**Uma Abordagem Orientada a Objetos de
uma Ferramenta de Auxílio à Programação
Paralela.**

Nivaldi Calônego Júnior

Tese apresentada ao Instituto de Física de
São Carlos, da Universidade de São Paulo,
para obtenção do Título de Doutor em
Ciências: "Física Aplicada".

Orientador: Prof. Dr. Marcos José Santana

São Carlos

1997

IFSC-USP SERVIÇO DE BIBLIOTECA
INFORMAÇÃO

Calônego Júnior, Nivaldi

Uma abordagem orientada a objetos de uma ferramenta para auxiliar o desenvolvimento de aplicações em arquiteturas paralelas - São Carlos, 1997.

144 p.

Tese (Doutorado)—Instituto de Física de São Carlos, 1997.

Orientador: Prof. Dr. Marcos José Santana

1. Programação Paralela. 2. Ferramentas de apoio a programação.

UNIVERSIDADE
DE SÃO PAULO

Instituto de Física de São Carlos

Av. Dr. Carlos Botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 272-6222
Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE NIVALDI
CALÔNEGO JÚNIOR APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO
CARLOS, UNIVERSIDADE DE SÃO PAULO, EM 31/10/1997.

COMISSÃO JULGADORA:

Prof. Dr. Marcos José Santana (ICMSC-USP)

Prof. Dr. Carlos Antônio Ruggiero (IFSC/USP)

Prof. Dr. Jan Frans Willem Slaets (IFSC-USP)

Prof. Dr. Onofre Trindade Júnior (ICMSC-USP)

Prof. Dr. Siang Wun Song (IME-USP)

IFSC-USP SERVIÇO DE BIBLIOTECA E
INFORMAÇÃO

ANEXOS: 1000

À minha esposa Reni, por toda dedicação, aos meus pais Nivaldi e Godiva pelos sacrifícios e educação, aos meus irmãos e sobrinhos, por todo carinho e dedicação.

Agradecimentos

Ao amigo e Prof. Dr. Marcos José Santana, pela orientação, acompanhamento e paciência que dedicou a este trabalho e a mim.

À Profª. Drª. Regina Helena C. Santana, pelas sugestões valiosas e marcantes neste trabalho.

Aos amigos Marcos e Regina, pela amizade sincera, incentivo pessoal durante esses anos e pelos churrascos.

Ao Paulo, à Simone e ao Felipe, por todos os bons momentos e por me aturarem nas horas críticas.

Ao amigo Pires, pelo incentivo, apoio incondicional, pelas pecarias e churrascos.

A todos os professores e funcionários do ICMSC que durante todos esses anos têm contribuído para a minha formação.

A todos os colegas do grupo, pelas constantes trocas de idéias e pelas críticas que, sem dúvida nenhuma, sempre contribuem.

À Vladerez e às secretárias do IFSC por sempre estarem dispostas a ajudar.

Aos companheiros e amigos: Ana Luísa, Boca, Robson, Walter, Roberta, Mário, Flávio, Laís, Sarita, Cristina, Célia, Renata, Regina Sousa, Maurício, Heli, Elisa, Trolha, Jacques, Paulinho, Cabral e Eduardo; Ângelo, Luciano, pela atenção dispensada e muitos outros, que de uma maneira ou de outra contribuíram para a realização deste trabalho.

À minha família e aos meus pais em especial, que souberam compreender minha ausência e sempre me incentivam.

À minha esposa Reni, por me aturar nas horas mais difíceis.

Ao Sr. Zé, Dª. Olga e Lú, pela simpatia no atendimento do "redondo".

A Deus, por me acompanhar em todos os momentos.

À UNIC, à Capes, ao CNPq e à FAPESP pelo apoio financeiro.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO 1

1.1 CONSIDERAÇÕES INICIAIS 1

1.2 DESCRIÇÃO DO PROBLEMA 5

1.3 UMA VISÃO MACROSCÓPICA DA FERRAMENTA EM ESTUDO 7

1.4 ORGANIZAÇÃO DO TEXTO 10

2. ANÁLISE E PROJETO DE SISTEMAS..... 12

2.1 CONSIDERAÇÕES INICIAIS 12

2.2 CONCEITOS..... 16

2.2.1 Encapsulamento 17

2.2.2 Herança..... 18

2.2.3 Polimorfismo 19

2.3 MÉTODOS ORIENTADOS A OBJETOS..... 20

2.3.1 Análise de Sistemas Orientada a Objetos..... 20

2.3.2 Análise Baseada em Objetos..... 21

2.3.3 O Método OMT 24

2.3.4 O Método Objectory..... 26

2.3.5 O Método Fusion..... 32

2.4 PROGRAMAÇÃO ORIENTADA A OBJETOS 36

2.4.1 Encapsulamento 39

2.4.2 Herança..... 39

2.4.3 Polimorfismo 40

2.5 CONSIDERAÇÕES FINAIS 41

3. UM MODELO ORIENTADO A OBJETOS PARA A TEORIA DOS GRAFOS 44

3.1 CONCEITOS, DEFINIÇÕES E OPERAÇÕES..... 44

3.1.1 Nós e Arcos 45

3.1.2 Grafos e Dígrafos.....	51
3.1.3 Supernós e Super-arcos	54
3.1.4 Supergrafos e Superdígrafos.....	59
3.1.5 Operações	63
3.2 UMA ESTRUTURA DE CLASSES HIERÁRQUICAS PARA MODELAR GRAFOS E DÍGRAFOS...	64
3.2.1 Modelo para Arcos e Nós	65
3.2.2 Modelo para Arcos e Nós Rotulados.....	67
3.2.3 Modelos para Grafos e Dígrafos.....	69
3.2.4 Modelo para Subgrafos e Subdígrafos	70
3.3 CONSIDERAÇÕES FINAIS	72
4. MODELO ORIENTADO A OBJETOS PARA ARQUITETURAS E PROGRAMAS PARALELOS.....	74
4.1 CONSIDERAÇÕES INICIAIS	75
4.2 TAXONOMIAS E MODELO PARA O HARDWARE	77
4.2.1 Taxonomias para Arquiteturas Paralelas.....	78
4.2.2 Modelo Orientado a Objetos para a Taxonomia de Duncan.....	80
4.3 DESENVOLVIMENTO DE PROGRAMAS PARALELOS/CONCORRENTES	88
4.3.1 Esqueletos para Aplicações Paralelas	90
4.4 CONSIDERAÇÕES FINAIS	92
5. INTERFACES	94
5.1 CONSIDERAÇÕES INICIAIS	94
5.2 INTERFACE BASEADA EM MENU	96
5.3 INTERFACE GRÁFICA	100
5.4 CONSIDERAÇÕES FINAIS	105
6. UM EXEMPLO DE APLICAÇÃO.....	106
6.1 CONSIDERAÇÕES INICIAIS	106
6.2 COMPONENTES DO HARDWARE	107
6.3 COMPONENTES DO SOFTWARE	110
6.3.1 Notação.....	111
6.3.2 Tipos de Dados, Canais e Protocolos de Comunicação.....	111
6.3.3 Subprogramas e Tarefas.....	113
6.3.4 O Comando PLACE.....	115
6.4 PARTIÇÃO DOS PROCESSOS	116
6.5 MODELO PARA O HARDWARE.....	119
6.6 MAPEAMENTO DOS PROCESSOS NOS PROCESSADORES	120
6.7 CONTROLE DE CONCORRÊNCIA PARA ACESSO AOS CANAIS FÍSICOS DE TRANSPUTERS	122

6.7.1 <i>Subsistema para Multiplexação Demultiplexação de Canais OCCAM</i>	123
6.7.2 <i>A Criação dos Processos Mux/Demux</i>	124
6.8 ESTENDENDO A FAPP PARA O AMBIENTE PVM.....	129
6.8.1 <i>Arquitetura do hardware</i>	129
6.9 ARQUITETURA DO SOFTWARE.....	130
6.10 CONSIDERAÇÕES FINAIS	131
7. CONCLUSÕES	135
7.1 CONSIDERAÇÕES INICIAIS	135
7.1.1 <i>Descrição do Problema</i>	135
7.1.2 <i>Orientação a Objetos</i>	136
7.1.3 <i>Modelo Matemático</i>	136
7.1.4 <i>Modelos Arquiteturais</i>	137
7.1.5 <i>Interface</i>	137
7.1.6 <i>Aplicação</i>	137
7.2 CONTRIBUIÇÕES RELEVANTES	138
7.3 TRABALHOS FUTUROS	138
7.4 CONSIDERAÇÕES FINAIS	139
8. REFERÊNCIAS BIBLIOGRÁFICAS	140

LISTA DE FIGURAS

Figura 1.1 - Visão macroscópica da ferramenta.....	8
Figura 2.1 - Encapsulamento.....	17
Figura 2.2 - Exemplo de herança.....	19
Figura 2.3 - Símbolos para a geração do diagrama de objetos.....	22
Figura 2.4 - Notação para o diagrama de estados.....	23
Figura 2.5 - Símbolos gráficos do modelo de objetos.....	24
Figura 2.6 - Exemplo de uma estrutura hierárquica de classes.....	25
Figura 2.7 - Símbolos do modelo de análise, Jacobson (1994).....	26
Figura 2.8 - O objetos Luiz e suas operações.....	27
Figura 2.9 - Relação de composição.....	28
Figura 2.10 - Hierarquia de classes para composição.....	29
Figura 2.11 - Dentro do objeto Luiz apenas o comportamento Salte está especificado.....	29
Figura 2.12 - Fases do desenvolvimento do sistema, Jacobson (1994).....	30
Figura 2.13 - Subfases da análise e do projeto.....	31
Figura 2.14 - Exemplo de composição e agregação.....	33
Figura 2.15 - Resumo da notação gráfica do modelo de análise.....	33
Figura 2.16 - Exemplo de composição e agregação.....	34
Figura 2.17 - Pré-condições x Ciclo de vida, Coleman; et. al. (1994).....	36
Figura 2.18 - Estrutura hierárquica ambígua.....	41
Figura 3.1 - Componentes possíveis para um grafo.....	45
Figura 3.2 - Semântica da interface de um nó.....	46
Figura 3.3 - Um exemplo de grafo.....	51
Figura 3.4 - Exemplo de um grafo com identificadores.....	52

Figura 3.5 - Um modelo de rede.....	52
Figura 3.6 - Exemplo de um dígrafo.....	54
Figura 3.7 - Nós e supernós rotulados.....	57
Figura 3.8 - Super-arco rotulado.....	58
Figura 3.9 - Super-arco rotulado e com peso.....	59
Figura 3.10 - Um exemplo de um dígrafo rotulado.....	60
Figura 3.11 - Exemplo de um super-dígrafo.....	61
Figura 3.12 - Classes GRFNo (a), GRFArco (b) e seus atributos.....	66
Figura 3.13 - Classe para a geração de objetos persistentes.....	67
Figura 3.14 - Classes para a representação de nós rotulados.....	67
Figura 3.15 - Classes hierárquicas para nó e arco rotulados.....	68
Figura 3.16 - Classes para o gerenciamento de objetos do sistema.....	69
Figura 3.17 - Classes grafo e dígrafo.....	70
Figura 3.18 - Níveis de visibilidade ou subgrafos.....	71
Figura 3.19 - Classe Super-nó e Super-Arco.....	71
Figura 3.20 - Classe GRFSuperDigrafo e suas derivações.....	72
Figura 3.21 - Visão macroscópica das classes.....	72
Figura 4.1 - Classes-base para o modelo dos componentes das arquiteturas.....	83
Figura 4.2 - Classe-base para os meios de transmissão.....	84
Figura 4.3 - Elemeto Processador.....	85
Figura 4.4 - Representação gráfica dos grafo de conexão dos elementos processadores.....	85
Figura 4.5 - A classe FAPPElemProcessador.....	86
Figura 4.6 - Hierarquia de classes para a taxonomia de DUNCAN.....	87
Figura 4.7 - Domínios para programação paralela/concorrente. (a) MIMD com memória compartilhada, (b) MIMD com memória distribuída, (c) Sistemas distribuídos.....	87
Figura 4.8 - Um modelo para programas paralelos.....	91
Figura 4.9 - Meta-modelo para programas paralelos.....	92
Figura 5.1 - Modelo de interface.....	95
Figura 5.2 - Modelo de interface de Hix; Hartson (1993).....	95
Figura 5.3 - Visões da aplicação.....	97

Figura 5.4 - Elementos da interface da ferramenta.....	99
Figura 5.5 - Interação usuário/interface/sistema.....	100
Figura 5.6 - Exemplo de elementos de processamento.....	101
Figura 5.7 - Classe base para formas baseadas em um par de pontos.....	102
Figura 5.8 - Estrutura das classes para gerar elementos de desenho.....	103
Figura 5.9 - Componentes para a interface.....	104
Figura 6.1 - Estrutura lógica do Servidor de Processamento Paralelo.....	108
Figura 6.2 - Hierarquia de classes para os Elementos Processadores.....	109
Figura 6.3 - Canais lógicos e protocolos.....	113
Figura 6.4 - Comandos PROC e FUNC da linguagem OCCAM2.....	115
Figura 6.5 - Estrutura lógica de um Transputer T800.....	115
Figura 6.6 - Projeto lógico das partições.....	116
Figura 6.7 - Estrutura lógica para as aplicações.....	117
Figura 6.8 - Topologia para a aplicação.....	119
Figura 6.9 - Estrutura para uma aplicação.....	122
Figura 6.10 - Representação lógica de um multiplexador.....	123
Figura 6.11 - Representação lógica de um demultiplexador.....	124
Figura 6.12 - Representação lógica dos processos mux/demux.....	125
Figura 6.13 - Esqueleto para uma aplicação genérica.....	125
Figura 6.14 - Topologia para a aplicação.....	126
Figura 6.15 - Elementos processadores PVM.....	130

LISTA DE TABELAS

TABELA 2.1 - EVOLUÇÃO DOS SISTEMAS E MÉTODOS COMPUTACIONAIS. CARMICHAEL (1994)..... 15

TABELA 2.2 - GABARITO PARA O MODELO DE OPERAÇÕES 35

TABELA 2.3 - SINTAXE E SEMÂNTICA DA LINGUAGEM PARA A DESCRIÇÃO DO CICLO DE VIDA. 36

TABELA 2.4 - LINGUAGENS DE PROGRAMAÇÃO ORIENTADAS A OBJETOS. 40

TABELA 2.5 - UTILIZAÇÃO DE CLASSE VIRTUAL EM C++ 42

TABELA 3.1 - RÓTULOS PARA IDENTIFICADORES E NÓS. 49

TABELA 3.2 - RELAÇÃO RÓTULO/IDENTIFICADOR. 54

TABELA 3.3 - RELAÇÃO IDENTIFICADOR/RÓTULO PARA A FIGURA 3.10. 60

TABELA 3.4 - TABELA DE OPERAÇÕES. 63

TABELA 3.5 - GERAÇÃO DE IDENTIFICADORES. 64

TABELA 3.6 - CLASSES-BASE GRF. 66

TABELA 3.7 - CLASSES GRFRÓTULO, GRFNORÓTULADO, GRFARCO. 68

TABELA 3.8 - CLASSES DE ARMAZENAMENTO PAR ARCOS E NÓS. 70

TABELA 4.1 - TAXONOMIA DE FLYNN..... 79

TABELA 4.2 - TAXONOMIA DE DUNCAN..... 80

TABELA 6.1 - EXEMPLO PARA INSTÂNCIAS PARA A CLASSE FAPPPROCESSADOR. 109

TABELA 6.2 - INSTÂNCIAS DA CLASSE FAPPMEMORIA. 110

TABELA 6.3 - ESTRUTURAS BASICAS PARA O ELEMENTO PROCESSADOR T800. 110

TABELA 6.4 - OBJETOS DA CLASSE SFTPROCARGS. 117

TABELA 6.5 - ESQUELETO PARA O PROCESSO APPLIC1..... 118

TABELA 6.6 - ESQUELETO PARA O PROCESSO APPLIC2..... 118

TABELA 6.7 - ESQUELETO PARA O PROCESSO APPLIC3..... 118

TABELA 6.8 - CONFIGURAÇÃO DOS ELEMENTOS PROCESSADORES. 119

TABELA 6.9 - DEFINIÇÃO DAS TAREFAS. 120

TABELA 6.10 - MAPA PROCESSOS/PROCESSADORES. 120

TABELA 6.11 - MAPA CANAL LÓGICO/FÍSICO..... 121

TABELA 6.12 - ESQUELETO PARA EXEMPLIFICAR O COMPARTILHAMENTO DE CANAIS FISICOS..... 127

TABELA 6.13 - GABARITO PARA UM PROCESSOS MULTIPLEXADOR. 128

TABELA 6.14 - GABARITO PARA UM PROCESSO DEMULTIPLEXADOR..... 128

RESUMO

Este trabalho contribui na busca de soluções para o problema de auxílio à programação paralela, apresentando uma abordagem orientada a objetos, como base para a construção de uma ferramenta que dá apoio ao desenvolvimento de programas paralelos. Diversas ferramentas com propostas análogas são revisadas e suas características principais são destacadas, visando a busca de um modelo adequado para a ferramenta a ser proposta. A ferramenta desenvolvida, implementada e validada neste trabalho (FAPP - Ferramenta de Auxílio à Programação Paralela) baseia-se na tecnologia de orientação a objetos. A teoria dos grafos, modelada segundo a orientação a objetos, serve de base para a criação de modelos tanto para arquiteturas paralelas (hardware) como para programas paralelos (software). Os modelos criados para o hardware e software, permitem ao programador criar o ambiente para a programação, definindo a sua arquitetura paralela, os processos componentes de seu programa e o mapeamento lógico desses processos nos processadores. A ferramenta FAPP gera automaticamente o esqueleto para a aplicação paralela. Todo o desenvolvimento efetuado é validado através de uma implementação básica da ferramenta e são apresentadas as diretrizes para futuras extensões, visando outros ambientes de hardware e software, bem como melhoramentos objetivando futuros trabalhos.

ABSTRACT

This work contributes to the solution of the parallel programming supporting problem, by proposing an object-oriented approach as the basis for building a tool to help the development of parallel programs. Several tools with similar goals are revised and their main features are highlighted aiming the search of an adequate model for the supporting tool to be developed. The tool developed, implemented and validated in this work (FAPP - Parallel Programming Supporting Tool) is based on the object-orientation technology. The graph theory was modeled according to the object-orientation and used as the basis for the creation of models for both parallel architectures (hardware) and parallel programs (software). This allows the programmer to create the programming environment by defining his parallel architecture, the program processes and the logical mapping of the processes on the processors. The FAPP tool automatically generates the skeleton for the parallel application. The work is validated by means of a basic implementation of the tool. The guidelines for future extensions aiming other hardware and software environments as well as for future works, are presented.

1. Introdução

Este capítulo estabelece o cenário no qual se insere o desenvolvimento desta tese de doutorado. Dessa forma, o desenvolvimento de ferramentas que auxiliam a programação paralela é colocado como o tema geral da pesquisa e direciona a análise inicial. O objetivo central do trabalho é estabelecido e as diretrizes básicas para o seu desenvolvimento são apresentadas.

1.1 Considerações Iniciais

A tecnologia envolvida no desenvolvimento do hardware permitiu, em quatro décadas, aumentar da ordem de 10^2 (ENIAC) para 10^9 (CRAY-1) FLOP/S a potência computacional das unidades centrais de processamento, para arquiteturas com um único processador, além de desenvolver arquiteturas paralelas, como o KSR1 que atinge pico teórico de 43,5 GFLOP/S (Steen (1993)), e as redes de computadores.

O desenvolvimento de software¹, entretanto, não tem ocorrido nas mesmas proporções que o desenvolvimento do hardware², causando a “crise do software” (Pressman (1995)). Essa crise é decorrente do fato de não haver suporte, suficientemente bem automatizado, para manter e/ou gerar aplicações estáveis, com as respectivas documentações atualizadas. Diversos paradigmas e metodologias têm sido propostos para solucionar esse problema, mas o consenso é que paradigmas e metodologias distintos devem ser aplicados a determinados conjuntos de problemas.

¹Recentemente incorporada ao dicionário Aurélio da língua portuguesa. significa o conjunto de *programas e documentos* que compõem o sistema.

²Dispositivos físicos que integram um sistema computacional informatizado

A indústria do software, desde a década de 60, não conseguiu estabelecer uma plataforma sólida para o desenvolvimento de aplicações estáveis para máquinas seqüenciais e está se deparando com novos problemas: as arquiteturas paralelas, os sistemas distribuídos e os requisitos de que as ferramentas utilizadas para o desenvolvimento de software têm que evoluir em função do hardware.

Em meados da década de 80 os temas “arquiteturas paralelas” e “redes de computadores” eram tratados separadamente. O avanço tecnológico das redes de computadores provocou o aumento significativo da vazão dos meios de comunicação (de cerca de 10 Mbps para 612 Mbps, com previsão de 1 Gbps até o final desta década) e a diminuição significativa das taxas de erro de comunicação nesses sistemas. Esses dois fatores aproximaram as pesquisas voltadas para a programação paralela/concorrente. Alguns autores passaram a caracterizar os sistemas distribuídos como um subconjunto das arquiteturas paralelas (Misra; Chandy (1989)) (Zaluska (1991)).

As técnicas de programação desenvolvidas para máquinas seqüenciais são, sob certos aspectos, de pouca valia para os novos modelos arquiteturais. Um problema básico da programação paralela/concorrente é gerenciar a complexidade, pois, dentre os diversos problemas decorrentes da utilização de arquiteturas paralelas os mais comuns são:

- desenvolvimento de algoritmos paralelos;
- determinação de condições para parada dos programas;
- gerenciamento de recursos ou dados para as subtarefas paralelas;
- granulação da aplicação e de suas subtarefas;
- utilização de mecanismos de comunicação, principalmente, no caso das redes de computadores;
- balanceamento de carga.

Há pesquisadores que, em analogia às teorias de sistemas distribuídos, apregoam ser o sistema operacional o responsável pela

alocação das tarefas, desobrigando o programador de escrever o código otimizado para uma determinada arquitetura (Zhou; et al. (1993)). O sistema Utopia, por exemplo, implementa um nível de abstração mais elevado, em relação a uma versão do sistema operacional UNIX, fazendo com que um sistema integrado de redes de computadores possa operar como um todo e de maneira transparente, estabelecendo o balanceamento dinâmico das cargas do sistema. Enquadram-se nesse tipo de sistema os ambientes que utilizam bibliotecas de *run time*, criando máquinas paralelas virtuais, tais como PVM (Beguelin; et al. (1994)), MPI (Gropp; Skjellum (1995)), LINDA (Carriero; et al. (1994)), PARMACS (Calkin; et al. (1994)).

Uma segunda linha de pesquisa defende que o ambiente é que deve ser projetado (ou ter “conhecimento”) para gerar o código executável para uma determinada arquitetura (Zima; Chapman (1992)). Em seu trabalho, Zima e Chapman propõem a intervenção do programador apenas quando estritamente necessário. O ambiente é suprido com informações relativas às arquiteturas, tanto de hardware quanto de software, e são aplicadas técnicas de inteligência artificial para promover o transporte do código fonte entre máquinas distintas. Nesse caso, as alterações da lógica dos algoritmos têm que ser efetuadas por programadores.

Há ainda aqueles que defendem a idéia de que os programas escritos para arquiteturas paralelas devem explorar o máximo de sua potência computacional, devendo o programador conhecer detalhes do hardware, do sistema operacional e do compilador. Neste último caso, fica claro que a intenção é minimizar o tempo de execução. Isso torna quase impraticável a migração dos programas para outras arquiteturas, pois a solução do problema está associada à plataforma de desenvolvimento.

O fato é que, aqueles que têm a necessidade de aumentar significativamente o desempenho de programas, percebem o potencial que o paralelismo do código pode oferecer e se deparam com as dificuldades impostas por essa abordagem. Os problemas normalmente encontrados, e citados na literatura, apontam a adequação dos algoritmos ao hardware como fator crucial na obtenção do aumento de desempenho. Em inúmeros

casos há a necessidade do desenvolvimento de novas técnicas, modelos matemáticos e métodos que solucionem o problema e minimizem o tempo de execução dos programas (Foster (1995)) (Hoare (1985)) (Andrews; Schineider (1983)).

Muitas pesquisas têm sido elaboradas com o objetivo de automatizar o processo de paralelização de algoritmos seqüenciais já desenvolvidos. Entretanto, nesse processo de transformação, a intervenção de programadores experientes tem sido contundente, pois, além das necessidades de eventuais alterações dos algoritmos, problemas casuais com a comunicação, sincronismo ou desbalanceamento das cargas do sistema, podem provocar falhas durante a execução ou causar o efeito colateral de perda de desempenho. Essas pesquisas abordam especialmente os casos nos quais os algoritmos são implementados na linguagem FORTRAN e devem ser adaptados para execução em máquinas com arquitetura MIMD, incluindo o caso de memória virtual compartilhada (Zima; Chapman (1992)).

A necessidade de adequação do software ao hardware, encontrada inicialmente nos ambientes científicos de desenvolvimento, tem ultrapassado esses limites e provocado reflexos na especificação dos sistemas computacionais modernos. Nesses sistemas definem-se os estilos arquiteturais para o software (estilo: Fluxo de Dados; *call* e *return*; Componentes Independentes; Máquinas Virtuais; e os *Data-Centered Systems*), que delimitam o domínio para o qual o software é desenvolvido. De acordo com Tracz (1994), não há consenso ou padrão para a diagramação das arquiteturas do software; mesmo assim, elas podem ser descritas através de gabaritos capazes de expressar atributos dos componentes e das conexões que as compõem.

As pesquisas em programação paralela/concorrente apontam para um futuro análogo ao que tem ocorrido com a programação seqüencial, isto é:

- é necessário que sejam desenvolvidas e aprimoradas técnicas de programação paralela/concorrente;

- é necessário o desenvolvimento de ferramentas para auxiliar a programação. Essas ferramentas têm que evoluir em função dos modelos arquiteturais modernos, sustentados pelo desenvolvimento tecnológico.

As ferramentas para apoio à programação paralela devem considerar as necessidades intrínsecas dessa atividade, bem como estabelecer uma arquitetura de software com domínio específico, objetivando atender o requisito de obtenção de desempenho máximo do sistema. Nesse contexto, estabelece-se a relação de composição de um conjunto de programas adequados a um conjunto de arquiteturas. Para essa composição é necessário estabelecerem-se as relações entre os componentes do software (para definição da arquitetura do software) e a relação entre os componentes do hardware (para a definição da arquitetura do hardware), bem como a relação entre o modelo do software e o modelo do hardware.

Este trabalho aborda a definição e o projeto de uma ferramenta que visa auxiliar a programação paralela/concorrente.

1.2 Descrição do Problema

A motivação para a investigação de ferramentas de apoio ao desenvolvimento de programas paralelos é bastante diversificada. Em particular, o assunto abordado nesta tese está relacionado com a experiência e as dificuldades acumuladas pelo autor e demais elementos do Grupo de Sistemas Distribuídos e Programação Concorrente do Departamento de Ciências de Computação e Estatística do ICMSC-USP, ao longo do desenvolvimento e utilização de uma máquina paralela de arquitetura MIMD com memória distribuída, denominada Servidor de Processamento Paralelo (SPP) (Trindade; Santana (1991)). O SPP é uma máquina com arquitetura paralela baseada em Transputers, cujo acesso é estabelecido através de uma rede local de computadores (Soares; et al. (1995)).

Essa máquina paralela tem características de um sistema multi-usuário, e permite que no máximo quatro usuários compartilhem

paralelamente o banco de processadores. A configuração da arquitetura do hardware, para cada um dos usuários, é programável por software, através de uma rede de chaveamento. A rede de chaveamento opera como uma interface de comunicação programável que define a interconexão dos meios de transmissão (*links*), estabelecendo-se assim uma arquitetura para a execução da aplicação. Os serviços oferecidos pelo SPP ficam disponíveis através de chamadas a procedimentos remotos [AND94].

O ambiente de desenvolvimento de aplicações para o servidor é o "Transputer Development System - TDS" (Inmos (1990)) e a linguagem de programação adotada é a OCCAM2 (Jones (1988)). A linguagem de programação OCCAM (Jones (1987)) foi desenvolvida para explorar de maneira eficiente o Transputer e tem sua concepção baseada na linguagem CSP (Hoare (1978)). A especificação CSP define uma gramática formal para a implementação de programas paralelos/concorrentes. Ambas oferecem a possibilidade de construção de processos compostos e de granulosidade variada, dependendo do algoritmo desenvolvido para a aplicação. Esses recursos da linguagem garantem a exploração do paralelismo com diversos graus de granulosidade.

Para a confecção de um programa OCCAM o programador tem que se preocupar em definir todos os seus processos, a granulosidade de cada um deles e, talvez o mais importante, o mapeamento dos processos na arquitetura, pois esse pode ser um fator de desequilíbrio na medida do desempenho do sistema.

As técnicas de programação seqüenciais limitam a capacidade de uma grande parte dos programadores em OCCAM, pois além de terem que criar algoritmos paralelos, têm também que supor uma arquitetura adequada ao seu modelo e mapear os processos e canais de comunicação de seus programas nessa arquitetura. Muitas vezes torna-se necessário que os processos sejam transportados entre os elementos de processamento, visando a melhoria do desempenho do sistema. Esse trabalho pode ser difícil sem o auxílio de uma ferramenta de apoio (Baker; Malvem (1991)).

Além de ter sido a motivação inicial para o desenvolvimento deste trabalho, o SPP foi também a primeira plataforma considerada para o desenvolvimento da ferramenta a ser discutida ao longo desta tese. Entretanto, o desenvolvimento do trabalho deixou claro que devido à aproximação das pesquisas em redes de computadores/sistemas distribuídos/arquiteturas paralelas, o surgimento de novos paradigmas de programação e novas metodologias/técnicas para análise e projeto de sistemas, uma ferramenta como a pretendida nesta pesquisa, tem que ser o mais flexível possível no sentido de permitir que novas tecnologias, métodos ou paradigmas, possam ser facilmente incorporados ao sistema.

Esta tese mostra que é possível elaborar o projeto de uma ferramenta desse tipo, com a característica fundamental de que possa evoluir de acordo com as necessidades tecnológicas. O projeto estabelece um núcleo comum às ferramentas que auxiliam a programação paralela, fazendo com que as evoluções tecnológicas sejam tratadas como especializações de elementos abstratos. A orientação a objetos é utilizada como o paradigma básico para o desenvolvimento de todo o processo de modelagem envolvido no projeto da ferramenta.

Os conhecimentos adquiridos com a prática de implementação de programas paralelos escritos na linguagem OCCAM e no desenvolvimento de subsistemas servidores (Trindade; et al. (1992)) (Andrade; Santana (1994)) (Nascimento; et al. (1995)) (Santana; et al. (1996)), baseados em mecanismos de RPC (Wilbur (1987)), permitiram vislumbrar uma ferramenta para estender os conceitos utilizados no desenvolvimento de programas na linguagem OCCAM2, permitindo a programação em ambientes de passagem de mensagem (Sunderam; et al. (1994)) (Souza; et al. (1996)) ou para a geração de esqueleto para aplicações em outras linguagens, tais como CC++, FORTRAN M (Foster (1995)).

1.3 Uma Visão Macroscópica da Ferramenta em Estudo

A Figura 1.1 apresenta uma visão macroscópica do sistema proposto nesta tese, na qual podem ser identificados três níveis de abstração:

- O primeiro nível: “Banco de dados”, é uma interface entre o nível imediatamente superior e o dispositivo de armazenamento das informações da ferramenta.
- O segundo nível: “Componentes do Hardware”, “Componentes do Software” e “Estrutura para os Modelos Arquiteturais Propostos”, implementa classes de objetos para a descrição do hardware, do software e um núcleo para a descrição da aplicação, respectivamente. O subsistema “Estrutura para os Modelos Arquiteturais Propostos” implementa classes que modelam os conceitos da teoria dos grafos criando tipos abstratos de dados.
- O terceiro nível corresponde à interface da ferramenta. Os conceitos modelados e implementados são manipulados de acordo com a “visão” que se deseja proporcionar para o usuário.

Como concebido nesta visão macroscópica, o núcleo da ferramenta corresponde à implementação de um metamodelo para arquiteturas de computadores e, em um nível mais abstrato, correspondente às interfaces gráficas, estão os elementos que permitem a geração de novos modelos arquiteturais.

O subsistema “Componentes do Hardware” representa os elementos que podem ser organizados para compor uma arquitetura de computador, por exemplo: memória, unidade central de processamento, dispositivos para

entrada e saída de dados, ou outros. O subsistema “Componentes do Software” modela as linguagens que podem ser utilizadas para a programação nas arquiteturas. O subsistema “Estrutura para os Modelos Arquiteturais Propostos” implementa estruturas que permitem a descrição das arquiteturas de hardware e arquiteturas de software (esqueletos para as aplicações). A manipulação dessas estruturas é consistente com a descrição dos componentes. Esse núcleo pode ser utilizado através da edição de texto no formato ASCII, entretanto seria pouco amigável para o desenvolvimento de aplicações. Assim, foi incluído um terceiro nível, que representa a interface com os programadores. Essa interface tem por objetivo facilitar a descrição de arquiteturas paralelas, dos esqueletos das aplicações e auxiliar o mapeamento do esqueleto da aplicação no projeto do hardware, fornecendo para o programador o texto do esqueleto da aplicação na linguagem especificada.

O paradigma de orientação a objetos, adotado como base no projeto, tem sido apontado como uma possível solução para a crise de software, pois permite a reutilização, tanto em nível de projeto quanto de código. O projeto da ferramenta segundo o paradigma de orientação a objetos, visa reutilizar ao máximo os componentes. Para tal foram desenvolvidos estudos relacionados com as arquiteturas dos computadores, ambientes para a programação paralela e a teoria que oferece suporte para a modelagem matemática do problema. As características comuns às arquiteturas de computadores e às linguagens de programação são fatoradas e modeladas segundo a teoria dos grafos. Os conceitos relativos à teoria dos grafos são modelados segundo o paradigma de orientação a objetos.

O projeto da ferramenta especifica:

- Um conjunto de classe-base fundamentado na teoria dos grafos que constitui o núcleo para as aplicações.
- Um conjunto de classes que modela os componentes básicos. Esses componentes têm que ser utilizados para a especificação do hardware da arquitetura paralela.

- Um conjunto de classes-base que permite a descrição do esqueleto do programa para aplicações paralelas.
- Um subsistema gráfico que permite o mapeamento do esqueleto da aplicação no hardware especificado.
- Um conjunto de interfaces gráficas para a manipulação das demais estruturas.

1.4 Organização do Texto

Considerando-se que o desenvolvimento deste trabalho envolve o estabelecimento de modelos para teorias distintas, optou-se por apresentar no corpo da tese, a teoria com o respectivo modelo orientado a objetos, encerrados no mesmo capítulo, quando for o caso. O trabalho está organizado através de uma introdução (capítulo 1) e mais sete capítulos e um apêndice.

O capítulo 2 apresenta a revisão bibliográfica relativa às metodologias de Análise Estruturada de Sistemas, Análise Essencial e Análise Orientada a Objetos. O objetivo desse capítulo é mostrar que a orientação a objetos fornece subsídios para a elaboração de um projeto suficientemente flexível para acompanhar a evolução tecnológica do hardware.

O capítulo 3 refere-se à teoria dos grafos e seu respectivo modelo orientado a objetos, com a definição das classes-base necessárias.

O capítulo 4 trata dos conceitos e taxonomias para as arquiteturas e linguagens paralelas. São apresentadas, também, a estrutura de classes hierárquicas que modelam a taxonomia apresentada segundo o paradigma de orientação a objetos.

O capítulo 5 discute a interface estabelecida para a implementação básica da ferramenta e aponta diretrizes para o estabelecimento de uma interface gráfica, como uma possível extensão do ambiente proposto.

O capítulo 6 apresenta um protótipo para a ferramenta considerando um ambiente de programação paralela específico. As diretrizes básicas para

que a ferramenta possa ser estendida a outros ambientes de programação paralela são também estabelecidas.

O capítulo 7 conclui o trabalho, apresentando uma visão global dos itens discutidos ao longo da tese, relacionando as contribuições relevantes identificadas na pesquisa desenvolvida e apontando diretrizes para trabalhos futuros.

O capítulo 8 lista as referências bibliográficas utilizadas para o desenvolvimento de todo o trabalho.

O apêndice A apresenta a implementação das classes-base para os componentes das arquiteturas de computador e foi incluído com o intuito de oferecer uma documentação completa ao leitor interessado em maiores detalhes.

2. Análise e Projeto de Sistemas

As ferramentas de desenvolvimento de sistemas implementam técnicas que são aprimoradas com o passar do tempo, o que provoca a manutenção do sistema no sentido de atender aos novos requisitos inerentes à evolução. A questão é: qual método e técnica devem ser utilizados para o desenvolvimento de uma ferramenta de auxílio à programação paralela? Para responder a essa pergunta é conveniente analisar o contexto do desenvolvimento de sistemas das últimas décadas.

Este capítulo apresenta as evoluções das tecnológicas, com a finalidade de estabelecer parâmetros que permitam a escolha de um método para descrever o protótipo da ferramenta de auxílio à programação paralela.

2.1 Considerações Iniciais

No final dos anos 60 e início da década de 70, a concepção de um sistema estava baseada nos métodos de decomposição funcional (Warnier (1974)) (Jackson (1975)). Durante esse período as arquiteturas dos computadores eram "Single Instruction Single Data - SISD" (Flynn (1972)), em sua grande maioria os programas eram implementados nas linguagens COBOL ou FORTRAN, processados em lotes, utilizando sistemas operacionais do tipo "batch" (Tanenbaum (1992)) e havia muita preocupação com o gerenciamento dos espaços físicos ocupados por programas e dados. Os *mainframes*, utilizados na indústria durante esse período, tinham como tarefa principal manipular grandes volumes de dados com formatos semelhantes. A necessidade de compreensão da organização dos dados em arquivos, acrescido da necessidade de gerenciar tais informações, induziu pesquisas para que fossem desenvolvidos os modelos

de dados (Codd (1970)) (Codasyl (1971)) (Chen (1976)). A proliferação de linguagens, com suporte aos tipos de dados estruturados (Wirth (1986)), apresentou novas perspectivas e mostrou como as estruturas de dados podem afetar a modularidade, interferir nos algoritmos e alterar o desempenho dos sistemas.

No final da década de 70 e início dos anos 80, a difusão de novas técnicas de análise de sistemas (Demarco (1979)) (Gane; Sarson (1982)), e suas extensões, mostram que a análise e o projeto devem ser elaborados independentemente do modelo arquitetural da máquina na qual a aplicação (ou o sistema) será executado.

Nesse mesmo período fica sacramentado o conceito de *máquina virtual*, surgem os sistemas operacionais que implementam esse conceito e ocorre a proliferação das redes de computadores (Tanenbaum (1992)). Até então as arquiteturas paralelas eram utilizadas, em sua grande maioria, para aplicações científicas.

Diversas variações da técnica de Gane; Sarson (1982) foram criadas com a intenção de embutir mais semântica e para permitir especificações de sistemas de tempo real. Porém, a técnica base faz uso do modelo entidade-relacionamento (Chen (1976)) e de uma hierarquia de especialização sobre os processos (denominado Diagrama de Fluxo de Dados - DFD), até chegar no nível da decomposição funcional (ou Diagrama Hierárquico de Funções - DHF), não ficando explícita a maneira através da qual os processos são transformados em funções e não havendo preocupação com a arquitetura do hardware.

A Análise Estruturada Moderna (Yourdon (1990)) e a Análise Essencial de Sistemas (McMenamim; Palmer (1991)), diminuem a lacuna semântica existente entre o DFD e o DHF, através de uma abordagem que busca em primeira instância identificar as necessidades essenciais do sistema, ao invés da análise "top-down" proposta por Gane; Sarson (1982). Entretanto, a complexidade dos sistemas aumentou sensivelmente com a difusão comercial das redes de computadores e das arquiteturas paralelas, que trouxeram consigo a palavra de ordem "downsizing" para dentro das

empresas. Os sistemas existentes tiveram seus projetos e dimensões alterados para a nova realidade.

No início da década de 90, a proliferação das redes de computadores, com unidades autônomas e interligadas, contribuiu significativamente com a difusão de novos paradigmas de programação. Mesmo as empresas que não migraram seus sistemas para esse novo modelo arquitetural, implementaram alguns subsistemas baseados no modelo cliente/servidor (Müllender (1993)).

O barateamento do custo do hardware levou para dentro das casas os computadores pessoais, um mercado emergente a ser explorado e marcante nas décadas de 80 e 90. A idéia de interação direta com o usuário tem exigido que as aplicações tenham uma boa parte de seu custo de desenvolvimento voltado para essa interface. A combinação de todos esses fatores tem feito com que diversas empresas decidam-se por uma abordagem orientada a objetos (Rumbaugh; et al. (1991)) (Jacobson (1992)) (Coleman; et al. (1994)) (Martin; Odell (1996)). A Tabela 2.1 ilustra o desenvolvimento de linguagens de programação, técnicas de análise e projeto de sistemas, modelos de dados e das arquiteturas dos computadores.

A análise da evolução das arquiteturas apresentada Tabela 2.1 na mostra que a tendência é a utilização de arquiteturas paralelas. Embora existam problemas para a produção de programas paralelos, conforme abordados no capítulo 1, a tarefa de distribuição de dados e código em máquinas paralelas pode ser efetuada automaticamente pelo sistema. Essa distribuição pode ser efetuada de maneira estática ou dinâmica. Quando a necessidade principal de ser gerado um programa paralelo é minimizar o tempo de execução, a migração dos programas para outras arquiteturas é quase impraticável (Zhou; et al. (1993)), pois a solução do problema está associada à plataforma de desenvolvimento e a migração afeta sensivelmente o desempenho global do sistema (Calônego Jr. (1994)).

Tabela 2.1 - Evolução dos sistemas e métodos computacionais, Carmichael (1994).				
2000				Arquiteturas Maciçamente Paralelas.
1990	Geradores de Código Usando ferram. CASE. para Anál. Orientada a Objetos. Ambientes de passagem de mensagem. Ferramentas para conversão de programas seqüências em paralelos. Paradigma SPMD. Ling. de progr. Paralelas.	Bancos de Dados Orient. A Objetos (Versant/Objective)	Análise Orientada a Objetos Análise Estruturada	Sistemas Abertos. Arquiteturas paralelas. Discos óticos.
1980	Geradores de Código Associados a Ferram. CASE. usando análise estruturada. C++, Ling. de 4ª. Geração, SQL	Modelo Entidade Relacionamento Modelos Relacionais (DB2/Oracle)	Moderna Análise Essencial Análise Estruturada	Computação Paralela/ Concorrente Arq. RISC. Redes
1970	Smalltalk	Modelo Hierárq. Modelo de Rede (CODASYL)	Análise baseada na decomp. Funcional	Sistemas de Bancos de Dados
1960	Ling. de 3ª Geração (COBOL, FORTAN, PL/1)	Sist. de Gerenc. De Arquivos (ISAM / VSAM)		Unidades de Disco
1950	Linguagem de máquina			Arq. SISD
	Linguagens	Bancos de Dados	Análise de Sistemas	Hardware

Aqueles que têm a necessidade de aumentar significativamente o desempenho de programas, percebem o potencial que o paralelismo do código pode oferecer, e se depararam com as dificuldades impostas por essa abordagem. Os problemas normalmente encontrados, e citados na literatura, apontam a adequação dos algoritmos ao hardware como fator crucial na obtenção do aumento de desempenho. Em inúmeros casos há a necessidade do desenvolvimento de novas técnicas, modelos matemáticos e métodos que solucionem o problema e minimizem o tempo de execução dos programas (Foster (1995)) (Hoare (1985)) (Andrews; Schineider (1983)).

A complexidade dos sistemas desenvolvidos durante a última década, tem feito com que as equipes que desenvolvem os sistemas busquem reutilizar ao máximo o trabalho já efetuado. O aumento da reutilização do código em sistemas de software é uma maneira importante para que sejam diminuídos os custos de desenvolvimento e de manutenção. Isso reduz o tempo requerido para que novos requisitos sejam atendidos e amplia as possibilidades de implementação impostas pela complexidade de novos sistemas. “As linguagens de programação orientadas a objetos, em conjunto com os ambientes de desenvolvimento, oferecem suporte fundamental para a verdadeira reutilização de código, efetuada através dos tipos abstratos de dados, herança e polimorfismo. A reutilização de código é um pré-requisito para um conceito significativamente mais fundamental: reutilização de projeto” (Graver (1992)). Entretanto, a reutilização somente é viável quando o software é projetado para essa finalidade, ou seja, quando permite alterações mais profundas que aquelas exigidas durante a manutenção.

Diante das considerações tecidas anteriormente, fica claro que dentre as opções de métodos¹, ou paradigmas, desenvolvidos para a engenharia do software, a orientação a objetos é uma abordagem recomendável para o problema proposto no capítulo 1.

Os itens a seguir apresentam abordagens diferentes, segundo o paradigma de orientação a objetos. O objetivo é definir conceitos, notações e o método a ser utilizado para a geração de um modelo para a ferramenta de auxílio à programação paralela/concorrente.

2.2 Conceitos

O paradigma de orientação a objetos permite que os entes do mundo real sejam abstraídos através de seus atributos e de suas interações. Um método é dito orientado a objetos se esse oferecer suporte ao

¹ “Os métodos de Engenharia de Software proporcionam paradigmas para construir o software” (Pressman (1995)).

encapsulamento, à herança e ao polimorfismo. Essas três premissas da orientação a objetos são detalhadas nos sub-itens a seguir.

2.2.1 Encapsulamento

O encapsulamento ocorre quando o acesso aos dados é efetuado somente através de primitivas definidas no protocolo de acesso.

Encapsulamento é o processo de encerrar em capsula, conforme ilustra a Figura 2.1. Ao contrário dos demais paradigmas para as linguagens de programação, nos quais são definidas estruturas de dados que são passadas como argumentos de funções ou procedimentos, a programação orientada a objetos exige que os objetos sejam definidos juntamente com as mensagens as quais ele responde.

Figura 2.1 - Encapsulamento.

Diversos autores buscam explicar e exemplificar a terminologia para o paradigma de orientação a objetos. As definições de Penson (Penson; Richard (1992)) são bastante claras e serão utilizadas no contexto deste trabalho.

- **“Objeto:** é uma representação de um elemento do mundo real. Cada elemento do mundo real possui informações que o caracterizam univocamente e tais informações são encapsuladas através dos protocolos de acesso” (Penson; Richard (1992)) ;
- **“Mensagem:** é um identificador, com ou sem parâmetros, que informa ao objeto uma ação (ou um conjunto de ações) a ser efetuada” (Penson; Richard (1992)) ;
- **“Classe:** é um elemento abstrato de dados capaz de representar um conjunto de objetos. A classe, portanto, determina tipos de objetos” (Penson; Richard (1992));
- **“Instância:** A criação de um elemento a partir de um tipo de classe é o que se denomina objeto. Pode-se dizer que um objeto é uma instância da classe a qual pertence” (Penson; Richard (1992)) ;
- **“Método:** é a maneira através da qual é implementada uma mensagem” (Penson; Richard (1992)).

“Objetos são instâncias de classes que respondem a mensagens de acordo com métodos determinados pelo protocolo de descrição de classe. Os objetos também possuem variáveis de estado definidas no protocolo de descrição de classe”.

Os métodos públicos definem o protocolo de acesso aos objetos da classe. Os métodos protegidos somente podem ser utilizados por classes derivadas a partir dessa. Os métodos privados são utilizados apenas por outros métodos pertencentes a mesma classe, não estando disponíveis para métodos de classes derivadas (Papaz; Murray (1995)). Nem todas as linguagens de programação orientadas a objetos oferecem esses três níveis de proteção.

2.2.2 Herança

É a relação existente entre uma classe-mãe (também denominada super-classe, super-tipo ou base) e uma classe derivada a partir dela. A classe derivada pode também ser denominada subtipo, subclasse ou

classe-filha. A herança permite que sejam criadas estruturas hierárquicas de classes. Em uma hierarquia de classes há uma ou mais classes bases, também denominada(s) raiz(es), e diversas classes-filhas. A Figura 2.2 apresenta um exemplo de classes hierárquicas.

O mecanismo de herança permite que as definições da classe base sejam reutilizadas e modificadas, especializando sua especificação. A utilização da herança para a criação de uma nova classe a partir de uma já existente possui três fatores importantes: a inclusão de novas informações e códigos sem alteração da classe base; a reutilização do código; a alteração do comportamento das classes, por exemplo através da redefinição ou sobrecarga² dos métodos herdados.

2.2.3 Polimorfismo

As Instâncias das classes (ou objetos) representam o comportamento dinâmico do sistema. Os objetos podem “conhecer” os destinatários de suas mensagens. O *Polimorfismo* ocorre quando o elemento que envia a mensagem não “sabe” quem trata a solicitação. Polimorfismo significa, pelo menos no contexto de orientação a objetos, que as instâncias que transmitem não precisam “conhecer” as instâncias das classes receptoras.

Polimorfismo “significa que o transmissor de estímulos não precisa conhecer as instâncias das classes receptoras. A instância receptora pode pertencer a uma classe qualquer” (Jacobson (1992)).

²É o mecanismo de utilizar mesmo identificador para métodos diferentes.

2.3 Métodos Orientados a Objetos

Alguns métodos apresentam apenas heurísticas, que caracterizam as suas concepções e apresentam abordagens direcionadas por modelo de dados, especificação de eventos ou pelo ambiente no qual a aplicação será inserida. Essas abordagens caracterizam evoluções na tecnologia da orientação a objetos. Os itens que seguem apresentam alguns desses métodos.

2.3.1 Análise de Sistemas Orientada a Objetos

O método “Object-Oriented Systems Analysis - Modeling the Word in Data” (Shlaer; Mellor (1990)), traduzido com o título “Análise de Sistemas Orientada para Objetos” (Shlaer; Mellor (1990)), estabelece uma classificação para tipos de objetos. Para Shlaer e Mellor o significado de um objeto “é uma abstração de um conjunto de coisas do mundo real de forma que: todas as coisas do mundo real - as instâncias - tenham as mesmas características; todas as instâncias estejam sujeitas a, e em conformidade com as mesmas normas” (Shlaer; Mellor (1990)).

Essa concepção de *objeto* estabelece apenas as características dos objetos, representados através de seus atributos. A seleção das características, ou *atributo*, devem ser abstraídas segundo as regras de:

- *independência* (estabelece que os atributos são disjuntos),
- *fatoração* (cada atributo representa apenas uma característica do objeto),
- *completitude* (agregado de dados que engloba todas as informações que caracterizam o objeto).

Pode-se estabelecer uma analogia entre a definição de atributos para o modelo de análise proposto de objetos e atributos de entidades definidos no *Modelo Entidade-Relacionamento (M-ER)* (Chen (1976)). Ao observar as regras desse modelo, pode-se interpretá-las de maneira semelhante às formas normais. Para Shlaer e Mellor um relacionamento é “uma abstração de um conjunto de associações que subsiste sistematicamente entre espécies diferentes de coisas do mundo real”, seguindo os mesmos

princípios de *cardinalidade* do M-ER e definem a ligação (ou vínculo) estático entre os objetos do sistema. As informações relativas às associações entre objetos e/ou atributos são inseridas no modelo, de forma que esse estende o M-ER para a representação da semântica dos dados.

Há a preocupação dos autores em representar as características principais do paradigma de orientação a objetos, estabelecendo uma notação para a representação de abstração, encapsulamento e herança, acrescido de um diagrama de transição de estados que modela o comportamento dinâmico do sistema.

O método “Análise de Sistemas Orientada para Objetos” (Shlaer; Mellor (1990)) sofre grande influência dos modelos de dados semânticos. Os modelos de dados influenciam o desenvolvimento. O efeito cascata produzido pelo modelo de dados é abordado por Carmichael (1994) e são observáveis em todos os métodos de análise.

Segundo Coad; Yourdon (1992) a “Análise de Sistemas Orientada para Objetos” é um conjunto de heurísticas, pois não descreve um método de análise, mas sim regras para modelar objetos e seus comportamentos.

2.3.2 Análise Baseada em Objetos

O método de “Análise Baseada em Objetos” (Yourdon; Coad (1992)) sugere que os analistas identifiquem, em primeira instância, “*um campo de atividade sob estudo ou consideração*”, denominado *domínio do problema*. Estabelecido e compreendido o domínio do problema, deve ser estabelecido o escopo da aplicação, o qual os autores denominam “*responsabilidades do sistema*” e definem como “uma organização de elementos de modo a formar um todo” dentro do domínio.

São definidos 7 (sete) princípios para a gerência da complexidade do sistema, denominados “Princípios para a Administração da Complexidade”, que são:

- abstrações,
- encapsulamento,

- herança,
- associação,
- troca de mensagens,
- métodos organizacionais, escala
- categorias de comportamento.

Esses termos definem heurísticas para a geração de um projeto, com notação gráfica, que permitem definir agregação e especialização (J. Smith; D. Smith (1977)), troca de mensagens, classes, atributos, serviços (métodos) e diagrama de estados (comportamento dinâmico). A Tabela 2.3 ilustra os elementos gráficos que podem ser utilizados para compor projetos de sistemas orientados a objetos, sendo:

- *Classe*: o identificador (ou nome) da classe de objetos; a palavra *Atributos* é um conjunto de características dos elementos que pertencem ao conjunto *Classe*; *Serviços* são ações (ou operações) que podem ser efetuadas por objetos.
- *Classe&Objeto*: a notação para expressar a classe e os objetos instanciados a partir da mesma.
- *Conexão*: a associação (ou ligação) entre dois elementos.
- *Parte/Todo*: a relação de composição.
- *Herança*: as hierarquias de especialização e generalização.

Os autores admitem que o método é apenas “um ponto de partida para a aplicação da Análise Baseada em Objetos” e que o “método deve ser expandido de acordo com as necessidades do projeto ou da organização”(Yourdon; Coad (1992)) .

Estabelecidos os conceitos, jargão e notações, são propostas regras para a identificação de objetos, classes e serviços. A especificação dos serviços está baseada em dois tipos de diagramas: *diagrama de estados* e *diagrama de serviços*. Os estados são modelados por retângulos interligados por seguimentos de retas orientados, que indicam o estado atual e os possíveis estados futuros, caracterizando a máquina de estados finitos definida pela interação entre os objetos. Os serviços são modelados através dos símbolos ilustrados na Figura 2.4, onde o significado de cada um dos itens, da esquerda para a direita, é:

- *condição*: estabelece pré e pós condições para que um serviço seja executado;
- *bloco*: texto contendo parte da especificação do serviço;
- *loop*: define estrutura de repetição para as atividades;
- *Conector*: indica as possíveis seqüências para as atividades.

Observa-se a preocupação dos autores em não utilizar notações e regras vinculadas aos métodos de análise estruturada, essencial ou moderna de sistemas, mas sim, uma busca em estabelecer uma nova metodologia.

Uma característica marcante, e que os autores salientam, é a eliminação do maior número possível de documentos de análise, com o objetivo de facilitar a comunicação intra e extra-equipe. Segundo Coad; Yourdon (1992), suas experiências têm mostrado que documentos longos introduzem redundância e inconsistências, prejudiciais ao bom andamento do projeto.

Contudo, esse método não oferece substância para que sejam tratados os conceitos de *polimorfismo*, *funções virtuais* e *classes bases puras*. Em contrapartida, não há a restrição da impossibilidade de ocorrência de mais de um eventos num determinado instante, o que é uma característica importante para a especificação de programas paralelos.

2.3.3 O Método OMT

Para os autores do método OMT (Rumbaugh; et al. (1991)) a expressão *orientação a objetos* diz respeito à organização do software, tratando principalmente dos aspectos do comportamento e da independência entre si. O método revela-se pragmático para a obtenção de modelos de classes e utiliza um diagrama análogo ao diagrama Entidade-Relacionamento (Chen (1976)), para apresentar a estrutura das classes e das associações entre os objetos. A utilização do M-ER como base para a geração de um método orientado a objetos sofre as mesmas influências da Análise Baseada em Objetos (Yourdon; Coad (1992)), estendendo a notação para acrescentar maior semântica ao modelo, conforme ilustra a Figura 2.5. Cada item do modelo de classes em um diagrama de classes, gerado através da utilização do item (a) da Figura 2.5, pode ser substituído por objetos, item (b), para a obtenção do diagrama de objetos.

A Figura 2.6 ilustra o exemplo de um modelo de classe na qual um conjunto de aeronaves, compostas por pelo menos um pneu e zero ou mais turbinas, é base para a especialização de aeronaves que podem ser de carga ou espaçonaves. Esse modelo permite que seja elaborada a estruturação lógica dos dados, estendendo os conceitos do M-ER e acrescentando um mecanismos para a representação de herança. Contudo, não é suficiente para a representação do dinamismo associado aos objetos, nem das mensagens que fluem entre os objetos. Para isso são utilizados dois outros diagramas denominados *diagrama de fluxo* e *diagrama de eventos* (ou *diagrama de estados*).

O diagrama de fluxo de dados é utilizado para representar as transformações aplicadas aos dados, quando manipulados por objetos. O diagrama de fluxo de dados não possui informações de controle. Assim, deve ser gerado um outro grafo, para representar o comportamento da máquina de estados finitos.

Analisando-se o método pode-se identificar que ele agrupa técnicas bastante utilizadas na análise estruturada de sistemas, estende notações e aprimora as notações dos métodos de Shlaer; Mellor (1990) e Coad; Yourdon (1992), estabelecendo fases bem definidas para o processo. O fato do método apresentar muitas das características da Análise Estruturada pode ser fator positivo, pois isso diminui o tempo de aprendizagem.

2.3.4 O Método Objectory

O OBJECTORY ou *Object-Oriented Software Engineering-OOSE* (Jacobson (1992)) é um método bastante diferente dos demais, pois a análise busca modelar o sistema a partir do ponto de vista do usuário, em vez da análise da estrutura ou do comportamento do sistema. Esse tipo de análise, denominada *use-case-driven*³, utiliza três elementos básicos para compor o modelo de análise do sistema:

- objetos de controle, que coordenam as atividades do sistema;
- objetos de interface, que estabelecem as interações com os usuários;
- objetos entidades, que incorporam os conceitos da aplicação.

A intenção desse tipo de modelagem é manter a estrutura interna, o comportamento e a dinâmica do sistema independentes, de tal maneira que cada uma dessas partes possam ser alteradas ou expandidas independentemente umas das outras. A Figura 2.7 ilustra os símbolos gráficos utilizados para a representação do modelo de análise. Utilizando esses componentes é possível isolar a interface da aplicação, criando-se um elemento que ajuste a interface e a aplicação.

As classes são modeladas segundo os atributos, o comportamento e as partes. Dessa maneira o modelo dos entes do mundo real equivale à tarefa de classificá-los segundo suas características mais expressivas, estabelecendo quais são as ações (ou reações) desses entes a determinados estímulos e de quais partes eles são constituídos. Essa

³ “*caso de utilização*”. representa seqüências de interação do usuário com o sistema.

relação de constituição é importante pois define uma árvore hierárquica de abstrações ou uma relação de composição de agregação.

Para Jacobson (1992) *objeto* é definido como: “uma entidade capaz de guardar um estado (informação) e que ofereça um número de operações (comportamento/ações) para analisar ou modificar o seu estado. Um **objeto** é caracterizado por um número de operações e um estado que lembra os efeitos dessas operações”. O acesso ao objeto é efetuado apenas através de suas operações, pois a orientação a objetos tem como um de seus requisitos o encapsulamento. Note-se que externamente ao objeto observa-se apenas as operações para cada comportamento e não como essas operações são efetuadas. Pode-se observar *como* os objetos diferentes têm comportamentos diferentes, apenas olhando-se dentro deles, conforme ilustra a Figura 2.8.

Figura 2.8 - O objetos Luiz e suas operações.

Para cada informação, quaisquer associações a outros objetos também são especificadas. Os modelos de objetos têm relações uns com os outros. Por exemplo o objeto Luiz pode ser composto por sua cabeça, braços, pernas e seu corpo, conforme ilustra a Figura 2.9.

Através dessa **composição** pode-se estruturar o objeto Luiz em partes. A razão dessa estruturação depende de muitos fatores, mas sempre depende do desejo de descrever detalhes do objeto, aumentando a sua compreensão e obtenção de partes reutilizáveis.

Um método similar de juntar partes diferentes é através do uso de **partições** hierárquicas. Isto significa que um objeto pode ser construído a partir de outros objetos. Essa relação é normalmente denominada **relação consiste-de**.

Uma maneira similar de mostrar a associação existente entre os entes, é mostrar o uso da **agregação**. As palavras hierarquia e agregação são utilizadas como sinônimos em algumas ocasiões, mas, Jacobson define muito bem a diferença: "a palavra *agregar* vem do Latim *aggregare* (verbo), cujo significado é *colocar junto*, e é o oposto de particionar". Como pode haver alguma dificuldade em expressar esse agrupamento de uma maneira simples, um novo objeto e uma partição hierárquica podem ser usados para expressar a agregação.

Essa abordagem proposta por Jacobson (1994) permite que o tratamento dado a alguns relacionamentos, utilizando o significado da palavra relacionamento definido para o M-ER, seja modelado na hierarquia das classes. Em um relacionamento familiar, os grupos de Homens, Mulheres e crianças estabelecem a agregação Família. Como não se pode representar esse relacionamento triangular, um objeto Família é adicionado para expressar esta junção de vários objetos, conforme ilustra a Figura 2.10. Assim, o objeto "representa" a agregação, mas ele não é por si só a agregação. Uma agregação é a união de vários objetos e essa união é muitas vezes representada por um objeto.

A visão interna de um objeto mostra a estrutura de suas informações e como os objetos operam. Podem-se observar os **atributos** que os objetos precisam armazenar, as partes das quais o objeto consiste e como o comportamento para as operações é definido, conforme ilustra a Figura 2.11. Isso faz com que os objetos ofereçam suporte ao conceito de **ocultação de informação**, isto é, eles escondem suas estruturas internas de seus vizinhos⁴.

Cada uma das operações dos objetos fazem parte do comportamento do objeto e podem modificar as informações nele contidas. Para usar um objeto não é necessário que se saiba o seu comportamento ou como as suas informações são representadas internamente, faz-se necessário apenas conhecer quais operações eles oferecem. Além de modelar a ocultação de informações, o OBJECTORY permite a modelagem da hierarquia de classes, do polimorfismo e dos aspectos dinâmicos do comportamento dos objetos instanciados a partir de uma classe.

O desenvolvimento do sistema é efetuado através de fases e sub-fases, analogamente ao que ocorre nos demais modelos, conforme ilustram a Figura 2.12 e a Figura 2.13, respectivamente. Os modelos utilizados são:

- o **modelo de requisitos** tenta capturar os requisitos funcionais;
- o **modelo de análise** objetiva dar ao sistema uma estrutura de objetos robusta e fácil de alterar;
- o **modelo de projeto** adapta e refina a estrutura de objetos para o ambiente de implementação;
- o **modelo de implementação** visa implementar o sistema;
- o **modelo de teste** visa realizar uma verificação do sistema;

O **modelo de requisitos** conta com a participação ativa dos usuários, pois deve representar a expectativa do usuário com relação ao sistema. Quando ocorre a estabilidade do modelo⁵ as especificações produzidas são re-estruturadas gerando o **modelo de análise**. A reestruturação é efetuada por uma equipe de analistas com a finalidade de

⁴ Vizinhos: objetos que co-existem com o objeto em questão.

apresentar uma perspectiva lógica do sistema para garantir a sua manutenção e robustez (Pressman (1995)).

O modelo de análise está dividido em três fases que permitem definir a interface do sistema, o núcleo da aplicação e o subsistema que desvincula a aplicação da interface, conforme ilustra o item Análise, Figura 2.13. A necessidade de retomar-se o modelo de objetos, após ter sido estabelecido o modelo funcional, é para garantir a robustez e a capacidade de manutenção do sistema. Quando esse modelo parece estar suficientemente estável passa-se para a fase de projeto.

O **modelo de projeto** deve possuir poucas variações, especializando apenas os aspectos intrínsecos à implementação. O refinamento e formalização do projeto, resultantes do **modelo de projeto**, estabelecem as diretrizes do **modelo de implementação**. A partir desse ponto o sistema é gradualmente implementado em alguma linguagem de programação. Finalmente, o **modelo de teste** é desenvolvido para permitir a verificação do sistema desenvolvido. Este envolve principalmente a documentação das especificações e resultados de testes.

Figura 2.13 - Subfases da análise e do projeto.

⁵Estabilidade do modelo significa haver consenso entre usuários e analistas, não havendo um ponto determinado que defina onde deve terminar essa fase da modelagem.

2.3.5 O Método Fusion

O método *Fusion* (Coleman; et al. (1994)) oferece uma abordagem sistemática que busca congrega e estender as boas características de outros métodos. Para atingir esse objetivo o método estabelece três fases:

- *fase de análise*,
- *fase de projeto*,
- *fase de implementação*;

havendo em cada uma das fases sub-fases que refinam cada uma das etapas.

A utilização do método pressupõe a existência de um documento de requisitos do sistema, que deve ser produzido pelo cliente do sistema. A *fase de análise* é, portanto, um instrumento para capturar a essência do sistema, gerando modelos que forneçam uma descrição formal, porém declarativa, das ações e reações do sistema como um todo. Nessa fase são definidas as classes dos objetos que irão compor o sistema, o relacionamento entre elas, as operações e as possíveis seqüências. O método, em analogia ao OBJECTORY (Jacobson (1992)), subdivide a *fase de análise* em duas subfases: *modelo de objetos*, *modelo de interface*.

O *modelo de objetos* utiliza notação semelhante ao M-ER, analogamente ao método OMT (Rumbaugh; et al. (1991)), para representar a semântica dos elementos que compõem o sistema, representando suas características e suas associações. A Figura 2.15 ilustra os símbolos gráficos, definidos para o método Fusion, que permitem a representação estática do sistema.

O exemplo ilustrado na Figura 2.6, utilizando o método OMT, está representado através da Figura 2.14, utilizando o método Fusion. O exemplo ilustra aeronaves que podem possuir zero ou mais turbinas, de 0 (zero) até 18 (pneus) e que aeronaves de carga, objetos da classe ArCarga, *transportam* objetos da classe Carga. A notação fica mais clara que aquela apresentada na Figura 2.6.

Analogamente ao M-ER, os relacionamentos podem possuir atributos

e identificadores, denominados *papéis*, que são adicionados ao modelo de objetos para esclarecer as ligações utilizadas nos relacionamentos. É importante notar que a agregação permite a omissão de relacionamentos entre classes, e pode ser utilizada para apresentar uma visão macroscópica mais clara do todo, conforme ilustra a Figura 2.16 (Coleman; et al. (1994)).

O paradigma de orientação a objetos tem como propriedade a comunicação entre objetos, através de troca de mensagens. Essa propriedade é modelada no Fusion, assim como em outros métodos, através de um diagrama de estados, denominado *Modelo de Interface*.

Figura 2.16 - Exemplo de composição e agregação.

O *modelo de interface*, analogamente ao Objectory, modela o sistema como uma entidade ativa que interage com outras entidades (no caso do Objectory são os “actors”), para o Fusion são os *agentes*. Os *agentes* causam *eventos* que provocam as *operações* no sistema. Para cada operação há um conjunto de especificações para pré e pós condições que garantem a integridade do sistema. Essas especificações foram introduzidas no modelo como características adquiridas dos *métodos formais*. O *modelo de interface* é dividido no *modelo de operações* e no *modelo de ciclo de vida*.

O *modelo de operações* tem por objetivo determinar as possíveis ações e reações da interface do sistema. Para isso é gerado um documento, descritivo e em linguagem natural, no qual são especificadas as condições atuais do sistema, o evento⁶ e as condições posteriores ao tratamento do evento, conforme ilustra a Tabela 2.2. As definições das operações não descrevem o algoritmo para tratar a solicitação, é apenas uma caixa preta que efetua uma solicitação mantendo o sistema íntegro. Assim, o modelo de operação não diz nada relativamente aos estados intermediários, através dos quais o sistema passa enquanto a operação está ativa. Esses estados

⁶ “Unidade instantânea e atômica de comunicação entre o sistema e o ambiente” (Coleman; et al. (1994)).

são dependentes da maneira pela qual são implementadas as operações e não podem ser legítimas na fase de análise.

Tabela 2.2 - Gabarito para o modelo de operações.

Operation:	Nome - identificador da operação.
Description:	Texto - Descrição da operação.
Reads:	Itens - Argumentos de entrada.
Changes:	Itens - Argumentos de saída.
Sends:	Agentes/Eventos
Assumes:	Condições iniciais.
Results:	Condições finais.

A expressão ciclo de vida, não tem o mesmo significado que a expressão utilizada na análise estruturada. A expressão, no contexto do método Fusion, “define seqüências permissíveis de interação que o sistema pode participar durante o seu tempo de vida. Se a qualquer instante o sistema recebe um evento que não está de acordo com o seu ciclo de vida, então o sistema ignora o evento e mantém o seu estado inalterado” (Coleman; et al. (1994)).

A Tabela 2.3 apresenta a sintaxe e a semântica das expressões do ciclo de vida. Assim como numa linguagem de programação as expressões

do ciclo de vida definem autômatos finitos que descrevem a interação do sistema com o ambiente.

Tabela 2.3 - Sintaxe e semântica da linguagem para a descrição do ciclo de vida.

<ul style="list-style-type: none">• <i>Alfabeto. Qualquer evento de entrada ou saída que possa ser usado em uma expressão.</i> <i>Eventos de saída são prefixados com #.</i>• <i>Operadores. Sejam x e y expressões ciclo de vida, então</i> <i>x.y denota que x é seguido por y.</i> <i>x y denota que x e y são mutuamente exclusivos.</i> <i>x* denota a ocorrência de x zero ou mais vezes.</i> <i>x+ denota a ocorrência de x uma ou mais vezes.</i> <i>[x] denota que x é opcional.</i> <i>x y significa a intercalação arbitrária de x e y.</i>• <i>Substituições. Uma expressão pode ser nomeada para substituições:</i> Nome = expressão <i>Nome pode ser utilizado em outras expressões, mas as substituições não podem ser recursivas.</i>• <i>Precedência de Operadores. Em ordem decrescente de precedência:</i> [] * + , . . <i>As operações podem ser parentizadas para alterar a precedência padrão" [COL94].</i>

O comportamento do sistema é descrito pelo ciclo de vida e pelo modelo de operação juntos, conforme ilustra a Figura 2.17 - Pré-condições x Ciclo de vida, Coleman; et. al. (1994). (Coleman; et al. (1994)).

Ciclo de vida \ Pré-condições	Verdade	Falso
	efetuar ignorar	indefinido ignorar
Aceitar		
Rejeitar		

Figura 2.17 - Pré-condições x Ciclo de vida, Coleman; et. al. (1994).

2.4 Programação Orientada a Objetos

A maneira através da qual a Orientação a Objetos viabiliza a *abstração do mundo real*, aliada à percepção natural do ser humano, é

que tem tornado tão difundida essa técnica. A Engenharia de Software tem mostrado que um aspecto importante da programação envolve a comunicação de um fenômeno através da compreensão e descrição do mesmo, dentro de um domínio. A POO⁷ permite que entes do mundo real sejam abstraídos e associados a aplicações dentro de um domínio.

Um problema bastante conhecido da Engenharia de Software é a possibilidade de reutilização de módulos já existentes para construir outros novos. A funcionalidade dos componentes existentes nos sistemas são similares às necessidades dos sistemas novos. Existem, entretanto, muitas diferenças que tornam impossível a reutilização dos componentes existentes. Por essa razão, os componentes novos normalmente são implementados copiando-se os módulos já existentes e modificando-os. Isso significa que esses módulos novos têm que ser testados novamente. Problemas maiores podem ser observados quando se deseja modificar partes do sistema que são afetadas por ambos os módulos ou quando encontram-se erros em algum dos módulos. As linguagens orientadas a objeto trazem como benefício os *construtores*, que permitem modificações incrementais dos programas através da definição um novo componente, a partir da extensão de um outro já existente, preservando a relação entre eles.

“A orientação a objetos tem se mostrado eficaz quando aplicada com as finalidades de se obter abstrações do mundo real e consiste numa técnica de projeto estável que permite a reutilização de código” (Madsen; et al. (1993)).

Programação Orientada a Objeto é uma expressão que tem sido utilizada para definir um novo paradigma para as linguagens de programação. O paradigma estabelece que as soluções dos problemas devem ser implementadas de tal maneira que os objetos possam trocar informações através do envio de mensagens. Esse conceito contrasta com a programação estruturada, na qual passam-se estruturas de dados como

⁷POO - Programação Orientada a Objetos

parâmetros para procedimentos bem definidos (Wilbur (1987)). Os itens a seguir apresentam conceitos relacionados ao paradigma de POO.

O paradigma para as Linguagens de Programação Orientadas a Objetos (LPOO) estabelece cinco componentes elementares:

“Objeto, é uma abstração encapsulada que inclui informações de estado e um conjunto claramente definido de protocolos de acesso (correspondendo às mensagens que o objeto responde). Um objeto é caracterizado por seus atributos, os quais representam (ou modelam) as propriedades de algum fenômeno do mundo real que se deseja modelar.

Mensagem: é um símbolo, identificador ou palavra(s) chave(s), com ou sem parâmetros, que representa uma ação a ser praticada pelo objeto.

Classe: representa um tipo específico de objeto e é definida pela descrição de classe, que identifica tanto as variáveis de estado quanto os protocolos de acesso para um objeto da classe.

Instância: os objetos são instâncias de uma classe. As propriedades de quaisquer instâncias (objetos) são determinadas pela descrição da classe à qual o objeto pertence.

Método: existe um método para cada mensagem definida para cada objeto de qualquer classe. O método define como deve ser implementada uma mensagem para um objeto. Tipicamente, consiste em uma série de expressões objeto-mensagem e podem usar protocolos de outras classes.

Propriedades mensuráveis podem variar com o tempo. Um objeto deve permitir a representação dessa variação em função do tempo. Assim, diz-se que um objeto tem associado a ele um *estado* que varia em função do tempo” (Winer; Pinson (1991)).

2.4.1 Encapsulamento

Abstração é o ato de separar mentalmente um ou mais elementos de uma totalidade complexa.

Encapsular é o processo de incluir ou encerrar em cápsula. Em linguagem de programação orientada a objetos a unidade de encapsulação é o objeto. A cápsula ou objeto inclui entidades específicas como os dados de estado e os protocolos de mensagens que os objetos respondem.

Todas as propriedades de uma abstração, incluindo valores das abstrações de dados (variáveis de estado), estão encapsuladas em um objeto. Assim, falamos de objetos como se fossem encapsulações de abstrações.

Os dois componentes essenciais dos objetos como encapsulações de abstrações são determinados:

- pelas definições das abstrações (definições do campo de dados e da função membro contidos no protocolo de descrição da classe),
- pelo estado interno (valores reais dos campos de dados de um objeto).

Um objeto caracteriza-se por sua descrição de classe. A descrição de classe define todos os dados de estado de um objeto e todas as mensagens (como funções membros - métodos) que um objeto pode responder.

Podem-se oferecer várias implementações das funções membros de uma descrição de classe em arquivos separados e compatíveis. A implementação apropriada será selecionada no momento da compilação ou *link* com apenas uma versão por programa. Normalmente o vínculo entre os métodos é estabelecido no *link*.

2.4.2 Herança

O conceito de herança, em geral, é descrito em termos de classes biológicas. Uma subclasse herda propriedades de todas suas classes superiores e normalmente possui também suas próprias características. Essa abordagem de herança tem *especialização crescente* nos níveis das classes mais inferiores e mais genéricos nas classes de níveis superiores.

As classes organizadas hierarquicamente estabelecem os conceitos de *subclasses*, correspondendo às classes das classes inferiores. As subclasses herdam as propriedades das *superclasses* que correspondem às classes superiores a ela na hierarquia. Em algumas linguagens de programação orientadas a objetos as classes são tratadas como objetos.

Quando uma classe é derivada de uma única cadeia hierárquica, diz-se que há herança simples e a classe derivada herda as propriedades de todos os seus ancestrais. No caso em que uma classe pode ser derivada de mais de uma cadeia hierárquica, diz-se que há herança múltipla.

2.4.3 Polimorfismo

Em linguagens interpretadas, como Smalltalk, Sather e Eiffel, o método a ser usado para uma mensagem está vinculado ao objeto no momento da execução. Esse *vínculo tardio* oferece flexibilidade máxima para o envio de mensagens a objetos em detrimento da eficiência. O vínculo tardio exige consultas a tabelas ou cálculo de deslocamento do código. As linguagens compiladas, como C++, Smalltalk, Sather, Eiffel e outras, executam um vínculo precoce, o método é vinculado ao objeto em tempo de compilação. O vínculo tardio em funções são efetuados através da declaração de uma função *virtual*. A Tabela 2.4 apresenta algumas das principais características das linguagens orientadas a objetos.

Tabela 2.4 - Linguagens de programação orientadas a objetos.					
	<i>Coleta de lixo</i>	<i>Herança</i>	<i>Vínculo tardio</i>	<i>Compilada</i>	<i>Tratamento de exceção</i>
<i>C++</i>	<i>Não</i>	<i>Múltipla</i>	<i>Sim</i>	<i>C</i>	<i>Sim</i>
<i>Objective-C</i>	<i>Não</i>	<i>Simples</i>	<i>Sim</i>	<i>C</i>	<i>Não</i>
<i>Turbo Pascal</i>	<i>Não</i>	<i>Simples</i>	<i>Sim</i>	<i>C</i>	<i>Não</i>
<i>Parasol</i>	<i>Não</i>	<i>Simples</i>	<i>Sim</i>	<i>C</i>	<i>Não</i>
<i>C+@</i>	<i>Sim</i>	<i>Múltipla</i>	<i>Sim</i>	<i>C</i>	<i>Não</i>
<i>Liana</i>	<i>Sim</i>	<i>Simples</i>	<i>Sim</i>	<i>I</i>	<i>Não</i>
<i>Smalltalk</i>	<i>Sim</i>	<i>Simples</i>	<i>Sim</i>	<i>I/C</i>	<i>Não</i>
<i>Eiffel</i>	<i>Sim</i>	<i>Múltipla</i>	<i>Sim</i>	<i>I/C</i>	<i>Sim</i>
<i>Sather</i>	<i>Sim</i>	<i>Múltipla</i>	<i>Sim</i>	<i>I/C</i>	<i>Sim</i>
<i>Beta</i>	<i>Sim</i>	<i>Simples</i>	<i>Sim</i>	<i>C</i>	<i>Sim</i>
<i>Ada</i>	<i>Não</i>	-----	---	<i>C</i>	<i>Sim</i>

C - Linguagem compilada

I - Interpretada

O tratamento de exceção é um elemento que, apesar de impor sobrecarga na execução dos programas, é importante no sentido de evitar o término incondicional da execução.

2.5 Considerações Finais

Há alguns pontos a serem ponderados relativos às linguagens de programação e aos métodos de análise apresentados neste capítulo.

A análise histórica mostra que as linguagens de programação precedem o desenvolvimento de métodos e técnicas para o desenvolvimento de sistemas. Com relação à orientação a objetos isso não tem sido diferente. Como representar em uma estrutura hierárquica de especializações uma classe derivada de duas outras, porém com ascendente único, conforme ilustra a Figura 2.18 - Estrutura hierárquica ambígua.. No exemplo dado as classes B e C herdam todas as características da classe A e, conseqüentemente, a classe D herda duas vezes essas mesmas características.

Figura 2.18 - Estrutura hierárquica ambígua.

Há casos nos quais a necessidade de haver duas características é real. Por exemplo, suponha-se o conjunto dos números reais $\mathbb{R}$, conjunto dos números imaginários I e o conjunto dos números complexos. Os números complexos podem ser modelados como uma especialização de $\mathbb{R}$ e de I , e o conjunto I como uma especialização de $\mathbb{R}$. Assim, todo número complexo herda duas vezes as características de $\mathbb{R}$. Por outro lado, há casos nos quais a classe base deve ser única. Por exemplo, a classe dos animais especializada nas subclasses répteis e mamíferos. Assim a classe

dos ornitorrincos não pode herdar duas vezes as caraterísticas da classe animais.

A linguagem de programação C++ oferece suporte para esse tipo de problema, implementando o conceito de classe virtual, conforme ilustra a Tabela 2.5. A classe virtual faz com que as características da classe base sejam herdadas apenas uma vez.

Tabela 2.5 - Utilização de classe virtual em C++.

class Repteis : public Animais { ... };
class Mamiferos : public Animais { ... };
class Ornitorrinco : virtual public Mamiferos, public Repteis { ... };

Outro ponto que deve ser fortemente considerado é a experiência dos autores Coad; Yordon (1992), que ao proporem o método Análise Baseada em Objetos, deixaram bastante clara a visão de que o método não deve gerar pilhas de documentos que acabam gerando inconsistências. “O modelo deve auxiliar na compreensão do problema”.

Os métodos desenvolvidos tendo como base extensões do M-ER, não tecem comentários relativos a obtenção de conjuntos disjuntos gerados por hierarquias de especialização/generalização. Os modelos gerados através do M-ER normalmente levam a estruturas que caracterizam a 3ª. forma normal, eliminando das t-uplas as possíveis redundâncias. O método Fusion é pragmático nesse ponto e oferece notação específica para especializações/generalizações não disjuntas. De acordo com os outros métodos apresentados, as características comuns devem ser agregadas para compor uma classe mais abstrata, ou superclasse.

Considerando-se a ordem cronológica com que os métodos foram desenvolvidos e as notações utilizadas, o método Fusion representa uma boa opção, por se tratar da síntese de características de outros métodos.

Os capítulos seguintes utilizam a notação do método para representar os modelos que constituem o protótipo da ferramenta.

O problema original, proposto no capítulo 1, está desmembrado em três outros domínios: modelos para autômatos, modelos para arquiteturas paralelas e interface para manipulação de máquinas de estados finitos, os quais serão apresentados nos capítulos 3, 4 e 5 respectivamente.

3. Um Modelo Orientado a Objetos para a Teoria dos Grafos

A informatização dos sistemas existentes no mundo real, requer a criação de modelos matemáticos que representem conceitualmente a abstração desejada, a qual pode ser trabalhada e transformada em um programa de computador. Há uma gama de aplicações que podem fazer uso da teoria dos grafos para a geração de modelos formais, por exemplo: as linguagens de programação, as redes de computadores, as arquiteturas de computadores, entre outros.

A análise das taxonomias das arquiteturas (Steen (1993)), das aplicações para essas arquiteturas e da teoria de orientação a objetos, sugere que os conceitos desenvolvidos para a teoria dos grafos sejam modelados segundo o paradigma de orientação a objetos. Esses conceitos podem ser especializados, ou pode ser elaborada uma hierarquia de abstrações para representar os conjuntos de arquiteturas e suas aplicações.

No item 3.1 são apresentados os conceitos e as definições relativos à teoria dos grafos. No item 3.2 são propostos os respectivos modelos, segundo o paradigma de orientação a objetos.

3.1 Conceitos, Definições e Operações

Desde o primeiro problema que envolveu conceitos matemáticos relativos à teoria dos grafos, cuja solução foi elaborada por Euler¹, o desenvolvimento dessa teoria favoreceu diversas nomenclaturas e notações (Malkevitch; Meyer (1974)) (Boaventura (1979)). Como não há consenso, nem um dicionário que padronize essas notações e nomenclaturas, serão

¹ Leonhard Euler - Matemático suíço que, ao solucionar o problema das pontes de Königsberg em 1736, introduziu os conceitos para o desenvolvimento da teoria dos grafos.

apresentados alguns conceitos, definidas notações e nomenclaturas para este trabalho, similares às encontradas na literatura (Tenenbaum; et al. (1995)).

3.1.1 Nós e Arcos

Um nó pode representar um elemento do mundo real ou um conjunto de características comuns a vários elementos. Por exemplo: todos os cidadãos têm o seu respectivo *nome*, todas as cidades têm o seu respectivo nome, a cidade de *Uchoa*, a cidade de *Jataí*, o cidadão *João* e o cidadão *José*, o conjunto de cidades *Cidade* e o conjunto de cidadãos *Cidadãos*, podem ser representados por seis nós, um para cada um desses entes. Nesse exemplo, notam-se que os nós designam um conjunto ou cada um dos elementos do conjunto, conforme ilustra a Figura 3.1. Isso significa que os nós são abstrações e as semânticas dessas abstrações têm que ser muito bem definidas.

No exemplo ilustrado na Figura 3.1, o conjunto Cidadão generaliza as características dos elementos que possuem um nome. Assim, *João*, *José*, *Uchoa*, *Jataí*, podem ser elementos pertencentes a qualquer um dos dois conjuntos, *Cidade* ou *Cidadão*. Por exemplo, uma cidade denominada “João”.

A Figura 3.1 expressa, ainda, duas outras características comuns a todos os elementos, que são:

- o fato de que cada um deles é um nó;
- a representação gráfica desses elementos.

Assim, é conveniente que um nó seja compreendido através de três partes:

- o conceito intrínseco aos nós, fundamentado na teoria dos grafos;
- a abstração do(s) ente(s) do mundo real;
- a interface visual, que explicita de maneira clara e objetiva o significado da abstração. Suponha-se que:

⇒ uma elipse seja a interface para representar o conjunto das cidades, Figura 3.2a;

⇒ um círculo represente elementos do conjunto cidades, Figura 3.2c e Figura 3.2d;

⇒ um retângulo represente o conjunto de cidadão, Figura 3.2b;

⇒ um losango represente elementos do conjunto cidadãos, Figura 3.2e e 3.2f. Isso elimina ambigüidades e faz com que a interface visual represente a noção exata do significado dos elementos abstraídos, ou modelados, a partir do mundo real.

Analogamente ao conceito de nó, um arco pode representar a associação entre elementos do mundo real ou o conjunto dessas associações. Por exemplo, *João* “conhece” *José*, estabelece que associação “conhece” está relacionada aos dois elementos em questão, mas poderia haver a generalização de que todos os elementos pertencentes ao conjunto Cidadão conhecem uns aos outros. Novamente, é necessário estabelecerem-se regras que definam a interface, não ambígua, para diferenciar associação entre elementos e entre conjuntos. Portanto, os nós podem ser utilizados para modelar entes do mundo real (ou conceitos) e os

arcos podem representar as suas associações. A interface visual atribui semântica aos componentes do sistema, modelados através da teoria dos grafos.

Definição: o conjunto *No* é um conjunto não vazio que representa elementos abstraídos do mundo real. Todo elemento pertencente ao conjunto *No* tem como característica o número de associações com outros nós.

Definição: o conjunto *Arco* é um conjunto que representa as associações entre elementos que pertencem ao conjunto *No*.

3.1.1.1 Identificadores e Rótulos para Nós e Arcos

Independentemente do objetivo para o qual deseja-se a utilização de nós e arcos, é necessário que cada um dos elementos seja identificado univocamente, ou seja, cada nó e cada arco possui o seu respectivo identificador.

Notação: O conjunto de nós com os respectivos identificadores é denotado por:

$$No = \{n_1, n_2, \dots, n_m\}$$

e representa o conjunto formado por $n_i \in No \mid i, m \in \mathbb{N} \text{ e } 0 < i \leq m$, onde n_i é o identificador do i -ésimo nó. A associação entre dois nós $i, k \in No$, representada por intermédio de um arco, é denotada por um par $\langle n_i, n_k \rangle$. A introdução de identificadores para os arcos é denotada por $j \langle i, k \rangle$ e significa que o j -ésimo arco associa o i -ésimo nó ao k -ésimo nó.

Na maior parte das aplicações, cuja base do modelo é a teoria dos grafos, é necessária a utilização de rótulos para identificar os nós ou os arcos. O intuito não é modelar esse conjunto de aplicações, mas sim identificar os elementos pertencentes aos conjuntos *No* a *Arco* de maneira única, e não por intermédio de uma relação de ordem no conjunto. Por exemplo, o conjunto $No = \{1, 2, 3, 4, 5\}$ é o mesmo que $No = \{5, 1, 4, 3, 2\}$ e o conjunto $Arco = \{1 \langle 1, 2 \rangle, 2 \langle 2, 3 \rangle, 3 \langle 1, 5 \rangle, 4 \langle 2, 5 \rangle, 5 \langle 3, 4 \rangle, 6 \langle 2, 4 \rangle, 7 \langle 4, 5 \rangle\}$ é igual ao conjunto $Arco = \{2 \langle 2, 3 \rangle, 1 \langle 1, 2 \rangle, 3 \langle 1, 5 \rangle, 5 \langle 3, 4 \rangle, 6 \langle 2, 4 \rangle, 7 \langle 4, 5 \rangle, 4 \langle 2, 5 \rangle\}$.

É importante salientar que o identificador de um nó não é o rótulo associado ao mesmo, mas sim um atributo associado a cada um dos elementos. Esse atributo é utilizado para controle, sendo definido e gerenciado automaticamente pelo sistema, e faz com que os elementos nunca sejam idênticos. Por exemplo, imagine-se uma rede de computadores com máquinas idênticas, interligadas e que utiliza o rótulo para representar os equipamentos que possuem as mesmas características. Assim, a utilização de um identificador permite o reconhecimento exato de cada um dos equipamentos. Caso não houvesse a definição de identificadores únicos, máquinas distintas, porém com a mesma característica, não poderiam ser diferenciadas.

Definição: um rótulo é um conjunto R formado por elementos $r_j \in R \mid j \in \mathbb{N}$ e $\text{Rotulo}(n_i)$ é uma relação cujo domínio é No ou $Arco$ e cuja imagem é R .

Essa definição é a proposta do modelo de dados Modelo Relacional (Chen (1976)) acrescentando-se a característica de que cada relação possui uma chave única (o identificador do objeto), o que restringe as operações sobre relações para operações sobre funções, ou seja, cada elemento do domínio associa um único elemento na imagem.

Notação: $R(n_i) = \{r_1, r_2, r_3, \dots, r_m\}$, onde m é o número de rótulos válidos do conjunto e r_i é o i -ésimo rótulo.

Por exemplo, suponhamos os conjuntos $No = \{1, 2, 3\}$, $Arco = \{1<1,2>, 2<2,3>\}$ e as relações $R_1(n_i)$ e $R_2(n_i)$ definidas Tabela 3.1.

O conjunto *Rotulo* (R_1) relacionado ao conjunto No é composto por dois elementos, $R_1 = \{\text{"Livro"}, \text{"Caderno"}\}$ e o conjunto *Rotulo* (R_2) relacionado ao conjunto $Arco$ possui, também, dois elementos $R_2 = \{\text{"livro/caderno"}, \text{"caderno/livro"}\}$. Apesar de haver dois nós com o mesmo rótulo ("Livro"), está caracterizado que se trata de dois nós distintos, por possuírem identificadores diferentes (1, 3). A relação *No/Rotulo* ou *Arco/Rotulo* pode ser representada por dois novos conjuntos: *NoRotulado* e *ArcoRotulado*. Esses conjuntos representam, diretamente, as relações entre os rótulos e os respectivos identificadores associados aos nós ou arcos.

Tabela 3.1 - Rótulos para identificadores e nós.

R1(ni)			R2(ni)		
	Identificador	Rótulo		Identificador	Rótulo
n ₁	1	"Livro"		1	"livro/caderno"
n ₂	2	"Caderno"		2	"caderno/livro"
n ₃	3	"Livro"			

Definição: *NoRotulado* é um conjunto não vazio de nós no qual cada um de seus elementos adquirem a característica de representar rótulo.

Definição: *ArcoRotulado* é um conjunto de arcos cujos elementos adquirem a característica de representar rótulo.

Notação: *NoRotulado*{*No*, (*Rotulo*)} representa as relações válidas entre os nós e os rótulos. Por exemplo: *NoRotulado*{1("Livro"), 2("Caderno"), 3("Livro")}; onde, 1, 2, 3 são os identificadores do nós.

Notação: *ArcoRotulado*{id[(*Arco*, *Rotulo*)]} representa as relações válidas entre os arcos e os rótulos. Por exemplo:

ArcoRotulado{1(<1,2>,"livro/caderno"), 2(<2,3>,"caderno/livro")}; onde, 1,2 são os identificadores dos arcos, <i,j> representam os nós associados ao arco e o que aparece entre aspas é o rótulo do arco.

3.1.1.2 Arcos e Nós com Peso

Em sistemas cujas atividades possam ser representadas por nós e as associações entre essas atividades representadas por arcos, é razoável supor que há casos nos quais a eficiência, ou o custo, das atividades e das suas associações possam ser parametrizados.

Definição: denomina-se *Peso* o conjunto das funções que podem ser utilizadas para parametrizar o desempenho das atividades e/ou associações em sistemas análogos ao sistema proposto no capítulo 1.

Notação: *Peso*(*f_i*) é o conjunto composto por todas as funções *f_i*, tal que $i \geq 0$, *f_i* representa o custo da atividade desejada e *i* o número de

funções que compõem o conjunto *Peso*.

Um nó com peso é um nó que adquire a característica de poder representar o custo ou eficiência da atividade que o mesmo representa. Por exemplo, a taxa com que uma determinada unidade central de processamento é capaz de efetuar operações, ou a capacidade de comunicação entre equipamentos.

3.1.1.3 Arcos e Nós Rotulados e com Pesos

Nos itens anteriores são definidos conjuntos e notações que permitem a representação de elementos rotulados ou com pesos, porém não com ambas características. Nós ou arcos rotulados e com pesos permitem que elementos que possuam essas características formem um novo conjunto.

Definição: o conjunto *NoRotuladoPeso* é um conjunto cujos elementos agregam as características de serem nó, rótulo e peso.

Notação: $NoRotuladoPeso\{No(Rotulo, Peso)\}$, onde: *No* é o identificador do nó, *Rotulo* é o rótulo associado ao nó e *Peso* é o peso do nó. Exemplo: $NoRotuladoPeso\{1("casa", 15), 2("tijolo", 7)\}$, onde 1 é o identificador do nó cujo rótulo é "casa" e cuja função peso é a constante 15; o mesmo raciocínio é válido para o nó cujo identificador é 2.

Definição: o conjunto *ArcoRotuladoPeso* é um conjunto cujos elementos possuem as características de serem arco, rótulo e peso.

Notação: $ArcoRotuladoPeso\{Id(<i,j>, Rotulo, Peso)\}$, onde *Id* é o identificador do arco; $<i,j>$ representam os identificadores dos nós *i* e *j* associados ao arco; *Rotulo* é do arco e *Peso* é o peso do arco. Exemplo: seja $Arco = \{1<1,2>, 2<2,3>\}$ e seja $ArcoRotulado = \{1(<1,2>, "Arco1"), 2(<2,3>, "Arco2")\}$. Suponha que os pesos para os dois arcos (1 e 2) sejam 45 e 50 respectivamente, então, $ArcoRotuladoPeso = \{1(<1,2>, "Arco1", 45), 2(<2,3>, "Arco2", 50)\}$.

3.1.2 Grafos e Dígrafos

Grafos e dígrafos são conjuntos compostos através dos conjuntos No e $Arco$, independentemente da representação gráfica desses conjuntos.

3.1.2.1 Grafos

Definição: Um grafo G consiste de um conjunto não vazio No e um conjunto $Arco$.

Notação: Seja $No = \{n_1, n_2, n_3, \dots, n_m\}$ um conjunto finito que contém m nós, e $Arco = \{1\langle n_1, n_2 \rangle, 2\langle n_1, n_3 \rangle, \dots, k\langle n_j, n_2 \rangle\}$ um conjunto finito que contém k pares de nós. Um grafo G é denotado por $G = (No, Arco)$. $G(No)$ é o conjunto não vazio de nós do grafo e $G(Arco)$ é o conjunto de arcos do grafo.

Definição: O grau de um nó corresponde ao número de arcos associados ao mesmo.

Notação:

- Seja $Gr(n_i)$, onde $Gr \in \mathbb{N}$, o grau do nó n_i , tal que $n_i \in No$.
- $Gn(No) = |No|$, onde $Gn \in \mathbb{N}$, o número de elementos do conjunto No .
- $Ga(Arco) = |Arco|$, onde $Ga \in \mathbb{N}$, o número de elementos do conjunto $Arco$.

Figura 3.3 - Um exemplo de grafo.

A Figura 3.3 é uma das possíveis representações visuais para um grafo que possui 5 (cinco) nós e 7 (sete) arcos; não é o grafo. Essa representação abstrata não possui significado associado nem aos nós nem aos arcos. Pode-se dizer que cada um dos arcos associa dois nós e que

cada nó tem como característica estar associado a um determinado número de nós. Essas características são intrínsecas aos arcos e nós, respectivamente.

Aos nós e arcos, que compõem o grafo ilustrado pela Figura 3.3, podem ser atribuídos identificadores, conforme ilustra a Figura 3.4. É importante notar que, salvo raras exceções, os identificadores não devem ser apresentados, mas sim os rótulos e/ou outras características mais significativas para o modelo, conforme ilustra a Figura 3.5.

No caso da Figura 3.5, estão definidos os conjuntos $R_1 = \{\text{Pentium}, 68020\}$ e $R_2 = \{C1, C2\}$ de rótulos para os nós e arcos, respectivamente. Supondo-se que os identificadores para os nós sejam $No = \{1, 2, 3\}$, pode-se estabelecer a relação $Rótulo_1$ da seguinte maneira :

$Rótulo_1(1) = \text{Pentium},$ onde $1 = n_1 \in No;$

$Rótulo_1(2) = 68020,$ onde $2 = n_2 \in No;$

$Rótulo_1(3) = \text{Pentium},$ onde $3 = n_3 \in No;$

estabelecendo a relação entre os conjuntos R_1 e No .

Analogamente à relação $Rótulo_1$, pode-se supor uma relação $Rótulo_2$ e um conjunto de identificadores para o conjunto $Arco = \{1<1, 2>, 2<2, 3>\}$ tal que:

$$Rótulo_2(1) = C1, \quad \text{onde } 1 = r_1 \in Arco;$$

$$Rótulo_2(2) = C2, \quad \text{onde } 2 = r_2 \in Arco;$$

Cabe ressaltar que, caso não tivesse sido definido um conjunto de identificadores para os nós, não seria possível explicitar a relação entre dois elementos distintos portadores de mesmo rótulo, e o conjunto $Arco$ seria composto por um único elemento, dado que $\langle Pentium, 68020 \rangle$ seria o mesmo que $\langle 68020, Pentium \rangle$ no caso de um grafo.

3.1.2.2 Dígrafo

Definição: Um dígrafo consiste de dois conjuntos No e $Arco$. O conjunto $Arco$ é formado por pares ordenados de nós $\langle i, j \rangle \mid i, j \in No$, onde No é um conjunto não vazio de nós, tal que i e j são denominados nó-origem e nó-destino, respectivamente.

Notação: Seja $No = \{n_1, n_2, n_3, \dots, n_m\}$ um conjunto finito que contém m nós, e $Arco = \{1\langle n_1, n_2 \rangle, 2\langle n_1, n_3 \rangle, \dots, k\langle n_j, n_2 \rangle\}$ um conjunto finito que contém k pares de nós. Um grafo D é denotado por $D = (No, Arco)$. $D(No)$ é o conjunto não vazio de nós do grafo e $D(Arco)$ é o conjunto de arcos do grafo.

Definição: O grau de entrada de um nó n_i do dígrafo D , tal que $n_i \in No$, corresponde ao número de elementos do conjunto $Arco$ que têm como nó-origem um nó qualquer n_k , tal que $n_k \in No$, e têm como destino o nó n_i .

Definição: O grau de saída de um nó n_i do dígrafo D , tal que $n_i \in No$, corresponde ao número de elementos do conjunto $Arco$ que têm como nó-destino um nó qualquer n_k , tal que $n_k \in No$, e têm como origem o nó n_i .

Notação: $Dre = |Arco|$, onde $Dre \in \mathbb{N}$ é o número de elementos pertencentes ao conjunto $Arco$.

Uma possível representação gráfica para um dígrafo pode ser efetuada através da utilização de segmentos lineares orientados que estabeleçam alguma relação de ordem para os pares ordenados $\langle i, j \rangle \in Arco$,

conforme ilustra a Figura 3.6. Supondo-se a relação apresentada na Tabela 3.2 para os elementos de *No* e supondo-se que o primeiro elemento do par ordenado é o início do segmento (não possui seta), obtém-se o seguinte conjunto *Arco* = {1<1,3>, 2<3,1>, 3<3,4>, 4<3,2>, 5<3,5>}.

Tabela 3.2 - Relação rótulo/identificador.

n_i	Identificador	Rótulo (i)
n_1	1	Assis
n_2	2	Itu
n_3	3	Jales
n_4	4	Jataí
n_5	5	Jau

3.1.3 Supernós e Super-arcos

Há casos nos quais determinados subconjuntos de nós podem ser agrupados para que seja gerado um novo nó, formado a partir de características fatoradas dos elementos que integram o subconjunto. Por exemplo, ao invés de serem representadas cada uma das máquinas que compõem uma determinada rede de computadores, pode-se desejar representar, num determinado instante, grupos de máquinas que apresentem alguma característica comum. Nesse caso, podem ser gerados nós que representam apenas as abstrações desejadas, por exemplo o tipo de processador.

Definição: O conjunto *supernó* é composto por elementos que pertencem ao conjunto $G(No, Arco)$, possuem a característica de serem representados por um único nó e são identificados univocamente.

Notação: O conjunto de *supernós* é denotado por $SNo(No, G) = \{id_1[Gs(No)], id_2[Gs(No)], \dots, id_m[Gs(No)], \}$, onde $m > 0$, $m \in \mathbb{N}$ é o número de *supernós* pertencentes a SNo , e $Gs(No)$ são subconjuntos de $G(No)$. Essa notação explicita a característica de que um *supernó* é um grafo que possui identificador único, como um nó. Assim, dado o identificador do *supernó* é possível identificar o subconjunto representado.

Exemplo: seja $No = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $Arco = \{1<1,2>, 2<2,3>, 3<3,1>, 4<3,4>, 5<2,5>, 6<5,3>, 7<4,1>\}$ e $SNo = \{6[1, 2, 5], 7[3, 4, 5], 8[1, 2, 5]\}$. Os subconjuntos $1[1, 2, 5]$, $2[3, 4, 5]$, $3[1, 2, 5]$ podem representar visões macroscópicas do sistema modelado. Os subconjuntos $1[1, 4, 5]$, $3[1, 4, 5]$ contêm os identificadores dos mesmos elementos e possuem identificadores diferentes, logo trata-se de subconjuntos distintos. Note-se que nesse caso não está representada a semântica que gerou essa representação, mas isso indica visões diferentes de um mesmo subconjunto.

Para o conjunto de arcos, há a possibilidade de existir mais do que uma associação entre dois nós. Por exemplo, nos casos em que há mais de um meio físico interligando os computadores em uma rede. Nesse caso, cada arco representa um caminho alternativo para os dados e o modelo pode apresentar esses detalhes em níveis diferentes de abstração, dependendo da relevância dessa informação. O conjunto de arcos é formado por subconjuntos e cada um dos subconjuntos é representado por um elemento. Por exemplo, no conjunto $Arco = \{1<1,2>, 2<1,2>, 3<2,1>, 4<2,4>, 5<4,2>, 6<1,3>, 7<3,4>\}$, nota-se que os arcos $\{1,2,3\}$ associam os mesmos nós; analogamente tem-se os arcos $\{4,5\}$ e finalmente as outras duas partes $\{6\}$ e $\{7\}$.

Definição: $SArco$ é o conjunto no qual os elementos são subconjuntos do conjunto $Arcos$ e que associam pares de nós ou de *supernós*. Cada subconjunto possui identificador único.

Notação: o conjunto $SArco(id[Arco])$ é composto por subconjuntos do conjunto de *Arcos*. Os identificadores permitem que subconjuntos compostos com os mesmos elementos possam representar abstrações distintas.. Exemplo: $SArco = \{1[1, 2, 3], 2[4, 5], 3[6], 4[7]\}$

3.1.3.1 Super-arcos e Supernós Rotulados

Os conjuntos *SNo* e *SArco* não possuem a característica de serem rotulados. A necessidade de representar níveis de abstração nos modelos pode fazer com que seja indispensável atribuir rótulos aos elemento, seja ele superno ou super-arco.

Definição: supernó-rotulado é um conjunto gerado a partir de subconjuntos do conjunto de *NoRotulado*.

Notação: $SNoRotulado\{Id(NoRotulado)\}$; onde *Id* é o identificador do subconjunto gerado a partir do conjunto *NoRotulado*.

A Figura 3.7 ilustra um conjunto *SNoRotulado* composto por três subconjuntos de *NoRotulado*, estando implícito na representação da mesma:

- o conjunto de nós,

$$No = \{1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13\};$$

- o conjunto de nós rotulados,

$$NoRotulado = \{(1,a), (2,b), (3,c), (4,d), (5,e), (6,f), (7,g), (8,h), (9,i), (10,j), (11,k), (12,o), (13,u)\};$$

- o conjunto de supernós,

$$SNo = \{1[1, 5, 9, 12, 13], 2[2, 3, 4, 6, 7, 8, 10, 11]\};$$

- o conjunto de supernós rotulados,

$$SNoRotulado = \{1("Vogais", [1,a], [5,e], [9,i], [12,o], [13,u]), 2("Consoantes", [2,b], [3,c], [4,d], [6,f], [7,g], [8,h], [10,j], [11,k]), 3("Letras", [1,a], [5,e], [9,i], [12,o], [13,u], [2,b], [3,c], [4,d], [6,f], [7,g], [8,h], [10,j], [11,k])\};$$

Definição: *SArcoRotulado* é um conjunto gerado a partir de subconjuntos do conjunto de *ArcoRotulado*.

Notação: $SArcoRotulado\{Id([Ar, \langle No, No \rangle, Rotulo])\}$, onde, Id é o identificador do subconjunto, Ar é o identificador de algum arco pertencente ao conjunto $Arco$, $\langle No, No \rangle$ são os nós associados a Ar e $Rotulo$ é o rótulo do super-arco.

A Figura 3.8b apresenta um arco cujo rótulo é Sa , tem peso 15, e a linha pontilhada indica que é um super-arco. O super-arco Sa representa uma possível visão macroscópica dos arcos $A1$, $A2$ e $A3$ que aparecem na Figura 3.8a, podendo-se assumir, para a representação em forma de grafo, o seguinte:

$No = \{1, 2\}$; $NoRotulado \{1("1"), 2("2")\}$

$Arco = \{1\langle 1, 2 \rangle, 2\langle 1, 2 \rangle, 3\langle 1, 2 \rangle\}$;

$ArcoRotulado = \{1(\langle 1, 2 \rangle, "A1"), 2(\langle 1, 2 \rangle, "A2"), 3(\langle 1, 2 \rangle, "A3")\}$

$SArcoRotulado = \{1[(1, \langle 1, 2 \rangle, "A1"), (2, \langle 1, 2 \rangle, "A2"), (3, \langle 1, 2 \rangle, "A3")]\}$

3.1.3.2 Super-arcos e Supernós Rotulados e com Peso

Analogamente ao conceito de nós e arcos com peso, super-arcos e supernós rotulados e com peso, permitem a representação de grandezas tais como o custo ou a eficiência com que um conjunto de atividades é efetuado.

Definição: $SNoRotuladoPeso$ é um conjunto de supernós rotulados tal que, para cada supernó-rotulado é possível agregar uma medida que

represente o custo e/ou a eficiência das atividades embutidas no nó.

Notação: $SNoRotuladoPeso\{Id[(No, Rotulo), P_i]\}$, onde Id é o identificador do supernó, $(No, Rotulo)$ é um par ordenado que representa identificadores de nós com os seus respectivos rótulos e P_i é o identificador da função associada ao supernó.

Exemplo: considerem-se os conjuntos

$No=\{1,2,3,4,5\}$;

$Rotulo=\{"No1", "No2", "No3", "No4", "No5"\}$;

$NoRotulado=\{1("No1"), 2("No2"), 3("No3"), 4("No4"), 5("No5")\}$;

$RotuladoPeso=\{1("No1",3), 2("No2",7), 3("No3",1), 4("No4",3), 5("No5",19)\}$;

$SNoRotulado=\{1[(1,"No1"), (2,"No2"), (3,"No3")], 2[(3,"No3"), (4,"No4"), (5,"No5")] \}$;

$SNoRotuladoPeso=\{1[(1,"No1",3), (2,"No2",7), (3,"No3",3), 13], 2[(3,"No3",1), (4,"No4",3), (5,"No5",19), 23] \}$.

No exemplo acima, o peso do supernó é equivalente à soma dos pesos parciais.

Definição: $SArcoRotuladoPeso$ é um conjunto cujos elementos são subconjuntos de $SArcoRotulado$ e a cada um desses subconjuntos está associada alguma função peso que equivale ao peso de todos os arcos do subconjunto.

Notação: $SNoRotuladoPeso\{[id(Arco, Rotulo, Peso), SPeso]\}$, onde Id é o identificador do super-arco rotulado e com peso; $(Arco, Rotulo, Peso)$ é uma terna ordenada na qual $Arco$ é o identificador de algum arco; $Rotulo$ e $Peso$ são o rótulo e o peso do arco, respectivamente; e $SPeso$ é o peso do super-arco. Os conjuntos $Peso$ e $SPeso$ são conjuntos não necessariamente distintos.

A Figura 3.9a ilustra um conjunto de arcos que associam os nós 1 e 2. A Figura 3.9b ilustra o super-arco rotulado e com peso gerado a partir dos arcos A1 e A2. Os conjuntos associados a esse exemplo são:

$$NoRotulado = \{1("1"), 2("2")\}$$

$$Rotulo = \{ "A1", "A2" \},$$

$$Arco = \{1<1,2>, 2<1,2>, 3<1,2>\},$$

$$ArcoRotulado = \{1(<1,2>, "A1"), 2(<1,2>, "A2"), 3(<1,2>, "A3")\}$$

$$ArcoRotuladoPeso = \{1(<1,2>, "A1", 2), 2(<1,2>, "A2", 5), 3(<1,2>, "A3", 7)\}$$

$$SArcoRotuladoPeso = \{1[(1, <1,2>, "A1", 2), (2, <1,2>, "A2", 5), SA, 9]\}$$

3.1.4 Supergrafos e Superdígrafos

O dígrafo ilustrado na Figura 3.10 apresenta cinco pessoas e o seu grau de parentesco. Para esse exemplo não é difícil imaginar uma abstração que represente apenas os subconjunto pais e filhos gerados a partir do conjunto $No = \{1, 2, 3, 4, 5\}$, de acordo com a tabela 3.3. Isso pode ser

modelado criando-se as partições {1, 2} e {3, 4, 5} a partir do conjunto *No*. A Figura 3.10a ilustra essa abstração.

Tabela 3.3 - Relação identificador/rótulo para a Figura 3.10.

n_i	Identificador	Rótulo ₁ (i)	a_i	Identificador	Rótulo ₂ (i)
n_1	1	Luís	a_1	$1\langle n_1, n_4 \rangle$	Pai
n_2	2	André	a_2	$2\langle n_2, n_3 \rangle$	Pai
n_3	3	Leonor	a_3	$3\langle n_2, n_5 \rangle$	Pai
n_4	4	André	a_4	$4\langle n_2, n_1 \rangle$	Irmão
n_5	5	Felipe	a_5	$5\langle n_1, n_2 \rangle$	Irmão
			a_6	$6\langle n_4, n_1 \rangle$	Filho
			a_7	$7\langle n_3, n_2 \rangle$	Filho
			a_8	$8\langle n_5, n_2 \rangle$	Filho

Além disso, pode-se notar que para toda relação “pai” existe o oposto “filho”. Esses dois arcos podem ser “agrupados” com a finalidade de, por exemplo, apenas simplificar a notação, conforme ilustra a Figura 3.11b. A inserção de um segmento de reta pontilhado com orientação nos dois sentidos, é colocada com a intenção de sugerir alguma diferenciação entre arcos e super-arcos. As duas elipses concêntricas são utilizadas para representar que cada um dos nós é um *super-nó*. A associação entre dois *super-nós* pode ser omitida em função da visibilidade almejada.

Figura 3.11 - Exemplo de um super-dígrafo.

Definição: *Supergrafo*, SG , é um conjunto, não vazio, no qual cada um de seus elementos são identificados univocamente e são compostos por pares pertencentes aos conjuntos SNo , $SArco$,

Notação: $SG(SNo, SArco) = \{id_1[<Gs_1(No)>, <Gs_1(Arco)>], id_2[<Gs_2(No)>, <Gs_2(Arco)>], \dots, id_m[<Gs_k(No)>, <Gs_n(Arco)>]\}$; onde:

$m, k > 0, n \geq 0$, tal que $m, k, n \in \mathbb{N}$;

$Gs(No)$ e $Gs(Arco)$ são subconjuntos de No e $Arco$, respectivamente;

$SG(SNo)$: é o subconjunto de supernós;

$SG(SArco)$: é o subconjunto de *super-arco*s;

$SGU(SNo)$: é o conjunto No ;

$SGU(SArco)$: é o conjunto $Arco$.

Exemplo:

$No = \{1, 2, 3, 4, 5, 6, 7\}$,

$Arco = \{1<1,2>, 2<2,3>, 3<3,4>, 4<4,5>, 5<1,3>, 6<1,4>, 7<1,5>\}$,

$G(No, Arco) = \{[1, 2, 3, 4, 5], [1<1,2>, 2<2,3>, 3<3,4>, 4<4,5>, 5<1,3>, 6<1,4>, 7<1,5>]\}$

$SNo = \{6[1, 2, 3], 7[3, 4, 5]\}$

$SArco = \{8[2, 5], 9[6,7]\}$

$SG(SNo, SArco) = \{1[<1, 2, 3>, <2, 5>], 2[<3, 4, 5>, <6, 7>]\}$

Definição: O *Superdígrafo* SD é um dígrafo composto por super-arcos e supernós, tal que a relação de ordem no subconjunto *SArco* é preservada, de acordo com a definição de *dígrafo*.

Notação: $SD(SNo, SArco)$, onde:

$SD(SNo)$: é o subconjunto de supernós;

$SD(SArco)$: é o subconjunto de *super-arcos*;

$SDU(SNo)$: é o conjunto *No*;

$SDU(SArco)$: é o conjunto *Arco*;

Supergrafos e Superdígrafos Rotulados

Definição: *SGRotulado* é um conjunto, não vazio, no qual cada um de seus elementos, além de possuírem as mesmas características dos elementos pertencentes ao conjunto SG, têm os seus respectivos rótulos.

Notação: $SGRotulado(SNo, SArco, Rotulo) = \{id_1[<Gs_1(No)>, <Gs_1(Arco)>, R(id_1)], id_2[<Gs_2(No)>, <Gs_2(Arco)>, R(id_2)], \dots, id_m[<Gs_k(No)>, <Gs_n(Arco)>, R(id_m)]\}$; tal que, $m, k > 0, n \geq 0$, tal que $m, k, n \in \mathbb{N}$; $R(id_m)$ é a relação que tem como domínio o conjunto de identificadores dos nós do supergrafo e contradomínio o conjunto *Rotulo*.

Supergrafos e Superdígrafos Rotulados e com Pesos

Definição: *SGRotuladoPeso* é um conjunto, não vazio, no qual cada um de seus elementos agrega características dos elementos pertencentes ao conjunto *SGRotulado* e têm as suas respectivas funções-peso.

Notação: $SGRotuladoPeso(SNo, SArco, Rotulo, Peso) = \{id_1[<Gs_1(No)>, <Gs_1(Arco)>, R(id_1), P(id_1)], id_2[<Gs_2(No)>, <Gs_2(Arco)>, R(id_2), P(id_2)], \dots, id_m[<Gs_k(No)>, <Gs_n(Arco)>, R(id_m), P(id_m)]\}$; tal que, $m, k > 0, n \geq 0$, tal que $m, k, n \in \mathbb{N}$; $P(id_m)$ é a relação que tem como domínio o conjunto de identificadores dos nós do supergrafo e contradomínio o conjunto *Peso*.

3.1.5 Operações

A intenção não é listar todas as operações possíveis para os conjuntos *No*, *Arco*, *SNo*, *SArco*, *G*, *SG*, *D* e *SD*. Esses conjuntos representam conceitos que serão modelados, no item 3.2, e servirão como classes-base para especializações que constituirão tipos diferentes de aplicações. Assim, o conjunto das operações, para os componentes dos grafos, contém operações básicas implementadas como funções membro das classes-base.

A definição do conjunto *G* pressupõe a criação dos conjuntos *No* e *Arco*, que, por sua vez, utilizam-se de identificadores para abstrair e representar os entes que são modelados. A Tabela 3.4 apresenta uma relação dos conjuntos com as operações selecionadas.

Tabela 3.4 - Tabela de operações.

Conjunto	Operação	Conjunto	Operação
<i>Identificador</i>	Obter um identificador.	<i>Arco</i>	Obter <i>No</i> origem
<i>Identificador</i>	Devolver um identificador.	<i>Arco</i>	Obter <i>No</i> destino
<i>No</i>	Criar um novo elemento, com o respectivo identificador	<i>Arco</i>	Armazenar em meio não volátil
<i>No</i>	Obter grau de entrada	<i>Arco</i>	Alterar origem
<i>No</i>	Obter grau de saída	<i>Arco</i>	Alterar destino
<i>No</i>	Obter o grau do nó.	<i>Arco</i>	Recuperar de meio não volátil
<i>No</i>	Recuperar de meio não volátil	<i>G</i>	Verificar a conexidade
<i>No</i>	Armazenar em meio não volátil	<i>G</i>	Armazenar em meio não volátil
<i>NoRotulado</i>	Obter rótulo	<i>G</i>	Destruir um grafo
<i>NoRotulado</i>	Alterar o rótulo	<i>G</i>	Criar um grafo
<i>NoRotulado</i>	Recuperar de meio não volátil		Inserir nó
<i>NoRotulado</i>	Armazenar em meio não volátil	<i>G</i>	Listar os componentes do grafo
<i>NoPeso</i>	Obter peso.	<i>G</i>	Inserir arco
<i>NoPeso</i>	Alterar o peso	<i>G</i>	Eliminar arco
<i>NoPeso</i>	Recuperar de meio não volátil	<i>G</i>	Eliminar nó
<i>NoPeso</i>	Armazenar em meio não volátil	<i>G</i>	Obter caminho entre dois nós
<i>Arco</i>	Criar um novo elemento, com o respectivo identificador	<i>G</i>	Recuperar de meio não volátil

3.2 Uma Estrutura de Classes Hierárquicas para Modelar Grafos e Dígrafos

Os modelos são apresentados de maneira tal que apenas os elementos estritamente necessários são citados. Os detalhes de projeto e implementação seguem em anexo no apêndice A.

Tanto os nós quanto os arcos possuem identificadores únicos, conforme argumentado no item 2.2. A classe GRFIdList permite que sejam instanciados objetos para o controle de identificadores. Nada impede que seja utilizado um único objeto instanciado a partir da classe GRFIdList para todo o sistema, conforme ilustra a Tabela 3.5. Entretanto, quando o sistema é relativamente complexo, há restrições para a capacidade de representação. Por isso é sugerido que existam objetos de controle diferentes para elementos instanciados a partir de classes diferentes.

Tabela 3.5 - Geração de identificadores.

```
#define IDENTIFICADOR unsigned long int
class GRFId {
public:
const IDENTIFICADOR GetId ( void );
void Delete ( const IDENTIFICADOR & );
private:
class GRFIdList // Gerencia a criação de identificadores
{
public:
GRFIdList ( void ) {id = 0; GRFIdList * pt = (GRFIdList *)
NULL;}
IDENTIFICADOR GetId ( void ); // Obtém um identificador
const GRFIdList * GetNext ( void ) const; // Pega o prox. da
lista.
private:
IDENTIFICADOR id;
GRFIdList *p;
};
IDENTIFICADOR id; // Numero do proximo valor de identificacao
IDENTIFICADOR size; // Tamanho da pilha
GRFIdList *p;
};
```

A geração de identificadores é controlada por uma estrutura do tipo pilha alocada dinamicamente. Quando um novo identificador é solicitado, a lista é consultada, se houver algum Id que tenha sido devolvido para o sistema, esse Id é utilizado novamente; caso contrário é gerado um novo Id.

Essa classe implementa uma pilha que permite que identificadores de objetos já destruídos sejam utilizados por objetos a serem criados. Os identificadores citados no item 3.1 são gerados por essa classe. Há uma lista de identificadores para cada classe de objetos.

3.2.1 Modelo para Arcos e Nós

Cada um dos nós de um dígrafo possui como elementos mais abstratos o número de arcos que chegam e que saem do mesmo. Isso sugere uma classe abstrata na qual seja representado o grau do nó. Classe essa que virtualmente virá a ser especializada em função de características de novos conceitos e/ou aplicações. A utilização de uma classe virtual abstrata é consequência direta da elaboração de um modelo orientado a objetos do conceito em questão, uma vez que conceitos não podem ser instanciados.

Notação: as linhas pontilhadas indicam que a classe é abstrata², ou seja, não podem ser instanciados objetos desse tipo de classe, embora essa notação seja utilizada no método Fusion para indicar conjunto. Os atributos são escritos utilizando-se fonte "true type" normal, as funções membro no formato itálico e as funções membro virtuais estão no formato itálico e grifadas.

Por uma questão de consistência da documentação será utilizada a sigla GRF (GRaFo) como prefixo para as classes que modelam os conceitos da teoria dos grafos. Por exemplo: GRFNo e GRFArco, apresentados na Figura 3.12, são as classes que representam, respectivamente, as fatorações dos atributos comuns a todos os nós e arcos de um dígrafo. O agrupamento de atributos modela a representação dos conjuntos *No* e *Arco*, apresentados no item 3.1.1. A nomenclatura utilizada para os atributos da classe GRFArco, apresentados na Figura 3.12b, induz representação de semântica no modelo, dado que é conceito de dígrafo a característica de representação de nó-origem e nó-destino. Contudo, esse significado é parte

² Classes abstratas são também denominadas classes virtuais puras.

apenas das aplicações arquitetadas em função das hierarquias de especializações, propostas a partir dessas classes-base. Portanto não devem ser confundidos os identificadores de objetos com o significado da palavra escrita.

Cada elemento do grafo deve possuir identificador único. Isso garante a propriedade da geração de objetos persistentes para a reconstituição do grafo a partir de arquivo. As linhas pontilhadas indicam essa modelagem, conforme ilustra a Figura 3.13. A Tabela 3.6 ilustra a estrutura das classes GRFNo e GRFArco.

Tabela 3.6 - Classes-base GRF.

<pre>class GRFNo : public GRFFile{ public: GRFNo (const IDENTIFICADOR &, UNIT ge=0, UNIT gs=0); protected: UINT grauEnt, grauSai; private: IDENTIFICADOR id; };</pre>
<pre>class GRFArco : public GRFFile{ public: GRFNo (const IDENTIFICADOR &, GRFNo * or = (GRFNo *) NULL, & GRFNo * des = (GRFNo *) NULL); protected: GRFNo *origem, *destino; private: IDENTIFICADOR id; };</pre>

3.2.2 Modelo para Arcos e Nós Rotulados

Considerem-se as aplicações nas quais os elementos do mundo real são abstrações identificadas por nomes, ou rótulos, em analogia ao conceito de nó rotulado. Dessa maneira, um nó rotulado é um nó acrescido da característica de possuir um rótulo, e tanto os nós quanto os arcos rotulados podem armazenar e restaurar os conteúdos de seus atributos a partir de arquivos. A Figura 3.14 e a Tabela 3.7 ilustram a hierarquia das classes e os seus respectivos protótipos.

Os elementos básicos que compõem as aplicações almejadas com esse modelo são máquinas abstratas, ~~representadas~~ representadas por nós e arcos rotulados. Assim, é razoável propor que nós e arcos possuam funções que representem esforços, trabalho, custo ou outros, em analogia aos ~~grafos~~ grafos rotulados com peso.

Pode-se, portanto, supor uma classe-base abstrata, que modele diversos conjuntos distintos de funções e duas classes derivadas de GRFNoRotulado e GRFArcoRotulado, conforme ilustram a Figura 3.15a e a Figura 3.15b, respectivamente.

Tabela 3.7 - Classes GRFRotulo, GRFNoRotulado, GRFArco.

<pre> class GRFRotulo{ public: GRFRotulo (const string &); protected: string rotulo; }; </pre>
<pre> class GRFNoRotulado : public GRFNo{ public: GRFNoRotulado (const IDENTIFICADOR &. const strinf &); GRFRotulo * rotulo; //Permite manipular qualquer filho. }; </pre>
<pre> class GRFNoRotuladoPeso : public GRFNoRotulado{ public: GRFNoRotuladoPeso (const IDENTIFICADOR &. const strinf &. GRFFuncaoPeso *); GRFFuncaoPeso * peso; }; </pre>
<pre> class GRFArcoRotulado : public GRFArco{ public: GRFArcoRotulado (const IDENTIFICADOR &. const strinf &); GRFRotulo * rotulo; //Permite a manipular qualquer filho. }; </pre>
<pre> class GRFArcoRotuladoPeso : public GRFArcoRotulado{ public: GRFArcoRotuladoPeso (const IDENTIFICADOR &. const strinf &. GRFFuncaoPeso *); GRFFuncaoPeso * peso; }; </pre>

O atributo "funcao", pertencente à classe GRFNoRotuladoPeso, permite que quaisquer objetos, instanciados a partir dessa classe, estejam

associados à algum objeto, derivado a partir da classe GRFFuncaoPeso, através do polimorfismo.

3.2.3 Modelos para Grafos e Dígrafos

Por definição, um grafo é composto por um conjunto de arcos e um conjunto de nós. O modelo proposto requer a criação de duas novas classes. Essas classes são estruturas de listas lineares que referenciam objetos instanciados a partir das classes GRFNo e GRFArco, conforme ilustra a Figura 3.16. Essa classe permite que quaisquer elementos das classes GRFNo e GRFArco, ou de alguma de suas subclasses, possam ser referenciados. Isso garante que futuras especializações, ou implementações de aplicações, possam fazer uso das classes de armazenamento GRFNos e GRFArcos.

Um grafo é um conjunto capaz de representar todos os nós e todos os arcos. Essa característica pode ser modelada através do mecanismo de herança múltipla, ou seja, através de um conjunto que especialize as características das classes de armazenamento GRFNos e GRFArcos, conforme ilustra a Figura 3.17. Ao ser imposta a restrição de que todo arco parte de um nó origem e alcança um nó destino, fica estabelecido o conceito de dígrafo. Assim, pode-se dizer que um dígrafo é uma especialização de um grafo. Essa hierarquia está ilustrada na Figura 3.19 e seus protótipos estão ilustrados na tabela 3.8.

Tabela 3.8 - Classes de armazenamento par arcos e nós.

<code>typedef TIArrayAsVector<GRFNo> GRFNos(100,0,10);</code> <code>typedef TIArrayAsVectorIterato<GRFNo> IteratorGRFNos;</code>
<code>typedef TIArrayAsVector</code> <code><GRFArco> GRFArcos(100,0,10);</code> <code>typedef TIArrayAsVectorIterator<GRFArco> IteratorGRFArcos;</code>
<code>class GRFGrafo{</code> <code>protected:</code> <code>GRFArcos *arcos; // Classe de armazenamento</code> <code>GRFNos *nos; // Classe de armazenamento</code> <code>};</code>

3.2.4 Modelo para Subgrafos e Subdígrafos

A utilização de objetos polimórficos permite que tanto os entes do mundo real quanto as suas associações sejam modelados através de grafos ou dígrafos. Mas, há casos nos quais é necessário representar agrupamentos de objetos, de maneira análoga aos subgrafos. Por exemplo, nos casos onde são desejáveis representações de níveis diferentes de visão ou a representação de hierarquias de abstração, conforme o exemplo ilustrado na Figura 3.18.

Os nós $\langle y, w, u \rangle$, que aparecem na Figura 3.18b, são agrupados no subgrafo T da Figura 3.18a. A explosão do nó, ou o desejo de visualizar maiores detalhes, deve ser parte do modelo para o problema. Para modelar a visão macroscópica do sistema pode ser necessário o agrupamento de diversos arcos. Para isso foram criados os conceitos de *super nó*, *super arco* e *super dígrafo*. Um super nó é um dígrafo rotulado. Um super arco é um arco rotulado e um conjunto de arcos, conforme ilustra a Figura 3.19.

Um *super dígrafo* é um dígrafo, um conjunto de *super nós* e um conjunto de *super arcos*, ou seja, é uma especialização do conceito de dígrafo. Para a implementação dessa proposta, o fato dos nós e arcos formarem as bases para a classe GRFDigrafo é suficiente. No entanto, para maior clareza e estruturação do modelo é interessante que sejam modeladas classes análogas às classes bases da classe GRFDigrafo. As classes GRFVetorSuperNo e GRFVetorSuperArco são classes de armazenamento, análogas às classes GRFNosRotulados e

GRFArcosRotulados, com a característica de ser um dígrafo cujos nós podem ser outros dígrafos (ou subdígrafos), conforme ilustra a Figura 3.20.

Figura 3.20 - Classe GRFSuperDigrafo e suas derivações.

3.3 Considerações Finais

A Figura 3.21 apresenta a visão macroscópica da estrutura de classes, na qual nota-se que a proposta é um meta-modelo para sistemas de gerenciamento de bancos de dados orientados a objetos (SGBDOO). Caso haja a disponibilidade de um SGBDOO que ofereça suporte à edição de meta-esquemas, as especializações que modelam as aplicações podem ser tratadas como versões dos esquemas (Traina (1986)).

Figura 3.21 - Visão macroscópica das classes.

Os módulos Componentes de Hardware e Componentes de Software, ilustrados na Figura 1.1, são especializações das classes-base apresentadas neste capítulo, cujos projetos constam do apêndice B. O capítulo 4 apresenta uma revisão bibliográfica das arquiteturas de computadores, de algumas linguagens de programação e de suas taxonomias, constituindo-se na base teórica para o modelo proposto.

O capítulo 5 mostra a estrutura lógica de um conjunto de classes-base para a manipulação dos objetos instanciados a partir das classes descritas nos capítulos 4 e 5.

4. Modelo Orientado a Objetos para Arquiteturas e Programas Paralelos

O objetivo almejado com o desenvolvimento das arquiteturas paralelas é melhorar o desempenho dos sistemas computacionais. Infelizmente, atrelados ao alto desempenho estão os altos custos, tanto para o desenvolvimento do hardware quanto do software. Quanto mais próximo do hardware estiverem as implementações dos sistemas, melhores serão os desempenhos (Foster (1995)). Por outro lado, a utilização de máquinas virtuais (McBryan (1994)) ou *coordination models* (Ciancarini (1996)), estabelecidas sobre o hardware, faz com que o software seja mais flexível. Essa flexibilidade pode diminuir os custos de manutenção e permitir a execução de um mesmo código fonte em plataformas de hardware diferentes.

Uma outra maneira de flexibilizar o software é através da utilização de ferramentas análogas à FAPP, como por exemplo do sistema CAPTool - *Computer Aided Parallelisation Tools* (Jonson; et al. (1996)). A ferramenta CAPTool busca automatizar o processo de geração de código paralelo em FORTRAN, utilizando heurísticas gerar partições de dados e é especializada para aplicações numéricas.

Este capítulo aborda alguns conceitos relativos a arquiteturas paralelas, programação paralela/concorrente e ambientes de programação paralelos/concorrentes. Essa síntese tem por objetivo definir as características principais, de hardware e de software, para o estabelecimento dos modelos orientados a objetos das classes-base que representam esses elementos na ferramenta FAPP, apresentando uma alternativa os problemas de programação paralela .

4.1 Considerações Iniciais

As arquiteturas paralelas podem ser definidas como “um conjunto de elementos de processamento que podem se comunicar e cooperar para resolver grandes problemas rapidamente” (Almasi; Gottlieb (1994)). Essa definição impõe alto grau de abstração sobre o conceito de arquiteturas paralelas, permitindo que sejam elaboradas questões relativas aos elementos de processamento, aos mecanismos de comunicação, à cooperação entre esses elementos de processamento e sobre o que significa resolver grandes problemas rapidamente.

O conjunto de elementos de processamento é uma característica intrínseca ao hardware e as soluções dos problemas geram o software. Entretanto, quando se deseja estabelecer a cooperação e a comunicação entre os elementos de processamento, deve-se considerar a interação hardware/software. No nível do hardware deve-se considerar quais são os mecanismos de interconexão entre os elementos de processamento, qual é a topologia do mesmo e qual é o custo da comunicação. No nível do software deve-se ponderar sobre a granulosidade da aplicação, qual é o modelo computacional¹ a ser utilizado, qual é o melhor algoritmo e qual é a visão que o programador tem da máquina (Treleven (1985)).

Analisar todas as arquiteturas paralelas com as possíveis variações para os dispositivos de entrada e saída de dados não é uma tarefa recomendável, dado que as combinações possíveis crescem exponencialmente (Steen (1993)). Porém, essas premissas são fundamentais para se compreender o escopo da Ferramenta de Auxílio à Programação Paralela - FAPP, que é o objeto deste trabalho.

Para que o auxílio oferecido por essa ferramenta seja significativo é necessário que se armazenem informações associadas à plataforma do hardware. Essas informações são utilizadas para auxiliar a geração do esqueleto da aplicação, analogamente à proposta de Zima; Chapman (1992), e devem ser suficientes para que a ferramenta possa auxiliar a

¹ Modelos computacionais: programas orientados por controles, dados e sob demanda.

adaptação do programa à arquitetura, buscando melhorar o desempenho do mesmo. O refinamento, ou adequação do programa à arquitetura através de uma ferramenta, é facilitada quando se possui informações sobre a arquitetura. Essas informações caracterizam grupos, ou conjuntos, de máquinas com atributos semelhantes.

O modelo para a ferramenta incorpora as informações propostas por Almasi; Gottlieb (1994) e Zima; Chapman (1992). O sistema proposto por Zima implementa uma ferramenta para paralelizar código FORTRAN e, segundo o autor, mesmo utilizando técnicas de inteligência artificial, faz-se necessária a intervenção do programador em diversas situações.

Estendendo os conceitos de arquiteturas paralelas para as redes de computadores, Grimshaw (1993) criou o sistema Mentat a partir das seguintes observações:

- os programadores podem compreender melhor o domínio da aplicação e, conseqüentemente, tomar decisões de particionamento melhores que o compilador, fato já constatado por Zima;
- os compiladores efetuam o trabalho de definição e gerenciamento de processos assíncronos melhor que os programadores, particularmente em ambientes que não utilizam o paradigma SPMD² (Foster (1995)).

O sistema Mentat não é apenas um ambiente de chamada a procedimentos remotos ("Remote Procedure Calls" (Wilbur (1987))), nem um ambiente de programação que utiliza máquinas virtuais, como por exemplo PVM (Beguelin; et al. (1994)) ou MPI (Gropp; Skjellum (1995)). Esse sistema utiliza a tecnologia de orientação a objetos para encapsular a cooperação e interconexão, em nível de linguagem de programação, criando um ambiente de programação paralela. As classes base do Mentat, escritas na linguagem C++, permitem que a aplicação faça uso dos objetos instanciados a partir dessas classes. O ambiente gera o grafo de execução

instanciados a partir dessas classes. O ambiente gera o grafo de execução do programa, eximindo o programador da responsabilidade de sincronizar os processos gerados. O compilador gera o código e executa o grafo de dependência de dados, no qual os nós são chamadas a funções-membro de objetos do Mentat e os arcos são as dependências de dados encontradas nos programas.

Pesquisas têm sido desenvolvidas com a finalidade de simular a execução de programas paralelos, com a finalidade de descobrir-se a priori qual é o melhor modelo arquitetural do software para melhorar o desempenho do sistema (Manacero (1997), Sahni; Thanvantri (1996)).

A maioria dos ambientes para programação paralela operam segundo as características dos sistemas citadas anteriormente, adicionando-se a esses as linguagens de programação. Esses ambientes geram um grafo, ou uma máquina de estados finitos, do programa para que sejam aplicadas transformações nesse grafo. Essas transformações correspondem aos novos programas ou às novas versões.

O ambiente FAPP reúne essas características e as estende, segundo o paradigma de orientação a objetos, para que sejam utilizadas como base para a geração do grafo do programa, do esqueleto da aplicação e do grafo da arquitetura do hardware.

Os itens a seguir apresentam os conceitos e modelos orientados a objetos para o hardware e software utilizados pela FAPP, empregando os conceitos e notações descritos nos capítulos 2 e 3.

4.2 Taxonomias e Modelo para o Hardware

A análise das taxonomias permite selecionar os elementos básicos comuns a todas arquiteturas. Esses elementos básicos compõem o conjunto de atributos da classe-base do modelo orientado a objetos para as arquiteturas paralelas.

4.2.1 Taxonomias para Arquiteturas Paralelas

Uma taxonomia ideal deve consistir dos seguintes atributos: hierárquico, universal, extensível e conciso (Bem-Dyke (1993)). sendo:

- Hierárquico: o modelo que iniciando de um nível mais abstrato, permite que as arquiteturas sejam refinadas em níveis. Arquiteturas semelhantes devem ser agrupadas até que as máquinas sejam muito bem descritas em seus detalhes.
- Universal: Cada computador deve possuir identificação única, de tal maneira que esse identificador o associe a uma taxonomia.
- Extensível: O mesmo nome deve ser aplicado à arquiteturas futuras com o menor número possível de alterações da nomenclatura.
- Conciso: utilizar nomenclatura simples e que permita realmente a sua utilização.

A taxonomia de Flynn está baseada no conjunto de instruções que pode ser unitário³ ou múltiplo⁴ e no conjunto de dados aos quais aplicam-se as instruções (Flynn (1972)). A Tabela 4.1 apresenta as possibilidades para as variações do número de instruções e da quantidade de dados.

Essa taxonomia tem sido utilizada principalmente pela abrangência de arquiteturas representadas, permitindo até mesmo a representação de arquiteturas MISD, embora não haja registro na literatura da implementação de máquinas com essa arquitetura.

Uma classificação mais refinada foi proposta por Duncan (1990), a qual conserva a taxonomia de Flynn e particiona as arquiteturas em síncronas e assíncronas, conforme ilustra o resumo apresentado na Tabela 4.2 (Souza; et al. (1996)). O modelo arquitetural baseado em sistemas distribuídos (Tanenbaum (1992)) (Müllender (1993)) pode ser classificado como sendo MIMD e, nesse caso, há que se preocupar com os mecanismos de comunicação (ou troca de mensagens), dado que esses mecanismos influenciam diretamente no desempenho do programa. O desenvolvimento

³Também é dito *simples*, pois é executada apenas uma instrução. ou código de operação.

de programas paralelos baseados em redes de computadores é atualmente também efetuado utilizando-se *máquinas virtuais*, tais como o PVM⁵ (Beguelin; et al. (1994)) (Sunderam; et al. (1994)) (Geist; et al. (1994)) e o MPI⁶ (McBryan (1994)) (Walker (1994)).

Tabela 4.1 - Taxonomia de Flynn.

SISD	“Single Instruction Single Data”: nesse caso há apenas uma instrução atuando sobre um dado. Exemplo: Bull DPX 500, Control Data série 4000, IBM-PC.
SIMD	“Single Instruction Multiple Data”: nesse caso uma única instrução atua sobre dados diferentes e é caracterizado por unidades de controle síncronas, porém distribuídas. Exemplo: Convex 3210 e o NEC SX-3/11.
MISD	“Multiple Instruction Single Data”: não há nenhuma máquina construída até o momento.
MIMD	“Multiple Instruction Multiple Data”: nesse caso existem cadeias não formatadas de instruções atuando em paralelo sobre dados distintos. Exemplo: Cray, Y-MP, C92 e o Parsys SN1000.

A Taxonomia de Johnson (1988) estende a classificação para arquiteturas paralelas que utilizam diversos processadores, organizados no estilo MIMD, classificando-as segundo a organização da memória. A memória do sistema obedece a uma hierarquia e pode ser global ou distribuída. O acesso aos dados depende da organização de memória utilizada, permitindo que o acesso às variáveis seja compartilhado ou efetuado através de troca de mensagens. Assim, as arquiteturas MIMD podem ser classificadas como:

- GMSV (*Global Memory Shared Variable*) para sistemas fortemente acoplados (Coulouris; et al. (1994));
- DMMP (*Distributed Memory Message Passing*) para sistemas fracamente acoplados;
- DMSV (*Distributed Memory Shared Variable*) para sistemas híbridos.

⁴ São executadas diversas instruções simultaneamente.

⁵“Parallel Virtual Machine” - PVM: Máquina Paralela Virtual

⁶“Message Passing Interface” - MPI: Interface para Passagem de Mensagens

A variedade de taxonomias encontradas na literatura é grande (Gilo (1981)) (Bem-Dyke (1993)). Embora a taxonomia proposta por Flynn seja bastante abrangente, optou-se por modelar a taxonomia de DUNCAN, pois esta encerra todas as características propostas por Flynn e estabelece adicionalmente uma visão macroscópica para uma hierarquia de classes.

Tabela 4.2 - Taxonomia de Duncan.

Arquiteturas Síncronas	- Processadores Vetoriais	
	- Processadores Matriciais (SIMD)	
	- Arquiteturas Sistólicas	- Memória Compartilhada
	- Arquiteturas MIMD	- Memória Distribuída
Arquiteturas Assíncronas		- Arquiteturas Híbridas
		- Arquiteturas a Fluxo de Dados
		- Arquiteturas de Redução
	- Arquiteturas não Convencionais	- Arquiteturas de Frente de Onda

4.2.2 Modelo Orientado a Objetos para a Taxonomia de Duncan

Para criar o modelo orientado a objetos para a taxonomia de Duncan, considerou-se uma plataforma básica para o hardware, composta por elementos fundamentais que definem o sistema, conforme ilustra o módulo *componentes do hardware* apresentado no capítulo 1 - Figura 1.1. Esses componentes são:

- **Processador:** elemento de processamento (Almasi; Gottlieb (1994)), não importando as características da sua organização e possuindo apenas capacidade de executar instruções a uma determinada taxa. Isto generaliza os processadores existentes em máquinas paralelas. Seus atributos representam características gerais comuns aos processadores, tais como: capacidade de endereçamento, *clock*, fabricante, família a que pertence, dentre outras.

- **Memória:** componente com capacidade predefinida para armazenamento ao qual o processador tem acesso direto. Toda memória possui uma função, denominada “tempo de acesso”, que define a velocidade com que a mesma pode atender às solicitações do processador. Tem como atributos principais: tempo de acesso, tamanho máximo e organização do(s) módulo(s).
- **Meio de transmissão:** estabelece o elo físico de comunicação entre duas ou mais interfaces, definindo o nível mais elementar de associação entre os componentes dos sistemas. É equivalente a: camada de enlace de dados (para máquinas paralelas baseadas em sistemas distribuídos fracamente acoplados), *links* (para máquinas paralelas baseadas em Transputers) (Inmos (1990)) ou ao meio físico de comunicação em sistemas fracamente acoplados.
- **Interface:** estabelece a conexão entre outros componentes e seus respectivos meios de transmissão de dados, controlando a comunicação entre os subsistemas. Devido ao fato de que diversos componentes podem utilizar um mesmo meio físico em um determinado *elemento de processamento*, a classe *interface de comunicação* deve ser especializada para permitir a utilização *protocolo de comunicação* e meios de *transmissão* distintos. Para os protocolos da família TCP/IP um objeto *interface de comunicação* representa as características encontradas no nível IP (Comer; Stevens (1991)). Para aplicações em ambiente Novell esse objeto representa o nível IPX.
- **Periféricos:** dispositivos de entrada e saída de dados aos quais o processador não tem acesso direto. O processador não possui a interface embutida em seu hardware, mas utiliza-se de uma interface para efetuar o acesso ao periférico.
- **Mecanismo de Interconexão:** abstração dos mecanismos de interconexão das máquinas paralelas citadas na literatura, e os barramentos amplamente utilizados em máquinas comerciais, tais como: ISA, EISA, PCI; dentre outros. Cada mecanismo impõe

restrições ao conjunto "Processador"/"Memória"/"Entrada/Saída" caracterizando o desempenho ótimo para cada conjunto. Em muitas arquiteturas o "Processador" tem acesso aos dispositivos "Entrada/Saída" através do espaço de endereçamento reservado à memória principal. Contudo, esse fato não é relevante para o projeto do software e é deixado para que seja tratado apenas durante a implementação. Nesse casos os objetos *entrada/saída* (instâncias das classes "Entrada/Saída") armazenam tais informações.

- **Software Operacional:** conjunto de *sistemas operacionais*, no sentido de que caracteriza uma máquina virtual com a capacidade de flexibilizar a utilização de algum *mecanismo de interconexão*. Representa informações intrínsecas relativas à adaptação de seus objetos aos objetos *mecanismo de interconexão*, tais como: endereços dos *drivers* controladores dos dispositivos de comunicação e outros elementos disponíveis num dado hardware, ou seja, representa o conjunto de dispositivos lógicos e suas respectivas associações ao "Mecanismo de Interconexão".
- **Elemento Processador:** essa classe é composta abstraindo-se as características gerais que podem ser obtidas da composição de um *software* operacional com um *mecanismo de interconexão*, resultando em *elementos de processamento*. Os *elementos de processamento* podem ser representados por algumas características tais como: suporte à multiprogramação, orientação a eventos, orientação a processos, mecanismos para troca de mensagens, desempenho do conjunto *software* operacional/*mecanismo de interconexão*.
- **Protocolos de Comunicação:** Representa o conjunto de protocolos que garantem a integridade e consistência dos dados armazenados na "Memória". Estabelecem o sincronismo inerente a todas as aplicações paralelas ou concorrentes, tais como:

"Comunicação entre Processos", "Chamadas a Procedimentos Remotos" e "Canais de Comunicação".

- Rede de Comunicação: para que haja comunicação efetiva entre os objetos *elemento de processamento* os objetos *protocolos de comunicação* são configurados para que operem com uma determinada *interface de comunicação*. Assim, os objetos *rede de comunicação* são elementos estritamente lógicos que definem quais protocolos estão associados a quais interfaces.
- Elemento de Processamento Paralelo: um objeto *elemento de processamento paralelo* é obtido através da interconexão de objetos *elemento de processamento* que utilizam objetos *rede de comunicação*. Dessa maneira, a classe "Elemento de Processamento Paralelo" representa uma máquina paralela ou um conjunto de máquinas paralelas interligadas. Nos casos em que se têm máquinas paralelas virtuais, como por exemplo PVM ou MPI, representam-se apenas as máquinas (e/ou estações de trabalho) que executam parte(s) da aplicação (Calônego Jr.; et al. (1996)).

O modelo proposto para as arquiteturas pressupõe a conexão de seus componentes, ou seja, não pode existir nenhum componente em alguma arquitetura que não esteja associado a pelo menos um outro componente através de algum meio de transmissão. Assim, cada um dos itens acima citados são modelados com a finalidade de serem utilizados como classe-base para elementos mais específicos, conforme ilustram as Figuras 4.1 e 4.2.

O prefixo FAPP (Ferramenta de Auxílio à Programação Paralela) é utilizado para diferenciar as classes-base dos componentes do hardware das demais classes. Os elementos processadores, a serem utilizados pelos programadores, devem ser modelados segundo esse conjunto de classes-base. Para cada tipo de elemento processador a ser acrescentado à ferramenta deve ser elaborada uma classe, derivada da classe GRFNoRotuladoPeso, que encapsule as demais características desse elemento; uma aplicação mais detalhada desse procedimento é apresentado no capítulo 6.

O apêndice A apresenta as implementações das classes-base para os componentes básicos, incluindo programas-exemplo para cada uma das classes, com os respectivos arquivos de saída gerados.

A classe GRFArcoRotuladoPeso, definida no capítulo 3, é base para a classe FAPPMeioTransmissao, pois define as características dos arcos de um dígrafo rotulado e ponderado. A especialização acrescenta os atributos intrínsecos aos meios de transmissão. Os demais componentes básicos devem pertencer a classes que representem suas respectivas características e cujas instâncias sejam identificadas univocamente, possuam um rótulo e permitam que seja definido o custo de utilização de cada unidade. Por exemplo: um mecanismo de interconexão de uma determinada máquina, possui um rótulo ("ISA", "PCI" ou outro) e define a taxa de comunicação máxima entre os elementos do sistema. Esses atributos são característicos dos objetos que pertencem à classe GRFNoRotuladoPeso, permitindo que as classes-base para os componentes do hardware sejam derivadas da classe GRFNoRotuladoPeso, conforme ilustra a Figura 4.3.

Modelados os componentes e os respectivos elementos de processamento, a arquitetura paralela é descrita por meio de um grafo. Os elementos processadores são os nós e os meios de ligação dos processadores são os arcos. A Figura 4.4 ilustra a representação gráfica de uma arquitetura paralela que contém 5 elementos processadores ($EP_i, 0 \leq i \leq 4$), 7 canais de comunicação ($C_i, 0 \leq i \leq 6$) e uma possível topologia.

Os objetos elemento-processador são sub-grafos ou supernós de acordo com o modelo proposto no capítulo 3. A Figura 4.5 representa a classe GRFSuperNo como superclasse de FAPPElemProcessador, sendo a

representação do encapsulamento de todos os componentes que compõem os objetos desse tipo.

Os elementos processadores podem possuir números diferentes de portas para a conexão física com os demais elementos. Assim, cada elemento é composto por um conjunto de interfaces, conforme é apresentado no apêndice A, classe FAPPElemProcessador.

A composição de uma arquitetura paralela utilizando-se um conjunto de elementos processadores é modelada por dígrafos conexos, caracterizando que todos os elementos de processamento estão interligados. A estrutura apresentada na Figura 4.6 representa uma hierarquia de classe que modela a taxonomia de Duncan. É utilizado o prefixo AP (Arquiteturas Paralelas) para caracterizar que a classe APArquiteturaParalela é uma especialização de um dígrafo conexo. As demais subclasses são prefixadas com AP para indicar qual a sua classe-base.

Os sistemas distribuídos fortemente acoplados (Müllender (1993)) apresentam como característica principal a proximidade dos processadores e a utilização de mecanismos de interconexão que permitem o fluxo de informações com taxas relativamente altas (Almasi; Gottlieb (1994)). Dentre as arquiteturas assíncronas distinguem-se as arquiteturas MIMD com memória compartilhada (Figura 4.7a) e MIMD com memória distribuída (Figura 4.7b) (Duncan (1990)). A diferença básica entre essas duas

arquiteturas (ou dois sistemas distribuídos fortemente acoplados) reside na maneira pela qual o acesso à memória é efetuado.

A facilidade com que as máquinas paralelas virtuais podem ser implementadas, utilizando como plataforma sistemas distribuídos fracamente acoplados, tem permitido a aplicação das teorias desenvolvidas para máquinas MIMD a esses sistemas e vice-versa. Essa abordagem especializa os domínios de desenvolvimento de aplicações para arquiteturas paralelas e torna possível representar esse conjunto de máquinas paralelas, de maneira simples, através da hierarquia de abstrações (Calônego Jr.; et al. (1996)).

Essa representação não pretende modelar as características do hardware de todas as máquinas MIMD construídas atualmente (Steen (1993)), mas sim servir como um meta-modelo-base de representação de arquiteturas, segundo o qual seja possível, a partir dos objetos, extrair informações imprescindíveis que auxiliem o projeto do software, o balanceamento de cargas e a simulação de sistemas.

4.3 Desenvolvimento de Programas Paralelos/Concorrentes

O desenvolvimento de programas paralelos/concorrentes é uma área de pesquisa muito explorada no meio científico, especialmente para as aplicações numéricas, havendo a necessidade de desenvolvimento de técnicas para o projeto e implementação de programas. As propostas nesse sentido buscam apresentar modelos de programação, normalmente específicos para as respectivas arquiteturas.

O modelo de máquina de von Neumann assume que um processador é capaz de executar uma seqüência de *códigos de operações*, que caracteriza o primeiro nível lógico de abstração do hardware do computador, denominado também de programa executável. Programar em linguagem de máquina demanda altos custos de desenvolvimento para o software e muitas vezes torna impraticável o desenvolvimento do mesmo. Para evitar a utilização das linguagens de máquina e facilitar o desenvolvimento do software, foram desenvolvidas linguagens de programação, tais como: FORTRAN, PASCAL, COBOL, C, Ada, Lisp, Prolog, C++. Essas linguagens

além de possuírem as características já citadas, implementam paradigmas de programação, permitindo que os problemas sejam abordados de maneiras diferentes. Entretanto, o processo de transformação de um programa-fonte em um programa executável é possível somente porque o compilador é capaz de estabelecer alguma equivalência entre as máquinas de estados finitos descritas pelas linguagens. Assim, quaisquer algoritmos podem ser escritos em alguma linguagem de programação e as linguagens podem ser modeladas segundo alguma máquina de estados finitos, fazendo com que cada programa implemente uma determinada máquina de estados finitos.

A programação paralela introduz fontes adicionais de complexidade, fazendo com que o programador tenha que se preocupar com o gerenciamento dos processos, processadores, balanceamento de cargas, desempenho, “dead-locks”, “live-locks”, comunicação e qual o modelo de computação que deve ser utilizado. “Um modelo computacional é a maneira através da qual a computação é efetuada” (Almasi; Gottlieb (1994)) e consiste dos seguintes atributos:

- Primitivas de computação: que são ações básicas do computados ou um conjunto de tipo de dados e operações definidos para o conjunto de instruções.
- Espaço de endereçamento: define a quantidade de memória disponível e como os dados são armazenados e recuperados.
- Unidades escaláveis: regras para particionar e escalonar as unidades do problema definindo unidades de controle sobre as primitivas.
- Comunicação: definição de padrões através dos quais as unidades de trabalho trocam informações.
- Sincronismo: mecanismo que impõem restrições temporais na ordem de execução das atividades do sistema.

Os itens a seguir apresentam a abordagem para programação paralela, proposta por Foster (1995), e o modelo orientado a objetos para os

conceitos propostos. A implementação desses conceitos traduz a técnica para a programação paralela implementada na ferramenta.

4.3.1 Esqueletos para Aplicações Paralelas

Para auxiliar o desenvolvimento de programas paralelos e oferecer uma maneira uniforme para a elaboração dos mesmos, Foster (1995) propôs uma série de passos baseados em quatro princípios: partição, comunicação, aglomeração, e mapeamento.

Assume-se que um programa paralelo consiste de uma ou mais tarefas que têm suas execuções concorrentes ou paralelas e podem ser criadas ou destruídas dinamicamente. As tarefas encapsulam uma memória local, as portas de comunicação com outras tarefas e um programa. Cada tarefa pode executar as operações comunicação (transmissão/recepção de mensagem), criar outras tarefas ou terminar sua execução. As tarefas podem ser alocadas nos diversos processadores e assume-se que o mapeamento não altera a semântica do programa. A transmissão das mensagens são assíncronas, porém a recepção é síncrona.

O conceito de memória local, definido para as tarefas (Foster (1995)), está associado ao conceito de *localidade de dado*, que reflete a disponibilidade para o acesso aos mesmos, podendo o acesso ser remoto ou local. Definem-se canais de comunicação para o acesso aos dados remotos.

No nível de projeto de programa as tarefas são modeladas por nós e os canais são modelados por arcos, conforme ilustra o dígrafo apresentado na Figura 4.8. As tarefas são pedaços de código que podem ser executadas seqüencialmente em um único processador. Caso duas tarefas, que compartilham um canal, sejam alocadas em processadores diferentes, então tem que haver um canal de comunicação físico interligando os processadores no qual os canais lógicos são mapeados.

O projeto do programa define um mecanismo único de comunicação entre as tarefas (canais), fazendo com que os programas sejam executados independentemente da localização desses dados. Assim, o algoritmo pode

ser proposto sem que haja preocupação do programador com a alocação das tarefas nos processadores.

Assumindo-se que cada processo é um comando básicos da linguagem, tem-se que cada comando é um nó para um grafo que gera processos. Assim, um processo é um supergrafo e as tarefas representam a aglomeração desses supergrafos, conforme ilustra a Figura 4.9.

Os objetos da classe SFTProcesso são consequência das partições geradas pelo algoritmo e os objetos da classe SFTTarefa são as aglomerações criadas para distribuir os processos nos processadores. O prefixo SFT caracteriza o conjunto de classes-base para modelar os modelos computacionais para o software.

As técnicas de programação seqüenciais limitam a capacidade de uma grande parte dos programadores em OCCAM, pois além de terem que criar algoritmos paralelos, têm que supor uma arquitetura adequada ao seu modelo e mapear os processos e canais de comunicação de seus programas a essa arquitetura. Muitas vezes torna-se necessário que os processos sejam transportados entre os elementos de processamento, visando-se a melhoria do desempenho do sistema. Esse trabalho pode ser difícil sem o auxílio de uma ferramenta automática (Baker; Malvem (1991)).

As máquinas paralelas virtuais genéricas (incluindo-se nestas os ambientes PVM (Beguelin; et al. (1994)), Linda (McBryan (1994)), MPI (Walker (1994)), e outros), implementadas a partir de sistemas distribuídos fracamente acoplados, podem diminuir o trabalho do programador. Entretanto, ainda existe a possibilidade de criação de uma máquina virtual que possa comprometer o desempenho do sistema.

4.4 Considerações Finais

Neste capítulo apresentaram-se os modelos computacionais, os respectivos meta-modelos e como uma ferramenta pode implementar os conceitos de programação paralela por esqueleto.

O paradigma proposto para a linguagem CSP permite que os algoritmos paralelos sejam elaborados independentemente da arquitetura. Esse avanço na área da programação paralela/concorrente, inicialmente proposto por Hoare (Hoare (1985)), é apresentado como uma técnica para desenvolvimento de programas (Foster (1995)). Os conceitos apresentados por Foster mostram que é possível o desenvolvimento de programas paralelos de maneira análoga ao desenvolvimento de programas seqüenciais, ou seja, resolver o problema sem associá-lo a uma arquitetura específica. A técnica proposta por Foster (1995), é análoga à programação por esqueleto desenvolvida por Zima; Chapman (1992).

Os problemas são particionados em processos que se comunicam através de mensagens, e a aglomeração dos processos compõem as tarefas. Entretanto, não há nessa técnica suporte para o mapeamento dos

processos nos processadores. Assim, há a necessidade do programador intervir para definir esse mapeamento. Nesse sentido a ferramenta pode auxiliar, informando ao programador dos recursos de hardware disponíveis e facilitar a geração dos processos controladores dos canais de comunicação, conforme é apresentado para a aplicação descrita no capítulo 6.

5. Interfaces

Este capítulo apresenta o modelo de interfaces utilizado para a implementação da ferramenta, caracterizando um conjunto de interfaces para os componentes básicos, tratando de maneira uniforme os elementos e oferecendo as diretrizes para as novas interfaces que venham a ser desenvolvidas em função da extensão do ambiente.

5.1 Considerações Iniciais

O método FUSION descreve o *modelo de interface* para definir os elementos que estabelecem a interação entre o sistema e o ambiente no qual o mesmo está inserido. Essa abordagem isola o projeto e implementação do sistema da interface, a qual é utilizada para interagir com o ambiente. A palavra *interface* é utilizada para estabelecer os limites da implementação do sistema.

Segundo Coleman et al. (1996), os elementos que fazem parte do ambiente são denominados *agentes*, que podem solicitar através de *eventos*, que o sistema efetue *operações*. Os eventos são unidades atômicas de comunicação que podem ou não possuir argumentos, conforme ilustra a Figura 5.1.

O projeto de interfaces tem significado mais abrangente do que a definição de um conjunto de primitivas, requerendo esforços colaboracionistas por parte dos usuários, no sentido de auxiliarem os projetistas na definição de uma interface amigável para o sistema. Segundo Hix; Hartson (1993), os projetistas de interfaces devem ter como diretrizes:

- utilização fácil;
- permitir que o usuário execute suas tarefas mais rapidamente;

- possuir baixa taxa de erros;
- satisfazer o usuário;
- manter o usuário por mais tempo utilizando o sistema.

Figura 5.1 - Modelo de interface.

Mesmo estabelecendo-se diretrizes, ou padrões, para o projeto de interfaces, os fatores humanos fazem com que as interfaces sejam elaboradas e validadas empiricamente. Para Hix; Hartson (1993) o projeto da interface vai além daquele proposto por Coleman et al. (1996), subdividindo a elaboração da interface no projeto dos componentes e implementação dos comandos, conforme ilustra a Figura 5.2.

Figura 5.2 - Modelo de interface de Hix; Hartson (1993).

Para este trabalho são utilizados os componentes predefinidos no ambiente de programação Borland¹ para a linguagem C++ e disponíveis para o sistema operacional Windows95. Esse ambiente permite que as implementações das interfaces sejam elaboradas de acordo com o modelo de HIX; Hartson (1993).

Para a implementação e testes das classes-base da ferramenta foi utilizada uma interface textual, conforme ilustra o apêndice A, dado que seria necessário um projeto de interface para cada classe. Entretanto, utilizar interface textual faz com que a interação da ferramenta com o ambiente seja menos amigável, pois o usuário é responsável por garantir a consistência e a integridade de todo o trabalho de edição efetuado. Visando melhorar e flexibilizar o ambiente, elaborou-se o projeto e implementou-se uma interface mais sofisticada, baseada em menus, para ilustrar a utilização do protótipo da ferramenta. Os itens a seguir apresentam detalhes dessa interface definidas para a ferramenta.

5.2 Interface Baseada em Menu

A interface da ferramenta oferece ao usuário uma possível visão do ambiente no qual ele está desenvolvendo suas aplicações, sendo específica para cada modelo arquitetural. A Figura 5.3 ilustra esse esquema de visões, análogo às visões dos Sistemas de Gerenciamento de Banco de Dados - SGBD, apresentadas por Korth (1986).

Os componentes são descritos e armazenados em arquivos com a finalidade de serem utilizados para compor o hardware e o software das aplicações, sendo reutilizados quando não houver alteração do ambiente de desenvolvimento da mesma. Cada visão representa um ambiente de desenvolvimento distinto.

A interface para a definição dos componentes estabelece quais tipos de componentes podem ser utilizados. Cada arquitetura é composta a partir desses componentes básicos, contudo, tais elementos são utilizados

¹ Marca registrada.

apenas para definir os atributos de arquiteturas específicas. Tem-se, sob essa ótica, três visões distintas do sistema para cada ambiente: visão da ferramenta, visão do projetista da arquitetura e a visão do programador.

A visão da ferramenta corresponde à estrutura de classes e grafos de interação entre os objetos, que permitem ao projetista especializar o sistema em função da necessidade do ambiente. Os projetistas são, portanto, especialistas em projetos orientados a objetos que têm conhecimento e suporte suficientes para implementarem as novas arquiteturas, estendendo as características da ferramenta. Os projetistas definem:

- os componentes para o software, que reflete o nível de abstração do ambiente de programação que o programador poderá utilizar. Para isso o projetista tem que definir os gabaritos que serão utilizados para a geração do esqueleto da aplicação.
- os componentes do hardware, que definem os tipos de dispositivos dos quais o projetista fará uso ao definir a arquitetura paralela;

- os elementos processadores, que estabelecem quais e quantos elementos podem ser utilizados para processar as informações;
- as interfaces de comunicação, que são utilizadas por elementos processadores para terem acesso ao meio de transmissão;
- a arquitetura, que define a maneira através da qual o programador informa à ferramenta como estão interligados os elementos processadores.

A visão do programador é aquela especificada pelo projetista, que permite a utilização:

- dos componentes de software para descrever os programas e/ou tarefas do sistema;
- dos elementos de processamento para descrever quantos são utilizados pela aplicação e as suas interligações.

Definir uma interface para a ferramenta pressupõe a descrição de todos os comandos/interações possíveis. Isso requer a descrição de todas as arquiteturas possíveis, uma vez que é necessário que existam armazenados todos os componentes de hardware e de software. Como isso é uma questão de evolução tecnológica, e o número de combinações cresce em função da combinação desses elementos, essa prática torna-se inviável. Analogamente à proposta para os modelos arquiteturais, optou-se por definir um conjunto de classes-base para a confecção de uma interface que possa evoluir de acordo com as especializações arquiteturais acrescentadas à ferramenta.

Para que o projeto da interface seja o mais independente possível da aplicação, optou-se por definir um conjunto de janelas com características semelhantes, vinculadas a um documento principal, cujas características visuais podem ser observadas na janela “FAPP”, ilustrada na Figura 5.4. Os itens que aparecem na parte superior da janela são denominados *itens do menu principal*.

As classes-base com prefixo FAPP, apresentadas no capítulo 4, descrevem os atributos comuns aos elementos que compõem o hardware, permitindo o incremento do número de atributos em função do nível de

especialização de suas subclasses. Esses atributos são comuns a todas os elementos que compõem uma arquitetura. Assim, descreveu-se um conjunto de objetos básicos instanciados das classes FAPP para ilustrar a interface para esses componentes. O exemplo de aplicação apresentado no capítulo 6 ilustra algumas especializações para a ferramenta.

Figura 5.4 - Elementos da interface da ferramenta.

Cada item do menu principal é composto por sub-itens que geram eventos (mensagens para os objetos da janela cliente) que são capturados pelo sistema operacional e passados para a janela cliente através da janela da aplicação (definida para o sistema), conforme ilustra a Figura 5.5.

O tratamento desses eventos implica na criação de janelas específicas para cada tipo de componente, conforme ilustram as janelas

FAPPIGProcessador e FAPPIGMemoria. As letras IG informam que as classes FAPPIGMemoria e FAPPIGProcessador são as respectivas interfaces para objetos instanciados a partir da classe FAPPMemoria e FAPPProcessador.

Cada uma das linhas das tabelas é uma tupla² na qual os usuários podem efetuar alterações, através da edição dos atributos, ou utilizar para a elaboração de novos componentes, conforme ilustra a Figura 5.6.

Ao comparar-se as janelas FAPPIGMemoria e FAPPIGProcessador da Figuras 5.4, pode-se observar a uniformidade da interface, fazendo com que o usuário aprenda a utilizar esses componentes com facilidade.

Excetuando-se o item “edit” do menu principal, os demais elementos estão dispostos ordenadamente da esquerda para a direita. Essa disposição tem por finalidade induzir a uma seqüência lógica de passos para a geração de aplicações, conforme o exemplo que é apresentado no capítulo 6.

5.3 Interface Gráfica

Apesar da adoção de uma interface gráfica para a FAPP ser altamente recomendável, em função dos benefícios que isto pode trazer, ela

² Conjunto de atributos de um objeto.

não foi completamente implementada neste trabalho.

Figura 5.6 - Exemplo de ementos de processamento.

A interface baseada em menus, apresentada no item 5.2, é suficiente para a validação da proposta e, mesmo não sendo tão amigável, oferece ao usuário condições para descrever e manipular os elementos que compõem o seu software.

A implementação de uma interface gráfica, conforme sugerida a seguir, requer um volume de trabalho de programação considerável, devendo ser elaborada em trabalhos futuros.

A análise dos elementos que compõem os grafos permite que a cada um deles esteja associada um forma gráfica. Assim, apresentar na tela do computador os grafos ou dígrafos não é tarefa complicada. Adotando-se a classe ilustrada na Figura 5.7 como base para as classes GRFNo, GRFArco, GRFSuperNo, GRFSuperArco, as classe derivadas têm que sobrescrever os métodos que das aos objetos da classe uma determinada forma. Entretanto, há necessidade de que os elementos que compõem essa interface possuam significados específicos. Isso demanda um projeto

minucioso dos componentes e das suas representações. Por exemplo, elementos processadores que diferem apenas pela quantidade de memória local, devem possuir alguma diferenciação visual que informe essa diferença para o usuário.

A classe FGPonto apenas define as características de um ponto na tela. A Classe FGPontos define um par de pontos e os objetos de desenho que podem ser utilizados para descrever a forma e o preenchimento da figura, essas características são imprescindíveis para a programação em ambiente Windows, pois definem o contexto do dispositivo de saída gráfica.

O método *forma* é virtual puro, permitindo que as subclasses da classe FGPontos definam formas geométricas variadas, que podem ser traçadas utilizando-se um par de pontos.

O ambiente de desenvolvimento IDE³, versão 4.52, possui as classes TPoint, TDC, que possuem essas características. A classe TDC utiliza o contexto de dispositivo e encapsula diversos métodos que implementam formas geométricas.

³ IDE - Integrated Development Environment - faz parte do pacote de desenvolvimento da Borland.

A estrutura de classes, apresentada na Figura 5.8, foi utilizada para implementar um protótipo de um editor gráfico, com a finalidade de explorar as dificuldades que eventualmente poderiam ocorrer ao utilizar-se a ferramenta com uma interface gráfica.

A classe TStreamableBase encapsula o sistema de gerenciamento de arquivos, permitindo que os objetos sejam armazenados em cadeias não formatadas ("streams"). Assim, os elementos desenhados na tela podem ser armazenados e recuperados de arquivos. A classe FAPPIGrafica é a base para todos os objetos de desenho. Cada uma das suas FAPPIGReta, FAPPIGElipse e FAPPIGRetangulo apenas definem as respectivas formas que devem ser apresentadas na tela.

A classe FAPPIGRotulo permite criar rótulos ou legendas para as figuras. As figuras rotuladas são criadas através do mecanismo de herança múltipla. A Figura 1.8 ilustra apenas como criar um retângulo rotulado, entretanto, a criar dos demais componentes é análoga.

A definição dos objetos de desenho não é suficiente para a utilização do sistema, é necessário que sejam criadas as classes que estabelecem a interface entre o sistema e o usuário. A Figura 1.9 ilustra a estrutura hierárquica para os objetos de interface que constróem as figura solicitadas pelos usuários.

A classe TClienteDC é base para quaisquer classes que utilizem algum contexto de dispositivo. A classe FAPPCnstDC especializa as características da classe Tclient para o contexto a aplicação FAPP.

A classe FAPPIGConstrutor encapsula as características dos contextos dos dispositivos, permitindo que cada construtor utilize o seu próprio conjunto de ferramentas de desenho, definidos pelas classes FAPPCnstReta, FAPPCnstElipse, FAPPCnstRetangulo.

A classe TDialog cria objetos de diálogo na forma de caixas de texto, que devem ser especializadas de acordo com o diálogo que se pretende estabelecer com o usuário do sistema. Assim, a classe FAPPDialRotulo define como deve ser o diálogo entre o usuário e o sistema para que seja definido o rótulo do objeto de desenho.

A classe `FAPPCRentanguloRot` é um exemplo de derivação múltipla com qual criam-se objetos de desenho retangulares e rotulados. Os demais objetos de desenho seguem o mesmo padrão.

5.4 Considerações Finais

Interfaces baseadas em menus podem ser alteradas com facilidade, pois não requerem alterações dos eventos gerados, mais sim do tratamento a eles aplicados. Para substituir uma janela da aplicação basta criar a classe que estabelece a interface e substituir o objeto que recebe a mensagem (ou evento) pelo novo objeto instanciado da nova classe.

Interfaces gráficas são mais amigáveis e dão ao ambiente mais elegância, porém são inevitavelmente mais específicas, pois deve-se criar um objeto de interface diferente para cada tipo de componente. Como a variedade de elementos é grande e evolutiva em função da tecnologia disponível (novos processadores, novas interligações, novas linguagens, dentre outros) a interface gráfica deixa, eventualmente, de atender às expectativas dos usuários.

A inconveniência do modelo descrito no item 5.3 é que não estão previstos mapas de bits. Contudo, essa estrutura permitiu o domínio das necessidades relativas ao projeto e implementação de interfaces gráficas e pode ser utilizada como diretriz para a confecção de uma interface mais sofisticada para a ferramenta.

6. Um Exemplo de Aplicação

Este capítulo apresenta um protótipo para a ferramenta FAPP em um ambiente de programação paralela. Os passos apresentados neste exemplo devem ser repetidos para instanciá-la para outros ambientes.

6.1 Considerações Iniciais

Nessa abordagem os módulos dos programas e os parâmetros formais compõem o esqueleto da aplicação. Os módulos são definidos pelo programador através da interface da ferramenta.

Procedimentos e funções que executam concorrentemente, utilizam um determinado conjunto de *headers*¹, que necessitam ser gerados independentemente para cada conjunto de processos que compõem uma tarefa e que serão executados em paralelo.

A utilização da ferramenta é efetuada através das fases seguintes:

- Definição dos componentes do hardware. O hardware é composto por elementos processadores que são utilizados para definir as arquiteturas. Uma arquitetura é composta por diversos elementos processadores e a interligação desses elementos processadores define o modelo arquitetural do hardware no qual a aplicação é mapeada. O item 6.2 descreve esses detalhes utilizando uma arquitetura paralela baseada em Transputers (Trindade; et al. (1992)).
- Definição dos componentes do software. A definição da linguagem é necessária para que sejam gerados os modelos dos comandos,

¹Headers - cabeçalhos de subprogramas utilizados para a definição de constantes, variáveis e configuração do ambiente

que definem o nível de abstração para a descrição dos programas. Assume-se, no exemplo, que o nível mais baixo de abstração são os processos, entretanto, o modelo prevê manipulação ao nível de comandos da linguagem. Esse nível corresponde à descrição dos componentes do software, conforme ilustra a Figura 1.1. O item 6.3 descreve os comandos da linguagem OCCAM2, que é utilizada para gerar o exemplo da aplicação.

- **Elaboração do esqueleto para a aplicação.** São descritos todos os processos que compõem a aplicação, caracterizando o modelo de partições descrito no capítulo 4. O item 6.4 apresenta um exemplo de aplicação modelado por partição de processos.
- **Elaboração do modelo arquitetural do hardware.** A interconexão dos elementos processadores define o modelo de máquina no qual os processos devem ser mapeados. O item 6.5 descreve como elaborar esse modelo.
- **Elaboração do mapeamento.** O mapeamento dos processos nos processadores não exige a descrição de tarefas, conforme proposto por Foster (1995), dado que essa atividade corresponde a aglomerar e encapsular os processos, conforme descrito no capítulo 4. Para o ambiente proposto para aplicações análogas a essa, o sistema é capaz de gerenciar automaticamente essas atividades, conforme é apresentado no item 6.6.
- **Geração da aplicação.** A geração da aplicação é dependente do ambiente, sendo o programador responsável por preencher o esqueleto gerado com os algoritmos específicos.

6.2 Componentes do Hardware

A descrição dos componentes é utilizada para compor os elementos processadores que descrevem o modelo arquitetural do hardware da aplicação. A seguir destacam-se e modelam-se os atributos principais de cada ente do mundo real que é parte do hardware do sistema.

O Servidor de Processamento Paralelo (SPP), adotado como uma plataforma de testes para validação do exemplo, é uma máquina com arquitetura maciçamente paralela e multi-usuário (Trindade; et al. (1992)). Esse sistema pode ser logicamente desmembrado em dois subsistemas: o Processador de Entrada (PE) e o Banco de *Transputers* (BT). O BT permite que no máximo quatro usuários compartilhem um total de no máximo 916 *Transputers*, interligados por uma chave, configurável dinamicamente através de software.

O BT possui um subsistema controlador responsável por gerenciar as atividades do banco e estabelecer a interface com o PE, conforme ilustra a Figura 6.1. O PE é um microcomputador no qual tem-se em execução um programa servidor para as operações básicas de segurança do sistema, operando também como um servidor para o BT nos casos em que há necessidade de acesso às informações armazenadas em disco, podendo ainda operar como roteador de pacotes, nos casos em que as primitivas são executadas no BT.

Figura 6.1 - Estrutura lógica do Servidor de Processamento Paralelo.

Observa-se nesse sistema três tipos básicos de elementos de processamento: estações de trabalho (ET), o processador de entrada (PE) e os Transputers. O Processador de entrada é composto e organizado tal

como uma estação de trabalho, acrescido da característica de servir como interface entre as estações de trabalho e o *Banco de Transputers*. A Figura 6.2 apresenta a estrutura hierárquica que é utilizada para modelar esse conceito. O prefixo SPP indica que os elementos de processamento, que encapsulam a classe FAPPElemProcessador, modelam os conceitos utilizados para a elaboração do Servidor de Processamento Paralelo (Trindade; et al. (1992)). A classe EPT800 representa o conjunto de todos os processadores modelo T800 produzidos pela INMOS² (Inmos (1990)).

A criação de objetos das classes apresentadas na Figura 6.2 pressupõe a existência de objetos instanciados das classes-base, tais como: FAPPProcessador, FAPPMemoria, FAPPIInterface, FAPPSoftwareOper; cujos exemplos de instâncias são apresentadas na Tabela 6.1 e Tabela 6.2.

Tabela 6.1 - Exemplo para instâncias para a classe FAPPProcessador.

Identificador	Tamanho	Unidade	Tempo de Acesso	Palavra
1	512	K	20	8
2	4	M	100	16
3	8	M	60	32

² Marca registrada.

Tabela 6.2 - Instâncias da classe FAPPMemoria.

Identificador	Familia	Fabricante	Clock	benchmark
1	80486	Intel	66	10
2	80586	Cyrrix	100	18
3	T800	INMOS	10	8

As demais instâncias para os exemplos de objetos das classes-base são apresentadas no apêndice A. Os atributos *identificador* e *benchmark/Tempo de acesso* são herdados da classe GRFNoRotuladoPeso e correspondem ao identificador do nó e ao seu peso, respectivamente. O rótulo é atributo para ser utilizado apenas como interface com o usuário, tendo sido omitido neste exemplo.

Definidas as características dos componentes básicos, modelam-se os elementos processadores. A Tabela 6.3 ilustra os atributos de um elemento processador do tipo T800 (Inmos (1990)).

A coluna T800 indica qual tipo de processador é utilizado como componente básico para o elemento processador, referenciando a linha 3 da Tabela 6.3. A coluna 2 indica o tipo de memória e assim sucessivamente. Isso mantém a consistência entre o modelo arquitetural e os componentes predefinidos.

Tabela 6.3 - Estruturas básicas para o elemento processador T800.

T800	memoria	link0	link1	link2	link3	canal	nome	codigo
3	2	3	3	3	3	3	P0	1
3	2	3	3	3	3	3	P1	2
3	2	3	3	3	3	3	P2	3
3	2	3	3	3	3	3	P3	4

6.3 Componentes do Software

A seguir serão apresentados os comandos da linguagem OCCAM-2, suas sintaxes e os respectivos exemplos. Esses elementos são os componentes básicos para a elaboração do software.

6.3.1 Notação

<xxx> : denota que a informação xxx é obrigatória. Por exemplo, <tipo> tem que ser substituído por algum tipo de dado.

[, xxx] : a informação xxx é opcional e caso haja uma lista de argumentos formais ou identificadores de variáveis, os itens da lista são separados por virgula “,”.

[; xxx] : a informação xxx é opcional e caso haja uma lista de tipo de dados que compõem os protocolos, os itens da lista são separados por ponto-e-virgula “;”.

Identificador : cadeia alfanumérica válida para a linguagem que identifica subprogramas ou variáveis.

TIPO : Tipo de dado.

Os demais símbolos são descritos no alfabeto da linguagem e na sua carta sintática (Jones (1988)).

6.3.2 Tipos de Dados, Canais e Protocolos de Comunicação

Além dos tipos primitivos de dados a linguagem permite a utilização de conjuntos e tipos estruturados de dados, que podem também ser utilizados para a definição de canais lógicos de comunicação.

6.3.2.1 Tipos Primitivos de Dados

- **Tipo de dado lógico**
 - ⇒ BOOL assume TRUE ou FALSE.
- **Tipo de dado caracter**
 - ⇒ BYTE são 8 bits.
- **Tipo de dado inteiro**
 - ⇒ INT16 são 16 bits.
 - ⇒ INT32 são 32 bits.
 - ⇒ INT64 são 64 bits.
- **Tipo de dado real**
 - ⇒ REAL16 números representados com 16 bits.

⇒ REAL32 números representados com 32 bits.

⇒ REAL64 números representados com 64 bits.

A comunicação através de canais é efetuada segundo o protocolo definido para o canal. A definição de canais para a troca de informações é baseada em tipos primitivos de dados e tem a seguinte sintaxe:

CHAN OF <TIPO> <nome>[, <nome>] :

onde,

nome: é um identificador para o canal.

Exemplos:

- Canal de comunicação para dados do tipo *inteiro*.

CHAN OF INT CanalLogico1:

- Dois canais de comunicação para dados do tipo *inteiro* representados com 32 bits.

CHAN OF INT32 CanalLogico2, CanalLogico3:

- Canal de comunicação para dados de números *reais* (notação em ponto flutuante) representados com 64 bits

CHAN OF REAL64 CanalLogico4:

- Canal de comunicação para dados do tipo *lógicos*.

CHAN OF BOOL CanalLogico5:

- Canais do tipo *stream*.

CHAN OF ANY CanalParaQualquerStream:

6.3.2.2 Tipos Estruturados de Dados

Nos casos em que os canais utilizam estruturas de dados mais complexas que os tipos primitivos, há a possibilidade da definição dessas estruturas através da utilização do comando **PROTOCOL**. A sintaxe para o comando **PROTOCOL** é:

PROTOCOL <identificador> **IS** <tipo>[; <tipo>]:

Embora exista o tipo primitivo de dado ANY em OCCAM, não é permitido que sejam declarados protocolos contendo canais do tipo ANY. Em situações abstratas é interessante a possibilidade da existência de protocolos que utilizam o tipo primitivo ANY. Entretanto, os protocolos são

padrões que devem ser muito bem definidos para estabelecerem as interfaces de comunicação entre processos, não cabendo portanto a utilização de tipos genéricos abstratos. A Figura 6.3 ilustra as abstrações de especialização elaboradas para modelar esses conceitos.

Exemplos:

- Os protocolos mais simples podem sobrescrever os tipo primitivos de dados da linguagem OCCAM.

PROTOCOL protocolo.int **IS** INT;

PROTOCOL protocolo.misto **IS** BOOL; REAL64;

CHAN OF protocolo.int dado;

CHAN OF protocolo.misto dado1, dado2;

- Os protocolos mais complexos são definidos a partir da combinação dos tipos primitivos da linguagem OCCAM.

PROTOCOL NumeroComplexo **IS** REAL32; REAL32;

PROTOCOL MeuProtocolo **IS** INT; REAL32; BOOL;

CHAR;

Cada tipo de canal a ser utilizado na aplicação deve ser definido como componente básico e compõe a lista de protocolos e tipos permitidos.

6.3.3 Subprogramas e Tarefas

O comando PROC define que um determinado subconjunto de comandos constitui um processo com granulosidade mais grossa que os comandos. Nas linguagens OCCAM2 e CSP os comandos assumem o "status" de processos (Jones (1988)) havendo palavras reservadas que definem algoritmos com granulosidade maior.

- O Comando PROC

PROC <identificador> (< lista de argumentos formais>)

-- Insira aqui a definição das variáveis locais

-- Insira o texto do procedimento aqui

:

A definição do comando estabelece o gabarito para a geração dos esqueletos dos subprogramas, para quaisquer programas OCCAM. A Figura 6.4 ilustra o modelo utilizado para inserir esse comando no contexto da ferramenta FAPP.

Exemplo

PROC soma (INT x, y , VAL z)

-- Insira aqui a definição das variáveis locais

-- Insira o texto do procedimento aqui

z := x + y

:

- O Comando FUNC

Funções são subprogramas que retornam valores através dos registradores da unidade central de processamento ou através da pilha de execução do programa, constituindo uma estrutura de dados da memória local de cada elemento processador. Isso implica na codificação de todos os processos que trocam mensagens através de procedimentos.

<tipo>[, <tipo>] **FUNCTION** <identificador_funcao> (< lista de argumentos formais >)

-- Insira aqui a definição das variáveis locais

-- Insira o texto do procedimento aqui

result <identificador>[, <identificador>]

:

Exemplo

INT FUNCTION soma (INT x, y)

-- Insira aqui a definição das variáveis locais

INT z :

-- Insira o texto do procedimento aqui

$z := x + y$

result z

:

Um programa é constituído por diversos módulos, caracterizando um dígrafo. A Figura 6.4 ilustra essa característica, mostrando que todos os procedimentos ou funções constituem um supernó para um dígrafo.

6.3.4 O Comando PLACE

O programador deve ter em mente que cada Transputer possui 4 (quatro) *links*, conforme ilustra a Figura 6.5. Cada *link* possui um canal físico de entrada e um canal físico de saída, ou seja, cada um dos canais dos *links* é um meio de transmissão *simplex*. Cada um dos *links* é uma interface *full-duplex*, pois os Transputers podem receber e transmitir

O comando **PLACE** permite que o programador estabeleça a associação entre um canal lógico e um determinado *link* de um Transputer. A sintaxe do comando é:

PLACE <canal.logico> **AT** <canal.fisico> :

onde, canal.logico é um identificador de variável do tipo CHAN.

O exemplo dado a seguir informa que o canal físico *link0.in* pode receber quaisquer tipos (**ANY**) de informações. Na verdade, *link0.in* é uma constante, sendo os comandos dentro das caixas abaixo equivalentes.

CHAN OF ANY canal :

PLACE canal **AT** link0.in : ⇔ **PLACE** canal **AT** 0 :

CHAN OF INT cin, cout:

PLACE cout **AT** link0.out : ⇔ **PLACE** cout **AT** 1 :

6.4 Partição dos Processos

Este exemplo foi escrito e testado no Servidor de Processamento Paralelo (SPP), mas pode ser executado em uma malha de Transputers, desde que os *links* dos mesmos sejam interligados de acordo com a definição do programa.

As elipses representam os processos. Os arcos representam as comunicações, ou ligações lógicas, entre os processos. O processo *applic1* atribui um valor inicial para a variável *X* e transmite seu valor para *applic2*. O processo *applic2* adiciona o valor 10 a *X* e transmite esse valor para *applic3*. O processo *applic3* adiciona mais 10 ao valor recebido e transmite-o para *applic1*, conforme ilustra a Figura 6.6.

Figura 6.6 - Projeto lógico das partições.

O ambiente de desenvolvimento permite que os processos sejam compilados separadamente e os códigos devidamente "carregados" nos respectivos elementos de processamento.

Cada um dos subprogramas da aplicação deve possuir pelo menos um canal lógico que o associe a outro subprograma. Caso isso não ocorra é porque existe código que não é utilizado durante a execução da aplicação. Assim, toda aplicação possui as características de ser um dígrafo conexo, composto por tarefas, conforme ilustra a Figura 6.7.

Tabela 6.4 - Objetos da classe SFTProcArgs.

Código	Tipo	Nome
1	CHAN OF SP	ent
2	CHAN OF SP	sai
3	CHAN OF INT	x
4	CHAN OF INT	z
5	CHAN OF INT	y
6	CHAN OF INT	x
7	CHAN OF INT	z
8	CHAN OF INT	y

A comunicação entre os processos é definida pelos canais lógicos X, Y e Z. A Tabela 6.4 apresenta a lista de argumentos formais para os processos, obedecendo à seguinte relação de ordem:

- applic1 argumentos 1,2,3 e 4;
- applic2 argumentos 5 e 6;
- applic3 argumentos 7 e 8.

Essas informações são suficientes para que os objetos que representam os processos, e que gerenciam suas próprias listas de argumentos, gerem os respectivos gabaritos, ilustrados na Tabela 6.5, Tabela 6.6 e Tabela 6.7.

Tabela 6.5 - Esqueleto para o processo applic1.

```
#USE strmhdr
#USE linkaddr
#USE strmhdr
#USE sphdr
#USE splib
#USE spinterf
#USE solib
PROC applic1(CHAN OF SP ent, CHAN OF SP sai , CHAN OF INT x, CHAN OF INT y)
  -- Insira aqui a identificação das variáveis

  -- Insira aqui o texto da aplicação
:
```

Tabela 6.6 - Esqueleto para o processo applic2.

```
#USE strmhdr
#USE sphdr
#USE splib
#USE spinterf
PROC applic2(CHAN OF INT x, y)
  -- Insira aqui a identificação das variáveis
  -- Insira aqui o texto da aplicação
:
```

Tabela 6.7 - Esqueleto para o processo applic3.

```
#USE sphdr
#USE splib
#USE spinterf
PROC applic3(CHAN OF INT y, z)
  -- Insira aqui a identificação das variáveis
  -- Insira aqui o texto da aplicação
:
```

O conjunto de cabeçalhos é padrão, devendo o programador editá-lo em função da necessidade de outras bibliotecas.

6.5 Modelo para o Hardware

A arquitetura paralela, para o modelo computacional utilizado nessa aplicação, instancia tantos elementos EPT800, Figura 6.8, quantos forem os processadores necessários e existentes no banco de processadores.

A topologia utilizada é apresentada na Figura 6.8. Os valores agregados às variáveis $link_{x.in}$ e $link_{x.out}$ (Figura 6.5), onde $0 \leq x \leq 3$, estão definidos nas bibliotecas *strhdr* e *linkaddr* do ambiente de desenvolvimento. Esses valores podem ser definidos diretamente no código fonte, pelo programador, através do uso de constantes. Entretanto, assumem-se os valores ilustrados na Figura 6.5 caso isso não ocorra.

A rede de comunicação define os elos de ligação entre os elementos processadores, determinado o processador-origem, o processador-destino, qual é o canal de origem, qual é o canal destino e o identificador (nome) do elo de ligação (meio de transmissão), conforme ilustra a Tabela 6.8.

Tabela 6.8 - Configuração dos elementos processadores.

Código	T800Origem	LinkSaída	T800Destino	LinkEntrada	NomeCanal
1	4	0	1	3	from.host
2	1	3	4	0	to.host
3	1	0	2	3	c0
4	2	0	3	0	c1
5	3	1	1	2	c2

Definido o mapa de interligação dos elementos processadores, ou modelo arquitetural do hardware, esse conjunto de informações permite a elaboração de um gabarito, conforme ilustra a Tabela 6.9.

Tabela 6.9 - Definição das tarefas.

```

CHAN OF ANY from.host, to.host, c0, c1, c2 :
PLACED PAR
  PROCESSOR 0 T8
    PLACE c0 AT link0.out :
    PLACE c1 AT link2.in : -- LINK0
    PLACE from.host AT link3.in : -- LINK3
    PLACE to.host AT link3.out :
    -- Insercao das chamadas dos procedimentos que
    -- que executam no processador 1

  PROCESSOR 1 T8
    PLACE c0 AT link3.in : --3
    PLACE c2 AT link0.out : --3
    -- Insercao das chamadas dos procedimentos que
    -- que executam no processador 2

  PROCESSOR 2 T8
    PLACE c2 AT link3.in : --3
    PLACE c1 AT link1.out : --3
    -- Insercao das chamadas dos procedimentos que
    -- que executam no processador 3

```

6.6 Mapeamento dos Processos nos Processadores

O mapeamento dos processos nos respectivos elementos processadores é elaborado em duas etapas:

- definir quais processos estão associados a quais processadores,
- quais canais lógicos estão mapeados em quais canais físicos.

Os canais listados na primeira linha da Tabela 6.9 têm que estar associados aos respectivos argumentos formais dos procedimentos, estabelecendo o mapeamento canal lógico/físico. Para isso é elaborada a classe SFTMapa que equivale às estruturas apresentadas na Tabela 6.10 e Tabela 6.11.

Tabela 6.10 - Mapa processos/processadores.

Código	Processador	Processo
1	1	1
2	2	2
3	3	3

Tabela 6.11 - Mapa canal lógico/físico.

Código	Canal Lógico	Canal Físico
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	3
7	7	4
8	8	5

Definidos os mapas, a ferramenta permite que seja gerado o esqueleto da aplicação, conforme ilustra o texto do programa fonte dado a seguir.

```
#USE strmhdr
#USE linkaddr
#USE strmhdr
#USE sphdr
#USE splib
#USE spinterf
#USE solib
PROC applic1(CHAN OF SP from.isv, CHAN OF SP to.isv, CHAN OF INT CHAN OF INT z,
CHAN OF INT x)
:
#USE strmhdr
#USE sphdr
#USE splib
#USE spinterf
PROC applic2(CHAN OF INT y, x)
:
#USE strmhdr
#USE sphdr
#USE splib
#USE spinterf
PROC applic3(CHAN OF INT z, y)
:
CHAN OF ANY from.host. to.host, c0, c1, c2 :
PLACED PAR
  PROCESSOR 0 T8
    PLACE c0 AT link0.out :
    PLACE c1 AT link2.in : -- LINK0
    PLACE from.host AT link3.in : -- LINK3
    PLACE to.host AT link3.out :
    applic1(from.host,to.host,c1,c0)

  PROCESSOR 1 T8
    PLACE c0 AT link3.in : --3
    PLACE c2 AT link0.out : --3
```

applic2(c0,c2)

PROCESSOR 2 T8

PLACE c2 AT link3.in : --3

PLACE c1 AT link1.out : --3

applic3(c2,c1)

No exemplo de aplicação gerada não ocorre concorrência entre processos que necessitam compartilhar por canais físicos. Assim, o item 6.7 apresenta como a ferramenta FAPP gerencia esse tipo de atividade.

6.7 Controle de Concorrência para Acesso aos Canais Físicos de Transputers

Um exemplo simples, de controle de concorrências em canais físicos, pode ser criado a partir algumas pequenas alterações no exemplo anterior. Suponhamos que apenas as aplicações applic1 e applic2 devam ser executados paralelamente e que não seja apenas uma soma, mas sim duas somas em paralelo, conforme ilustra a Figura 6.9. Nesse novo programa os valores das variáveis x e y devem ser acrescidos de 10 unidades paralelamente. Suponhamos que para efetuar essa operação optou-se utilizar apenas um canal físico.

Figura 6.9 - Estrutura para uma aplicação.

É claro que por se tratar de um exemplo simples e com apenas duas aplicações, o sincronismo poderia ser tratado na aplicação. A intenção é a de se generalizar o exemplo. Imagine-se diversos processos concorrentes em um processador e que trocam informações com outros processos noutro processador e devem compartilhar o mesmo *link* (canal físico).

Uma possível solução para o problema é a criação de dois processos: um que efetua a multiplexação das informações e outro que efetua a demultiplexação dessas informações.

6.7.1 Subsistema para Multiplexação/Demultiplexação de Canais OCCAM

Os multiplexadores/demultiplexadores, também denominados mux/demux), operam de maneira análoga a uma chave seletora. Nesses tipos de sistemas uma variável de controle é fornecida ao subsistema para seu estado possa ser alterado e permitir que apenas um dos elementos de entrada esteja conectado à saída. A Figura 6.10 apresenta a estrutura lógica de um mux que possui n canais de entrada.

Pode-se assumir para o exemplo da Figura 6.10 que ao ser colocada uma variável de controle com valor 1 (um), o multiplexador conecta o Elemento_1 à saída; caso o controle seja 2 (dois), então a saída passa a estar conectada com o Elemento_2; e assim sucessivamente para todas as entradas.

Um subsistema demultiplexador opera de maneira análoga ao multiplexador. Contudo, o demultiplexador possui entrada única e, mediante uma variável de controle, distribui os dados através dos canais de saída,

conforme ilustra a Figura 6.11.

Em ambos os subsistemas o estabelecimento da conexão entre o meio físico de transmissão e o canal de entrada, ou saída (respectivamente para mux ou demux) é estabelecido através da disposição física dos elementos.

No caso da comunicação entre processos, os componentes mux/demux são processos concorrentes aos elementos que desejam trocar informações. Não há a possibilidade de que sejam identificados os processos através da disposição física dos mesmos. Além disso, o subsistema de controle intrínseco aos multiplexadores e demultiplexadores, é utilizado para estabelecer o sincronismo da comunicação e também dos tipos de dados definidos para cada um dos canais.

6.7.2 A Criação dos Processos Mux/Demux

O compartilhamento de um determinado meio físico por um conjunto de processos deve estar sujeito a alguma disciplina. Essa disciplina é imposta pelo mecanismo de controle de acesso ao meio, que nesse caso trata-se de um mecanismo de multiplexação de dados.

A Figura 6.12 ilustra um conjunto de $n+1$ processos em execução concorrentes em um determinado processador, um dos quais corresponde ao processo que efetua o controle acesso ao *link*. Do outro lado existem também $n+1$ processos, dos quais um deles é o demultiplexador, enquanto os demais aguardam dados.

Embora haja um identificador por processo na linguagem de programação OCCAM, ao criar o esqueleto da aplicação o programador não deixa explícito quais processos devem compartilhar quais canais físicos, definindo apenas todos os canais lógicos que são utilizados para a troca de mensagens, conforme ilustra a Figura 6.13.

A definição dos canais lógicos, com seus respectivos protocolos, pode ser utilizada para a geração dos processos mux/demux tão logo sejam definidas a topologia, Figura 6.14, e o mapa de execução dos processos.

Embora os protocolos de comunicação possam expressar um mesmo tipo de dado, a Figura 6.13 mostra que cada aplicação (Aplicação 0i, Aplicação 1i; $1 \leq i \leq n$) possui seu próprio protocolo.

Pode-se supor que esse esqueleto para a aplicação deva ser executado na topologia ilustrada na Figura 6.14. Assuma ainda que os processos Aplicações 0i (onde, $1 \leq i \leq n$) são executados no processador T0 e os processos Aplicações 1i (onde, $1 \leq i \leq n$) são executados no processador T1.

Diante desse quadro, nota-se que há necessidade da criação de um processo multiplexador que controle os dados entre o *link1* do Transputer T0 e o *link4* do Transputer T1. Para os dados que fluem de T1 para T0 o procedimento é análogo.

O processo MUX deve conter n canais lógicos de entrada e apenas um canal físico de saída. Os n canais lógicos de entrada devem associar as entradas do MUX aos respectivos canais lógicos das aplicações. Além disso, o MUX deve associar valores aos canais lógicos a fim de que o demux possa rotear os dados corretamente. Isso leva a seguinte definição do protocolo para um canal lógico de saída do MUX/DEMUX:

CHAN OF ANY sai.mux, ent.demux :

No módulo principal do programa fonte em OCCAM apareceriam os comandos apresentados na Tabela 6.12. As linhas 12 e 22 mostram que o canal lógico "sai.mux" do processo MUX do processador 0 está associado ao canal lógico ent.demux do processo DEMUX do processador 1. Assim, tudo aquilo que o MUX rotear através do canal "sai.mux", será recebido pelo

processo DEMUX através de seu canal "ent.demux". Falta, portanto, a lógica ou o algoritmo para a implementação dos processos MUX e DEMUX.

Um gabarito para a criação de um multiplexador, segundo a lista de canais que aparecem na linha 18 do programa da Tabela 6.12, é apresentado na Tabela 6.13. É importante que seja ressaltado que, embora o canal lógico "sai.mux" tenha sido declarado do tipo "CHAN OF ANY", para esse canal é definido um protocolo, composto por número inteiro (identificador do processo) e pelo protocolo de comunicação do processo da aplicação.

Tabela 6.12 - Esqueleto para exemplificar o compartilhamento de canais físicos.

1	protocolo.A01 C01:
2	protocolo.A02 C02:
3	...
4	protocolo.A0n C0n:
5	
6	protocolo.A10 C10:
7	protocolo.A12 C12:
8	...
9	protocolo.A1n C1n:
10	PLACED PAR
11	PROCESSOR 0 T8
12	PLACE sai.mux AT 1 :
13	PAR
14	Aplicação01(C01)
15	Aplicação02(C02)
16	...
17	Aplicação0n(C0n)
18	MUX(C01, C02, ..., C0n, sai.mux)
19	DEMUX(< lista de canais de recepção>, ent.demux)
20	
21	PROCESSOR 1 T8
22	PLACE ent.demux AT 4 :
23	PAR
24	Aplicação11(C11)
25	Aplicação12(C12)
26	...
27	Aplicação0n(C1n)
28	MUX(<lista de canais de saída >. sai.mux)
29	DEMUX(C11, C12, ..., C1, ent.demux)

Tabela 6.13 - Gabarito para um processos multiplexador.

```
PROC MUX(C01, C02, ..., C0n, sai.mux, parada)
```

```
  BOOL pare:
```

```
  protocolo.AO1 C001
```

```
  protocolo.AO2 C002
```

```
  ...
```

```
  protocolo.AOn C00n
```

```
  SEQ
```

```
    pare := TRUE
```

```
  WHILE pare
```

```
    SEQ
```

```
      ALT
```

```
        C01 ? C001
```

```
          sai.mux ! 1; C001
```

```
        C02 ? C002
```

```
          sai.mux ! 2; C002
```

```
        ...
```

```
        C0n ? C00n
```

```
          sai.mux ! n; C002
```

```
      parada ? pare
```

```
        SKIP
```

```
  :
```

O processo associado ao MUX deve reconhecer o protocolo de comunicação para a entrada de dados e distribuir essas informações através dos respectivos canais das aplicações. A Tabela 6.14 apresenta o código correspondente ao DEMUX associado ao código fonte da Tabela 6.13.

Tabela 6.14 - Gabarito para um processo demultiplexador.

```
PROC DEMUX(C01, C02, ..., C0n, ent.demux, parada)
```

```
  BOOL pare:
```

```
  protocolo.AO1 C001
```

```
  protocolo.AO2 C002
```

```
  ...
```

```
  protocolo.AOn C00n
```

```
  INT processo:
```

```
  SEQ
```

```
    pare := TRUE
```

```
  WHILE pare
```

```
    ALT
```

```
      ent.demux ? processo
```

```
        SEQ
```

```
          CASE processo
```

Tabela 6.14 - Continuação

1	SEQ
	ent.demux ? C001
	C01 ! C001
2	SEQ
	ent.demux ? C002
	C02 ! C002
...	
n	SEQ
	ent.demux ? C00n
	C0n ! C00n
	parada ? pare
	SKIP
:	

6.8 Estendendo a FAPP para o Ambiente PVM

Muitas aplicações têm sido desenvolvidas utilizando-se máquinas virtuais, visando a flexibilidade do software. Entretanto, esses ambientes não apresentam ferramentas que auxiliem o projeto e manutenção de seus programas. Esta seção apresenta uma análise de como a ferramenta FAPP pode ser instanciada para dar apoio à programação paralela nesses ambientes, tomando-se como exemplo o PVM.

6.8.1 Arquitetura do hardware

O desenvolvimento de aplicações em ambiente PVM exige que o programador defina os elementos de processamento que compõem a sua arquitetura, o conjunto de processos que são executados e o paradigma de programação (*single program multiple data* ou *multiple program multiple data*). A definição da arquitetura do hardware, no PVM, pode ser especificada através de um arquivo de configuração, constando de um identificador lógico para o processador.

Essa característica pode ser facilmente definida no ambiente FAPP através da especialização da classe FAPPElemProcessador. Cada um dos tipos de máquinas utilizadas para compor a máquina virtual deve ser modelado por uma classe, conforme ilustra a Figura 6.15. Cada um dos

objetos elemento-processador instanciados dessas classes tem como atributo o seu identificador lógico. Assim, gerar a estrutura do hardware, para uma aplicação PVM, consiste em fornecer aos usuários o elenco de elementos processadores que podem ser selecionados, deixando que a ferramenta gerencie a configuração do sistema.

A arquitetura paralela é, portanto, uma especialização da classe

APMemoriaDistribuida, ilustrada na figura 4.6, cuja característica especial é o gerenciamento da configuração.

6.9 Arquitetura do software

Os processos conforme as definições de CSP/OCCAM podem ser comandos. Isso difere do conceito de processo aplicado na teoria de sistemas operacionais (Tanenbaum (1992)), que corresponde a um programa em execução juntamente com suas variáveis de estado. Assim, nessa seção utiliza-se o conceito de processo segundo a teoria de sistemas operacionais.

Essa diferenciação é necessária para a elaboração do modelo para os processos. Um processo é a especialização de um dígrafo-rotulado-completo, pois todos os subprogramas que compõem o processos têm seu código alcançável, definem-se os canais lógicos de comunicação entre eles e cada processo é identificado univocamente.

Os subprogramas, para a linguagem C, não podem ter seus códigos locais a algum subprograma, por definição da linguagem. Assim, modela-se o conjunto de subprogramas como uma especialização da classe GRFSuperNoRotuladoPeso, que é definida como um conjunto de nós que

possui rótulo e peso. Os comandos básicos da linguagem são nós rotulados e com peso. Essa abordagem permite que a ferramenta:

- possa ser instanciada para auxiliar a programação até o nível de geração do texto completo da aplicação;
- armazene informações que auxiliam a distribuição dos processos, visando o balanceamento de cargas;
- ofereça ao usuário níveis distintos de abstração durante a fase de projeto do sistema.

O ambiente PVM permite que os programas sejam desenvolvidos utilizando-se os paradigmas SPMD ou MPMD. Assim é necessário que a ferramenta ofereça suporte a esses mecanismos.

O modelo implementado na ferramenta FAPP é MPMD, utilizando a técnica de programação proposta por Foster (1994). Para o auxílio na geração do código na linguagem C, é necessário que os comandos de definição de subprogramas sejam modelados, analogamente aos comandos OCCAM. Nesse caso, o auxílio da ferramenta, para a geração do código no atual estado de desenvolvimento não é vultoso, pois o esqueleto da aplicação resume-se a um conjunto de protótipos, gerados em função das aglomerações definidas por Foster (1994). Entretanto, considerando-se os aspectos mais genéricos do desenvolvimento de software, a ferramenta pode ser muito útil para a manutenção do sistema.

Assim, considerando-se a discussão acima, o paradigma SPMD é uma especialização da proposta MPMD, pois há restrições nas partições da aplicação, sendo gerado um arquivo único, contendo todos os subprogramas e o programa principal.

6.10 Considerações Finais

A utilização de referências para os elementos básicos, permite que a alteração, ou manutenção dos programas e processos seja efetuada de

maneira consistente e, em alguns casos, quase transparente para o programador.

As especializações a partir da classe GRFSuperNo, para subprogramas, permite que a ferramenta seja estendida ao nível de comandos. Isso faz com que a programação sejam elaborada utilizando-se dígrafos. Entretanto, essa possibilidade não está implementada.

O exemplo apresentado na listagem acima ilustra uma maneira de se desenvolver programas paralelos utilizando-se a topologia em uma malha de Transputers. Contudo, no exemplo apresentado não ocorre concorrência em relação os canais físicos. Quando há concorrência por canais físicos, há a necessidade do estabelecimento do sincronismo na execução do programa. O item 6.7 apresenta maiores detalhes com relação a esse problema e apresenta solução utilizando processos para multiplexação/demultiplexação de canais físicos.

A interface para aplicações, utilizando o contexto apresentado neste capítulo, são apresentas em seqüências nas amostras da tela de execução da ferramenta.

FAPPFiles Edit Componentes EPs Arquitetura Mapa Esqueleto Help

Processador

Memoria

Interface

SoftwareOper

Mechter

OCCAM2

PVMlib

MPIlib

FABRICANTEFAMILIA

CLOCKBENCHMA

Intel8048633

Cyrix8048633

IBM8048633

TAMANHOUNIDADEPALAVRAORGANIZACA

TEMPOACCESSIDENTIFICA

1 M8 8x1701

2 M8 8x1702

4 M8 8x1703

Iniciar

C++Builder

Fappprj

Microsoft Word - Doc...

22:27

FAPPFiles Edit Componentes EPs Arquitetura Mapa Esqueleto Help

SPP

SDFA

C_DIGO

T800_ORIGE

LINK_SA_DA

T800_DESTI

LINK_ENTRA

NOME_CANAL

1403 from.host

21340 to.host

31023 c0

42030 c2

531c1

Iniciar

C++Builder

Fappprj

Microsoft Word - cap...

22:32

7. Conclusões

Neste capítulo são tecidas algumas considerações relativas ao desenvolvimento deste trabalho, ressaltadas as contribuições relevantes da pesquisa desenvolvida, comentados os pontos básicos que estabelecem a continuidade da pesquisa e, finalizado o capítulo, é elaborada uma análise global dos resultados obtidos.

7.1 Considerações Iniciais

No contexto do desenvolvimento de programas para arquiteturas paralelas, há um conjunto de modelos arquiteturais que definem regras para o desenvolvimento de software. Diversos ambientes têm sido propostos com a finalidade de auxiliar o programador no desenvolvimento das aplicações, não havendo consenso em relação a um modelo arquitetural deva ser utilizado.

Esta tese contribui para a solução do problema discutido acima, apresentando uma abordagem para modelagem de programas e arquiteturas paralelas, que pode ser utilizada para a implementação de uma ferramenta de apoio ao desenvolvimento de programas paralelos.

A abordagem, baseada em orientação a objetos, foi proposta e validada no trabalho, como discutido nos itens que seguem.

7.1.1 Descrição do Problema

O capítulo 1 discutiu a evolução do hardware e do software, mostrando a necessidade do desenvolvimento de novas ferramentas para a elaboração de programas, particularmente, no caso de programas paralelos.

Foi estabelecido que modelos arquiteturais para software, têm sido propostos em função da plataforma de hardware no qual devem ser

executados. O custo de desenvolvimento e manutenção das aplicações faz com que os programadores utilizem mecanismos mais flexíveis de programação (como, por exemplo, os ambientes de passagem de mensagem), arcando com a perda de desempenho de suas aplicações.

Considerando-se as diversas ferramentas de programação discutidas na literatura, pode ser estabelecido um modelo geral para a ser a base da investigação desta tese.

7.1.2 Orientação a Objetos

A revisão dos métodos orientados a objetos e dos conceitos associados a essa teoria (capítulo 2), mostram que essa tecnologia de desenvolvimento de software tem evoluído bastante, com reflexos nos conceitos e notações adotadas.

Dos métodos analisados, decidiu-se adotar a notação do método FUSION, por permitir a representação das características e conceitos capturados na etapa anterior (características principais das ferramentas) e, por apresentar notações já familiares, diminuiu o tempo de aprendizado.

O paradigma de orientação a objetos foi escolhido como base para a abordagem, em função da facilidade com que é possível projetar um sistema com a finalidade de estenderem-se as aplicações, ou reutilizarem-se os códigos.

7.1.3 Modelo Matemático

Os computadores e as linguagens de programação podem ser modelados por autômatos finitos ou, mais genericamente, por grafos. Assim, no capítulo 3, a notação matemática, os conceitos e as respectivas classes de objetos para a teoria dos grafos foram definidos.

Como resultado, aliando-se a orientação a objetos, obteve-se um conjunto de classes-base que é utilizados para modelar tanto os componentes do hardware quanto do software, além de ser útil para outras aplicações.

7.1.4 Modelos Arquiteturais

O capítulo 4 trata em detalhes os modelos arquiteturais, com a finalidade de eleger uma taxonomia como base para os respectivos modelos orientados a objetos.

A taxonomia de Duncan é a mais indicada, segundo os critérios definidos no capítulo. O modelo orientado a objetos da taxonomia de Duncan, é criado como uma especialização de um dígrafo conexo, utilizando-se para isso a notação do método FUSION.

Os componentes do software também são modelados com a mesma teoria. Entretanto, é necessário que seja definido o nível de abstração em que a ferramenta de apoio atua. Pode-se, com a estrutura proposta, modelar as linguagens até o nível dos comandos, de modo análogo às respectivas cartas sintáticas das linguagens. Porém, adotou-se o nível de subprograma como plataforma básica para o projeto de programas paralelos.

7.1.5 Interface

No capítulo 5 são apresentadas as características desejáveis em uma interface, um modelo de interfaces baseado em menus e a proposta de uma hierarquia de classes para criar objetos de desenho em uma interface gráfica para uso com a ferramenta, desenvolvida com base nos modelos propostos nesta tese.

A interface, baseada em menus, é implementada para oferecer ao usuário visões de um ambiente, no qual cada janela representa uma visão diferente. Essa interface é suficiente para mostrar que o modelo para a ferramenta é viável.

7.1.6 Aplicação

O capítulo 6 apresenta uma especialização da ferramenta para uma arquitetura específica, com a finalidade de validar a utilização do modelo proposto e apresentar os passos que devem ser seguidos para estender a utilização da ferramenta FAPP para outros modelos arquiteturais.

7.2 Contribuições Relevantes

Dentre as diversas contribuições obtidas com a modelagem, implementação e validação deste trabalho, destacam-se:

- um ambiente de desenvolvimento de programas paralelos com flexibilidade para ser estendido para diversas arquiteturas;
- uma ferramenta que utiliza a descrição do hardware para auxiliar a geração do esqueleto da aplicação.
- geração de um conjunto de classes-base para a teoria dos grafos, que pode ser aplicado a outros problemas que têm como modelo-base a teoria dos grafos;
- projeto dos elementos baseados em grafos/dígrafos ponderados, o que oferece uma plataforma para definir estratégias para alocação estática de subprogramas e simulação de código.
- sistematização do desenvolvimento de programas paralelos aplicando a técnica proposta por Foster (Foster (1994)).
- uso da tecnologia de orientação a objetos como base para o modelo da teoria dos grafos, arquiteturas paralelas e linguagens de programação.

7.3 Trabalhos Futuros

O trabalho apresentado nesta tese é apenas o embrião para uma linha de pesquisa nessa área. Diversos trabalhos podem ser estabelecidos a partir deste, destacando-se:

- a análise das linguagens de programação paralelas para definir-se uma taxonomia, permitindo que outras linguagens possam ser agregadas à ferramenta com maior facilidade.
- o estabelecimento do escopo das arquiteturas para o qual a ferramenta será efetivamente utilizada, para que possa ser especificada uma interface gráfica, mais robusta e amigável.

- inserção de algoritmos que auxiliem os programadores na tarefa de particionar e aglomerar os processos em função da distribuição dos dados, implementando novos paradigmas de programação.
- inserção de algoritmos que auxiliem o balanceamento de carga.
- extensão do ambiente para simulação de programas paralelos.
- o desenvolvimento de um estudo da eficácia da ferramenta no auxílio da programação paralela. Essa necessidade decorre do fato dos usuários terem que assimilar os conceitos e técnicas implementados, antes de iniciarem o desenvolvimento das aplicações.

7.4 Considerações Finais

O trabalho desta tese contribui significativamente para a solução do problema de apoio ao desenvolvimento de programas paralelos, pois além de estabelecer um meta-modelo para o hardware e software, oferece a implementação de uma técnica de programação que auxilia o programador durante as fases de desenvolvimento do software.

A sistematização para o desenvolvimento da aplicação permite que a manutenção do programa seja efetuada com menor grau de dificuldade, pois nem sempre o programador deve interagir diretamente com o texto do código gerado, por exemplo quando ocorre alteração da configuração do hardware.

8. REFERÊNCIAS BIBLIOGRÁFICAS

- ANDREWS, G.R.; SCHINEIDER, F.B. *Concepts and Notations for Concurrent Programming*. ACM Computing Survey, v.15, n.1, p.3-43, 1983.
- ANDRADE, A.L.P.; SANTANA, M.S. *Servidor de Processamento Paralelo: Análise e Extensão dos Protocolos*. São Carlos, 1994. Dissertação (Mestrado) - Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo.
- ALMASI, G.S; GOTTLIEB A. **Highly Parallel Computing**, 2.ed, Califórnia, The Benjamin Cummings Publishing Company, Inc, 1994.
- BAKER, S.A; MALVEM, R.S.R.E. *A Process Migration Harness for Transputers*. SNARC91-01, Controller HMSO, London, 1991.
- BEGUELIN, A, GEIST; A., DONGARRA; J.; JIANG, W.; MANCHEK, R.; SUNDERAM V. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for NetWork Parallel Computing*, The MIT Press, s.l., 1994.
- BEM-DYKE, A.D. *Architectural Taxonomy - A Brief Review*. Computation Parallel Group, adb@cs.bham.ac.uk, Universidade de Birmingham, junho, 1993.
- BOAVENTURA NETTO, P.O. **Teoria e Modelos de Grafos**, Blücher, s.l., 1979.
- CALÔNEGO JR., N. *UTOPIA: uma Facilidade para Balanceamento de Cargas em Sistemas Distribuídos*. São Carlos, 1994. Monografia (Exame Geral de Qualificação) - Instituto de Física e Química de São Carlos, Universidade de São Paulo.

- CALÔNEGO JR., N.; Santana, M.J., Santana, R.H.C *Parallel Programming Tool-kit*. Anais do I Workshop de Sistemas Distribuídos (WoSiD'96), Salvador, 1996.
- CALKIN, R; HEMPEL, R.; HOPPE, H. C.; WYPIOR, P. *Portable programming with the PARMACS message-passing library*. *Parallel Computing*, v.20, p.615-632, 1994.
- CARMICHAEL, A. **Advances in Object Technology - Object Development Methos**, SIGS Books, Inc. 1994.
- CARRIERO, N.J.; GELERNTER, D.; MATTSON, T.G.; SHERMAN, A.H. The Linda alternative to message-passing systems. *Parallel Computing*, v.20, p.633-655, 1994.
- CHEN, P.P. *The entity-relationship model: toward a unified view of data*. *ACM Transaction on Database Systems*, v.1, n.1, p.9-36, 1976.
- CIANCARINI, P. *Coordenation Models and Languages as Software Integrators*. v.28, n.2, 1996
- CODD, E.F. *A relational model of data for large shared data banks*. *Communications of the ACM*, v.13, n.3, p.377-387, 1970.
- CODASYL Data Base Task Group. April 71 Report, ACM, New York, 1971.
- COLEMAN, D; ARNOLD, P.; BODOFF, S.; DOLLIN, C.; GILCHRIST, H.; HAYES, F.; JEREMAES, P. **Desenvolvimento Orientado a Objetos: O Método Fusion**, Rio de Janeiro, Editora Campus, 1994.
- COMER, D.E; STEVENS, D.L. **Internetworking with TCP/IP Design, Implementation and Internal**, Prentice Hall, v.2, New Jersey, 1991.
- COULOURIS, G; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems Concepts and Design*, 2^a.ed. Addison-Wesley Publishing Company, s.l.,1994.
- DEMARCO, T. **Structured Analysis and System Specification**, Prentice Hall, New Jersey, 1979.

- DUNCAN, R. *A Survey of Parallel Computer Architectures*. IEEE Computer, p.5-16, fev., 1990.
- FLYNN, M.J. *Some computer organizations and their effectiveness*. IEEE Transactions on Computers, v.C, n.21, p.948-960, 1972.
- FOSTER, I. **Designing and Building Parallel Programs**, Addison-Wesley Publishing Company, s.l. 1995.
- GANE, T; SARSON, C. **Strutured System Analysis**, McDonnnell Douglas, 1982.
- GEIST, A; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM V. **PVM 3 User's Guide and Reference Manual**, Oak National Laboratory, set. 1994.
- GILOI, W.K. *A complete taxonomy of computer architecture based on the abstract*. Proceedings of the IFIP: Wokshop on Taxonomy in Computer Architecture, p.19-38, jun. 1981.
- GRAVER, J.O. *The evolution of an object-oriented compiler framework*. Software-Practice and Experience, v.17, p.519-535, jul. 1992.
- GRIMSHAW, S.A. *Easy to use object-oriented parallel processing with Mentat*. IEEE Computer, p.39-51, 1993.
- GROPP, W; SKJELLUM, A. **Using MPI: Portable Parallel Programing with the Message Passing Interface**, MIT Press, 1995.
- HOARE, C.A.R. *Communicating sequential processes*. Comunications of The ACM, v.21, n_8, p.666-677, 1978.
- HOARE, C.A.R. **Communicating Sequential Processes**, Prentice Hall, 1985
- INMOS Limited, **Transputer Development System**, 2^{ed}. Prentice Hall, U.K. 1990.
- JACKSON, M.A. **Principles of the Program Design**, Academic Press, s.l. 1975.

- JACOBSON, I. **Object-Oriented Software Engineering**, ADDISON-WESLEY, 1992.
- JOHNSON, E.E. *Completing a MIMD multiprocessor taxonomy*. *Computer Architecture News*, v.16, n.3, p.44-47, 1988.
- JOHNSON, S.P.; et al; *Automatic parallel code generation for message passing on distributed memory systems*. v. 22, p.227-258, 1996.
- JONES, G. **Programming in OCCAM**, Prentice-Hall, 1987.
- JONES, G.; GOLDSMITH, M. **Programming in OCCAM2**, Prentice-Hall, 1988.
- MADSEN, O.L; PEDERSEN, M.B; NYGAARD, K. **Object-Oriented Programming in the BETA Programming Language**, Addison-Wesley, 1993.
- MALKEVITCH, J; MEYER, W. **Graphs, Models & Finite Mathematics**, 1^a. ed, Prentice-Hall, 1974.
- MANACERO JR., A.; *Predição do Desempenho de Programas Paralelos por Simulação do Grafo de Execução*. Campinas, 1997. Tese (Doutorado) - Faculdade de Engenharia Elétrica e Computação, Universidade de Campinas.
- MARTIN, J; ODELL, J.J. **Análise e Projeto Orientados a Objetos**, Makron Books do Brasil, 1^a. ed, 1996.
- McBRYAN, O.A. An overview of message passing Environments. *Parallel Computing*, v.20, p.417-444, 1994.
- McMENAMIM, S.M; PALMER, J.F. **Análise Essencial de Sistemas**, Makron Books do Brasil, 1991.
- MISRA, J; CHANDY, K.M. **Parallel Program Design**, Addison-Wesley, vol I, 1989.
- MÜLLENDER, S.J. *Distributed Systems*, 2^a. ed, ACM PRESS Frontier Series, Addison-Wesley Publishing Company, 1993.

- NASCIMENTO, H.H.A; SANTANA, M.J.; CALÔNEGO JR., N.; SANTANA, R.H.C. **Detalhes de Desenvolvimento do Sistema Gerador de Stubs - GAPS**, Relatórios Técnicos do ICMSC, no. 37, São Carlos, 1995.
- PAPAS, C.H; MURRAY **Borland C++ 4.0**. Makron Books do Brasil Ltda, São Paulo, 1995.
- PENSON, L.J; Richard, S.W. **C++ Programação Orientada para Objeto - Manual Prático e Profissional**, do Brasil Ltda, São Paulo, 1992.
- PRESSMAN, R.S. **Engenharia de Software**, 2ª. ed. Kakron Books do Brasil Ltda, 1995.
- RUMBAUGH, J.M; BLAHA, W; EDDY, F; LORENSEN, W. **Object-Oriented Modeling and Design**, Prentice-Hall, 1991.
- SAHNI, S.R.; THANVANTRI, V. *Performance Metrics: keeping focus on run time*. IEEE Parallel and Distributed Technology. v.4, n.1, p.43-56, 1996.
- SANTANA, R.H.C; MORSELI JR., J.C.; SANTANA, M.J.; CALONEGO JR., N.C. *Distributed Simulation Using the CMB Method in a Transputer Network*, The Proceedings of the 1996 Summer Conference, p.144-149, Portland, 1996.
- SHLAER, S; MELLOR, S.J. **Análise de Sistemas Orientada para Objetos**, McGraw-Hill, São Paulo, 1990.
- SMITH, J.M; SMITH, D.C.P. *Data Base Abstractions: Aggregation and Generalization*. ACM Transactions On Data-base Systems, v.2, n.2 p.105-133, 1977.
- SOARES, L.F.G; LEMOS, G; COLCHER, S. **Redes de Computadores - Das LAN's, MAN's e WAN's às redes ATM**, Editora Campus, 1995.
- SOUZA, P.S.L; Santana, M.J. *Máquina Paralela Virtual em Ambiente Windows*. São Carlos, 1996. Dissertação (Mestrado) - Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo.

- STEEN, A.J.V.D. **An overview of (almost) available parallel systems.**
Stichting Nationale Computer Faciliteiten, 3^a ed. pp.1-46, 1993.
- SUNDERAM, V.S; GEIST, A.; DONGARRA, J.; MANCHEK, R. The PVM concurrent computing system: evolution, experiences and trends.
Parallel Computing, v.20, p.531-545, 1994
- TANENBAUM, A.S. **Modern Operating Systems**, 2^a. ed. Prentice Hall International Inc. 1992.
- TENENBAUM, A.M; LANGSAN, Y.; AUGENSTEIN, M.J. **Estruturas de Dados Usando C**, 1ed. Makron Books do Brasil LTDA, 1995.
- TRAINA, C. *Máquina e Modelo de Dados Dedicados para Aplicações em Engenharia*. São Carlos, 1986. Tese (Doutorado) - Instituto de Física e Química de São Carlos, Universidade de São Paulo.
- TRACZ, W. Domain-Specific Software Architecture (DSSA) Frequently Asked Questions. *ACM SIGSOFT*, Software Engineering Notes, v.19, no 2, p.52-56, 1994.
- TRELEVEN, P. Control-Driven, Data-Driven, and Demand-Driven Computer Achitecture (Abstract). *Parallel Computing*, 2, 1985.
- TRINDADE Jr, O.; Santana, M.J. *Um Servidor de Processamento Paralelo Baseado em Transputers*. São Carlos, dezembro de 1991. Tese (Doutorado) - Instituto de Física e Química da Universidade de São Paulo.
- TRINDADE JR., O; SANTANA, M.J; GERMANO, F.S.R. Um Servidor de Processamento Paralelo Baseado em Transputers - Aspectos de Implementação, IV Simpósio Brasileiro de Arquiteturas de Computadores - Processamento de Alto Desempenho, São Paulo, 1992.
- YOURDON, E.N. **Modern Structured Analysis**, Prentice-Hall, 1990.
- YOURDON, E.N; COAD, P. **Análise Baseada em Objetos**, Editora Campus, 1992.

- WALKER, D.W. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, v.20, p.657-673, 1994.
- WARNIER, J.D. Logical Construction of Programs. Van Nostrand Reinhold, 1974.
- WILBUR, S. **Building Distributed Systems with Remote Procedure Call.** Software Engineering Journal, Setembro 1987.
- WINER, S.R; PINSON, L.J. **C++ Programação Orientada para Objeto: Manual Prático e Profissional.** Makron Books, São Paulo, 1991.
- WIRTH, N. **Algorithms and Data Structure**, Prentice-Hall, 1986.
- ZALUSKA E.J. Research lines in distributed computing systems and concurrent computation. *Anais do Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software*, p.132-155, 1991.
- Zima, P.H.; CHAPMAN, B.M. Software Tools for Parallel Program Development. Programming Environment for High Level Scientific Problem Solving, Elsevier Science Publishers, IFIP, 1992.
- ZHOU, S; ZHENG, X.; WANG, J.; DELISLE, P. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer. *Software-Practice and Experience*, vol23(12), p.1305-1336, 1993.

1. Introdução

Os programas apresentados neste apêndice foram gerados utilizando-se o ambiente de desenvolvimento integrado fornecido pela Borland “Integrated Development Environment - IDE”, na versão 4.52.

As classes definidas neste trabalho foram implementadas inicialmente utilizando-se uma interface textual, devido à complexidade de implementação de interfaces para o ambiente Windows 3.1 e Windows95. As classes prefixo FAPP (Ferramenta de Auxílio à Programação Paralela) foram implementadas e testadas, sendo apresentadas na seguinte sequência:

- arquivos de cabeçalhos, contendo a definição das classes, necessários para a geração do código;
- implementação dos métodos da classe;
- programa-exemplo para a utilização da classe;
- arquivo-exemplo apresentando as informações armazenadas por objetos instanciados a partir dessas classes.

As informações correspondentes às variáveis de estado dos objetos aparecem no formato <dados>, para facilitar a compreensão do mesmo. A função *GetToken*, extrai os dados que são armazenados nas variáveis de estado dos objetos. Essa primitiva foi eliminada do código em versões posteriores, que passaram a utilizar um sistema de gerenciamento de banco de dados.

A macro `TxxxxyyASzzz<classe>`, disponível para os compiladores C++ utilizados, faz com que a classe passada como argumento seja manipulada segundo um tipo abstrato de dado, apresentando uma solução rápida e prática para a implementação de estrutura de dados análogas à estrutura apresentada na figura 3.16.

2. Classe Memória

Classe-base que representa as possíveis configurações de memória utilizadas para implementar um Elemento de Processamento.

2.1 Cabeçalho para a classe base *FAPPMemoria*

```
#ifndef FAPPMemoria_H      // prevent header from
#define FAPPMemoria_H 1    // being included twice

#include <classlib/arrays.h>
#include <cstring.h>
#include <iostream.h>
#include <fstream.h>

class FAPPMemoria{
public:
    FAPPMemoria();
    FAPPMemoria(const string& s);
    FAPPMemoria(const FAPPMemoria& mc);
    ~FAPPMemoria();
    FAPPMemoria& operator=(const FAPPMemoria& mc);
    int operator==(const FAPPMemoria& mc) const;
    int operator!=(const FAPPMemoria& mc) const;
    int operator<(const FAPPMemoria& mc) const;
    friend ostream& operator <<(ostream& os, const FAPPMemoria& mc);
    int GetObjeto(ifstream&);
    void MemInic(string, int, char, int, int);
    string IdGet() const;
    void IdSet(string);
    void TamanhoSet(int);
    int TamanhoGet();
    void UnidadeSet(char);
    char UnidadeGet();
    void TempoAcessoSet(int);
    int TempoAcessoGet();
    void PalavraSet(int);
    int PalavraGet();

private:
    string objetoId; // Identificador do objeto
    int tamanho; // Tamanho da memoria
    char unidade; // Unidade da memoria: K/M/G/T bytes
    int tempoAcesso; // Em nano-segundos
```

```

    int palavra; // Largura da palavra 8/16/32/64 bits
    // string organizacao; // 4x4 / 8X1 / ...

};

typedef TArrayAsVector<FAPPMemoria> TArrayMemoria;
typedef TArrayAsVectorIterator<FAPPMemoria> TArrayIterMemoria;
const int FAPP_MAX_MEM=20;
typedef TArrayAsVector<FAPPMemoria> ContainerType;
typedef TArrayAsVectorIterator<FAPPMemoria> IteratorType;
#endif

```

2.2 Implementação dos Métodos da Classe FAPPMemoria

```

// -----
// Copyright (C) 1994 Borland International
// FAPPMemoria.cpp
// -----

#include "memoria.h"
#include "gettoken.h"

FAPPMemoria::FAPPMemoria(){}

FAPPMemoria::FAPPMemoria(const FAPPMemoria& mc){
    objetoId = mc.objetoId;
    tamanho = mc.tamanho;
    unidade = mc.unidade;
    tempoAcesso = mc.tempoAcesso;
    palavra = mc.palavra;
}

FAPPMemoria::~FAPPMemoria(){}

FAPPMemoria& FAPPMemoria::operator=(const FAPPMemoria& mc){
    objetoId = mc.objetoId;
    tamanho = mc.tamanho;
    unidade = mc.unidade;
    tempoAcesso = mc.tempoAcesso;
    palavra = mc.palavra;

    return *this;
}

int FAPPMemoria::operator==(const FAPPMemoria& mc) const{

```

```

int val=1; // retorna 1 caso sejam iguais
if ((tempoAcesso != mc.tempoAcesso) | (tamanho != mc.tamanho) |
    (palavra!=mc.palavra) | (unidade != mc.unidade)
    ) val = 0;

return val;
}

int FAPPMemoria::operator!=(const FAPPMemoria& mc) const{
    int val=1; // retorna 1 caso sejam iguais
    if ((tempoAcesso == mc.tempoAcesso) & (tamanho != mc.tamanho) &
        (palavra == mc.palavra) & (unidade == mc.unidade)
        ) val = 0;

    return val;
}

int FAPPMemoria::operator<(const FAPPMemoria& mc) const{
    return (tamanho == mc.tamanho);
}

ostream& operator<<(ostream& os, const FAPPMemoria& mc){
    os << "Identificador <" << mc.objetoId << "> " << endl;
    os << " Tamanho <" << mc.tamanho << "> " << endl;
    os << " Unidade <" << mc.unidade << "> Bytes" << endl;
    os << " Tempo de acesso  <" << mc.tempoAcesso << "> ns" << endl;
    os << " Palavra  <" << mc.palavra << "> bits" << endl << endl;
    return os;
}

int FAPPMemoria::GetObjeto(istream& file){
    string id, tam, uni, ta, pal;
    if (GetToken(file, id)) return 1;
    if (GetToken(file, tam)) return 1;
    if (GetToken(file, uni)) return 1;
    if (GetToken(file, ta)) return 1;
    if (GetToken(file, pal)) return 1;

    // Existem dados: Nao ha verificacao de consistencia
    // Caso o arquivo de dados contenha dados em ordem errada
    // ocorrerá erro ao iniciar o objeto.
    objetoId = id;

```



```

tamanho = atoi(tam.c_str());
unidade = (uni.c_str())[0];
tempoAcesso = atoi(ta.c_str());
palavra = atoi(pal.c_str());
return 0;
}

void FAPPMemoria::MemInic(string id, int tam, char uni, int ta, int pal)
{
    objetoId = id;
    tamanho = tam;
    unidade = uni;
    tempoAcesso = ta;
    palavra = pal;
}

string FAPPMemoria::IdGet() const { return objetoId; }
void FAPPMemoria::IdSet(string obj){ objetoId = obj; }
void FAPPMemoria::TamanhoSet(int t){ tamanho = t; }
int FAPPMemoria::TamanhoGet(){ return tamanho ;}
void FAPPMemoria::UnidadeSet(char c){ unidade = c; }
char FAPPMemoria::UnidadeGet(){ return unidade; }
void FAPPMemoria::TempoAcessoSet(int t){ tempoAcesso = t; }
int FAPPMemoria::TempoAcessoGet(){ return tempoAcesso; }
void FAPPMemoria::PalavraSet(int p){ palavra = p; }
int FAPPMemoria::PalavraGet(){ return palavra; }

```

2.3 Programa-exemplo para a Classe FAPPMemoria

```

#include <classlib/arrays.h>
#include <iostream.h>
#include <strstrea.h>
#include <fstream.h>
#include "..\include\memoria.h"
class FAPPGPersisMem{
public:
    FAPPGPersisMem(const char far*);
    ~FAPPGPersisMem();

```

```

void inserir(FAPPMemoria);
void remover(FAPPMemoria&);
FAPPMemoria& obter(FAPPMemoria);
void gravar();

protected:
    TArrayMemoria arrayMem(FAPP_MAX_MEM);
    fstream file;
    int mode;
};

#include "gerpmem.h"

FAPPGPersisMem::FAPPGPersisMem(const char far* arquivo){
    mode = ios::in | ios::app;
    file.open( arquivo, mode );
}

FAPPGPersisMem::~FAPPGPersisMem(){
}

void FAPPGPersisMem::inserir(FAPPMemoria& p){
    arrayMem.Add(p);
}

void FAPPGPersisMem::remover(FAPPMemoria& p){
    arrayMem.Destroy(p);
}

FAPPMemoria& FAPPGPersisMem::obter(FAPPMemoria m){
    TArrayIterMemoria i(arrayMem.GetItemsInContaner());
    int achou = 0;
    while (!achou && i!=0){
        if (m == *i.Current())
            return i;
        i++;
    }
    return ((FAPPMemoria&) NULL)
}

void FAPPGPersisMem::gravar(){
    TArrayIterMemoria i(arrayMem.GetItemsInContaner());

```

```

        while (i != 0){
            file << *i.Current();
            i++;
        }
    }

// -----
// Copyright (C) 1995 - Nivaldi Calonego Junior
// Geral.cpp
// Utiliza process.h e a definição da classe em process.cpp.
// -----

extern const int FAPP_MAX_MEM;

int CriarArrayMem(TIArrayMemoria& arrayMem, int& numItems){
    int i=0;
    int stop=0;
    ifstream MEMORIA_FILE;
    MEMORIA_FILE.open("memoria.txt");
    if (!MEMORIA_FILE){
        cerr << "Nao pode abrir o arquivo " << endl;
        return 1;
    }
    FAPPMemoria p;
    while ( !stop ){
        stop = p.GetObjeto(MEMORIA_FILE);
        if (!stop){
            i++;
            arrayMem.Add(new FAPPMemoria(p));
        }
    }
    numItems = i;
    MEMORIA_FILE.close();
    return 0;
}

void ContainerSave(TIArrayMemoria cont, ofstream& file){
    TIArrayIterMemoria i(cont);
    while (i != 0){

```

```

        file << *i.Current() << " ";
        i++;
    }
}

int main(){
    int numItems;
    TIArrayMemoria arrayMem(FAPP_MAX_MEM);
    if (CriarArrayMem(arrayMem, numItems))
        return 0; // Nao ha dados no arq.
    ofstream file("Saida");
    ContainerSave(arrayMem, file);
    file.close();
    return 0;
}

```

2.4 Arquivo-exemplo Gerado

```

Identificador <M/1/m/8-70>
    TAMANHO <1> <M>   TEMPO DE ACESSO <70>   LARGURA PAL. <8>
Identificador <M/2/m/16-70>
    TAMANHO <2> <M>   TEMPO DE ACESSO <70>   LARGURA PAL. <16>
Identificador <M/4/m/32-70>
    TAMANHO <4> <M>   TEMPO DE ACESSO <70>   LARGURA PAL. <32>
Identificador <M/8/m/32-70>
    TAMANHO <8> <M>   TEMPO DE ACESSO <70>   LARGURA PAL. <32>
Identificador <M/16/m/32-70>
    TAMANHO <16> <M>   TEMPO DE ACESSO <70>   LARGURA PAL. <32>

```

3. Classe Processador

Unidade central de processamento composta apenas por circuitos capazes de efetuar operações lógicas e/ou aritméticas.

3.1 Cabeçalho para a Classe FAPPProcessador

```

#ifndef FAPPProcessador_H    // prevent header from
#define FAPPProcessador_H 1  // being included twice
#include <classlib/arrays.h>

```

```

#include <cstring.h>
#include <iostream.h>
#include <fstream.h>
#include "gettoken.h"
class FAPPProcessador{
public:
    FAPPProcessador();
    FAPPProcessador(const string& s);
    FAPPProcessador(int cl, const string& s);
    FAPPProcessador(const FAPPProcessador& mc);
    ~FAPPProcessador();
    FAPPProcessador& operator=(const FAPPProcessador& mc);
    int operator==(const FAPPProcessador& mc) const;
    int operator!=(const FAPPProcessador& mc) const;
    int operator<(const FAPPProcessador& mc) const;
    friend ostream& operator <<(ostream&, const FAPPProcessador&);
    string IdGet()const;
    int ClockGet() const;    // Retorna o clock do processador
    string FabricanteGet() const; // Retorna o fabricante do processador
    string FamiliaGet() const; // Retorna a familia do procesador
    virtual int BenchMarkGet(void); // Retorna o benchmark do processador
    int GetObjeto(ifstream &); // Le um objeto processador do stream
    void IdSet(string);
    void FabricanteSet(string);
    void FamiliaSet(string);
    void ClockSet(int);
    virtual void BenchMarkSet(int);
protected:
    string objetoId;    // Identificador para o objeto
    string fabricante; // Intel - INMOS - ZILOG
    string familia;    // 80xxx - T800 - 68xxx
    int clock;        // Clock do processador em MHz
    int benchMark;    // Benchmark do processador
};
const int FAPP_MAX_PROC=20;

```

```
typedef TArrayAsVector<FAPPProcessador> ContainerType;
typedef TArrayAsVectorIterator<FAPPProcessador> IteratorType;
#endif
```

3.2 Implementação dos Métodos para a Classe FAPPProcessador

```
// -----
// Copyright (C) 1994 Borland International
// FAPPProcessador.cpp
// -----

#include "process.h"

FAPPProcessador::FAPPProcessador(){ benchMark = 0;}

FAPPProcessador::FAPPProcessador(const FAPPProcessador& mc){
    objetoId = mc.objetoId;
    fabricante = mc.fabricante;
    familia = mc.familia;
    clock = mc.clock;
    benchMark = mc.benchMark;
}

FAPPProcessador::~FAPPProcessador(){ }

FAPPProcessador& FAPPProcessador::operator= (const FAPPProcessador& mc){
    objetoId = mc.objetoId;
    fabricante = mc.fabricante;
    familia = mc.familia;
    clock = mc.clock;
    benchMark = mc.benchMark;
    return *this;
}

int FAPPProcessador::operator==(const FAPPProcessador& mc) const{
    int val=1;
    if ((clock != mc.clock) | (familia != mc.familia) |
        (fabricante!=mc.fabricante)
        ) val = 0;
    return val;
}

int FAPPProcessador::operator!=(const FAPPProcessador& mc) const{
```

```

int val=1;
if ((clock == mc.clock) & (familia == mc.familia) &
    (fabricante==mc.fabricante)
    ) val = 0;

return val;
}

// A comparação entre processadores é controvertida
// Drystrones x Weatstones x ... Assim pode-se definir a
// capacidade de processamento dos mesmos e o operadoe de comparação
int FAPPProcessador::operator<(const FAPPProcessador& mc) const{
    return (benchMark == mc.benchMark);
}

ostream& operator<<(ostream& os, const FAPPProcessador& mc){
    os << "Identificador <" << mc.objetoId << ">" << endl;
    os << "  Familia   <" << mc.familia << ">" << endl;
    os << "  Fabricante <" << mc.fabricante << ">" << endl;
    os << "  Clock     <" << mc.clock << ">" << endl;
    os << "  BenchMark <" << mc.benchMark << ">" << endl;
    return os;
}

int FAPPProcessador::GetObjeto(ifstream& file){
    string id, fam, fab, cl, ben;
    if (GetToken(file, id)) return 1;
    if (GetToken(file, fam)) return 1;
    if (GetToken(file, fab)) return 1;
    if (GetToken(file, cl)) return 1;
    if (GetToken(file, ben)) return 1;
    objetoId = id;
    familia = fam;
    fabricante = fab;
    clock = atoi(cl.c_str());
    benchMark = atoi(ben.c_str());
    return 0;
}

string FAPPProcessador::IdGet() const{

```

```

    return objetoId;
}
int FAPPProcessador::ClockGet()const{
    return clock;
}
string FAPPProcessador::FabricanteGet() const{
    return fabricante;
}
string FAPPProcessador::FamiliaGet() const{
    return
        familia;
}
int FAPPProcessador::BenchMarkGet(){
    return benchMark;
}
void FAPPProcessador::IdSet(string in){
    objetoId = in;
}
void FAPPProcessador::FabricanteSet(string in){
    fabricante = in;
}
void FAPPProcessador::FamiliaSet(string in){
    familia = in;
}
void FAPPProcessador::ClockSet(int in){
    clock = in;
}
void FAPPProcessador::BenchMarkSet(int in){
    benchMark = in;
}

```

3.3 Programa-exemplo para a Classe FAPPProcessador

```

// -----
// Copyright (C) 1995 - Nivaldi Calonego Junior
// Geral.cpp

```



```
// Utiliza process.h e a definição da classe em process.cpp.
// -----
#include <classlib/arrays.h>
#include <iostream.h>
#include <strstream.h>
#include <fstream.h>
#include "..\include\process.h"

const int FAPP_MAX_PROC=20; //Numero maximo de processadores permitido
ifstream PROC_FILE;

int CriarArrayProc(TIArrayProcessador& arrayProc, int& numItems){
    int stop=0;
    ifstream PROC_FILE;
    PROC_FILE.open("process.txt");
    if (!PROC_FILE){
        cerr << "Nao pode abrir o arquivo " << endl;
        return 1;
    }
    numItems=0;
    FAPPProcessador p;
    while ( !stop ){
        stop = p.GetObjeto(PROC_FILE);
        if (!stop){
            numItems++;
            arrayProc.Add(new FAPPProcessador(p));
        }
    }
    return 0;
}

void ContainerSave(TIArrayProcessador& cont, ofstream& file){
    TIArrayIterProcessador i(cont);
    cout << "Numero de processadores lidos: ";
    cout << cont.GetItemsInContainer() << endl;
    while (i != 0){
        file << *i.Current() << endl;
    }
}
```

```

        i++;
    }
}

int main(){
    int numItems;
    TIArrayProcessador IArrayProc(FAPP_MAX_PROC);
    if (CriarArrayProc(IArrayProc, numItems)) return 0; // Nao ha dados no arq.
    ofstream file("Saida");
    ContainerSave(IArrayProc, file);
    file.close();
    return 0;
}

```

3.4 Arquivo-exemplo Gerado

```

Identificador <proc1>
  FAMILIA<T800>
  FABRICANTE<INMOS>
  Clock<10>
  BENCHMARK<8>

```

```

Identificador <proc2>
  FAMILIA<80486SX>
  FABRICANTE<Intel>
  Clock<25>
  BENCHMARK<8>

```

```

Identificador <proc3>
  FAMILIA<80486SX>
  FABRICANTE<Intel>
  Clock<25>
  BENCHMARK<8>

```

```

Identificador <proc4>
  FAMILIA<Pentium>

```

FABRICANTE<Intel>

Clock<100>

BENCHMARK<10>

4. Classe dos Dispositivos de Entrada/Saída de Dados

Essa classe-base permite que sejam instanciados objetos que representam elementos básicos para entrada e saída de dados.

4.1 Cabeçalho para a Classe *FAPPEntSai*

```
#ifndef FAPPEntSai_H      // prevent header from
#define FAPPEntSai_H 1   // being included twice
#include <classlib/arrays.h>
#include <cstring.h>
#include <iostream.h>
#include <fstream.h>
class FAPPEntSai{
public:
    FAPPEntSai();
    FAPPEntSai(const string& s);
    FAPPEntSai(int cl, const string& s);
    FAPPEntSai(const FAPPEntSai& mc);
    ~FAPPEntSai();
    FAPPEntSai& operator=(const FAPPEntSai& mc);
    int operator==(const FAPPEntSai& mc) const;
    int operator!=(const FAPPEntSai& mc) const;
    int operator<(const FAPPEntSai& mc) const;
    friend ostream& operator <<(ostream&, const FAPPEntSai&);
    string IdGet() const;
    void IdSet(string);
    int TaxaGet() const;
    void TaxaSet(int);
    char UnidadeGet() const;
    void UnidadeSet(char);
    int GetObjeto(ifstream&);
private:
    string objetoId; // nome do dispositivo
    int taxa; // Taxa de transferencia
```

```

        char unidade; // K/M/G/T bits por segundo
    };

    const int FAPP_MAX_ES=20;
    typedef TArrayAsVector<FAPPEntSai> ContainerType;
    typedef TArrayAsVectorIterator<FAPPEntSai> IteratorType;
#endif

```

4.2 Implementação dos Métodos para a Classe FAPPEntSai

```

// -----
// Copyright (C) 1994 Borland International
// FAPPEntSai.cpp
// -----

#include "entsai.h"
#include "gettoken.h"

FAPPEntSai::FAPPEntSai(){ }

FAPPEntSai::FAPPEntSai(const FAPPEntSai& mc){
    objetoId = mc.objetoId;
    unidade = mc.unidade;
    taxa = mc.taxa;
}

FAPPEntSai::~FAPPEntSai(){ }

FAPPEntSai& FAPPEntSai::operator=(const FAPPEntSai& mc){
    objetoId    = mc.objetoId;
    unidade     = mc.unidade;
    taxa        = mc.taxa;
    return *this;
}

string FAPPEntSai::IdGet()const{
    return objetoId;
}

int FAPPEntSai::TaxaGet() const{
    return taxa;
}

char FAPPEntSai::UnidadeGet() const{
    return unidade;
}

```

```
}  
void FAPPEntSai::IdSet(string in){  
    objetoId = in;  
}  
void FAPPEntSai::TaxaSet(int in){  
    taxa = in;  
}  
void FAPPEntSai::UnidadeSet(char in){  
    unidade = in;  
}  
int FAPPEntSai::operator==(const FAPPEntSai& mc) const{  
    int val=1; // retorna 1 caso sejam iguais  
    if ((taxa!=mc.taxa) | (unidade != mc.unidade))  
        val = 0;  
    return val;  
}  
int FAPPEntSai::operator!=(const FAPPEntSai& mc) const{  
    int val=1; // retorna 1 caso sejam iguais  
    if ((taxa==mc.taxa) & (unidade == mc.unidade))  
        val = 0;  
    return val;  
}  
int FAPPEntSai::operator<(const FAPPEntSai& mc) const{  
    return (taxa == mc.taxa);  
}  
ostream& operator<<(ostream& os, const FAPPEntSai& mc){  
    os << "Dispositivo <" << mc.objetoId << ">" << endl;  
    os << " Taxa <" << mc.taxa << "> ";  
    os << " <" << mc.unidade << "> bits por segundo" << endl;  
    return os;  
}  
int FAPPEntSai::GetObjeto(istream& file){  
    string id, uni, tax;  
    if (GetToken(file, id)) return 1;  
    if (GetToken(file, tax)) return 1;
```

```

if (GetToken(file, uni)) return 1;
// Existem dados: Nao ha verificação de consistência
// Caso o arquivo de dados contenha dados em ordem errada
// ocorrerá erro ao iniciar o objeto.
objetoId = id;
unidade = (uni.c_str())[0];
taxa = atoi(tax.c_str());
return 0;
}

```

4.3 Programa-exemplo para a Classe FAPPEntSai

```

// -----
// Copyright (C) 1995 - Nivaldi Calonego Junior
// Geral.cpp
// Utiliza process.h e a definição da classe em process.cpp.
// -----

#include <classlib/arrays.h>
#include <iostream.h>
#include <strstrea.h>
#include <fstream.h>
#include "..\include\entsai.h"
typedef TIArrayAsVector<FAPPEntSai> TIArrayEntSai;
typedef TIArrayAsVectorIterator<FAPPEntSai> TIArrayIterEntSai;
int CriarArrayES(TIArrayEntSai& arrayES, int& numItems){
    int stop=0;
    ifstream ENTSAI_FILE;
    ENTSAI_FILE.open("entsai.txt");
    if (!ENTSAI_FILE){
        cerr << "Nao pode abrir o arquivo " << endl;
        return 1;
    }
    FAPPEntSai p;
    while ( !stop ){
        stop = p.GetObjeto(ENTSAI_FILE);
        if (!stop){arrayES.Add(new FAPPEntSai(p)); numItems++;}}
}

```

```

    return 0;
}

void ContainerSave(TIArrayEntSai& cont, ofstream& file){
    TIArrayIterEntSai i(cont);
    while (i != 0){ file << *i.Current() << endl; i++; }
}

int main(){
    int numItems;
    TIArrayEntSai arrayES(FAPP_MAX_ENTSAI);
    if (CriarArrayES(arrayES, numItems)) return 0; // Nao ha dados no arq.
    ofstream file("Saida");
    ContainerSave(arrayES, file);
    cout << "acabou";
    return 0;
}

```

4.4 Arquivo-exemplo Gerado

```

Dispositivo <p1>
    Taxa <10> <M> bits por segundo
Dispositivo <p3>
    Taxa <1000> <M> bits por segundo
Dispositivo <p4>
    Taxa <1000> <G> bits por segundo

```


5. Classe Software Operacional

Classe que representa o conjunto de programas necessários para que o hardware funcione a contento, podendo ser algum tipo de firmware.

5.1 Cabeçalho para a Classe XXXX

```
#ifndef FAPPSoftwareOper_H    // prevent header from
#define FAPPSoftwareOper_H 1    // being included twice
#include <classlib/arrays.h>
#include <cstring.h>
#include <iostream.h>
#include <fstream.h>
#include "gettoken.h"
#define MAX_TAM_VISAO_SO 20
struct FAPPSOBuffer{
    char objetoId[MAX_TAM_VISAO_SO]; // Identificador do objeto
    char sistema[MAX_TAM_VISAO_SO]; // Nome do sistema operacional
    char versao[MAX_TAM_VISAO_SO];
    char benchMark[MAX_TAM_VISAO_SO]; // Índice para a comparação
                                     // entre dois sistemas
};
class FAPPSoftwareOper{
public:
    FAPPSoftwareOper();
    FAPPSoftwareOper(const FAPPSoftwareOper&);
    FAPPSoftwareOper(struct FAPPSOBuffer);
    ~FAPPSoftwareOper();
    FAPPSoftwareOper& operator=(const FAPPSoftwareOper& mc);
    int operator==(const FAPPSoftwareOper& mc) const;
    int operator<(const FAPPSoftwareOper& mc) const;
    friend ostream& operator <<(ostream&, const FAPPSoftwareOper&);
    string SistemaGet() const; // Obtem o nome do sistema
    void SistemaSet(string); // Modifica o nome do sistema
```

```

string VersaoGet() const; // Obtem versao
void VersaoSet(string); // modifica versao
int BenchMarkGet() const; // Obtem desempenho
void BenchMarkSet(int); // Modifica desempenho
int GetObjeto(ifstream&);
string IdGet() const;
void IdSet(string);

protected:
    string objetoId; // Identificador do objeto
    string sistema; // Nome do sistema operacional
    string versao;
    int benchMark; // Indice de comparação entre dois sistemas
};

typedef TArrayAsVector<FAPPSoftwareOper> TArraySoftwareOper;
typedef TArrayAsVectorIterator<FAPPSoftwareOper> TArrayIterSoftwareOper;
typedef TArrayAsVector<FAPPSoftwareOper> ContainerType;
typedef TArrayAsVectorIterator<FAPPSoftwareOper> IteratorType;
#endif

```

5.2 Implementação dos Métodos para a Classe FAPPSoftwareOper

```

// -----
// Copyright (C) 1994 Borland International
// FAPPSoftwareOper.cpp
// -----

#include "softoper.h"
FAPPSoftwareOper::FAPPSoftwareOper(){ }
FAPPSoftwareOper::FAPPSoftwareOper(const FAPPSoftwareOper& mc){
    objetoId = mc.objetoId;
    sistema = mc.sistema;
    versao = mc.versao;
    benchMark = mc.benchMark;
}
FAPPSoftwareOper::FAPPSoftwareOper(struct FAPPSOBuffer obj){
    objetoId = obj.objetoId;

```

```

        sistema = obj.sistema;
        versao = obj.versao;
        benchMark = atoi (obj.benchMark);
    }
FAPPSoftwareOper::~FAPPSoftwareOper(){ }
FAPPSoftwareOper& FAPPSoftwareOper::operator=(const FAPPSoftwareOper& mc){
    objetoId = mc.objetoId;
    sistema = mc.sistema;
    versao = mc.versao;
    benchMark = mc.benchMark;
    return *this;
}

string FAPPSoftwareOper::SistemaGet()const { return sistema;}
void FAPPSoftwareOper::SistemaSet(string s) { sistema = s;}
string FAPPSoftwareOper::VersaoGet()const {return versao;}
void FAPPSoftwareOper::VersaoSet(string v) { versao = v;}
string FAPPSoftwareOper::IdGet()const{return objetoId;}
void FAPPSoftwareOper::IdSet(string id){objetoId = id;}
int FAPPSoftwareOper::BenchMarkGet() const{ return benchMark;}
void FAPPSoftwareOper::BenchMarkSet(int b) {benchMark = b;}
int FAPPSoftwareOper::operator==(const FAPPSoftwareOper& mc) const{
    int val=1; // retorna 1 => mesmo nome e versão
    if ( (sistema != mc.sistema) | (versao != mc.versao)) val = 0;
    return (val);
}

int FAPPSoftwareOper::operator<(const FAPPSoftwareOper& mc) const{
    return (benchMark == mc.benchMark);
}

ostream& operator<<(ostream& os, const FAPPSoftwareOper& mc){
    os << "Sistema Operacional <" << mc.objetoId << ">" << endl;
    os << "    nome do sistema <" << mc.sistema << ">" << endl;
    os << "    versao    <" << mc.versao << ">" << endl;
    os << "    benchmark <" << mc.benchMark << ">" << endl;
    return os;
}

```

```

int FAPPSoftwareOper::GetObjeto(ifstream& file){
    string tk0, tk1, tk2, tk3;
    if (GetToken(file, tk0)) return 1;
    if (GetToken(file, tk1)) return 1;
    if (GetToken(file, tk2)) return 1;
    if (GetToken(file, tk3)) return 1;
    objetoId = tk0;
    sistema = tk1;
    versao = tk2;
    benchMark = atoi(tk3.c_str());
    return 0;
}

```

5.3 Programa-exemplo para a Classe FAPPSoftwareOper

```

// -----
// Copyright (C) 1995 - Nivaldi Calonego Junior
// tstso.cpp
// Um teste para a classe FAPPSoftwareOper
// Utiliza process.h e a definição da classe em process.cpp.
// Arquivo: V_SO.H
// -----
#ifndef FAPPSOVisao_H // prevent header from
#define FAPPSOVisao_H 1 // being included twice
#include <fstream.h>
#include "softoper.h"
#include <owl\listbox.h>
#define MAX_ITEMS 30
class FAPPSOVisao{
public:
    FAPPSOVisao();
    ~FAPPSOVisao();
    int Existe(const FAPPSoftwareOper&);
    int IniciaContainer(string); // Cria um container a partir de um arquivo
    void SalvaContainer(string); // Grava container no arquivo
    void Insere(FAPPSoftwareOper); // Insere um elemento no container

```

```

void Apaga(const char*); // Apaga um elemento do container
unsigned ObtemNumItens(); // Número de elementos do container
void GetListId(string**, int&); // Monta uma lista contendo o
    // identificadores dos objetos existentes no container
int GetObjeto(FAPPSoftwareOper&); // Obtem copia do objeto da lista
    // deve ser passado o atributo de identificacao do objeto: objetold
int GetObjeto(int, FAPPSoftwareOper&); // Obtem uma copia do elemento

    // cujo indice eh passado como argumento.

protected:
    TArrayAsVector<FAPPSoftwareOper>* container;
    int MaxItems;

private:
    string* Messages[];
};

#endif

#include <classlib\arrays.h>
#include "v_so.h"
FAPPSOVisao::FAPPSOVisao(){
    MaxItems = MAX_ITEMS;
    container = new TArrayAsVector<FAPPSoftwareOper>(MAX_ITEMS, 0, 3);
}
FAPPSOVisao::~FAPPSOVisao(){
    delete container;
}
int FAPPSOVisao::IniciaContainer(string fileName){
    int stop=0;
    ifstream SOFTOPER_FILE;
    SOFTOPER_FILE.open(fileName.c_str());
    if (!SOFTOPER_FILE){
        cerr << "Nao pode abrir o arquivo " << endl;
        return 1;
    }
    while ( !stop ){
        FAPPSoftwareOper p;

```

```

    stop = p.GetObjeto(SOFTOPER_FILE);
    if (!stop)
        container->Add(p);
}
return 0;
}

int FAPPSOVisao::GetObjeto(FAPPSoftwareOper& so){
    IteratorType i(*container);
    while (i){
        if ( i.Current().IdGet() == so.IdGet() ){
            so = i.Current(); // Retorna uma copia do objeto
            return 0;
        } // Armazena os objetos

        i++;
    }
    return 1;
}

void FAPPSOVisao::GetListId(string* lista[MAX_ITEMS], int& n){
    IteratorType i(*container);
    n=0;
    while (i){
        lista[n] = new string(i.Current().IdGet());
        n++;
        i++;
    }
}

void FAPPSOVisao::SalvaContainer(string fileName){
    ofstream file;
    file.open(fileName.c_str()); // Cria stream e associa a file
    IteratorType i(*container); // Instancia um iterador para o container
    while (i){
        file << i.Current() << endl; // Armazena os objetos
        i++;
    }
}

```

```
file.close();
}

int FAPPSOVisao::GetObjeto(int indice, FAPPSoftwareOper& obj){
    IteratorType i(*container); // Instancia um iterador para o container
    while (i && (indice > 0)){
        i++;
        indice--;
    }
    if (i){
        obj = i.Current(); // Armazena os objetos
        return 0;
    }
    return 1;
}

int FAPPSOVisao::Existe(const FAPPSoftwareOper& so){
    return ( container -> HasMember(so) );
}

void FAPPSOVisao::Insere(FAPPSoftwareOper so){
    if (container->GetItemsInContainer() < MaxItems)
        container->Add(so);
    else
        ::MessageBox(0, "Capacidade de armazenamento foi esgotada",
            CLASSE: FAPPSOVisão - ERRO",
            MB_OK | MB_ICONEXCLAMATION);
}

void FAPPSOVisao::Apaga(const char* id){
    IteratorType i(*container);
    int indice = 0;
    while (i){
        if ( strcmp (i.Current().IdGet().c_str(), id) == 0 ){
            container->Destroy(indice);
            break;
        } // Armazena os objetos
    }
```

```

        i++;
        indice++;
    }
}

unsigned FAPPSOVisao::ObtemNumItens(){
    return(container->GetItemsInContainer());
}

#include "v_so.h"

int main(){
    FAPPSOVisao v;
    string obj;
    if (v.IniciaContainer("SOFTOPER.TXT")){
        cout << "Nao ha dados no arquivo de netrada " << endl;
        return 0; // Nao ha dados no arq.
    }
    v.SalvaContainer("saida");
    cout << "Salvou saida - ok";
    cin >> obj;
    v.Apaga(obj);
    v.SalvaContainer("saida1");
    return 0;
}

```

6. Classe Elemento Processador

Classe-base que armazena as informações elementares contidas nos elementos de processamento descritos no capítulo 4.

6.1 Cabeçalho para a classe base *FAPPElemProcessador*

```

#ifndef FAPPElemProcessador_H           // Evita que o cabeçalho seja incluído
#define FAPPElemProcessador_H 1        // duas vezes
#include <fstream.h>
#ifndef FAPPProcessador_H
#include "..\include\process.h"
#endif
#ifndef FAPPMemoria_H

```



```
#include "..\include\memoria.h"
#endif
#ifndef FAPPEntSai_H
#include "..\include\entsai.h"
#endif
#ifndef FAPPMecInterconexao_H
#include "..\include\intercon.h"
#endif
#ifndef FAPPSoftOperacional_H
#include "..\include\softoper.h"
#endif
#include "..\include\gettoken.h"
class FAPPElemProcessador{
public:
    FAPPElemProcessador();
    FAPPElemProcessador(const FAPPElemProcessador& mc);

    ~FAPPElemProcessador();

    FAPPElemProcessador& operator=(const FAPPElemProcessador& mc);
    int operator==(const FAPPElemProcessador& mc) const;
    int operator!=(const FAPPElemProcessador& mc) const;
    int operator<(const FAPPElemProcessador& mc) const;
    friend ostream& operator <<(ostream&, const FAPPElemProcessador&);

    // Metodos
    int GetObjeto(istream&); // return 1 = erro
    int equivalente(FAPPElemProcessador, FAPPElemProcessador);
protected:
    char mem;// 0 não possui memoria local
    string objetoId; // Identificador do objeto
    FAPPSoftwareOper *so;
    FAPPProcessador *processador;
    FAPPMemoria *memorialLocal;
    FAPPMecInterconexao *mecInter;
```

```

    int nItensEntSai; // Número de itens na lista de dispositivos de I/O
    ListaEntSai entSai;
    int benchMark;
};

typedef TArrayAsVector<FAPPElemProcessador> TArrayElemProcessador;
typedef TArrayAsVectorIterator<FAPPElemProcessador> TArrayIterElemProcessador;
#endif

```

6.2 Implementação dos Métodos da Classe FAPPElemProcessador

```

// -----
// FAPPElemProcessador.cpp
// -----

#include "..\include\elemproc.h"

FAPPElemProcessador::FAPPElemProcessador(){
    processador = NULL;
    memoriaLocal = NULL;
    mecInter    = NULL;
    so          = NULL;
    mem         = '0';
}

FAPPElemProcessador::FAPPElemProcessador(const FAPPElemProcessador& mc){
    mem = mc.mem;
    objetoId = mc.objetoId;
    benchMark = mc.benchMark;
    nItensEntSai = mc.nItensEntSai;
    processador = new FAPPProcessador(*mc.processador);
    memoriaLocal = new FAPPMemoria(*mc.memoriaLocal);
    mecInter    = new FAPPMecInterconexao(*mc.mecInter);
    so = new FAPPSoftwareOper(*mc.so);
    IterListaEntSai i(mc.entSai);
    while (i){
        FAPPEntSai es;
        es = i.Current();
        entSai.Add(es);
    }
}

```

```

        i++;
    }
}

FAPPElemProcessador::~FAPPElemProcessador(){
    if (processador) delete processador;
    if (memoriaLocal) delete memoriaLocal;
    if (mecInter) delete mecInter;
    if (so) delete so;

}

FAPPElemProcessador& FAPPElemProcessador::operator=(const FAPPElemProcessador&
mc){

    mem = mc.mem;
    benchMark = mc.benchMark;
    nltensEntSai = mc.nltensEntSai;
    objetoId = mc.objetoId;
    processador = new FAPPProcessador(*mc.processador);
    memoriaLocal = new FAPPMemoria(*mc.memoriaLocal);
    mecInter = new FAPPMecInterconexao(*mc.mecInter);
    so = new FAPPSoftwareOper(*mc.so);
    IterListaEntSai i(mc.entSai);
    FAPPEntSai es;
    while (i){
        es = i.Current();
        entSai.Add(es);
        i++;
    }
    return *this;
}

int FAPPElemProcessador::equivalente(FAPPElemProcessador x,
FAPPElemProcessador y){
    int val=1; // retorna 1 caso sejam iguais
    // Verificar a lista de dispositivos de I/O

```

```

if ((*y.processador) != *(x.processador)) |
    ((*y.memoriaLocal) != *(x.memoriaLocal)) |
    ((*y.mecInter) != *(x.mecInter))
    ) val = 0;

return val;
}

int FAPPElemProcessador::operator==(const FAPPElemProcessador& mc) const{
    return objetoId == mc.objetoId;
}

int FAPPElemProcessador::operator!=(const FAPPElemProcessador& mc) const{
    int val=1; // retorna 1 caso sejam iguais
    if ((*processador == *(mc.processador)) &
        (*memoriaLocal == *(mc.memoriaLocal)) &
        (*mecInter == *(mc.mecInter))
        ) val = 0;

    return val;
}

int FAPPElemProcessador::operator<(const FAPPElemProcessador& mc) const{
    return (benchMark < mc.benchMark);
}

ostream& operator<<(ostream& os, const FAPPElemProcessador& mc){
    os << "Elem. Processador: <" << mc.objetoId << ">" << endl;
    os << " Tipo mem: <" << mc.mem << ">" << endl;
    os << " Num. disp. de E/S: <" << mc.nItensEntSai << ">" << endl;
    os << " BenchMarck: <" << mc.benchMark << ">" << endl;
    os << " Processador: " << *(mc.processador) << endl;

    switch(mc.mem) {
        case '0': os << " Nao tem memoria local" << endl; break;
        case '1': os << " Mem. Local: " << *mc.memoriaLocal << endl; break;
    }
}

```