionary* não ser comportamentalmente subclasse de *Set*. Grande parte destes conflitos são decorrentes da possibilita de redefinição de propriedades em subclasses que pode resultar em comportamentos inconsistentes entre instâncias de uma classe e instâncias de suas subclasses.

A divisão desta hierarquia em dois mecanismos distintos (para especificação de comportamento e de implementacão) está sendo adotado em algumas das novas linguagens orientadas a objetos. Tal distinção introduz um elemento a mais de complexidade, mas permite que soluções particulares sejam pensadas e adotadas em cada um dos casos separadamente.

3.5 - Mensagens versus Métodos

No modelo clássico de objetos o mecanismo básico de comunicação entre os objetos é a troca de mensagens. Linguagens como Smalltalk levam este preceito a extremos, implementando inclusive as estruturas de controle como mensagens.

Uma expressão do tipo:

5 estaEntre: 1 e: 10

pode ser lida como: o objeto 5 recebe a mensagem #estaEntre:e:, tendo os objetos 1 e 10 como argumento.

Ao receber a mensagem #estaEntre:e: o objeto 5 deve executar alguma ação (um método) em resposta a esta ação. A decisão de qual ação executar depende da busca de um método apropriado na hieraquia de classes. Diferentes estratégias de busca são encontradas nas linguagens orientadas a objetos, cujas distinções básicas são apresentadas nos tópicos seguintes.

3.5.1 - Ligação Estática versus Ligação Dinâmica

Desde haja informações suficientes, em tempo de compilação de um programa é possível inferir-se qual o método que será executado em resposta a uma mensagem e ligá-lo à mensagem diretamente no código executável. Nesta situação ocorre a ligação estática.

Contudo, nem sempre é possível realizar a ligação estática em função de ambiguidades ou de incompletezas as quais só são resolvidas pelo contexto de execução. Neste caso ocorre a ligação dinâmica, ou seja, a decisão sobre qual método será executado em resposta a uma mensagem só é tomada em tempo de execução do programa.

Em linguagens que não são tipadas, a exemplo de Smalltalk, a ligação é completamente dinâmica. Contudo, mesmo linguagens estaticamente tipadas, como Eiffel, há situações em que a ligação é realizada dinamicamente em função do polimorfismo (ver 3.4.4).

3.5.2 - Disparo Simples versus Disparo Múltiplo (multi-métodos)

Na maior parte das linguagens orientadas a objetos somente o objeto receptor de uma mensagem é utilizado para decidir qual ação (método) será executada em resposta a esta mensagem. Outros objetos podem ser passados como parâmetros, mas eles não tem nenhuma participação na procura do método a ser executado.

Considerando o fragmento de código escrito em Smalltalk:

```
| p |
p := 10 @ 20.        "atribui o objeto (10,20) da classe Point à variável p"
p + 5.               "a mensagem + é enviada ao objeto denotado por p,
                          tendo o objeto inteiro 5 como argumento"
p + (5 @ 15).        "a mensagem + é enviada ao objeto denotado por p,
                          tendo o objeto ponto (5,15) como argumento"
```

observa-se que em resposta ao envio da mensagem + ao objeto denotado por *p* o mesmo método será executado em ambos os casos. Isto ocorre por que a decisão sobre qual método executar depende exclusivamente o objeto receptor da mensagem. Esta característica força o programador a incluir pontos de decisão para tratar os diferentes casos (parâmetros), a exemplo de:

```
+ delta
        "Retorna um novo Ponto que é a soma do objeto receptor com delta.
        Delta pode ser um número ou um ponto. Se delta é um ponto, as
        coordenadas x são adicionadas e as coordenadas y são adicionadas.
        Se delta é um número, este valor é adicionado em ambas as coordenadas"

        delta class == Point
                ifTrue: [^ ( x + delta x) @ ( y + delta y)]
                ifFalse: [^ ( x + delta) @ ( y + delta)]
```

Uma abordagem distinta adotada em linguagens como Self e Cecil, utilizam um esquema alternativo de ligação de métodos a mensagens que leva em consideração os tipos dos argumentos. Considerando uma linguagem com disparo múltiplo, o método + poderia ser desdobrado em dois métodos distintos:

```
+ delta<Integer>
        "Retorna um novo Ponto que corresponde à soma de delta a ambas as
        coordenadas do objeto receptor"
        ^ ( x + delta) @ ( y + delta)

+ delta<Point>
        "Retorna um novo Ponto que corresponde à soma das coordenadas x e
        das coordenadas y de delta e do objeto rececptor"
        ^ ( x + delta x) @ ( y + delta y)
```

Neste caso a decisão sobre qual método executar em resposta à mensagem + passaria a considerar também o tipo do argumento:

p + 5. -> causa a execução do primeiro dos métodos acima

e

p + (5 @ 15) -> causa a execução do segundo dos métodos acima.

A opção de utilização de multi-métodos implica necessariamente em ter-se uma linguagem com tipagem forte.Contudo, uma abordagem de tipagem opcional e incremental é suficiente.

3.6 - Identidade versus Igualdade de Objetos

Uma fonte costumeira de confusão nas linguagens orientadas a objetos é a distinção entre identidade e igualdade de objetos. Diferentes noções de igualdade podem ser encontradas nas linguagens orientadas a objetos: identidade, igualdade estrutural e igualdade definida pelo usuário.

A relação de identidade entre duas variáveis significa que ambas as variáveis referem-se ao mesmo objeto na memória do computador. Linguagens como Smalltalk oferecem um operador primitivo (método ==) para verificação de identidade entre objetos.

A igualdade estrutural, por sua vez, considera que dois objetos são iguais se seus "slots" (variáveis de instância) são iguais. O processo de comparação é recursivamente repetido tendo como base a hierarquia de agregação, sendo que igualdade de objetos primitivos é obtida por métodos primitivos.

A última forma de igualdade entre objetos é baseada em condições (critérios) de igualdade definidas pelo usuário. Este critério pode inclusive estabelecer que objetos de classes diferentes sejam iguais, a exemplo da noção de ponto que pode ser definido em duas classes distintas, uma por coordenadas cartesianas e outra por coordenadas polares. No caso da linguagem Smalltalk, o operador "=" corresponde à noção de igualdade definida pelo usuário.

Obviamente dois objetos que são idênticos são também iguais, mas não vice-versa.

3.7 - Considerações

São inegáveis os benefícios introduzidos pela orientação a objetos enquanto perspectiva de desenvolvimento de software. Além de reduzir o *hiato (fosso) semântico*, ou seja, a distância entre o domínio computacional e mundo real, a orientação a objetos oferece mecanismos que possibilitam o efetivo reuso de artefatos de software em níveis até então inimagináveis. E como decorrência da reusabilidade, a orientação a objetos introduz a prototipação como uma fase viável e factível do ciclo de vida de um software.

Mas a programação orientada a objetos é conhecida por ocultar muitas das relações entre os objetos. Isto inclue situações como as relações temporais encontradas em especificação de comportamento de interfaces de aplicação [Hartson 89], como, também, a manutenção de requerimentos de consistência entre os objetos envolvidos numa aplicação. Estas relações estão presentes no modelo conceitual acerca de uma aplicação mas não são diretamente expressáveis no modelo de objetos. Em geral o programador as codifica em métodos que procuram implicitamente satisfazê-las.

Uma alternativa para tornar mais aparentes e claras as relações entre objetos é buscar a integração da perspectiva puramente imperativa da orientação a objetos com características declarativas de outras perspectivas como a programação por restrições e a programação em lógica. Está integração de pardigmas é alvo dos capítulos 4 e 5.

4 - A INTEGRAÇÃO DA PROGRAMAÇÃO ORIENTADA A OBJETOS E DA PROGRAMAÇÃO POR RESTRIÇÕES

A programação orientada a objetos oferece uma filosofia de programação que possibilita o desenvolvimento de "programas" que refletem de forma bastante aproximada a realidade que esta sendo modelada. Aliada a esta melhor apreensão da realidade, a programação orientada a objetos oferece um conjunto de ferramentas (encapsulamento, herança, polimorfismo, mensagens, etc) que provêm um mecanismo bastante eficiente para extensão do domínio computacional da linguagem. Mas a programação orientada a objetos é conhecida por ocultar muitas das relações entre os objetos. O programador cria um modelo mental de requerimentos de consistência entre os objetos envolvidos numa aplicação e codifica os métodos procurando implicitamente satisfazer a estes requerimentos. Esta é, em realidade, uma das falhas da programação orientada a objetos apontadas por vários pesquisadores.

Por outro lado, a programação baseada em restrições facilita a descrição de relações que devem ser mantidas entre elementos do domínio computacional da linguagem. Estas restrições são resolvidas por um sistema de satisfação de restrições que normalmente encontra-se embutido na linguagem de programação. Este sistema é que se encarrega de escolher os algoritmos que devem ser executados e de estabelecer a ordem na qual eles serão aplicados. Ela, contudo, carece precisamente de um mecanismo que possibilite a extensão do domínio computacional.

A integração de Objetos e de restrições parece, então, uma alternativa bastante natural. O principal objetivo desta integração é oferecer um mecanismo que permita tornar explícitas várias relações entre objetos, relações estas que normalmente aparecem implicitamente nas linguagens de programação mais tradicionais.

4.1 - Motivação para a Integração

Provavelmente a maior motivação para se buscar a integração da programação orientada a objetos e da programação baseada em restrições em uma linguagem única é a observação de que em aplicações típicas há algumas partes que são mais claramente e convenientemente descritas utilizando-se restrições, enquanto que outras partes são mais claramente descritas utilizando-se contruções imperativas.

Historicamente a aplicação que mais motivou esta integração foi a construção de interfaces gráficas com o usuário. Neste tipo de aplicação, há vários relacionamentos entre os elementos gráficos dispostos na tela que perduram durante toda a vida destes elementos (relações de longa vida). Supondo que se defina a disposição de uma janela por meio de dois pontos, seu canto superior esquerdo e seu canto inferior direito, então algumas relações podem ser estabelecidas, tais como:

$$\text{extensão} = \text{cantoInferiorDireito} - \text{cantoSuperiorEsquerdo}$$

$$\mathit{centro} = \mathit{extensão}/2$$

que deveriam ser satisfeitas sempre que algumas das variáveis fosse modificada. Elas deveriam ser bidirecionais e mantidas automaticamente.

Já ações disparadas por movimentação ou pressão de botão do "mouse" são melhor descritas por meio de ações imperativas que modificam (perturbam) as relações estabelecidas por meio destas restrições.

Apesar das interfaces com o usuário poderem ser escritas em linguagens imperativas de alto nível, algumas partes poderiam ser mais sucintamente e claramente descritas num estilo declarativo. Estas partes normalmente são fontes de problemas, particularmente na fase de manutenção, pois utilizando-se programação imperativa as relações são descritas em um nível baixo e os fragmentos de código são distribuidos ao longo do programa. Em suma, o programador é responsável por incluir nos pontos corretos o código necessário para a manutencão das restrições.

4.2 - Características Desejáveis nesta Integração

Ao se buscar a união da programação orientada e da objetos e a programação por restrições, algumas características deveriam ser preservadas.

4.2.1 - Ampliação da Flexibilidade e Expressividade

A integração das duas perspectivas deve expandir certas característas das modernas linguagens orientadas a objetos, em particular aquelas que se referem a flexibilidade e expressividade. Ao fazer isto, tanto quanto possível a parte declarativa deveria ser independente de alternativas ao modelo clássico de objetos, a exemplo da herança múltipla, da diferenciação nas hierarquias de tipo e de implementação, protótipos, etc.

A que se notar que o aumento de expressividade de uma linguagem de programação normalmente implica em redução de sua eficiência (aumento do tempo de execução). Esta relação é bem conhecida, como demostraram a a programação estruturada nos anos 70 e o surgimento da programação orientada a objetos nos anos 80. Todavia, ganha-se em tempo de programação.

4.2.2 - Suporte ao Estilo Orientado a Objetos

Uma característica que normalmente norteia as experiências de integração de programação por pestrições com a programação orientada a objetos é a de, tanto quanto possível, manter o estilo orientado a objetos. Isto inclue, por exemplo, permitir que o usuário estabeleça restrições sobre objetos/classes por ele criados, de forma similar à forma como ele define novos métodos para estes objetos/classes. Inclui, também, a interação consistente com os vários mecanismos da linguagem, a exemplo da herança e do polimorfismo.

Se for guardada inclusive a pureza de Smalltalk, mesmo as restrições deveriam ser objetos. Sendo assim, e de forma similar aos blocos (instâncias da classe Context) de Smalltalk, elas estariam sujeitas a serem armazenadas em coleções, a serem passadas como argumentos de

métodos e a serem avaliadas e/ou inseridas no sistema de satisfação de restrições por meio de mensagens específicas a ela enviadas.

4.2.3 - Manutenção do Estilo de Programação Orientado a Objetos

A exemplo da agregação de sistemas de tratamento de excessão às linguagens orientadas a objetos, a inclusão de um sistema de satisfação de restrição deveria ser tão transparente quanto possível. Isto implica em não se forçar alterações extensivas no estilo de programação. Caso o mecanismo de restrições não seja usado, deveria ser possível escrever programas segundo o padrão imperativo tradicional.

4.3 - Alguns Obstáculos para a Integração

Apesar de parecer óbvia e natural, a integração entre as duas perspectivas apresenta alguns problemas que precisam ser melhor analisados. Boa parte das dificuldades para a integração destes dois paradigmas são devidas às diferenças fundamentais entre eles. Ou seja, a mesma dicotomia que torna interessante e útil a sua integração é também responsável por uma série de obstáculos.

```
class Círculo subclassOf FiguraGeométrica
        Var     centro: Point;
                raio: Number;
        Methods
                centro  "retorna o ponto correspondente ao centro da círculo"
                        ^centro
end;

class Quadrado subclassOf  FiguraGeométrica
        Var     origem: Point;
                lado: Number;
        Methods
                centro  "retorna o ponto correspondente ao centro do quadrado"
                        ^origem + (lado / 2)
end;

Var     c1, c2: Círculo
        q1, q2: Quadrado
```

Figura 4.1 - Definição Das Classes *Círculo* E *Quadrado*

Para efeito de exemplificação de alguns obstáculos para esta integração descritos nos tópicos seguintes, considere as classes *Círculo* e *Quadrado* e as variáveis definidas na figura 4.1.

4.3.1 - Os modelos de Execução e de Armazenamento de Dados

O paradigma imperativo é baseado na arquitetura de Von Neumann que, grosseiramente, pode ser vista com composta um um processador de instruções e por um depósito de dados (que armazena instruções e valores). O processador de instruções executa quatro tipos básicos de instruções: leitura de um valor, escrita destrutiva de um valor, cômputo de alguma função sobre valores e desvio (condicional ou incondicional). Neste modelo, cada locação do depósito armazena um único dado em cada tempo, o qual pode ser modificado por uma atribuição destrutiva. Assim, este depósito de dados pode ser visto como uma função:

Tempo x Locação –> Valor

O paradigma de restrições, por sua vez, não inclue em si um processador de instruções nem um depósito de dados. Um programa com restrições pode ser representado por um hipergrafo no qual os nodos são variáveis e as arestas são restrições. Cada variável comporta o conjunto de todos os valores que são consistentes com o grafo de restrições, o qual é constante para este grafo. O único meio de alterar este conjunto de valores é operar com o grafo por meio de algum programa externo. O grafo pode, então, ser visto como uma função:

Locação –> ConjuntoDeValores

onde *Locação* é o nome de uma variável.

A programação imperativa com restriçoes deve preocupar-se em integrar de alguma forma estas duas diferentes perspectivas.

4.3.2 - Domínio Computacional da Linguagem

A programação orientada a objetos oferece um mecanismo bastante poderoso de extensão do domínio computacional da linguagem de programação. A aplicação das operações de classificação, de especialização e de generalização (as operações básicas para modelagem conceitual de domínios) fomentam a implementação de novas classes (programação diferencial ou programação por extensão).

A adição da programação por restrições numa linguagem orientada a objetos implica na incorporação de um sistema de satisfação de restrições nesta linguagem. Há bons algoritmos para tipos particulares de restrições, particularmente aquelas relacionadas com números reais ou restrições acíclicas sobre domínios arbitrários. Contudo, nem sempre há bons algoritmos para restrições arbitrárias sobre domínios arbitrários.

4.3.3 - Restrição sobre o Todo e/ou sobre as Partes

Em linguagens de programação baseadas em restrições, as restrições são declarações que estabelecem relações entre elementos de domínio computacional da linguagem. Mas numa linguagem orientada a objetos um objeto pode ser composto de várias partes, a hierarquia de agregação/decomposição, de forma que as restrições podem relacionar elementos de diferentes níveis desta hierarquia.

Uma restrição como:

$$c1 = c2$$

estabelece uma relação de igualdade entre os círculos *c1* e *c2*, provavelmente considerando algum relacionamento entre os centros e raios dos dois círculos. Já uma restrição como:

$$c2.centro = 10@20$$

relaciona diretamente as componentes *centro* do círculo *c2*.

Grande parte dos algoritmos para solução de restrições descritos em 2.10 são aplicáveis sobre domínios primitivos tais como números inteiros, números reais e valores lógicos (verdade e falso). Estes algoritmos nem sempre conseguem manipular diretamente informação mais estruturada.

4.3.4 - Definição de Novas Restrições

As noções de classe, herança e polimorfismo oferecem às linguagens orientadas a objetos um excelente mecanismo para definição de novos métodos (programação por refinamento/extensão). Mas estas linguagens em geral não oferecem mecanismos para a definição de novas restrições. Nos casos em que mecanismos de definição de restrições podem ser oferecidos, estes mecanismos nem sempre respeitam a natureza orientada a objetos da linguagem.

Considere, por exemplo, que se desejasse atestar que:

$$c2.centro = 10@20$$

A satisfação desta restrição implicaria em se atribuir o ponto 10@20 à variável de instância *centro* de *c2* , caracterizando uma quebra de encapsulamento/proteção.

Já uma restrição do tipo:

$$q2.centro = 10@20$$

apresenta um outro tipo de desconforto. Neste caso *centro* é uma ação definida na classe *Quadrado* que calcula e retorna o valor do centro do quadrado. A satisfação desta restrição implicaria em alguma modificação indireta dos valores das variáveis de instância *origem* e/ou *lado* (uma espécie de efeito colateral) como única forma de satisfazê-la.

4.3.5 - Atribuição Destrutiva versus Satisfação de Restrição

Grande parte das linguagens orientadas a objetos são imperativas, caracterizando-se por incluir operações de mudança de estado, em particular a operação de atribuição que é destrutiva. Já as Linguagens de Restrições são caracteristicamente declarativas. O programador estabelece um conjunto de restrições que são continuamente verificadas, em geral a cada modificação de valor de uma das variáveis envolvidas. Ao perceber que uma restrição não está sendo satisfeita, o sistema de satisfação de restrições trata de calcular novos valores para as variáveis de forma a que todas as restrições sejam satisfeitas. Esta visão distinta de como são modificados os valores das variáveis pode conduzir a situações contraditórias que precisam ser tratadas.

Por exemplo, suponha que se tenha uma restrição do tipo:

$$c1.centro \neq 32@20$$

que diz que centro de *c1* pode ser qualquer valor exceto o ponto 32@20, e suponha que seja atribuido a *c1.centro* exatamente este valor:

$$c1.centro \leftarrow 32@20$$

Há visivelmente uma contradição entre as duas declarações. Se considerarmos que a atribuição seja efetivada, então a restrição ou não poderá ser mantida ou permanecerá em um estado incosistente durante um período de tempo. Por outro lado, se considerarmos que a atribuição falhe, então tem-se um problema de estabelecimento de significado para a atribuição.

Mesmo que não haja exatamente uma contradição, a atribuição destrutiva pode causar dúvidas no que se relaciona com o período de validade de uma restrição. Considere que seja estabelecida uma restrição do tipo:

$$c1.centro = 32@20$$

e que em algum tempo futuro seja atribuido um valor qualquer a *c1.centro*:

$$c1.centro \leftarrow 10@15$$

Se considerarmos que imediatamente após a operação de atribuição o sistema de satisfação de restrições entre em ação, possivelmente o valor de *c1.centro* retornaria imediatamente ao valor 32@20 e, portanto, a atribuição seria completamente inóquoa.

A programação por restrições preocupa-se em como as restrições são satisfeitas, mas não se preocupa nem com o quando, nem com os passos procedurais necessários para satisfazê-las. Considerando sua integração com programação orientada a objetos, há a necessidade adicional de se definir claramente quando elas são resolvidas. As alternativas extremas são a de invocar automaticamente o sistema de satisfação de restrições a cada mudança de valor de uma variável, ou de transferir para o programador a responsabilidade de explicitamente indicar o momento de ativação. Há também que se definir um mecanismo de tratamento de contradições como as descritas acima.

4.4 - Os Diferentes Tipos de Restrições

Diferentes tipos de restrições surgem quando se oferece a possibilidade de estabelecer restrições sobre classes e sobre objetos.

4.4.1 - Restrições sobre valores

Restrição sobre valores são asserções sobre o conteúdo (valor) das variáveis de instância de um objeto. Exemplos como:

$$c1.centro = 10@20$$

$$c1.centro = c2.centro$$

restringe diretamente a componente *centro* dos círculos a valores particulares: o centro de *c1* a ser igual ao ponto (10,20); e os círculos *c1* e *c2* a terem o mesmo centro.

Este tipo de restrição é o mais comumentemente tratado na literatura corrente sobre sistemas de restrições.

4.4.2 - Restrições de identidade

Conforme visto em 3.6, a identidade e igualdade de objetos são características distintas. Se considerarmos que duas variávies x e y denotam o mesmo objeto (identidade), então uma mudança de estado causada por um envio de mensagem a x também deve ser vista quando acessando o objeto denotado por y. Em contraste, se x e y referem-se a objetos iguais, mas não idênticos, mudanças no objeto denotado por x não deveriam afetar o objeto denotado por y.

Quando da integração da programação orientada a objetos com programação por restrições, muitos efeitos similar ao de identidade podem ser obtido por uma restrição do tipo:

$$c1 = c2$$

Contudo, este tipo de construção normalmente refere-se à igualdade de objetos (possivelmente uma igualdade definida pelo usuário) e não à identidade. Se quando da integração dos dois paradigmas desejar-se manter as características da orientação a objetos, inclusive a manutenção da expressividade das linguagens orientadas a objetos, a restrição de identidade deveria ser explicitamente declarada pela de um operador similar ao método == de Smalltalk.

4.4.3 - Restrições de classe

Uma restrição de classe é uma asserção sobre a classe de uma variável (seu tipo). Esta forma de restrição é particularmente necessária se considerarmos que a liguagem inclua um sistema opcional e incremental de tipos (ver 3.4.3) e multi-métodos (ver 3.5.2). O sistema de satisfação para este tipo de restrições deve preocupar-se em como determinar classes para variávies sem classes (inferência de tipos). Há, também, a necessidade de se estabelecer critérios para a escolha de métodos quando estiverem envolvidas variáveis sem tipos.

Considere, por exemplo, uma definição como:

```
Var r, s, t;
r <- novo(Ponto);      "r passa a denotar um objeto da classe Ponto"
t <- novo(Ponto);      "t passa a denotar um objeto da classe Ponto"
```

e uma restrição:

$$r + s = t$$

O sistema de satisfação de restrições deve ser capaz de determinar a classe da terceira variável baseado numa regra do tipo:

```
+(a, b: Ponto) = (c: Ponto);
        a.x + b.x = c.x;
        a.y + b.x = c.y
fim +;
```

que define o critério de igualdade para objetos da classe *Ponto* baseado na comparação de suas coordenadas.

4.4.4 - Restrições sobre Estruturas

Exemplos de restrições sobre estruturas incluem restrições sobre o tamanho de objetos de classes como *Array*, assim como em operações como adição de matrizes que restringe tantos os valores como a estrutura das matrizes.

Considerando, por exemplo, uma definição como:

```
Var r, s, t;
r <-- #( 1 2 3 4 5);       "r passa a denotar o objeto #( 1 2 3 4 5) da classe Array"
s <-- #( 6 7 8 9 10);      "s passa a denotar o objeto #( 6 7 8 9 10) da classe Array"
```

e uma restrição:

$$r + s = t$$

O sistema de satisfação de restrição deve ser capaz de determinar que *t* passará a denotar um objeto da classe *Array* de tamanho 5 correspondente à soma de *r* e *s*.

4.5 - A Aplicabilidade das Técnicas Clássicas de Satisfação de Restrições

São discutidas a seguir algumas abordagens para a integração da programação por restrição com a programação orientada a objetos.

4.5.1 - Uso exclusivo de propagação local

Apesar de sua popularidade, principalmente em função da simplicidade e velocidade, a propagação local é incapaz de resolver restrições cíclicas (a exemplo de equações simultâneas) e restrições com informações parciais (como a relação <maior que>). Além disto, conforme visto em 2.10.1, a propagação local opera apenas com informações locais, de sorte que ela não trabalhará corretamente quando forem usadas restrições em diferentes níveis da hierarquia parte-todo, a não ser que estejam disponíveis informações que conectem as partes ao todo e vice-versa.

4.5.2 - Restrições sobre folhas (objetos primitivos)

A técnica de estabelecimento de restrições somente sobre objetos primitivos é menos usual que a propagação local. Ela pressupõe a separação do domínio computacional da linguagem em dois sub-domínios: do objetos primitivos (*Integer*, *Float*, *Boolean*, ...) e dos objetos complexos (compostos) definidos pelo programador. As restrições disponíveis são pré-definidas e só podem ser estabelecidas sobre os objetos primitivos, ou seja, sobre as folhas da árvore resultante da hierarquia de agregação.

Dado que as restrições possíveis compõem um conjunto fixo, o programador não pode criar novas restrições nem para os objetos primitivos nem para os objetos específicos da aplicação. É claro que ele pode dividir uma restrição sobre um objeto composto em um conjunto de restrições sobre objetos primitivos. Contudo, está divisão tem alcance limitado pois as restrições só valem no escopo onde foram definidas.Além disso, elas são baseada na implementação concreta dos objetos, fato que força o programador a violar o encapsulamento de objetos ao mesmo tempo que compromete algumas características da programação orientada a objetos como modificabilidade e reusabilidade.

4.5.3 - Adição de Novos Resolvedores de Restrições

Esta abordagem baseia-se na definição de novos resolvedores de restrições para manipular novos domínios. Cada resolvedor conterá algoritmos eficientes baseados em conhecimento específico do domínio.

Esta técnica, contudo, apresenta problemas de flexibilidade similares à tecnica de restrições sobre folhas. Mas possivelmente o pior defeito desta técnica é que em geral ela exclui a possibilidade de restrições inter-domínios, a exemplo de situações como:

$$c1.raio = 15.2 * x - 9.4 * y$$

que envolve objetos da classe *Círculo* (denotado por *c1)* e da classe Float. O problema é que a integração de dois domínios requer a integração de dois resolvedores, característica esta nem sempre suportada pelos resolvedores.

4.5.4 - Reescrita de grafo

A técnica de reescrita de grafo de restrições tem sido empregado em alguns sistemas como Bertrand [Leller 88] e Equate [Wilk 91]. Nestes sistemas, a descrição de um objeto inclui a definição de seu estado (variáveis de instância), de seu comportamento (ações, métodos) e das regras de reescrita que são a ele aplicáveis.

Apesar de preservar o encapsulamento dos objetos e de ser facilmente extensível, esta técnica inclui algoritmos de solução de restrição que geralmente são ineficientes dadas as transformações envolvidas. Além disto, ela introduz um modelo de execução bastante diferente do clássico modelo imperativo.

4.6 - A abordagem de constructores

As técnicas de satisfação descritas no capítulo 2 são utilizadas em linguagens e sistemas que operam exclusivamente com restrições. Praticamente todas estas técnicas falham quando se tenta integrá-las à perspectiva de orientação a objetos, o que parece indicar que uma outra alternativa deve ser buscada.

O que se nota é que cada uma das perspectivas assume um modelo computacional puro: declarativo no caso da programação por restrições e imperativo no caso da programação orientada a objetos. A efetiva integração destas duas perspectivas é buscada em linguagens como Kaleidoscope pela definição de uma máquina virtual especial, a máquina K [Lopez 94a], que reune características das duas perspectivas.

4.6.1 - Os construtores

Em Kaleisdoscope os construtores estendem a linguagem com restrições definidas pelo usuário, de forma similar à extensão provida pelos métodos nas linguagens orientadas a objetos. A diferença fundamental entre eles é que os métodos (procedimentos) somente são executados quando invocados explicitamente no corpo de um programa, enquanto que os construtores são re-executados automaticamente quando houver uma alteração em um dos objetos ou variáveis por eles restritos.

O conceito de construtores de restrições possibilita a definição de novas restrições em função de restrições mais primitivas. Em um certo tempo todas as restrições definidas pelo usuário reduzem-se a restrições primitivas, as quais são tratadas pelo sistema de satisfação de restrições sobre estes domínios primitivos.

```
constructor = (p: Point, q: Point);
        p.x = q.x;
        p.y = q.y;
end constructor =;

constructor + (a,b:Point) = (c: Point);
        a.x + b.x = c.x;
        a.y + b.y = c.y;
end constructor +;
```

Figura 4.2 - Construtores Para A Classe *Point*

A figura 4.2 apresenta dois exemplos de construtores. O primeiro deles define a igualdade entre dois objetos da classe *Point*[6] por estabelecimento de restrições de igualdade entre as coordenadas *x* e *y* dos pontos. De maneira similar o segundo construtor estabelece que o objeto denota por *c* será igual à soma dos objetos denotados por *a* e *b*. As restrições envolvidas

---

[6] A classe *Point* normalmente é implementada como uma agregação de dois valores numéricos correspondentes às suas coordenadas *x* e *y* num plano cartesiano.

são multidirecionais, de forma que no caso do construtor +, por exemplo, dados dois dos valores o terceiro será computado sem nenhuma instrução adicional.

As construções do tipo *a.x* que aparecem nas figuras 4.2 e 4.3 não são referências diretas à variável de instância *x* do objeto denotado por *a*. Ao contrário, esta construção preserva o encapsulamento do objeto ao enviar a mensagem *x* ao objeto *a*. Isto possibilita tanto o retorno do valor armazenado na variável *x* como um valor qualquer computado por *a*. Além disto, esta computação é realizada por meio de restrições multi-direcionais de forma que valores podem fluir tanto de dentro como para dentro do objeto.

```
procedure início();
        var p1,p2,p3,p4: Point;

        p1 <- 2@2;
        p2 <- 10@10;
        always: p1 + p2 = p3;                 (1)
        p4.x <- 100;  p4.y <- 100;            (2)
        p2 <- p4;                             (3)
end início;
```

Figura 4.3 - Um Programa Envolvendo Restrições

Para ilustrar a execução incremental de construtores, considere a execução do procedimento apresentado na figura 4.3. Na linha (1) é escolhido o construtor "+ (a,b:Point) = (c: Point)" desde que *p1* e *p2* denotam objetos da classe *Point*. O objeto *Point* em *p3* torna-se a soma de *p1* e *p2*. Este construtor não é re-executado após as atribuições da linha (2) pois estas mudanças de estado não tem influência sobre *p1*, *p2* ou *p3*. A atribuição da linha (3), contudo, requer que a restrição "p1 + p2 = p3" seja re-satisfeita já que ela não é mais verdadeira. O construtor apropriado é escolhido (neste caso o mesmo construtor usado anteriormente) e executado.

4.6.2 - A vidas das Restrições

Kaleidoscope utiliza uma abordagem estruturada para determinar quando as restrições devem estar ativas. Este mecanismo guarda certa similaridade com as estruturas de controle da programação estruturada.

No caso de não haver definição explícita é assumido que a restrição permanece ativa durante toda a vida do programa. Este mesmo comportamento pode ser obtido pela prefixação da palavra *always* a um restrição, a exemplo de:

always: mouse.position = cursor.position;

Outra alternativa são as restrições declaradas como *once* que instrui o sistema a estabelecer e satisfazer a restrição naquele instante e, então, imediatamente retirar. Esta opção é particularmente importante no sentido de estabelecer valores iniciais para variáveis.

A atribuição é vista como um caso particular restrição prefixada por *once*. A expressão do lado direito é avaliada e uma restrição unidirecional do tipo *once* é aplicada entre este valor e a expressão do lado esquerdo. Esta alternativa integra a atribuição destrutiva com o sistema de restrições.

Por último, a construção *during* especifica que uma restrição deve estar ativa somente duante a execução de um bloco sujeito a repetição. O exemplo seguinte estabelece que a posição da janela deve ser a mesma do "mouse" enquanto o botão do "mouse" estiver pressionado:

```
assert mouse.position = window.position
      during
            while mouse.button = down
                  ...
            end while;
```

4.6.3 - A Máquina K

A máquina K interpreta códigos-k, que incluem tanto instruções imperativas típicas (*Add*, *Load*, *Branch*, etc), como também operações baseadas em restrições sobre objetos. O armazenamento de dados é representado por meio de grafo cujos nodos representam variáveis e as arestas representam referências e restrições. Isto significa que há dois tipos de arestas: um tipo que representa referências tradicionais de variáveis para objetos (linhas pontilhadas na figura 4.4) e outro tipo que representa restrições entre variáveis (linhas contínuas na figura 4.4).

De forma a manter as restrições de acordo com as mudanças dos objetos, em cada mudança de estado o sistema identifica as ligações de restrição da variável que foi modificada para outras variáveis e objetos, encontra construtores que tratam estas mudanças, executa o código necessário para re-satisfazer as restrições, encontra novas variáveis que foram modificadas, e assim por diante.

No nível da máquina K, construtores e procedimentos tem representação idêntica: uma assinatura e um bloco de códigos-K. A nível de execução, contudo, o tratamento é distinto. As chamadas de procedimento são manipuladas de maneira similar à encontrada em outras linguagens orientadas a objetos. As restrições, por sua vez, são tratadas por meio de *padrões de restrição*, que, em essência, são chamadas continuamente repetidas a construtores.

Um padrão de restrição é criado para cada instância de uma restrição, o qual relaciona as variáveis que estão sendo restringidas e o nome da restrição a ser aplicada. Para uma restrição *always*, um padrão de restrição é adicionado e permanece vigorando. No caso de uma restrição *once*, um padrão de restrição é adicionado e imediatamente retirado.

Um padrão é executado assim que ele é adicionado ou quando ele pode ter sido afetado por uma mudança de estado, correspondente a uma execução incremental de restrições. Um exemplo desta execução incremental é apresentado em 4.6.1.

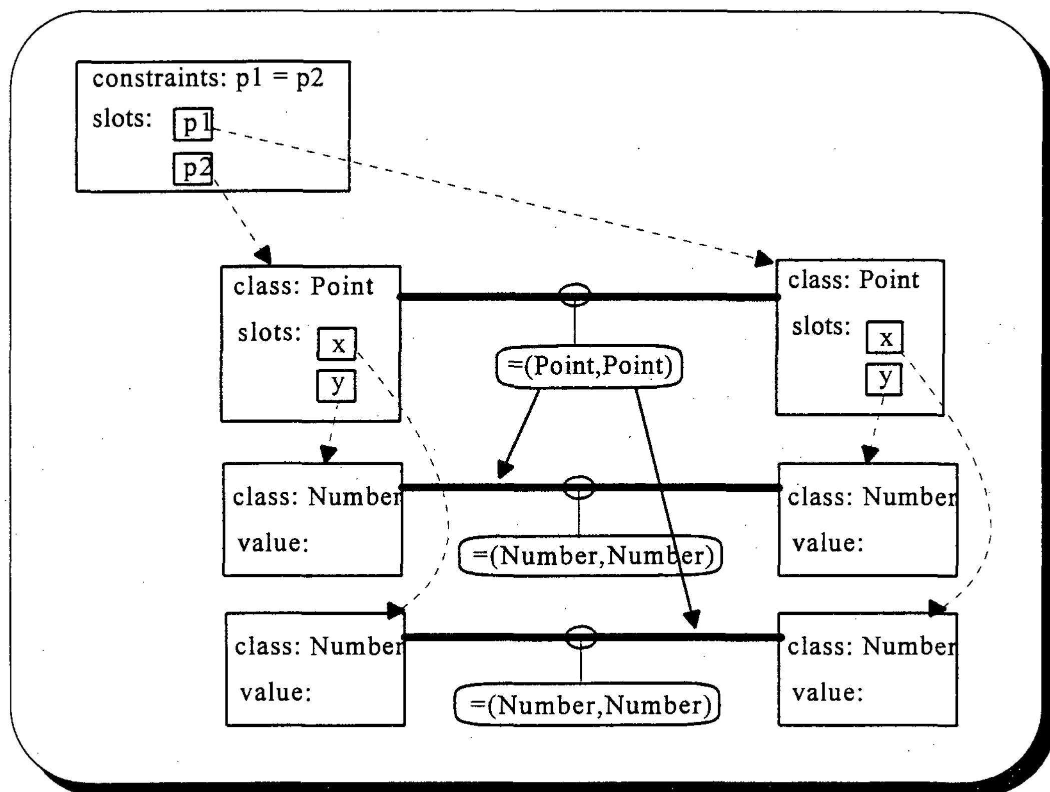


Figura 4.4 - Representação Do Grafo De Restrições

4.7 - Considerações

Poucas são abordagens que tentam efetivamente integrar a programação por restrições com a programação orientada a objetos. As linguagens Bertrand [Leler 88] e Siri [Horn 92] são baseadas numa máquina de reescrita aumentada de termos que, conforme visto em 4.5.4, apresenta algumas deficiências. Outra linguagem que busca esta integração é Leda [Budd93], mas a abordagem é extremamente rudimentar.

A noção de construtores de restrições e a definição de uma máquina virtual especial faz de Kaleidoscope[Lopez 94a] uma boa alternativa para integração das duas perspectivas. Esta abordagem é utilizada no capítulo 7 para estabelecimento de restrições quando da implementação do modelo back-propagation de redes neurais.

5 - INTEGRAÇÃO COM A PROGRAMAÇÃO EM LÓGICA

Conforme notado por Budd [Budd 93], alguns autores tem mostrado as dificuldades de se combinar diferentes paradigmas. Alguns deles consideram os paradigmas de programação em lógica e de programação orientada a objetos como sendo fundamentalmente incompatíveis. A principal argumentação de incompatibilidade apega-se às distinções relacionadas com os mecanismos de controle: a seqüência de controle é fundamental em paradigmas como o de programação orientada a objetos, enquanto que é irrelevante em paradimas declarativos como o de programação em lógica.

Contudo, Budd argumenta que os programadores que utilizam a perspectiva declarativa não trabalham em um mundo no qual o mecanismo de procura opera de forma não determinística. Ao contrário, eles exploram um conhecimento detalhado da ordem em que a busca será executada. Assim, adicionar objetos apenas introduz estruturas de dados mais complexas a serem manipuladas em programas em lógica.

Mas a maior argumentação no sentido de mostrar que esta integração é possível é a existência de linguagens como Leda [Budd 93] que mostram que algoritmos interessantes e não triviais podem ser desenvolvidos em uma estrutura multiparadigmática. Esta perspectiva é explorado nos tópicos seguintes.

5.1 - A Programação em Lógica

"Um programa em lógica é um modelo de um determinado problema ou situação expresso através de um conjunto finito de sentenças lógicas" [Casanova 87]. Divergindo da programação imperativa, um programa em lógica assemelha-se mais a um banco de dados (apesar de ter o escopo bem mais genérico), ficando a cargo de um interpretador/compilador definir o processo a ser adotado para a pesquisa de soluções.

O processo de busca de solução corresponde a uma dedução que é baseada nas condições expressas na consulta em presença das informações descritas pelo programa. Esta busca normalmente é caracterizada por uma pesquisa de refutação (uma dedução de uma contradição).

Mais do que apenas indicar se uma suposição acerca das informações contidas num programa é verdadeira ou não, uma resposta a uma consulta a um programa em lógica pode extrair informações que podem inclusive ser acompanhadas de uma explicação sobre como ela foi obtida.

A programação em lógica tem em Prolog (PROgramming in LOGic) sua maior expressão. Apesar da relativa jovialidade de Prolog, data do início da década de 70, a programação declarativa como um todo existe a mais tempo e engloba a programação funcional e quase todas as linguagens de consulta a base de dados [Casanova 87].

5.2 - A Programação em Lógica com Restrições (CLP)

Um ramo de integração de diferentes paradigmas são as CLPs, programação em lógica com restrições [Borning 92] [Lassez 87]. CLPs são esquemas gerais de extensão da programação em lógica para incluir restrições.

Em programação em lógica padrão, a exemplo de Prolog, as regras (cláusulas) assumem a forma:

$$p(t) \text{ :- } q_1(t), ..., q_m(t).$$

onde p, q_1, ..., q_m são símbolos de predicado e *t* denota uma lista de termos. Nas CLPs as regras assumem a forma de:

$$p(t) \text{ :- } q_1(t), ..., q_m(t), c_1(t), ..., c_n(t)$$

onde p, q_1, ..., q_m são tais como na programação em lógica padrão, e c_1, ..., c_n são restrições.

Operacionalmente, as CLPs podem ser pensadas como a execução usual da parte Prolog de um programa, com acumulação de restrições sobre variáveis lógicas à medida que se vai executando o programa, e verificando que as restrições sejam satisfeitas ou, em caso contrário, realizando um "backtracking". Contudo, o conceito de unificação sintática no Universo de Herbrand é susbstituído por satisfação de restrições no domínio da aplicação [Lassez 87].

5.3 - As Relações em Leda

Leda é um linguagem de programação que possibilita o desenvolvimento de programas multi-paradigmáticos, incluindo a programação orientada a objetos, a programação funcional e a programação em lógica.

Em Leda os programas em lógica são construídos em torno de uma função que retorna um tipo de dados chamado de *relation* (a função *items* descrita na figura 5.1 exemplifica esta construção). Este tipo de dados pode ser pensado como um valor lógico ("boolean"), apesar de apresentar características que o tornam um pouco mais complexo. Em particular ele está vinculado a uma forma especial de operação de atribuição (denotada pelo símbolo <–) que efetua a ligação (unificação) de um valor a um nome, além de retornar um valor *true* como resultado da função (a função *unify* apresentada na figura 5.2 exemplifica o uso deste operador de atribuição).

A classe *Set*, figura 5.1, exemplifica a integração dos paradigmas. De forma similar aos programas escritos em linguagens de programação em lógica, a função *items* é definida por uma série de disjunções ("|") sobre uma série de conjuncões ("&").

A classe *Set* é definida recursivamente em função dela própria. Nesta definição, o parâmetro de tipo *X* é restrito a ser um tipo que seja subclasse da classe *equality*[7]. Fato similar ocorre com o parâmetro de tipo *T* que aparece na definição da função *unify* descrita na figura 5.2.

---

[7] Em Leda, instâncias de subclasses de *equality* caracterizam-se por saber como se comparar por igualdade com outros objetos.

```
class Set [X : equality];
        var     value: X;
                next : Set [X];
        ...
        function items (byRef val : X) -> relation;
        begin
                return unify[X] (val, value)
                        | defined(next) & next.items(val);
        end;
end;
```

Figura 5.1 - A Classe *Set*.

```
function unify [T : equality] (byRef left : T, right : T) -> relation;
        "Se left é definido (não NIL), a função retorna o resultado
         da comparação de igualdade. Caso left seja indefinido,
         ela é unificada com right e a função retorna true"
begin
        if defined(left) then
                return left = right
        else
                return left <- right;
end;
```

Figura 5.2 - A Função *Unify*

A função *items* define uma relação que toma um valor simples como argumento. Se este valor é definido[8] quando da invocação da função, então a relação simplesmente retorna valor verdade ou falso indicando se o elemento está ou não no conjunto. Um efeito mais interessante ocorre quando o argumento é indefinido (*NIL*) no momento da invocação da função. Neste caso o valor do parâmetro é ligado (unificado) pela função *unify* (figura 5.2) ao primeiro elemento do conjunto e a função *items* retorna *true*. Como efeito colateral, o elemento do conjunto é retornado ao ponto de invocação da função *items* uma vez que o argumento está sendo passado por referência (*byRef*). Assim, a invocação da função *items* corresponde a uma consulta feita numa linguagem de programação em lógica.

Usada em conjunto com a instrução de iteração como *for* (figura 5.3), a relação encontrará, em seqüência, cada ligação para a variável, correspondendo a todos os elementos do conjunto.

---

[8] Em Leda um valor pode estar definido ou não. O estado de estar "indefinido" é representado por NIL, um valor polimórfico que significa que ele pode ser atribuído a qualquer tipo de dados.

```
val := NIL;
for aSet.items(val) do
        "Em cada iteração a variável val é unificada a um dos elementos
                do conjunto aSet"
        ... instruções
end;
```

Figura 5.3 - A Consulta Lógica Em Leda

5.4 - Definindo Relações

Para que uma linguagem de programação suporte o paradigma de programação em lógica ela deve incluir mecanismos para descrição de fatos e de regras de inferência, além de um mecanismo de pesquisa.

As caracterísicas da linguagem Leda descritas em 5.3 mostram que os mecanismos de pesquisa das linguagens lógicas podem ser incorporados a uma linguagem orientada a objetos. Leda, contudo, não apresenta nenhum mecanismo para definição explícita de relações entre objetos. As relações são embutidas na definição das classes (vide exemplo da classe *Set* apresentada na figura 5.1), cuja implementação favorece a aplicação dos mecanismos de pesquisa da programação em lógica.

Uma alternativa que se mostra interessante é estender os recursos de Leda por de um mecanismo explícito para definição de relações ordinárias sobre objetos. Ao mesmo tempo, parece igualmente interessante explorar as características da programação em lógica de, sob certo sentido, assemelhar-se a um sistema de banco de dados. Isto inclui permitir que, baseado nos tipos e formas de consultas contidas no programa, o interpretador/compilador defina a melhor forma de estruturar as informações com vistas a favorecer o processo de pesquisa de soluções.

A definição de relações pode ser feita por de um construtor de relações denominado *relation*. Este construtor deve permitir a definição de relações nos moldes da programação em lógica clássica (a exemplo das relações definidas na figura 5.4) e, possivelmente, nos moldes da CLPs (ver 5.2).

```
relation é_Pai_De ( p1, p2: Pessoa);

relation é_Avô_De(p1, p2:Pessoa) :- é_Pai_De(p1,z) & é_Pai_De(z,p2);
```

Figura 5.4 - Duas Relações Clássicas Entre Pessoas

Contudo, além de relacionar objetos, o construtor *relation* pode se espandido de forma a permitir a definição de estados locais à relação. Isto implica a possibilita de definição de variáveis locais e o estabelecimento de restrições invariantes entre as variáveis locais e os parâmetros. A figura 5.5 apresenta um exemplo que define uma relação binária entre dois objetos

do tipo *Neuronodo*[9]. Adicionalmente esta relação define um estado interno, o *peso* da conexão, e duas outras variáveis do tipo *Float* que completam a descrição do estado da relação.

```
relation conecta( n1: Neuronodo, n2: Neuronodo)
                    with(propaga, retroPropaga: Float);
        local peso;
        once: peso <-- aleatórioEntre(-1.0,1.0);
        always: peso <-- peso + Neta * n1.valAtivação * n2.erro;
        always: propaga <-- n1.valAtivação * peso;
end;
```

Figura 5.5 - A Definição De Relações

O construtor de relações guarda um certa similaridade com a noção de classes uma vez que funciona como um gabarito através do qual é possível estabelecer relações entre objetos particulares. Ela é particularmente interessante, também, por possibilitar uma integração com a programação por restrições.

5.5 - O Estabelecimento de Relações

O efetivo estabelecimento da relação[10] entre objetos particulares pode ser feito por meio de um operador *assert*. Considerando a definição da relação *conecta* apresentada na figura 5.5, uma declaração como:

assert: conecta(neurOrig, neurDest, _ , _);

estabelece uma relação de conexão entre os objetos denotados por *neurOrig* e *neurDest*. Da execução desta instrução resulta a criação de um objeto especial (instância da classe *Relationship*) contendo uma variável local *peso* e um conjunto de restrições que estabelece interdependências entre os parâmetros da relação e a variável local. O símbolo "_" utilizado no operador *assert* indica que os parâmetros *propaga* e *retroPropaga* são definidos internamente, quando do estabelecimento da relação.

Para desfazer a conexão entre dois objetos pode-se utilizar o operador *retract*. Uma declaração como:

retract: conecta(neurOrig, neurDest, _ , _);

desfaz a relação de conexão entre os neurônios denotados por *neurOrig* e *neurDest*.

5.6 - A Consulta a Relações

Uma consulta (pesquisa) a relações significa encontrar respostas que sejam derivadas das informações que forem fornecidas em questões ou perguntas. Para isto, à definição da relação *conecta* poderia ser associada uma função de consulta como:

---

[9] A classe *Neuronodo* e a sintaxe das restrições são apresentadas no capítulo 7.

[10] Na clássica prova proposicional de teoremas, estas relações entre objetos correspondem aos axiomas (fatos que se assumem sejam evidentemente verdade ou pelo menos aceitáveis para os propósitos do argumento).

```
function conecta (    byRef neurOrig, neurDest : Neuronodo;
                      byRef prop, retroProp: Float  ) -> relation;
```

que ofereceria funcionalidade de consulta similar à encontrada na linguagem Leda e descrita em 5.3.Um esquema diferente, contudo, parece igualmente interessante como alternativa para implementar funções de consulta.

As funções de consulta, como implementadas em Leda, retornam um valor lógico (*relation*) como resultado da função e retornam os objetos da relação por meio de efeito colateral ("byRef"). O poder destas funções pode ser aumentado se elas também retornassem coleção de tuplas que satisfazem a determinado tipo de consulta. Uma consulta como:

conecta(? , neurDest , ? , _);

resulta numa coleção contendo duplas de valores (Neuronodo, Float) correspondestes a todas as conexões nas quais o objeto denotado por *neurDest* é o segundo parâmetro. Já uma consulta mais simples como:

conecta(_ , neurDest , ? , _);

retorna a coleção dos objetos do tipo Float.

Neste tipo de consulta o símbolo "?" indica qual a informação que se deseja. Por sua vez o símbolo "_" tem uma funcionalidade similar ao do objeto *NIL* descrito em 5.3, indicando a unificação com qualquer objeto.

5.7 - Considerações

As inovações introduzidas pela orientação a objetos geraram uma pequena revolução no processo de desenvolvimento de software. Contudo, perspectivas como a programação por restrições e a programação em lógica igualmente introduziram inovações. Reunir as várias perspectivas em torno de uma estrutura única é um sonho perseguido por alguns autores. Mas esta integração não é uma atividade trivial.

Neste capítulo são introduzidas algumas idéias de como integrar as perspectivas de programação em lógica, de programação por restrições e de programação orientada a objetos. Em particular é descrito o mecanismo chamado *relation* que introduz a noção de estado em uma relação. Este mecanismo é utilizado nos capítulos 7 e 8 para implementar o modelo back-propagation de redes neurais e um diagrama PERT.

6 - REDES NEURAIS

Nos anos 50 duas abordagens distintas foram estabelecidas pela inteligência artificial: os sistemas simbólicos e os sistemas conexionistas, em especial as redes neurais. Ambas procuravam modelar processos do cérebro tais como a representação de conhecimento e o processo de raciocínio.

Divergindo dos sistemas simbólicos, que viam os processos do cérebro como "caixas pretas", os modelos computacionais conhecidos como redes neurais foram construídos baseados na forte interconexão de elementos processadores. Inspirados pelos modelos bilógicos do sistema nervoso, as redes neurais procuram ser análogas às redes de neurônios do cérebro. Neste modelo, elementos processadores (os neurônios) foram interconectados por meio de ligações valoradas (as sinapses) de forma a compor uma rede.

Nos anos 70 as pesquisas involvendo redes neurais foram parcialmente abandonadas em favor da abordagem adotada nos sistemas simbólicos, mas voltaram com força nos anos 80. Muitos pesquisadores continuaram a explorar aspectos dos modelos com inspiração neurológica e/ou psicológica. Estes estudos conduziram a novos modelos, assim como conduziram a soluções para alguns dos problemas encontrados em modelos anteriores.

Características gerais das redes neurais são apresentadas nos tópicos seguintes. Em particular é descrito o modelo *back-propagation* para o qual, no capítulo 7, será apresentada uma implementação utilizando os mecanismos descritos nos capítulos 4 e 5.

6.1 - Redes Neurais versus Arquiteturas Tradicionais

Os modelos computacionais clássicos baseados na arquitetura de Von Neumann, caracterizam-se pela utilização de uma única UCP (unidade central de processamento) poderosa capaz de realizar centenas de operações e pela execução sequencial de todas as computações.

As redes neurais, por outro lado, são sistemas paralelos baseados no arranjamento denso de interconexão de processadores bastante simples. Numa rede neural cada unidade de processamento é capaz de realizar alguns poucos cálculos. Seu poder de processamento é dado pela quantidade de elementos processadores (os neuronodos) e pelo número de interconexões modificadas por unidade de tempo.

Os modelos de redes neurais eliminam muitos detalhes biológicos, mas retém parte significativa da estrutura observada no cérebro. O nível apropriado de abstração é baseado no objetivo do modelador.

6.2 - Aplicações Típicas

A aplicação de redes neurais para a solução de um problema não requer desenvolvimento de algoritmos que sejam específicos ao problema a ser tratado. Isto sugere que o tempo e o esforço humano para obtenção de uma solução podem ser reduzidos. Há, contudo,

alguns problemas associados ao uso de redes neurais. Em particular, o risco de se necessitar um tempo relativamente grande para efetivar o treinamento dos padrões e a possibilidade de não haver convergência do processo de aprendizagem.

O processo de construção de uma aplicação utilizando-se redes neurais é relativamente complexo. O uso de redes neurais implica uma série de decisões de projeto que inclui: escolha do modelo de rede neurais (back-propagation, kohonen, etc) adequado à aplicação; topologia da rede (número de neuronodos, forma de interconecção, etc); parâmetros internos (pesos iniciais para as interconecções, taxas de aprendizagem, etc); definição dos padrões de entrada e saída; e seleção do conjunto de padrões de entrada e de saída a serem oferecidos para treinamento.

O espaço de possíveis aplicações é, contudo, impressionantemente grande. As principais aplicações para redes neurais em geral estão relacionadas a problemas que seriam intratáveis ou de difícil solução por métodos tradicionais. As redes neurais normalmente superam os sistemas clássicos em solução de problemas que involvam situações como: mapeamentos (a exemplo de sintomas em diagnóstico); complemento (a partir de partes obter o todo); e classificação de padrões (a exemplo de imagens de células de sangue).

6.3 - O Funcionamento das Redes Neurais

As redes neurais não são "programáveis"; elas aprendem por exemplos.

A operacionalidade de um rede neural pressupõe que ela inicialmente passe por um processo de treinamento. Nesta fase um conjunto de exemplos é sistematica e repetidamente apresentado à rede neural de forma a permitir um ajuste gradual de seus parâmetros internos.

Se completado com sucesso o treinamento, numa segunda fase a rede neural está apta a reconhecer os padrões que lhe forem apresentados. Quando não é possível um reconhecimento preciso, em geral ela oferece uma aproximação (por interpolação) para o resultado.

6.3.1 - As Formas de Aprendizagem

Os modelos de redes neurais apresentam dois tipos básicos de treinamento: o treinamento supervisionado e o não supervisionado.

No treinamento supervisionado, a resposta alvo (objetivo) é apresentada juntamente com o padrão que se deseja ensinar. O treinamento, então, pode ser visto como um processo de construção de um função de mapeamento entre o conjunto de padrões de entrada e o conjunto de padrões de saida (os objetivos). O exemplo mais clássico de treinamento supervisionado é o implementado pelas redes back-Propagation.

Divergindo do treinamento supervisionado, modelos com o das Redes de Kohonen implementam um estilo não supervisionado de aprendizagem. Neste modelo o ajuste de pesos das conexões é feito sem o conhecimento da resposta alvo. A rede procura, então, classificar os padrões de entrada em categorias similares.

6.3.2 - O Sincronismo

A operação de uma rede neural, tanto na fase de treinamento quanto na de reconhecimento, é baseada no cômputo e propagação de valores de ativação dos neuronodos. Na maior parte das implementações de redes neurais este cômputo é realizado de modo *síncrono*, ou seja, cada neuronodo é alterado em cada passo de simulação. O valor de saída do passo anterior é utilizado para calcular o novo valor do neuronodo.

Algumas sistemas, a exemplo do RCS (Rochester Connectionist Simulator) [Lutzy 93] expandem esta possibilidade ao oferecer outros modos para cômputo de valores de um neurodos: modo assíncrono e modo retardado. No modo *assíncrono* todos ou apenas alguns poucos neuronodos tem seu valor alterado em cada passo de simulação. Já o modo *retardado* possibilita o estabelecimento de um tipo especial de conexão que define um tempo de retardo. Com isto a operação de alteração é mantida suspensa por algum tempo.

6.4 - O Modelo Back-propagation

Dentre os modelos de redes neurais, o mais difundido e largamente aplicado é o de retro-propagação de erros (back-propagation). Back-propagation é uma das redes mais fáceis de se compreender por basear-se num conceito relativamente simples: se a rede oferece uma resposta errada em resposta a um padrão de entrada, então os pesos das conexões são alteradas de forma a reduzir o erro e propiciar um resposta mais próxima da real quando do re-oferecimento deste mesmo padrão.

A descrição da rede back-propagation e as equações que seguem foram retiradas de [Dayhoff 90].

6.4.1 - Uma Unidade Típida

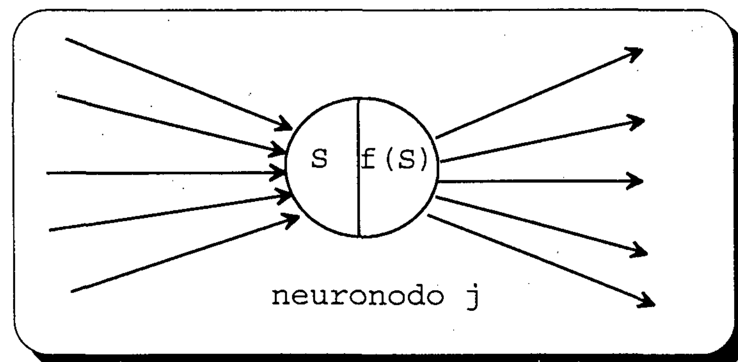


Figura 6.1 - Um Neuronodo Típico

Nas redes back-propagation uma típica unidade de processamento (neuronodo) toma a forma apresentada na figura 6.1. Um neuronodo recebe múltiplas entradas vindas de outras unidades. Cada conexão tem um peso associado que é utilizado pelo neuronodo para calcular seu

valor de ativação que será, então, propagado através das conexões de saída para outros neuronodos.

Para calculo do valor de ativação de um neuronodo *j* inicialmente é realizada uma soma ponderada dos valores de ativação dos neuronodos anteriores (a_i) e dos pesos das conexões com estes neuronodos (w_{ji}).

$$S = \sum_{i=1}^{n} a_i w_{ji} \quad \text{[equação 6.1]}$$

A esta soma aplica-se uma função de limiar não linear, cujo resultado passa a ser o valor de ativação do neuronodo. Uma função comumentemente utilizada é a função sigmóide:

$$f(x) = \frac{1}{1+e^{-x}}$$

de forma que o valor de ativação de um neuronodo é calculado por:

$$A = f(S) = \frac{1}{1+e^{-S}} \quad \text{[equação 6.2]}$$

6.4.2 - A Topologia

Os neuronodos geralmente são agrupados em camadas as quais são interconectadas de forma a comporem a rede back-propagation.

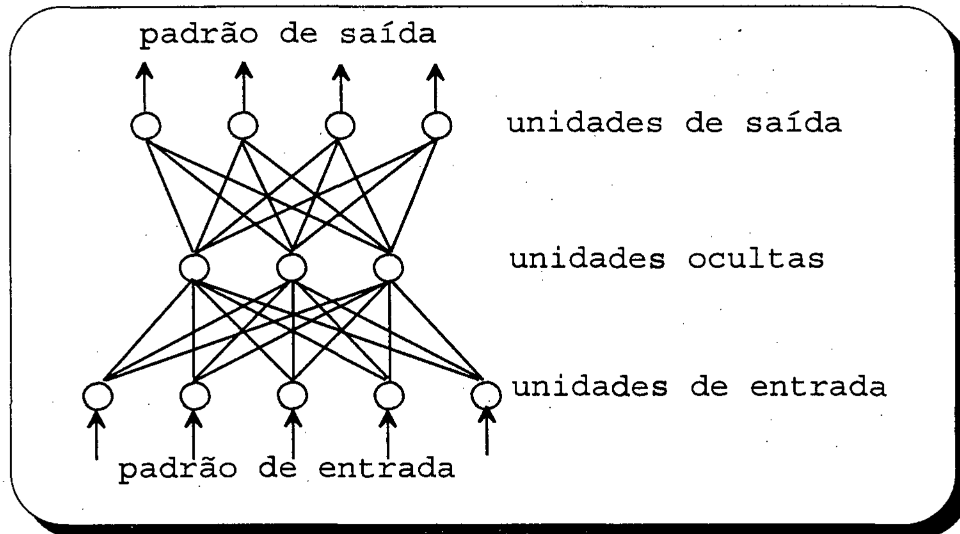


Figura 6.2 - Uma Rede Back-Propagation

Tipicamente as redes back-propagation trabalham com três ou mais camadas de unidades de processamento. A figura 6.2 mostra um rede back-propagation típica com 3 camadas. A camada mais a esquerda (camada de entrada) contém os únicos neuronodos que recebe estímulos externos. Os valores de ativação dos neuronodos da camada mais à direita (camada de saída) representam a saída calculada pela rede. As demais camadas (camadas ocultas) contém neuronodos cuja distribuição proporciona diferentes topologias para a rede back-propagation.

Nas redes back-propagation as camadas geralmente são completamente conectadas, isto é, cada neuronodo é conectado com todos os neuronodos das camadas anterior e sucessora. Os neuronodos não são, contudo, conectados a outros neuronodos da mesma camada. Apesar de grande parte das aplicações fazerem uso deste tipo de rede back-propagation, há variantes nais quais as camadas não estão necesariamente completamente conectadas.

6.4.3 - A Propagação de Valores de Ativação

A propagação de valores é iniciada quando um novo padrão de entrada (um vetor de números reais) é apresentado para a rede. Cada unidade da camada de entrada toma um destes valores como seu valor de ativação. Os valores passam, então, a ser propagados conforme descrito no ítem 6.4.1, até que os valores de ativação dos neuronodos da camada de saída sejam calculados.

Os neuronodos da camada de entrada apresentam um comportamento especial pois eles não executam a soma ponderada; eles simplesmente assumem os valores do padrão de entrada fornecido à rede.

Algumas topologias de rede back-propagation utilizam um neuronodo adicional em cada camada, à exceção da camada de saída. Estas unidades de ajuste tem valor de ativação constante igual a 1 e são conectadas a todos os neuronodos da camada seguinte. Sua função é a de prover um termo constante na soma ponderada, cujo resultado por vezes conduz a uma melhoria na convergência da rede.

6.4.4 - A Retro-Propagação (correção de erros)

Completado o passo de propagação dos valores de ativação, cada neuronodo da camada de saída terá produzido um número real, seu valor de ativação, que será utilizado para compor o padrão de saída para um dado padrão de entrada.

A fase de treinamento das redes back-propagation é baseada numa aprendizagem supervisionada. Assim, durante a fase de treinamento dois vetores são fornecidos: um para o padrão de entrada e outro para o padrão de saída esperado. O cálculo da diferença entre o padrão de saída esperado e o padrão de saída calculado pela rede permite estabelecer um valor de erro para cada unidade da camada de saída.

Os erros observados nos neuronodos de saída são progressivamente retro-propagados para os neuronodos anteriores. Durante esta retro-propagação os pesos das conexões entre os neuronodos vão sendo ajustados de forma que quando da re-apresentação de um padrão de entrada o erro seja menor que o atual.

Para os neuronodos da camada de saída, o erro (denotado por δ) é computado pela expressão:

$$\delta_j = (t_j - a_j) f'(S_j) \quad \text{[equação 6.3]}$$

onde:

t_j = o valor objetivo (padrão esperado) para o neuronodo *j*
a_j = o valor calculado para o neuronodo *j*
f'(x) = derivada da função sigmóide *f*
S_j = soma ponderada das entradas do neuronodo *j*

No caso dos neuronodos das camadas ocultas, o cálculo do erro é um pouco mais complicado, sendo obtido pela equação:

$$\delta_j = \left[\sum_k \delta_k w_{kj} \right] f'(S_j) \quad \text{[equação 6.4]}$$

ou seja, a soma ponderada é tomada entre os valores δ de todas os neuronodos que recebem saída do neuronodo *j*.

Cada peso de interconexão de dois neuronodos é então ajustado em valor determinado pela equação:

$$\Delta w_{ji} = \eta \delta_j a_i \quad \text{[equação 6.5]}$$

ou seja, a correção do peso é proporcional ao valor δ do neuronodo de destino da interconexão, ao valor de ativação do neuronodo de origem da interconexão e a um valor η que é a taxa de aprendizagem (normalmente um valor entre 0.25 e 0.75).

6.5 - A Implementação de Redes Neurais

Várias e diferentes técnicas têm sido utilizadas para implementação de redes neurais, incluindo pesquisas em arquiteturas fortemente paralelas para a construção de grandes redes neurais. Contudo, a grande parte das implementações usam simuladores sobre máquinas com um único processador.

6.5.1 - Sistemas Simuladores

Há hoje um número razoável de pacotes de simulação disponíveis para projeto e avaliação crítica de hípoteses alternativas relacionadas com modelos de redes neurais. Estes simuladores[11] apresentam uma funcionalidade similar, sendo compostos basicamente por um núcleo simulador e uma interface gráfica para operacionalização do ambiente.

O núcleo simulador destes sistemas incluem mecanismos para criação e conexão de células, assim como para definir as funções de ativação e as regras de aprendizagem. Sistemas como o UCLA-SFINX [Mesrobian 92] possibilita a construção de rede neurais estruturalmente irregulares por meio da definição de cada neuronodo e correspondentes conexões. Já redes muito grandes com padrão regular de conectividade são especificadas utilizando-se construções do tipo array.

Para especificação do comportamento da rede neural, estes sistemas, em geral, oferecem um conjunto de funções predefinidas, mas também permitem a definição de novas

[11] Vários simuladores de redes neurais são descritos em [Mesrobian 92] (Pablo, Boss, Genesis, Mirrors/II, SFINX) e em [Lutzy 93] (PlaNet, Pygmalion, RCS-Rochester Connectionist Simulator, SNNS-Stuttgart Neural Net Simulator).

operações e procedimentos para aplicações especiais. Estas especificações ou são escritas diretamente na mesma linguagem de implementação do simuladores (na maior parte dos casos linguagens procedurais clássicas) ou em uma linguagem de especificação de rede que normalmente são bastante restritas no que tange a expressividade.

A que se notar que a maior parte destes simuladores não oferecem mecanismos para modificação dinâmica, em tempo de execução, da topologia da rede neural [Lutzi 93]. No caso de haver mudanças é necessário reprojetá-la completamente.

6.5.2 - "Frameworks"

Outro tipo de implementação de simuladores de redes neurais toma a forma de estruturas de classes ("frameworks"). Utilizando as características de programação por extensão proporcionadas pela programação orientada a objetos, ambientes como smallBrain [Krishnan] oferecem um conjunto pré-implementado de classes (e alguns modelos de redes neurais) que podem ser especializadas para a implementação de modelos particulares de redes neurais.

Apesar de bastante flexível, esta alternativa apresenta a desvantagem de exigir que o projetista de redes neurais conheça o ambiente e a linguagem computacional na qual foi implementada a estrutura de classes.

6.5.3 - Os Elementos Básicos

A implementação de redes neurais normalmente envolve a definição e programação de quatro tipos de elementos: os neuronodos, as conexões, as camadas e as redes neurais propriamente ditas.

A unidade básica de uma rede neural é o **neuronodo** (uma abstração computacional para o neurônio). Os neuronodos podem ser vistos como elementos processadores simples que, no caso mais geral, tomam uma série de valores de entrada e computam um valor de saída. A busca dos valores de entrada e a propagação do valor de saída são baseadas nas **interconexões** entre os neuronodos. Estas interconexões são valoradas de forma que o neuronodo aumenta a contribuição de alguns neuronodos e ignora (ou reduz) a contribuição de outros.

Em praticamente todos os modelos de redes neurais, os neuronodos são tomados em conjuntos: as **camadas**. Estas camadas assumem diferentes formas, dependendo do modelo que se está implementando. Nas redes back-propagation (ver item 6.4) as camadas são lineares, sendo implementadas com o auxílio de "arrays". Já redes como as de Kohonen [Dayhoff 90] incluem uma camada bidimensional (um mapa auto-organizativo).

As **redes neurais** são, então, vistas como um conjunto de camadas. Dependendo do modelo elas podem ser compostas por várias camadas, a exemplo das redes back-propagation que é composta de no mínimo três camadas.

6.6 - A Neurociência Computacional

Os temas tratados nos tópicos anteriores deste capítulos concernem principalmente à utilização de redes neurais como ferramenta para modelagem e resolução de problemas. Contudo, outro ponto de vista que tem merecido a atenção de diversos pesquisadores é o estudo da "neurociência computacional" [Mesrobian 92].

Os sistemas neuronais computacionais tem sido utilizados para modelar propriedades dinâmicas de interações neuronais por representação dos processos biofísicos em vários níveis de abstração. À semelhança de outros fenômenos físicos, estes níveis de abstração incluem situações que fazem uso de modelos contínuos (como circuitos elétricos) e de modelos computacionais de redes neurais com padrão irregular de conexão e modificáveis (adaptáveis). O nível de abstração a ser utilizado é completamente dependente do objetivo do modelador.

Conforme notado por Mesrobian de Skrypek [Mesrobian 92], os avanços em neurociência, incluindo o estudo do cérebro, motivam a necessidade por ambientes simuladores de propósito geral capazes de manipular todas as possíveis abstrações de redes neurais. Isto inclue a definição de linguagens de alto nível capazes de descrever tanto neuronodos individuais como redes de neuronodos.

6.7 - Considerações

Boa parte dos sistemas de simulação de redes neurais descritos em 6.5.1 oferecem recursos significativos no sentido de construir grandes redes neurais com padrão regular de conectividade. Contudo, eles apresentam deficiências, em maior ou menor grau, em tópicos relacionados com: padrões irregulares de conexão entre neuronodos; recursos para descrição em alto nível da estrutura e do comportamento da rede neural; e modificabilidade dinâmica da arquitetura da rede neural.

A reunião dos mecanismos associados às diferentes perspectivas de desenvolvimento de software descritas nos capítulos anteriores mostra-se extremamente promissora no sentido de oferecer um ferramenta para descrição e implementação de redes neurais. Esta perspectiva é explorada no capítulo 7 através de uma implementação não usual do modelo back-propagation de redes neurais, a qual mostra a potencialidade da associação destes mecanismos.

7 - UMA IMPLEMENTAÇÃO DO MODELO BACK-PROPAGATION DE REDES NEURAIS

A implementação do modelo back-propagation de redes neurais descrita neste capítulo é baseada numa linguagem hipotética, um misto de programação orientada a objetos, programação por restricões e programação em lógica. Não é inteção oferecer uma implementação mais eficiente em termos de velocidade, mas sim buscar explorar os recursos de cada um dos paradigmas no sentido de oferecer uma linguagem de alta flexibilidade e alto poder de expressividade para, entre outras coisas, implementar redes neurais.

Vários termos e construções desta linguagem hipotética foram mantidos em língua inglesa por refletirem jargões computacionais.

7.1 - A Linguagem Hipotética

7.1.1 - Características da Linguagem

A programação orientada a objetos oferece à linguagem hipotética seus mecanismos de estruturação de aplicações. Um conjunto razoável de variantes são implementadas nas linguagens orientadas a objetos existentes, de sorte que algumas opções foram tomadas. Estas opções incluem:

a) classes (em oposição aos protótipos): ambos os mecanismos são igualmente expressivos. O mecanismo de classes, contudo, é mais conhecido e difundido;

b) checagem opcional e incremental de tipos (em oposição à checagem estática): este tipo de abordagem oferece um misto de programação exploratória (característica das linguagens não tipadas) e programação de sistemas (características das linguagens fortemente tipadas);

c) multi-método (em oposição ao mecanismo tradicional de ligação de mensagens e métodos): o mecanismo de multi-métodos é particularmente importante em linguagens de programação por restrições. Contudo, mesmo as linguagens orientadas a objetos clássicas se beneficiam sobremaneira com este tipo de abordagem;

d) hierarquia única de tipos e de implementação: reflete apenas uma simplificação da linguagem, a qual pode ser expandida de forma a diferenciar as hierarquias;

e) herança simples: à semelhança do ítem anterior, é apenas um simplificação;

Da programação por restrições a linguagem hipotética faz uso do mecanismo de estabelecimento de dependências entre objetos e do mecanismo de satisfação automática de restrições. Em particular é utilizado o conceito de construtores descritos em 4.6.

Por último, a linguagem hipotética busca na programação em lógica uma forma de estabelecer e manter relações ordinárias entre objetos, outras que não as relações de dependência

estabelecidas pela restricões. Isto é obtido por meio de um tipo particular de construção sintática ("relation"), e de dois operadores de administração de relações encontrados em linguagens de programação em lógica: "assert" e "retract", os quais são apresentados no capítulo 5.

7.1.2 - Biblioteca de classes primitivas

Por basear-se na programação orientada a objetos, a linguagem hipotética pressupõe a disponibilidade de uma biblioteca de classes primitivas nos moldes das hierarquias embutidas em linguagens como Eiffel e Smalltalk. Um fragmento desta hierarquia, que não é próprio de nenhuma das linguagens citadas, é apresentado na figura 7.1.

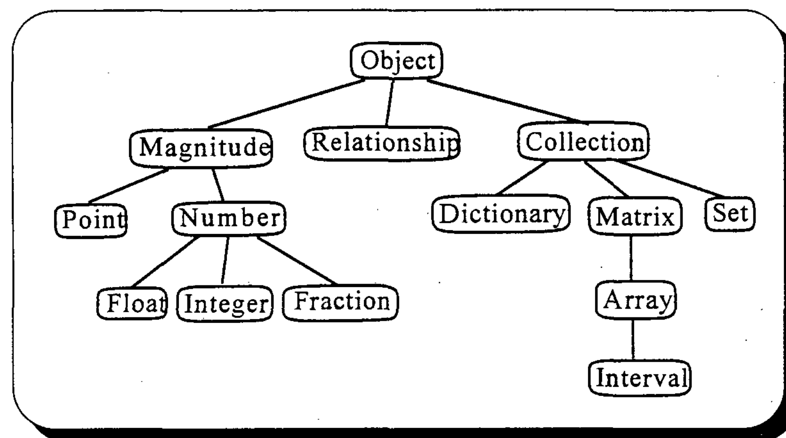


Figura 7.1 - Biblioteca De Classes Primitivas

Sobre estas classes é pressuposta a existência de um sistema de satisfação de restrições nos moldes descritos no capítulo 4.

7.1.3 - Algumas Construções Sintáticas

Algumas construções sintáticas não usuais que aparecem nas figuras deste capítulo são descritas nos tópicos seguintes.

7.1.3.1 - Variáveis Públicas

Duas das características da orientação a objetos são o encapsulamento e a proteção da estrutura interna dos objetos. Isto significa que os elementos que compõem um objeto são acessíveis somente via a interface (métodos) definida pelo usuário. Contudo, é freqüente a existência de variáveis de instância cujo comportamento se restrige a receber consultas e atribuições de objetos, forçando o programador a implementar métodos triviais de acesso e atribuição de valores a estas variáveis.

Linguagens como Cecil [Chambers 92, 93] oferecem recursos de acesso a este tipo de campos via definição automática de dois métodos especiais tipo *get* e *put*[12] para as variáveis de instância declaradas como *field*. Em linguagens de programação por restrições como

Kaleidoscope este mesmo comportamento é obtido pelo inclusão de um único construtor, a exemplo dos citados em 4.6.

Esta segunda abordagem é adotada na linguagem hipotética. Assim, a declaração de uma variável de instância como *public* (em oposição às variáveis declaradas como *local*) implica na definição automática de um construtor através do qual o valor de ativação pode fluir de e para um objeto. Estes construtores podem ser explicitamente implementados caso se deseje um comportamento diferente.

Declarações como:

public valAtivação

encontradas na figura 7.3 possibilitam a escrita de instruções do tipo:

n.valAtivação <– 5;
n.erro <– (valor - n.valAtivação) * derivadaSigmóide(n.soma);

A execução da primeira instrução resulta no estabelecimento do objeto 5 como valor de ativação do neuronodo denotado por *n*. Já a segunda instrução possibilita a consulta ao valor de ativação do objeto denotado por *n*.

7.1.3.2 - Restrições Invariantes

A cláusula *initially* inserida na definição de classes (ver figura 7.3) define restrições invariantes, que devem ser verificadas durante toda a vida das instâncias da classe onde elas são definidas. As restrições, em si, são descritas em 7.5.

7.1.3.3 - Contruções *Obj.Msg*

Boa parte das linguagens orientadas a objetos utilizam um ponto para separar o objeto e a mensagem que está sendo enviada para este objeto. Tratando-se de multi-métodos, contudo, este tipo de construção já não apresenta mais o mesmo sentido pois não há mais um objeto primordial para o qual seja enviada a mensagem. Nesta perspectiva todos os objetos envolvidos na mensagem tem igual importância. A construção *obj.msg* é mantida apenas como um "açucar sintático", ou seja, uma forma diferente de escrever *msg(obj)*. De forma similar, uma construção do tipo *obj1.msg(obj2)* é equivalente a *msg(obj1, obj2)*.

Na linguagem hipotética, as mensagens enviadas a objetos podem tanto representar invoção de métodos como de construtores.

7.1.3.4 - Estruturas Particulares De Iteração

Uma expressão do tipo:

a.do[x| s <– s + x];

---

[12] Outras linguagens como Self e Trellis também utilizam uma abordagem similar ao prover um mecanismo de acesso à variáveis de instância somente via métodos de acesso *set/get*.

pode ser lida como: para cada elemento da coleção *a* faça "s <– s + x", sendo que a variável *x* denota cada um destes elementos a seu tempo. Este tipo de construção é encontrada em linguagens como Smalltalk e assemelha-se muito à notação utilizada em teoria de conjuntos para definir um conjunto por intensão, como em X={x| x > 5} que define o conjunto *X* como sendo dos elementos *x* tal que "x > 5".

7.1.3.5 - Construtor De Objetos Da Classe *Interval*

A exemplo do operador "@" que toma dois valores numéricos e resulta na construção de um objeto da classe *Point*, o operador ".." tomas estes valores e retorna um objeto da classe *Interval* que é subclasse de *Collection.*

Sendo assim, uma construção como:

(1..n).do[i| ...];

apresenta comportamente similar ao descrito em 7.1.3.4, ou seja, a variável *i* assume cada um dos valores contidos no intervalo definido por (1..n).

7.2 - As Classes

7.2.1 - Estrutura de Classes para Implementação de Redes Neurais

Considerando os elementos básicos que constituem uma rede neural (descritos em 6.5.3) e a noção de "frameworks" (ver 6.5.2), a hierarquia da figura 7.1 pode ser expandida para acomodar classes específicas para a implementação de redes neurais. A figura 7.2 apresenta uma possível estrutura para estas classes. Nesta estrutura, as classes tem a seguinte funcionalidade:

a) *RedeNeural*: uma classe abstrata que descreve as características gerais de todas os modelos de redes neurais. Suas subclasses refletem a implementação de modelos específicos de redes neurais, tais como *Hopfield*, *Kohonen* e *BackPropagation;*

b) *Neuronodo*: uma classe abstrata que descreve as caracterísicas gerais de um neuronodo. Suas subclasses especializam a estrutura e o comportamento de neuronodos particulares a um modelo de rede neural. Na figura 7.2 a classe *Neuronodo* é especializa em três subclasses específicas para a implementação de redes back-propagation: *BPNeurEntrada*, *BPNeurOculto* e *BPNeurSaída*;

c) *Conexão*: uma classe abstrata que decreve as características gerais dos possíveis tipos de conexões entre neuronodos. Suas subclasses implementam conexões particulares a algum modelo de redes neurais;

d) *Camada*: uma classe abstrata que decreve as características gerais das camadas de neuronodos que compõem uma rede neural. Suas subclasses implementam camadas particulares a algum modelo de redes neurais, a exemplo das camadas lineares das redes back-propagation e das camadas bidimensionais das redes de Kohonen.

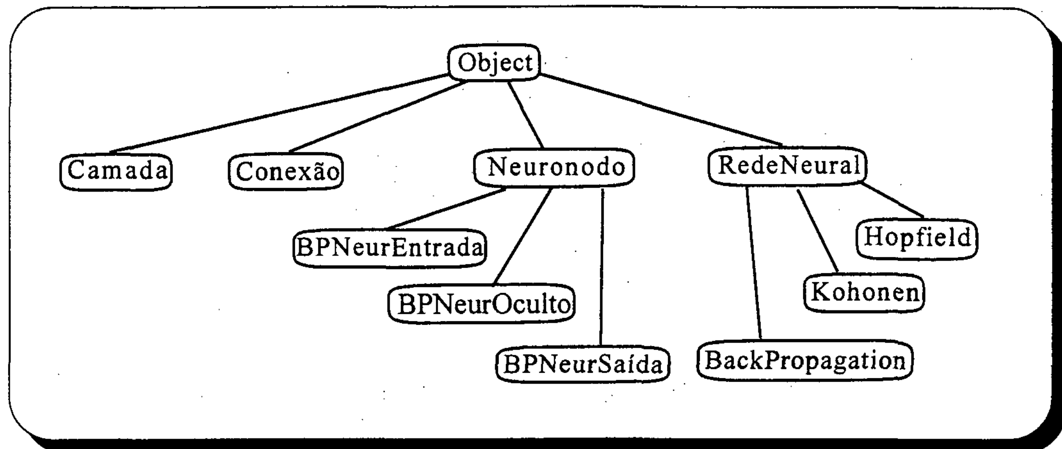


Figura 7.2 - Estrutura Para Implementação De Redes Neurais

7.2.2 - As Classes para Implementação de Redes Back-propagation

Apesar de terem sido incluídas quatro classes abstratas na descrição da estrutura para implementação de redes neurais apresentada na figura 7.2, a implementação da rede back-propagation prescindem das classes *Camada* e *Conexão*.

Numa rede neural, as camada são elementos agregadores de neuronodos, sendo utilizadas para estruturação (definição da topologia) da rede neural. Em geral sobre elas são também definidas as relações de prescedência entre os neuronodos. Esta relação de prescedência é que estabelece o sentido de propagação e retro-propagação de valores no modelo de back-propagation.

Na implementação aqui proposta, contudo, as relações de precedência e, por conseqüência a topologia da rede, são definidas diretamente sobre os neuronodos através da relação *conecta* descrita em 7.4. Assim, a classe *Camada* apresentaria apenas o papel de agregadora de neuronodos, papel este que pode exercido por alguma subclasse de *Collection,* tal como *Set* ou *Array.* Motivo similar torna desnecessário o uso da classe *Conexão*. A descrição das demais classes é apresentada na figura 7.3

```
class Neuronodo  subclassOf Objeto
        public  valAtivação;
end;

class BPNeurEntrada subclassOf Neuronodo;
end;

class BPNeurOculto  subclassOf Neuronodo;
        public soma, erro;
        initially
                always: soma <-- somatório( conecta( _ , self , ? , _ ) );
                always: valAtivação <-- sigmóide(soma);
                always: erro <-- somatório( conecta( self , _ , _ , ? ) ) *
                                derivadaDaSigmóide(soma);
end;

class BPNeurSaída subclassOf Neuronodo;
        public soma, erro;
        initially
                always: soma <-- somatório( conecta( _ , self , ? , _ ) );
                always: valAtivação <-- sigmóide(soma);
end;

class BackPropagation  subclassOf RedeNeural;
        local   neuronodosEntrada,
                neuronodosSaída: Array;
end;
```

Figura 7.3 - Descrição Das Classes Para Implementação De Rede Back-Propagation

Instâncias da classe *BPNeurEntrada* são utilizados para compor a camada de entrada de uma rede back-propagation. Conforme visto em 6.4.3, estes neuronodos limitam-se a assumir os valores do padrão de entrada fornecido à rede. Esta operação é desencadeada pela restrição:

once: bp.entrada = padraoEnt;

inserida nos procedimentos *reconhece* e *treina* descritos na figura 7.7.

Já instâncias da classe *BPNeurOculto* compõem as camadas intermediárias (ocultas) de uma rede back-propagation. Além do valor de ativação, estes neuronodos incluem dois outros valores correspondestes à *soma* ponderada dos valores de ativação dos neuronodos anteriores (calculada pela equação 6.1) e um valor de *erro* utilizado na atualização dos pesos das conexões (calculado pela equação 6.4). As restrições inseridas na secção *initially* da classe *BPNeurOculto* refletem diretamente as equações 6.1, 6.2 e 6.4, respectivamente.

Por último, instâncias da classe *BPNeurSaída* são utilizados para compor a camada de saída de uma rede back-propagation. Instâncias desta classe tem comportamento muito similar ao apresentado por instâncias da classe *BPNeurOculto*, à excessão do mecanismo de cálculo do

erro. Para estas instâncias o erro é calculado pela equação 6.3, a qual depende do padrão de saída esperado para uma dada entrada. Este padrão é repassado à rede neural por meio da restrição:

once: bp.objetivo = padrãoSai;

que é ativada quando da execução do procedimento *treina* inserido na figura 7.7.

Além das subclasses de *Neuronodo*, a figura 7.3 inclue a definição da classe *BackPropagation*, a qual é composta por duas variáveis de instância que guardam referências aos neuronodos de entrada e de saída. Estas coleções de neuronodos são definidas por execução do procedimento *montaRede* (ver figura 7.6) e manipulados pelos construtores apresentados na figura 7.10.

7.3 - As Funções/Procedimentos

Na linguagem hipotética as funções e procedimentos são implementadas na forma de multi-métodos. Sendo assim, os métodos não são associados a uma classe particular, mas a todas as classes envolvidas na tipagem dos parâmetros. Isto significa que o ambiente de desenvolvimento tem papel crucial na classificação e localização dos métodos associados a cada classe.

```
sigmóide(x: Number) -> Real;
        "Retorna a função sigmóide no ponto x"
        return 1 / (1 + exp(x));
end;

derivadaDaSigmóide(x: Number) -> Real;
        "Retorna a derivada da função sigmóide no ponto x"
        return ... ;
end;

somatório(a:Collection) -> Magnitude;
        "Retorna a soma dos elementos da coleção a"
        local s;
        s <- 0;
        a.do[elem| s <- s + elem];
        return s;
end;
```

Figura 7.4 - Funções Sobre Classes Primitivas

Os métodos estendem a clássica perspectiva imperativa ao permitir a inclusão de declarações que expressam restrições, assim como estabelecimento de fatos e consultas lógicas. Exemplos de uso de restrições aparecem nos procedimentos *reconhece* e *treina* inseridos na figura 7.7. Por sua vez, na figura 7.5 aparece um exemplo de estabelecimento de relação, neste caso a relação *conecta* sobre instâncias da classe *Neuronodo*.

```
conectaNeuronodos( orig, dest : Array);
        "Procedimento de conexão completa entre os neuronodos
                do array orig com os neuronodos do array dest"
        orig.do[neurOrig|
                dest.do[neurDest|
                        assert: conecta( neurOrig, neurDest, _ , _ );
                ];
        ];
end;
```

Figura 7.5 - Conexão Completa Entre Camadas De Neuronodos

```
montaRede( bp: BackPropagation , n: Array);
        "Procedimento de montagem da Rede Back Propagation"
        local   orig, dest;
        bp.neuronodosEntrada <– new(Array,  n.first);
        (1..n.first).do[i|
                bp.neuronodosEntrada.atPut(i, new(BPNeurEntrada))];
        orig<– bp.neuronodosEntrada;
        (2..(n.size-1)).do[j|
                dest <– new(Array, n.at(j));
                (1..(n.at(j))).do[i|
                        dest.atPut(i, new(BPNeurOculto))];
                conectaNeuronodos( orig, dest);
                orig <– dest;
        ];
        bp.neuronodosSaída <– new(Array, n.last);
        (1..n.last).do[i|
                bp.neuronodosSaída.atPut(i, new(BPNeurSaída))];
        conectaNeuronodos( orig, bp.neuronodosSaída);
end;
```

Figura 7.6 - Procedimentos De Montagem De Uma Rede Back-Propagation

```
reconhece( bp: BackPropagation , padrãoEnt: Array) -> Array;
        "Procedimento de reconhecimento do padrao padrãoEnt"
        local sai: Array;
        once: bp.entrada = padrãoEnt;
        once bp.saída = sai;
        return sai;
end;

treina( bp: BackPropagation , padrãoEnt, padrãoSai: Array);
        "Procedimento para treinamento do padrãoEnt, tendo
                padrãoSai como valor esperado para a saída"
        once: bp.entrada = padrãoEnt;
        once: bp.objetivo = padrãoSai;
end;
```

Figura 7.7 - Procedimentos De Reconhecimento E Treinamento

7.4 - As Relações

O mecanismo de definição de relações é descrito em 5.4. Particularmente para implementação das redes back-propagation é definida a relação *conecta*, conforme apresentado na figura 7.8. A duplicidade de definição desta relação deve-se a distinção existente entre a conexão de um neuronodo da camada de entrada com o neuronodo da segunda camada e as demais conexões. No primeiro caso não há valor de retro-propagação.

```
relation conecta( n1: BPNeurEntrada, n2: BPNeurOculto+BPNeurSaída;
                                        propaga: Float, _ );
        local peso;
        once: peso <- aleatórioEntre(-1.0,1.0);
        always: peso <- peso + Neta * n1.valAtivação * n2.erro;
        always: propaga <- n1.valAtivação * peso;
end;

relation conecta( n1: BPNeurOculto, n2: BPNeurOculto+BPNeurSaída;
                                        propaga, retroPropaga: Float);
        local peso;
        once: peso <- aleatórioEntre(-1.0,1.0);
        always: peso <- peso + Neta * n1.valAtivação * n2.erro;
        always: propaga <- n1.valAtivação * peso;
        always: retroPropaga <- n2.erro * peso;
end;
```

Figura 7.8 - Descrição Das Relações

O estabelecimento de conexão entre dois neuronodos implica da definição de um *peso* para a conexão, o qual é inicializado por ação da restrição:

once: peso <– aleatórioEntre(-1.0,1.0);

inserida na figura 7.8. Posteriormente o *peso* da conexão é alterado, segundo a equação 6.5, cada vez que o *erro* do neuronodo de destino da conexão for alterado. Este comportamento é descrito pela restrição:

always: peso <– peso + Neta * n1.valAtivação * n2.erro;

Além do peso da conexão, a relação *conecta* define outros dois valores por meio das restrições:

always: propaga <– n1.valAtivação * peso;
always: retroPropaga <– n2.erro * peso;

Estes valores são utilizados para cálculos das somas ponderadas descritas pelas equações 6.1 e 6.4, e inseridas na figura 7.3 quando da consulta à relação *conecta*.

7.5 - As Restrições

O mecanismo de estabelecimento e satisfação de restrições inseridos na linguagem hipotética é baseado na abordagem de construtores descrita em 4.6.

7.5.1 - Uso do contexto para definir o fluxo de dados

Conforme visto em 2.5, quando é utilizado o modelo de perturbação pode haver várias formas de se modificar o valor das variáveis de forma a re-satisfazer as restrições. Em alguns casos o próprio contexto pode ser utilizado para dirimir esta dúvida como na restrição:

always: valAtivação <– sigmóide (soma);

apresentada na figura 7.3. O fato da atribuição ser um caso particular de restrições unidirecionais (ver 4.6.2) é suficiente para indicar que a variável *valAtivação* será igualada ao resultado da aplicação da função *sigmóide*. Mesmo que esta restrição tivesse sido escrita como:

always: valAtivação = sigmóide (soma);

ela continuaria tendo o mesmo comportamento pois a *sigmóide* não tem função inversa declarada, tornando inviável outra interpretação. Fato similar acontece com as demais restrições inseridas na figura 7.3.

Outro caso em que o contexto é suficiente para decidir a direção do fluxo de dados é exemplificado pela restrição:

once: bp.entrada = padrãoEnt;

apresentada na figura 7.6. Aqui a variável *n* representa um objeto passado como parâmetro. Dado que ele não é passado por referência, não há como atribuir-lhe qualquer valor. Isto equivale a atribuir uma anotação de somente-para-leitura à variável *n*.

7.5.2 - Anotação de não-dependência

Um tipo particular de anotação (além das descritas em 2.6.2 e 2.6.3) é utilizada em algumas das restrições apresentadas neste capítulo: a anotação de *não-dependência*. Mais do que apenas reduzir as opções de escolha quando da satisfação das restrições, quando uma variável estiver sublinhada ela não é incluida no grafo de dependências de forma que se seu valor for modicado a restrição não é necessariamente re-satisfeita; ela permanece em um estado inconsistente até que haja uma modificação ou contexto em que force sua satisfaçao.

Um exemplo de aplicação da anotação de não-dependência é a restrição:

always: peso <– peso + Neta * n1.valAtivação * n2.erro;

definida na figura 7.8. Neste caso somente a modifição do erro do neuronodo de destino da conexão é que causará a re-satisfação da restrição e, consequentemente, atualização do peso da conexão. A restrição foi definida desta forma pois no caso das redes back-propagation somente quando da retro-propagação do erro é que os pesos são atualizados.

7.6 - Os Construtores

Os construtores utilizados para a implementação da rede back-propagation são apresentados nas figuras 7.9 e 7.10.

```
constructor = ( n: Neuronodo, v: Number );
        "Construtor de restrição bidirecional entre o valor de ativação do
                neuronodo n e o número denotado por v"
        n.valAtivação = v;
end =;

constructor objetivo ( n: BPNeurSaída) = ( v: Number );
        "Informa ao neuronodo n seu valor de valor de ativação esperado"
        n.erro <– (v - n.valAtivação) * derivaDaSigmóide(n.soma);
end objetivo;
```

Figura 7.9 - Constrututores Sobre A Classe *Neuronodo*

Por ser bidirecional, o construtor = apresentado na figura 7.9 pode tanto ser usado para estabelecer como para obter o valor de ativação de um objeto da classe *Neuronodo*.

Já o construtor *objetivo* é unidirecional em função da atribuição contida em seu corpo. Conforme visto em 4.6.2, a execução de uma restrição que inclua o operador de atribuição consiste da avaliação da expressão à direita e aplicação de uma restrição do tipo *once* entre este valor e a expressão do lado esquerdo. Este construtor é expressão direta da equação 6.3.

```
constructor objetivo ( bp: BackPropagation ) = ( v: Array );
        "Construtor de restrição bidirecional que define o valor
                objetivo da rede"
        local i:
        for i := 1 to v.size do
                objetivo(bp.neuronodosSaida.at(i)) = v.at(i);
        end;
end objetivo;

constructor entrada( bp: BackPropagation ) = ( v: Array );
        "Construtor de restrição bidirecional que define o valor
                de entrada da rede"
        local i:
        for i := 1 to v.size do
                bp.neuronodosEntrada.at(i) = v.at(i);
        end;
end entrada;

constructor saida( bp: BackPropagation ) = ( v: Array );
        "Construtor de restrição bidirecional que define o valor
                de saida calculado pela rede"
        local i:
        for i := 1 to v.size do
                bp.neuronodosSaida.at(i) = v.at(i);
        end;
```

Figura 7.10 - Constrututores Sobre A Classe *Backpropagation*

Os construtores *entrada, saída* e *objetivo* (figura 7.10) são invocados pelos procedimentos *reconhece* e *treina,* definidos na figura 7.7, para se descrever o comportamento de uma rede *Back-Propagation.*

O construtor *entrada*, por exemplo, permite que seja informado um novo padrão de entrada para a rede *Back-Progation.* A execução deste construtor define novos valores de ativação para os neuronodos da camada de entrada, com consequente resatisfação das restrições que envolvam estes valores. Por ser bidirecional, este mesmo construtor pode ser utilizado para se obter o valor atual de ativação dos neuronodos da camada de entrada.

Os outros construtores apresentam comportamento similar ao descrito para o construtor *entrada.* O construtor *objetivo*, contudo, não é bidirecional pois o construtor *objetivo* sobre neuronodos não o é.

7.7 - Considerações

O modelo back-propagation de redes neurais caracteriza-se por ser apresentar padrões regulares de conexão entre neuronodos, os quais são definidos pelas conexões completas entre camadas vizinhas que compõem a rede. Ele caracteriza-se, também, por manter uma arquitetura estática, ou seja, não inclue a possibilidade de aumento ou redução de neuronodos nas camadas, nem alterações nas conexões entre neuronodos.

A implementação apresentada neste capítulo preserva estas características das redes back-propagation, mas introduz uma arquitetura de implementação que possibilita mudanças dinâmicas tanto no padrão de conexão como na topologia interna da rede. Com um pequeno esforço adicional, mesmo as camadas de entrada e de saída não poderiam ser modificadas.

Não são apresentados neste capítulo detalhes sobre o modelo dinâmico da implementação proposta. Para este fim é introduzido no capítulo 8 o problema que cálculo do caminho crítico associado a um diagrama PERT. A implementação proposta para o digrama PERT apresenta um aquitetura muito similar à proposta neste capítulo, porém mais simples, fato que colabora para uma mais clara do modelo dinânico.

8 - DIAGRAMAS PERT

Um diagrama PERT caracteriza-se por ser uma rede[13] na qual os vértices representam eventos, os arcos representam atividades e os valores associados aos arcos representam a duração das atividades. A disposição dos vértices na rede exprimem uma relação de dependência: um evento não ocorre antes de se encerrarem todas as atividades que o precedem. Sendo assim, a rede define uma relação de ordem entre os eventos e atividades. Um exemplo bastante simples de rede é apresentada na figura 8.1.

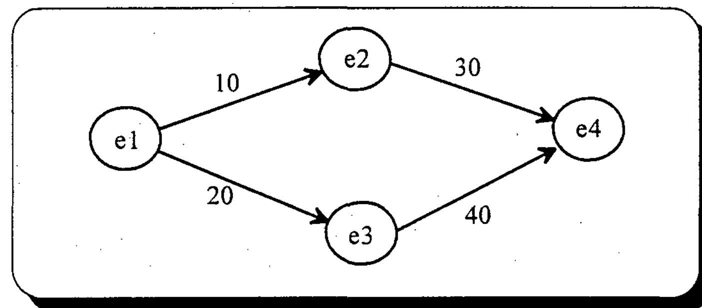


Figura 8.1 - Um Diagrama Pert

8.1 - O Problema do Caminho Crítico

Um dos problemas frequentemente relacionados com os diagramas PERT é o cálculo do caminho crítico[Furtado 73], ou seja, a determinação do caminho que vai do vértice inicial até o vértice final e cuja soma de durações das atividades seja máxima. Este caminho pode ser obtido pela observação dos eventos cuja folga é zero. A folga, neste sentido, é calculada por:

a) Tempo mais cedo em que um evento pode ocorrer (TMCE):

$$TMCE_e = \begin{cases} 0 & se \quad e = evento \ inicial \\ \max(TMCE_x + t_{xe}) & caso \quad contrário \end{cases}$$

b) Tempo mais tarde que se pode permitir que um evento ocorra (TMTE):

$$TMTE_e = \begin{cases} TMCE_e & se \quad e = evento \ final \\ \min(TMTE_x - t_{ex}) & caso \quad contrário \end{cases}$$

c) Folga do evento:

$$FE_e = TMTE_e - TMCE_e$$

d) Tempo mais cedo de conclusão da atividade (TMCA):

$$TMCA_a = TMCE_e + t_{ex}$$

---

[13] Uma rede é um grafo dirigido, sem laços, valorado e que possue uma única entrada e uma única saída [Furtado 73].

e) Tempo mais tarde de início da atividade (TMTA):

$$TMTA_a = TMTE_e - t_{xe}$$

onde t_{ij} representa a duração da atividade entre os eventos *i* e *j*.

8.2 - A Estruturação do Problema

O problema de se construir um diagrama PERT é modelado na linguagem hipotética, apresentada em 7.1, pela definição da classe *Evento* (figura 8.2) e da relação denomidada *atividade* (figura 8.3).

As instâncias da classe *Evento* são constituídas de um *título* (uma descrição para o evento que pode ser qualquer objeto uma vez que a variável não é tipada) e de três números (*maisCedo*, *maisTarde* e *folga*) que correspondem aos ítens (*a*), (*b*) e (*c*) descritos em 8.1. Os inter-relacionamentos entre estes valores são descritos por restrições inseridas na secção *initially*, as quais são expressão direta das equações apresentadas em 8.1

```
class Evento subclassOf Objeto
        public título;
                maisCedo, maisTarde, folga: Number;
        initially
                once: weak: maisCedo = 0;                              [A]
                always: weak: maisTarde <- maisCedo;                   [B]
                always: maisCedo <- max(atividade( _, self, ? , _ ));   [C]
                always: maisTarde <- min(atividade(self ,_ , _ , ?) );  [D]
                always: folga <- maisTarde - maisCedo;
end;
```

Figura 8.2 - A Classe *Evento*

As atividades expressas no diagrama PERT são modeladas por meio da definição de uma relação denomidade *atividade*. Esta relação é estabelecida entre dois eventos, mas envolve outros três objetos da classe *Number* que refletem o custo da atividade e as variáveis *maisCedo* e *maisTarde* descritas nos ítens (d) e (e) do tópico 8.1.

```
relation atividade( e1: Evento, e2: Evento )
                        with (custo, maisCedo, maisTarde: Number);
        always: maisCedo <- e1.maisCedo + custo;               [E]
        always: maisTarde <- e2.maisTarde - custo;             [F]
end;
```

Figura 8.3 - A Relação De *Atividade*

As funções *max* e *mim* utilizadas nas restrições [C] e [D] da figura 8.2 operam sobre as coleções de valores que resultam das consultas à relação *atividade*. A implementação destas funções é apresentada na figura 8.4.

```
max( col: Collection ) : Object;
        local m;
        if col.isEmpty then
                ^NIL
        else    m <– col.one;
                col.do[:x| if x > m then: [m <– x] ];
                ^m;
        end
end;

min( col: Collection ) : Object;
        local m;
        if col.isEmpty then
                ^NIL
        else    m <– col.one;
                col.do[:x| if x < m then: [m <– x] ];
                ^m;
        end
end;
```

Figura 8.4 - As Funções *Max* E *Min* Sobre A Classe *Collection*

8.3 - A Construção do Diagrama PERT

A figura 8.5 apresenta a implementação do procedimento *montaDiagramaPert* que monta o diagrama PERT apresentado na figura 8.1.

```
montaDiagramaPert;
        e1 <- new(Evento);   e1.titulo <- 'e1';          {1}
        e2 <- new(Evento);   e2.titulo <- 'e2';          {2}
        assert: atividade(e1,e2) with(10);               {3}

        e3 <- new(Evento);   e3.titulo <- 'e3';          {4}
        assert: atividade(e1,e3) with(20);               {5}

        e4 <- new(Evento);   e4.titulo <- 'e4';          {6}
        assert: atividade(e2,e4) with(30);               {7}
        assert: atividade(e3,e4) with(40);               {8}
end;
```

Figura 8.5 - Procedimento De Montagem Do Diagrama Pert

Quando da criação de uma instância da classe *Evento* (linhas {1}, {2}, {4} e {6} da figura 8.5), as consultas à relação *atividade* (linhas [C] e [D] da figura 8.2) retornam uma coleção vazia já que este evento não se relaciona com nenhum outro. Como conseqüência, as funções *max* e *min* retornam o valor indefinido (*NIL*). Nesta situação, apesar das restrições [C] e [D] terem maior peso que as restrições [A] e [B], estas últimas são utilizadas pois de forma contrária os valores das variáveis *maisCedo* e *maisTarde* permaneceriam indefinidos. O resultado da execução das instruções {1} e {2} é apresentado na figura 8.6. Nesta figura, e nas demais que se seguem, foi omitido o campo *folga* por questões de simplicidade visual. A *folga* corresponde a uma restrição de simples subtração entre os campos *maisTarde* e *maisCedo*, conforme apresentada na figura 8.2.

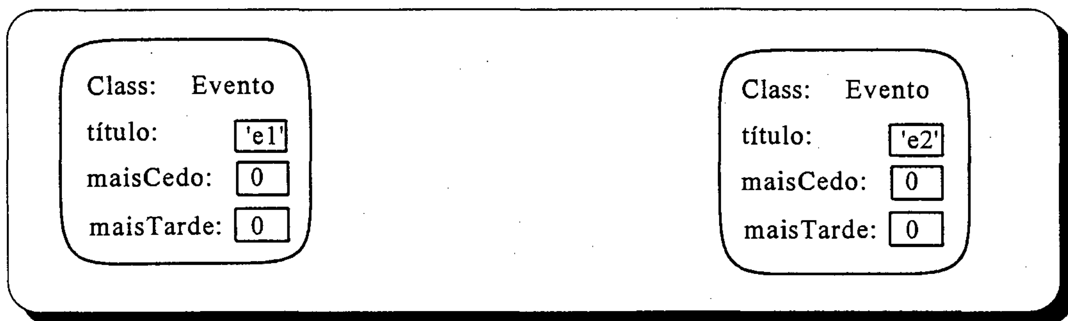


Figura 8.6 - A Instanciação Dos Eventos *E1* E *E2*

Após o estabelecimento da atividade entre os eventos *e1* e *e2* (linha {3} da figura 8.5), as variáveis *maisCedo* do evento *e1* e o *custo* da atividade são as únicas que estão completamente livres, ou seja, seus valores não estão sujeitos a nenhuma restrição que as modifique. Esta situação é apresentada na figura 8.7.

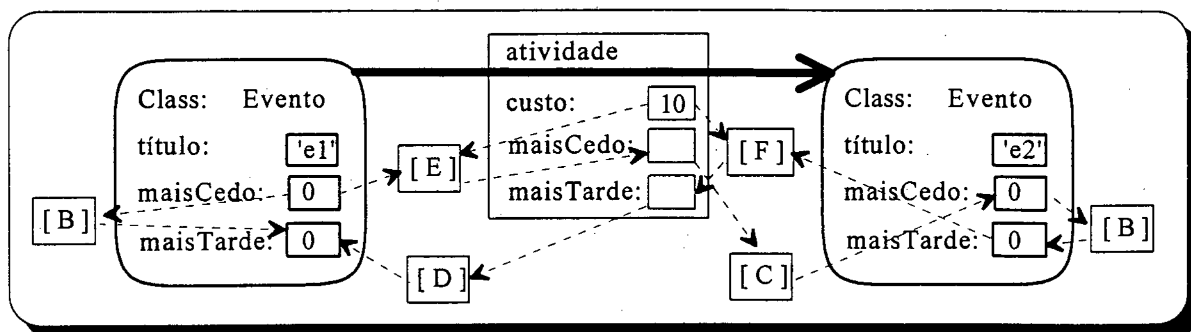


Figura 8.7 - O Estabelecimento Da Atividade Entre Os Eventos *E1* E *E2*

A restrição [E] da figura 8.7 pode, então, ser satisfeita, propagando o valor 10 para a variável *maisCedo* da atividade, resultando na figura 8.8 A restrição [B] sobre o evento *e1* não pode ser satisfeita pois a restrição [D] tem precedência. Nas figuras seguintes à medida que as restrições vão sendo satisfeitas suas linhas passam a ser representadas em traço contínuo.

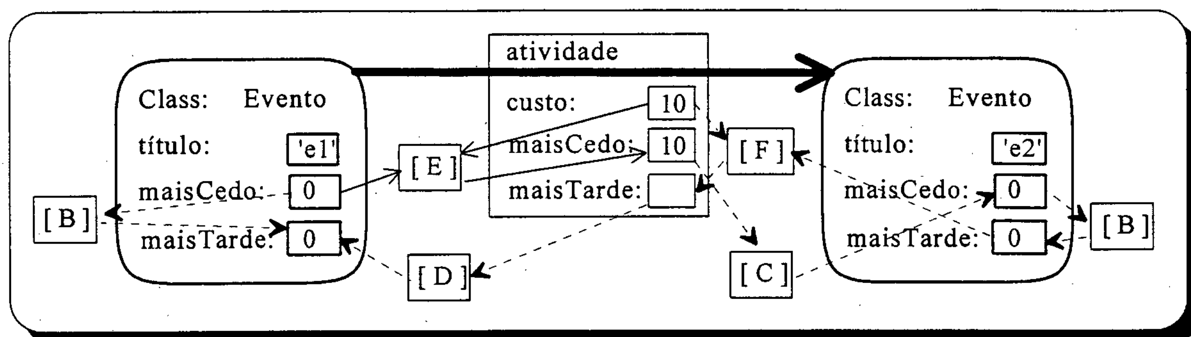


Figura 8.8 - Após A Satisfação Da Restrição [E]

Nesta altura a restrição [C] pode ser satisfeita, resultando na propagação do valor 10 para a variável *maisCedo* do evento *e2*, conforme apresentado na figura 8.9.

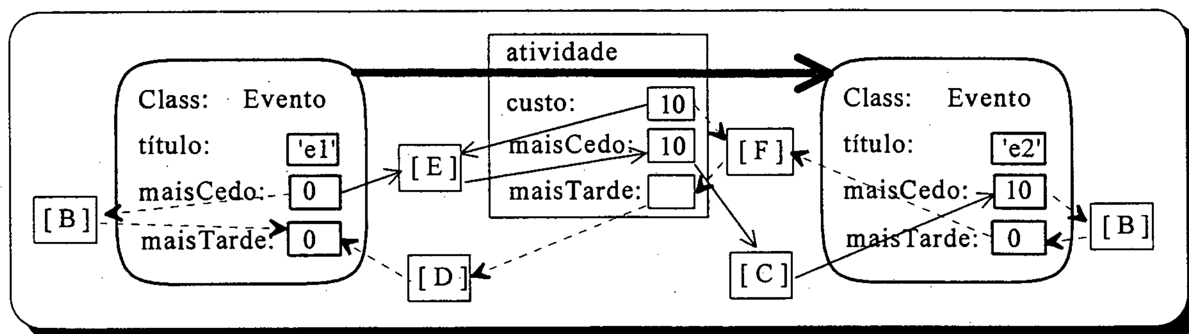


Figura 8.9 - Após A Satisfação Da Restrição [C]

Em seguida o valor da variável *maisCedo* do evento *e2* é propagado para a variável *masTarde* deste mesmo evento em função da satisfação da restrição preferencial [B] sobre o evento *e2*. O resultado é apresentado na figura 8.10.

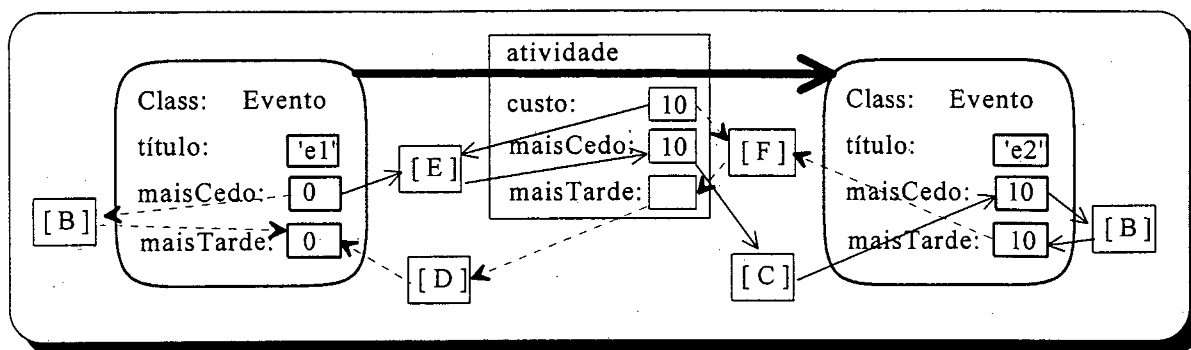


Figura 8.10 - Após A Satisfação Da Restrição [B] Sobre O Evento *E2*.

Por satisfação da restrição [F] o valor da variável *maisTarde* da atividade é fixada em 0 (zero), resultando na figura 8.11

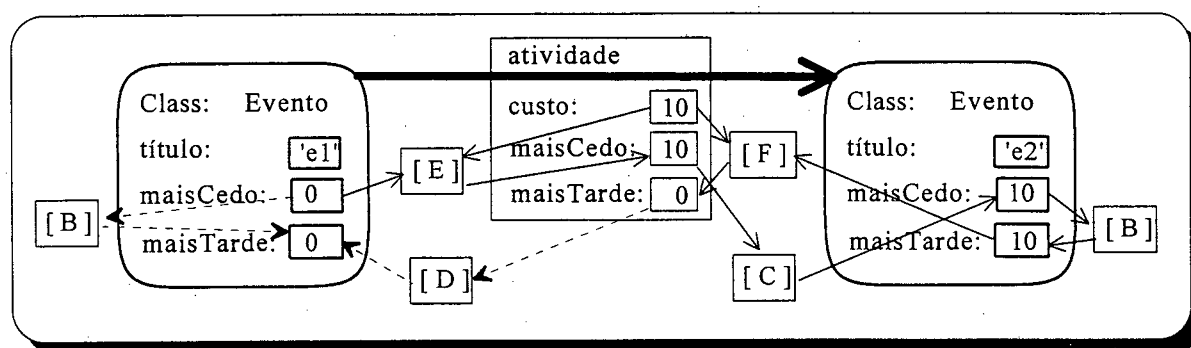


Figura 8.11 - Após A Satisfação Da Restrição [F].

Por último, dentre as restrições [B] e [D] restantes, a última delas tem mais força e, portanto, é escolhida. Neste caso, em particular, a restrição [D] já está satisfeita, de forma que o sistema entra em equilíbrio, conforme figura 8.12.

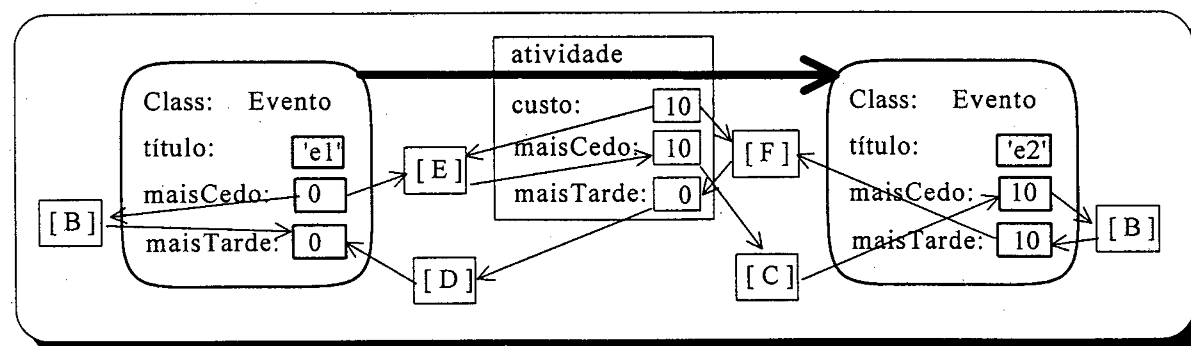


Figura 8.12 - Após A Satisfação De Todas As Restrições

A execução das linhas {4} e {5} da figura 8.5 introduzem um novo evento *e3* no diagrama e uma atividade do evento *e1* para *e3*. De maneira similar à conexão dos eventos *e1* e *e2* detalhada acima, as restrições vão sendo satisfeitas uma a uma, resultando no esquema apresentado na figura 8.13

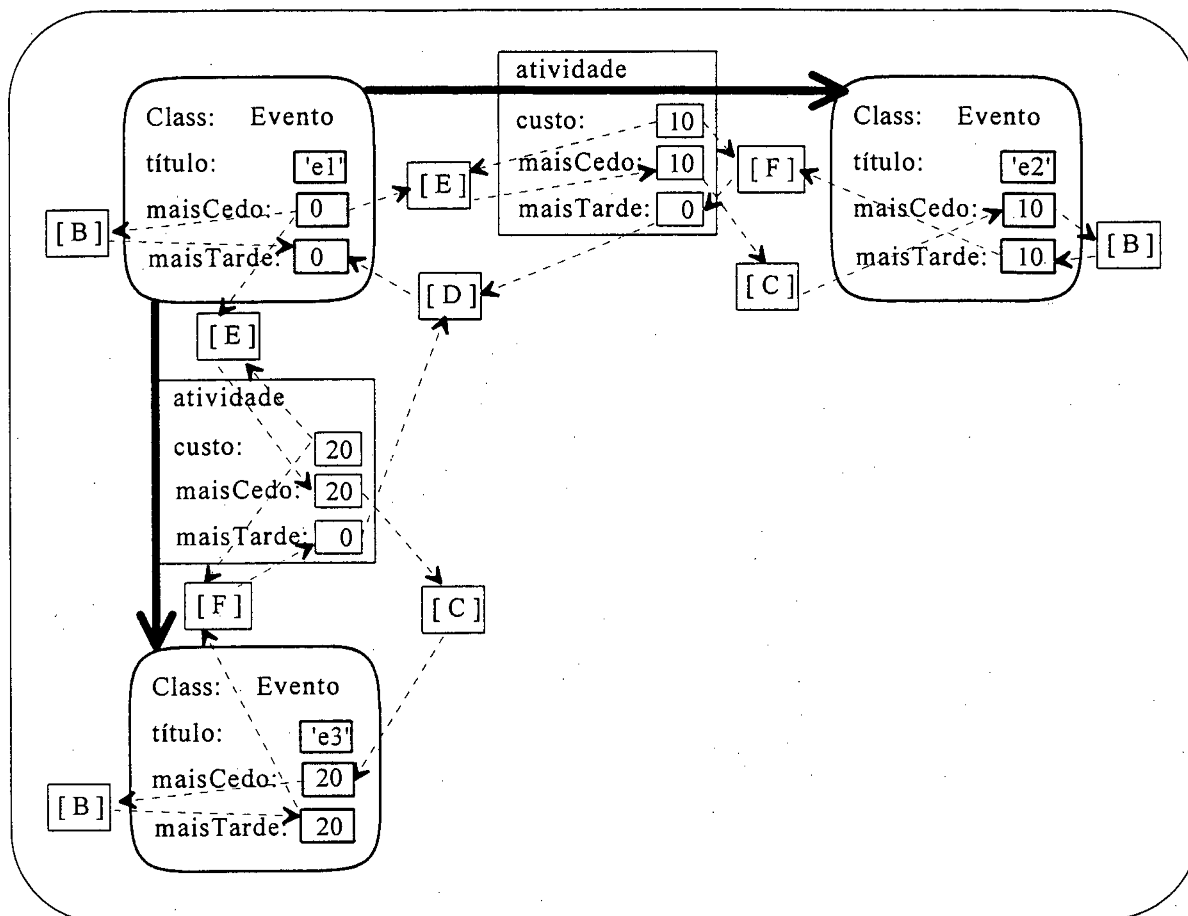


Figura 8.13 - A Do Evento *E3* E Da Atividade De *E1* Para *E3*.

Para completar o diagrama PERT, a execução das linhas {6}, {7} e {8} da figura 8.5 introduzem um o evento *e4* no diagrama e duas transições ligando os eventos *e2* e *e3* a *e4*. A figura 8.14 apresenta o resultado destas operações, já com a satisfação das restrições envolvidas.

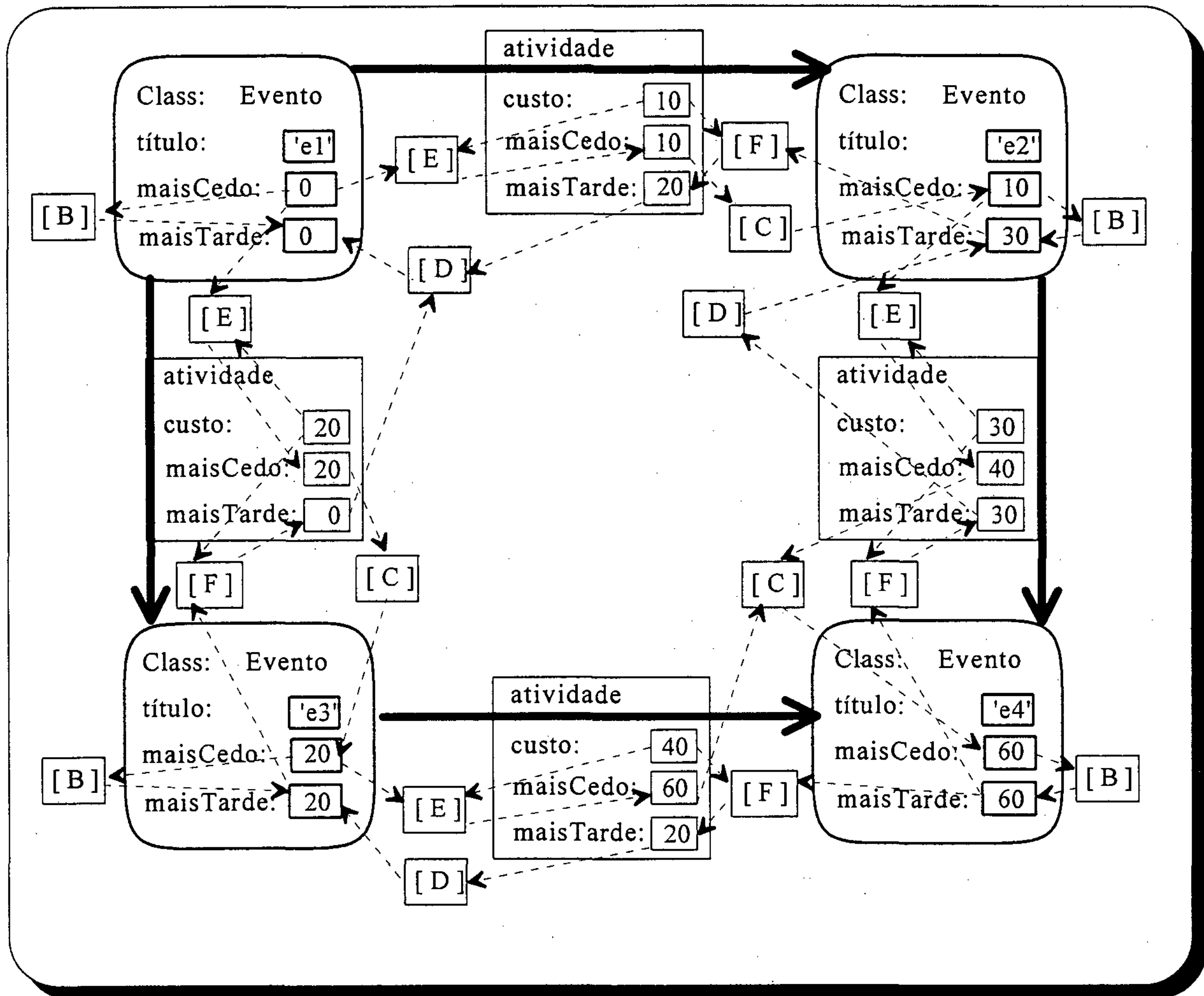


Figura 8.14 - A Do Evento *E4* E Da Atividade De *E2* E *E3* Para *E4*.

8.4 - A Modificação do Diagrama PERT

Completada a fase de construção do diagrama PERT correspondente à figura 8.1, a estrutura de armazenamento de dados assemelha-se ao esquema apresentado na figura 8.14. A adição de novos eventos e/ou novas atividades a este diagrama pode ser feita conforme descrito em 8.3. Mas outra modificação possível num diagrama PERT é a alteração do custo de uma atividade.

Ao se alterar o custo de uma atividade, o diagrama PERT deve ser revisto no sentido de se atualizar os valores dos tempos mais cedo, mais tarde e folga, conforme equações apresentadas em 8.1. Para a realização desta tarefa inicialmente pode-se utilizar o comando retract para desfazer a conexão (e, consequentemente, remover as restrições a ela relacionadas) e, em seguida, utilizar o comando assert para refazer a conexão, agora com o novo custo. A execução dos comandos:

retract: atividade(e3,e4);
assert: atividade(e3,e4) with(15);

por exemplo, resulta na alteração do custo da atividade entre os eventos *e3* e *e4* de 40 para 15 e re-estabelecimento e verificação das restrições conforme descrito em 8.2.

Contudo, por vezes as relações assume um caráter de função, a exemplo da relação *atividade* que associa custos às atividades do diagrama PERT. Assim, o comando *assert* também cumpre o papel de atualizador de função. Isto significa que em situações como expressa acima, não é necessário desfazer a conexão (comando *retract*). Basta ratificar a relação entre os eventos *e3* e *e4,* informando o novo custo da atividade. Imediatamente todas as restrições são re-satisfeitas.

9 - CONSIDERAÇÕES FINAIS

9.1 - Semântica da Linguagem Hipotética

A linguagem hipotética descrita no capítulo 7 é definida em função de mecanismos implementados em linguagens como Smalltalk (Classes e métodos), Kaleidoscope (construtores), Cecil (multi-métodos e tipagem opcional e evolutiva) e Leda (consultas lógicas). Além destes mecanismos, a linguagem hipotética introduz uma construção denominada de "*relation*" que expande os recursos de programação em lógica encontrados na linguagem Leda.

O presente trabalho caracteriza-se mais por abrangência do que por profundidade no tratamento destes temas. Sendo assim, a linguagem hipotética carece de uma análise mais profunda de questões relacionadas com a semântica da construção "*relation*" e de sua integração com os demais mecanismos.

9.2 - Ambiente para Experimentação de Redes Neurais

Vários autores [Smith 92] [Mesrobian 92] [Lutzy 93] apontam para a necessidade de se construir ambientes computacionais capazes de oferecer recursos para manipular diferentes níveis de abstração de redes neurais. Isto inclui disponibilizar uma linguagem de alto poder de expressão que possibilite tanto a descrição de nodos particulares como a descrição de redes de nodos com padrão regular e/ou irregular de conexão. Inclue, também, oferecer recursos visuais para construção, operação e visualização das redes.

A perspectiva multiparadigmática aqui apresentada pode ser uma alternativa significativa para a obtenção de uma linguagem de especificação/implementação de redes neurais. Ela, contudo, não inclue rescursos visuais para interação com o usuário, os quais podem ser obtidos por integraçaõ de características de perspectivas como a da programação visual e da programação por exemplos [Myers 90].

9.3 - Concorrência, Paralelismo e Distribuição

O modelo de computação inserido na linguagem hipotética é estritamente sequencial, não apresentando quaisquer características de distribuição, concorrência ou paralelismo. Os sistemas conexionistas, contudo, caracterizam-se justamente por apresentar alto grau de distribuição e paralelismo [Dayhoff 90].

Uma visão geral das linguagens de programação para sistemas de programação distribuída pode ser encontrada em [Bal 89] e [Marchini 94]. Para a perspectiva de programação inserida neste trabalho, contudo, a abordagem que parece mais relevante é a adotada em linguagens como Lucy [Kahn 90]. Esta linguagem exemplifica a programação por restrições concorrentes, cuja estrutura computacional consiste de coleções de agentes que interagem por meio de operações de restrições sobre variáveis compartilhadas. Ela provê, assim, uma meio de ligar as programações em lógica e por restrições com atores e a programação orientada a objetos.

10 - BIBLIOGRAFIA

[Bal 89] Bal, Henri E. & Steiner, Jennifer G. & Tanenbaum, Andrew S. Programming Languages for Distributed Computing Systems. ACM Computing Surveys, vol 21, num 3, Sepetember 1989.

[Borning 87] Borning, Alan & Duisberg, Robert & Freeman-Benson, Bjorn. Constraint Hierarchies. Proceedings of OOPSLA'87, October 1987.

[Borning 92] Borning, Alan & Freeman-Benson, Bjorn & Wilson, Molly. Constraint Hierarchies. Lisp and Symbolic Computation: An International Journal, 5, 223-270, Netherlands, 1992.

[Budd 93] Budd, Timothy A. Multiparadigm Programming in Leda. Technical Report, Oregon State University, December 1993.

[Casanova 87] Casanova, Marco A. & Giorno, Fernando A. C. & Furtado, Antonio L. Programação em Lógica e a Linguagem Prolog. Edgard Blücher. 461 p. São Paulo, 1987.

[Chambers 92] Chambers, Craig. Object-Oriented Multi-Methods in Cecil. Proceedings of ECOOP'92 Conference, Utrecht, the Netherlands, July 1992.

[Chambers 93] Chambers, Craig. The Cecil Language: Specification and Rationale. Technical Report 93-03-05, University of Washington, Seattle, March 1993.

[Costa 90] Costa, Antônio C. Rocha. A Noção de Tipos e a Modelagem Semântica de Linguagens de Programação. Notas de Aula. Insituto de Informática, UFRGS, 1990.

[Dayhoff 90] Dayhoff, Judith E. Neural Network Architectures: An Introduction. Van Nostrand Reinhold. 259 p. New York, 1990.

[Freeman-Benson 89] Freeman-Benson, Bjorn. A Module Mechanism for Constraints in Smalltalk. Proceedings of OOPSLA'89, October 1989.

[Freeman-Benson 90] Freeman-Benson, Bjorn & Borning, Alan. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. Proceedings of the ECOOP/OOPSLA'90, October 1990.

[Freeman-Benson 92] Freeman-Benson, Bjorn & Borning, Alan. Integrating Constraints with an Object-Oriented Language. Proceedings of the ECOOP'92, 268-286, Utrecht, the Netherlands, July 1992.

[Furtado 73] Furtado, Antonio Luz. Teoria dos Grafos: algoritmos. Livros Técnicos e Científicos, Rio de Janeiro, 1973.

[Hartson 89] Hartson, H. Rex & Hix, Deborah. Human-Computer Interface Development: Concepts and Systems for Its Management. ACM Computing Surveys, vol.21, no.1, March 1989.

[Heileman 92] Heileman, Gregory L. & Georgiopoulos, Michael & Roome, William D. A General Framework for Concurrent Simulation of Neural Network Models. IEEE Trans. Soft. Eng, vol.18, no.7, July 1992.

[Horn 92] Horn, Bruce. Constraint Patterns As a Basis For Object Oriented Programming. Proccedings of OOPSLA'92, 218-234.

[Johnson 86] Johnson, Ralph E. Type-Checking Smalltalk. Proceedings of OOPSLA'86. September 1986.

[Kahn 90] Kahn, Kenneth M. & Saraswat, Vijay A. Actors as a Special Case of Concurrent Constraint Programming. Proceedings of ECOOP/OOPSLA'90, October 1990.

[Krishnan] Krishnan, Murali. SmallBrain version 1.1: A Neural Network Simulator written in Smalltalk. Technical Report. University of Washington.

[Kristensen 91] Kristensen, Bent Bruun & Madsen, Ole Lehrmann & Birger, Moller-Pedersen & Nygaard, Kristen. Object Oriented Programming in the Beta Programming Language. 275 p. DRAFT. September 1991.

[Lassez 1987] Lassez, Catherine. Constraint Logic Programming. BYTE 171, August 1987.

[LEiB 87] LeiB, Hans. On Type Inference for Object-Oriented Programming Languages. Lecture Nots in Computer Science 329. 1st Workshop on computer Science Logic, Karlsruhe, FRG, October 1987.

[Leler 88] Leler, Wm. Constraints Programming Languages: Their Specification and Generation. Addison-Wesley, 202 p, 1988.

[Lippmann 87] Lippmann, Richard P. An Introduction to Computing with Neural Nets. IEEE ASSP Magazine, vol.3, no. 4, 4-22, April 1987.

[Liu 92] Liu, Chamond & Goetze, Stephen & Glynn, Bill. What Contributes to Successful Object-Oriented Learning? Proceedings of OOPSLA'92, 77-86.

[Lopez 93] Lopez, Gus & Freeman-Benson, Bjorn & Borning, Alan. Kaleidoscope: A Constraint Imperative Programming Language. Technical Report 93-09-04, University of Washington, September 1993.

[Lopez 94a] Lopez, Gus & Freeman-Benson, Bjorn & Borning, Alan. Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93 Virtual Machine. Proccedings of OOPSLA'94, Portland, october 1994.

[Lopez 94b] Lopez, Gus & Freeman-Benson, Bjorn & Borning, Alan. Constraints adn Object Identity. Proccedings of ECCOP'94, Bolonha, Italy, July 1994.

[Lutzy 93] Lutzy, Ottmar & Dengel, Andreas. A Comparison of Neural Net Simulators. IEEE Expert, 43-51, August 1993.

[Madsen 90] Madsen, Ole Lehrmann & Magnusson, Boris. Strong Typing of Object-Oriented Languages Revisited. Proceedings of ECCOP/OOPSLA'90. October 1990.

[Maloney 89] Maloney, John & Borning, Alan & Freeman-Benson, Bjorn. Constraint Technology for User-Interface Construction in ThingLab II. Proceedings of OOPSLA'89, October 1989.

[Marchini 94] Marchini, Márcio Quintaes. LindaTalk: Suporte Distribuído à Programação Concorrente Orientada a Objetos. Dissertação de Mestrado. UFSC/CTC/EPS, Janeiro de 1994.

[Mattos 89] Mattos, Nelson Mendonça. An Approach to Knowledge Base Management - requirements, knowledge representation, and design issues. Tese de Doutorado. Universidade de Kaiserslautern, 1989.

[Mesrobian 92] Mesrobian, Edmond & Skrzypek, Josef. A Software Environment For Studyng Computational Neural Systems. IEEE Trans.Soft.Eng, vol.18, no.7, 575-589, july 1992.

[Meyer 88] Meyer, Bertrand. Object-Oriented Software Construction. Prentice Hall, 1988.

[Myers 90] Myers, Brad A. Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints. ACM Trans. Prog. Lang and Systems. Vol.12, no.2, 143-177, April 1990.

[Passerino 90] Passerino, Liliana. O uso de Tipos em Linguagens de Programação. Trabalho Indiviual I. PGCC. UFRGS, 1990.

[Sannella 93] Sannella, Michael & Maloney, John & Freeman-Benson, Bjorn & Borning, Alan. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. Technical Report 92-07-05a, University or Washington, May 1993

[Smith 92] Smith, Leslie S. A Framework for Neural Net Specification. IEEE Trans. Soft. Eng, vol 18, no 7, July 1992.

[Takahashi 90] Takahashi, Tadao & Liesenberg, Hans K.E. & Xavier, Daniel Tavares. Programação Orientada a Objetos: Uma Visão Integrada do Paradigma de Objetos. VII Escola de Computação, São Paulo, 335 p, 1990.

[Wilk 91] Wild, Michael R. Equate: An Object-Oriented Constraint Solver. Proceedings of the OOPSLA'91, 286-298.