



PALAVRAS CHAVES/KEY WORDS

AUTORES/AUTHORS: TESTES DE "SOFTWARE" MULTIPROGRAMAÇÃO  
SISTEMAS DISTRIBUÍDOS MULTIPROCESSAMENTO  
SISTEMAS OPERACIONAIS

AUTORIZADA POR/AUTHORIZED BY

*Marco Antonio Raupp*  
Diretor Geral

AUTOR RESPONSÁVEL  
RESPONSIBLE AUTHOR

*Mauro de M. Spinola*  
Mauro de M. Spinola

DISTRIBUIÇÃO/DISTRIBUTION

INTERNA / INTERNAL  
 EXTERNA / EXTERNAL  
 RESTRITA / RESTRICTED

REVISADA POR / REVISED BY

*João José Neto*

CDU/UDC

681.3.06

DATA / DATE

Dez. 1.986

TÍTULO/TITLE	PUBLICAÇÃO Nº PUBLICACION NO INPE-4091-TDL/252
	MÁQUINA DE ARQUITETURA DISTRIBUÍDA - CONSIDERAÇÕES SOBRE TESTES DE "SOFTWARE"
AUTORES/AUTHORSHIP	Mauro de Mesquita Spinola

ORIGEM  
ORIGIN

DRH-DIN

PROJETO  
PROJECT

FRH/CAP

Nº DE PAG.  
NO OF PAGES

170

ULTIMA PAG.  
LAST PAGE

B.9

VERSÃO  
VERSION

Nº DE MAPAS  
NO OF MAPS

RESUMO-NOTAS / ABSTRACT - NOTES

Sistemas distribuídos têm sido utilizados em escalas crescentes em diversas aplicações. As diferenças existentes entre esses sistemas e os sistemas concentrados trazem impactos nas metodologias de desenvolvimento do "software" e dos seus testes. Uma metodologia para realizar testes de "software" distribuído é apresentada, identificando-se técnicas e ferramentas que podem ser adotadas com a finalidade de simplificar esta atividade. Este trabalho trata do desenvolvimento de uma metodologia para testes de "software" distribuído. Esta metodologia resulta da experiência adquirida durante os testes de "software" de uma máquina de arquitetura distribuída particular. Algumas propostas para aprimoramento dessa metodologia são também apresentadas.

OBSERVAÇÕES / REMARKS

Dissertação de Mestrado em Computação Aplicada, aprovada em 27 de Fevereiro de 1986.

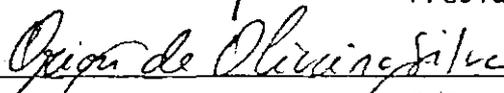
Aprovada pela Banca Examinadora  
em cumprimento a requisito exigido  
para a obtenção do Título de Mestre  
em Computação Aplicada

Dr. Celso de Renna e Souza



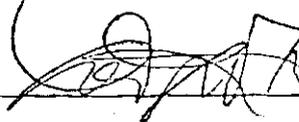
Presidente

Dr. Orion Oliveira Silva



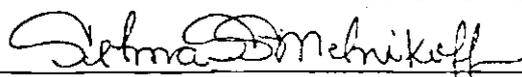
Orientador

Dr. João José Neto



Co-Orientador

Dra. Selma S. Shimizu Melnikoff



Membro da Banca  
-convidada-

Dr. Tatuō Nakanishi



Membro da Banca

Candidato: Mauro de Mesquita Spínola

São José dos Campos, 27 de fevereiro de 1986



A  
Diva,  
meus filhos e  
meus pais.

4

## AGRADECIMENTOS

Ao Prof. Dr. João José Neto pela efetiva orientação.

Ao Prof. Dr. Orion Oliveira Silva pelas sugestões.

Ao INPE - Instituto de Pesquisas Espaciais, pela oportunidade que tive de assistir ao Curso de Mestrado em Computação Aplicada e realizar este trabalho.

À FDTE - Fundação para o Desenvolvimento Tecnológico da Engenharia, pelo apoio à realização deste trabalho.

À Profª Dra. Edith Ranzini pelo interesse e sugestões.

Aos colegas da FDTE pelo trabalho que realizaram e as contribuições que deram para o desenvolvimento das idéias apresentadas.

A Sra. Gloria Cardozo Bertti pela minuciosa revisão de linguagem.

A Sra. Sonia Regina Izarelli e às Srtas. Laudelina Santa Scarparo, Andréa Pinesi, Mara Gomes, Cleusa Aparecida Martins e Elizabeth Fátima Hiromi Tokuda pelos serviços de datilografia.

Aos técnicos Alberto Barreto dos Santos e Evaristo Dutra da Costa Neto pelos serviços de desenho.

A minha esposa, Divanir Mouta Garcia de Mesquita Spinola, e meus filhos, Fernando Garcia Spinola e Maíra Garcia Spinola, pela compreensão nas minhas ausências e incentivo constante.

A meus pais, Geraldo de Souza Spinola e Neyde de Mesquita Spinola, pela confiança que sempre me transmitiram.

••

## PREFÁCIO

Este trabalho propõe uma metodologia de testes de "software" para sistemas de arquitetura distribuída e está dirigido aos técnicos dedicados ao desenvolvimento e aos testes de sistemas desse tipo. Uma das suas motivações é a constatação de que essa área é pouco explorada, apesar da crescente adoção de sistemas distribuídos para aplicações diversas. A metodologia aqui apresentada está embasada na experiência adquirida através da realização dos testes do "software" distribuído levados a efeito no Núcleo de Processamento Distribuído (NPD), um sistema de computação com arquitetura não-convencional desenvolvido no Laboratório de Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, em convênio com a Fundação para o Desenvolvimento Tecnológico da Engenharia - FDTE. As empresas Ferrovia Paulista S.A. - FEPASA, e Trens Urbanos de Porto Alegre - TRENSURB, patrocinaram o desenvolvimento e os testes do "hardware" e do "software" aplicativo para controle de tráfego ferroviário. O referido projeto contou com a participação de uma equipe de 40 engenheiros, a partir da contribuição dos quais foi possível ao autor a coleta e a presente formalização das idéias utilizadas. O trabalho está organizado da seguinte forma: os Capítulos 1 e 2 apresentam os conceitos e a terminologia adotados em relação aos sistemas de arquitetura distribuída e às técnicas para desenvolvimento de "software". No capítulo 3 são apresentadas propostas para elaboração e realização dos testes de sistemas de "software" distribuído, com base na experiência profissional da equipe e nas contribuições da literatura. No Capítulo 4 apresenta-se a metodologia de testes utilizada no projeto, e no Capítulo 5, as conclusões. As descrições do NPD e de seu ambiente de testes são feitos nos Apêndices A e B, respectivamente.

Mauro de Mesquita Spinola

#### ABSTRACT

*Distributed systems have been used at growing rates in many applications. The existing differences between these systems and centralized systems bring impacts on the software development and test methodologies. A methodology to make tests on distributed software is presented, identifying techniques and tools that can be adopted in order to simplify this task. This work is concerned with the development of a methodology for testing distributed software. This methodology had resulted of the experience acquired during the software tests of a particular distributed architecture machine. Some proposals to improve this methodology are also presented.*

## SUMÁRIO

	<u>Pág.</u>
LISTA DE FIGURAS .....	<i>xiii</i>
<u>CAPÍTULO 1 - SISTEMAS DE COMPUTAÇÃO DISTRIBUÍDOS</u> .....	1
1.1 - Sistemas operacionais .....	1
1.2 - Sistemas distribuídos .....	6
<u>CAPÍTULO 2 - TÉCNICAS PARA DESENVOLVIMENTO DE "SOFTWARE"</u> .....	9
2.1 - Considerações iniciais .....	9
2.2 - O ciclo de vida do desenvolvimento de "software" .....	13
2.2.1 - Fase 1: Análise do problema, definição de necessidades e especificação .....	14
2.2.2 - Fase 2: Projeto .....	15
2.2.3 - Fase 3: Codificação .....	17
2.2.4 - Fase 4: Testes .....	18
2.2.5 - Fase 5: Manutenção e operação .....	18
2.3 - Verificação, validação e teste do "software" .....	19
2.3.1 - Algumas definições .....	19
2.3.2 - Verificação durante todo o ciclo de vida do "software" ...	20
2.3.3 - A fase de testes .....	20
<u>CAPÍTULO 3 - TESTES DE "SOFTWARE" PARA SISTEMAS DE ARQUITETURA DISTRIBUÍDA</u> .....	27
3.1 - As dificuldades de testar "software" concorrente e distribuído .....	27
3.1.1 - Múltiplos processos assíncronos .....	28
3.1.2 - Múltiplos processadores .....	30
3.2 - Resenha das contribuições da literatura .....	32
3.3 - Diretrizes para a realização de testes .....	36
3.3.1 - Desenvolvimento de "software" visando os testes .....	37
3.3.1.1 - Aplicação dos objetivos da Engenharia de Software a sistemas distribuídos .....	38
3.3.1.2 - Especificação projeto e codificação .....	47
3.3.2 - Verificações em diversas fases do desenvolvimento .....	51
3.3.3 - Testes .....	52

	<u>Pág.</u>
3.3.3.1 - Ambiente de testes .....	53
3.3.3.2 - Ferramentas .....	56
3.3.3.3 - Metodologia .....	73
3.4 - Conclusão .....	76
<u>CAPÍTULO 4 - TESTES DE "SOFTWARE" DO NÚCLEO DE PROCESSAMENTO DISTRIBUÍDO</u> .....	79
4.1 - Características do NPD e do ambiente de testes .....	79
4.2 - Atividades voltadas para detecção, diagnóstico e depuração de erros .....	30
4.3 - Diretrizes básicas para o desenvolvimento de testes .....	84
4.4 - Etapas do desenvolvimento dos testes .....	87
4.5 - Testes em níveis funcionais .....	96
4.6 - Depuração de erros .....	100
4.7 - Documentação dos testes .....	107
4.8 - Proposta para aperfeiçoamento dos testes de "software" do NPD	113
4.9 - Conclusão .....	115
<u>CAPÍTULO 5 - CONCLUSÃO</u> .....	117
REFERÊNCIAS BIBLIOGRÁFICAS .....	119
BIBLIOGRAFIA COMPLEMENTAR .....	125
APÊNDICE A - DESCRIÇÃO DO NÚCLEO DE PROCESSAMENTO DISTRIBUÍDO	
APÊNDICE B - O AMBIENTE DE TESTES NO NPD	

## LISTA DE FIGURAS

	<u>Pág.</u>
2.1 - Ciclo de vida do desenvolvimento de "software" .....	13
3.1 - Modificação de funções que desrespeita a estrutura original .	41
3.2 - Modificação de funções que desrespeita a hierarquia original.	42
3.3 - Utilização dos recursos do paralelismo visando eficiência do processamento .....	45
3.4 - Exemplo de estrutura funcional .....	48
3.5 - Simplificação de transação .....	50
3.6 - Modelo de sistema de arquitetura distribuída para testes de "software" distribuído .....	54
3.7 - Envio e recebimento de mensagem pelo SO com testes de consistência de mensagens .....	59
3.8 - Gerenciador de arquivos em disco .....	63
3.9 - Utilização da Interface como SO para teste de um processo ...	64
3.10 - Geração e exames de "traces" .....	68
3.11 - Ferramentas para o ambiente de teste .....	72
3.12 - Estrutura de interconexão de processos a serem testados .....	75
4.1 - Representação gráfica de programas .....	82
4.2 - Representação das transações que compõem a função "pedido de calendário".....	83
4.3 - Bases para a realização dos testes do NPD .....	87
4.4 - Uso do SDPP no teste de um processo .....	89
4.5 - Uso do SDPP em testes integrados .....	91
4.6 - Teste de processos com variáveis compartilhadas .....	93
4.7 - Testes parcialmente integrados .....	94
4.8 - Níveis estruturais .....	98
4.9 - Níveis funcionais .....	99
4.10 - Relação hierárquica entre os níveis funcionais .....	99
4.11 - Depuração de erros .....	103
4.12 - Plano de teste .....	109
4.13 - Correções para uso durante a aplicação dos testes .....	110
4.14 - Correções para uso por parte da equipe de desenvolvimento ...	111
4.15 - Teste de aceitação .....	112
4.16 - NPD com recursos para desenvolvimento de "software" .....	114

Pág.

A.1 - Arquitetura do NPD .....	A.3
A.2 - Estrutura física dos nós PUG e PD .....	A.5
A.3 - NPD e periféricos .....	A.7
A.4 - Sistema de sinalização - TRENSURB .....	A.12
A.5 - Funções do "software" aplicativo .....	A.15
B.1 - Ambiente de testes do NPD .....	B.2
B.2 - Utilização do SDPP para teste .....	B.2

## CAPÍTULO 1

### SISTEMAS DE COMPUTAÇÃO DISTRIBUÍDOS

Os sistemas de computação distribuídos são hoje objeto de estudos e projetos com resultados práticos positivos. Crescem as aplicações desses sistemas à medida que se ampliam e diversificam os seus recursos de "hardware" e "software".

Neste capítulo são apresentados os conceitos e a terminologia relativos a esses sistemas, utilizados no contexto deste trabalho. Inicialmente apresentam-se os Sistemas Operacionais, cuja conceituação e terminologia são fundamentais para introduzir os sistemas distribuídos. Na seção 1.1 são discutidos alguns aspectos relevantes relativos aos Sistemas Operacionais e à programação paralela, com a introdução dos termos técnicos mais importantes. Na seção 1.2, definem-se os sistemas distribuídos propriamente ditos e discutem-se o seu "hardware" e "software".

#### 1.1 - SISTEMAS OPERACIONAIS

Segundo Guimarães (1980) os sistemas operacionais são programas de computador que possuem múltiplas funções, a saber:

- 1) apresentar ao usuário uma máquina mais flexível e adequada para programar do que aquela que o "hardware" nu apresenta;
- 2) possibilitar o uso eficiente e controlado dos vários componentes do "hardware" que constituem o sistema como um todo: processador, memória principal e secundária, canais de entrada/saída, controladores, periféricos etc.;
- 3) possibilitar a diversos usuários o uso compartilhado e protegido dos diversos componentes do "hardware" e do "software" do sistema.

Nota-se que cabe ao sistema operacional gerenciar a utilização dos recursos físicos e lógicos do sistema, apresentando para o usuário uma máquina com maiores facilidades de utilização.

A linguagem de comunicação do usuário com o sistema operacional é chamada *linguagem de controle de tarefas* (Hansen, 1973a). Esta linguagem habilita o usuário a se identificar e descrever as necessidades das tarefas computacionais a serem executadas: tipos e quantidades de recursos necessários e os nomes dos programas e arquivos de dados usados. Podem ser solicitadas ações diversificadas, tais como gravação de um conjunto de dados em disco, compilação de um programa, execução de um programa, abertura e fechamento de arquivos etc.

Uma outra linguagem definida pelo sistema operacional é chamada *linguagem de máquina virtual* (Hansen, 1973a), que abrange o conjunto de operações de máquina disponíveis ao usuário durante a execução do programa. O usuário tem a ilusão de trabalhar numa máquina que pode executar programas escritos nessa linguagem; na verdade apenas um subconjunto das operações de máquina está disponível. A máquina vista daquela forma pelo usuário dá-se o nome de *máquina virtual*.

Costuma-se caracterizar os sistemas operacionais pelo principal tipo de serviço que eles proporcionam aos usuários. Desta forma, pode-se classificá-los, para efeito de estudos, em três tipos principais (Guimarães, 1980):

- 1) *Sistemas do tipo lote (batch)* em que as tarefas dos usuários são agrupadas fisicamente e processadas seqüencialmente, uma após a outra. O sistema impõe um escalonamento na execução das tarefas e nas saídas dos resultados, o que faz com que o número de tarefas processadas por unidade de tempo seja normalmente grande. Por outro lado, os tempos de resposta para o usuário são geralmente altos.
- 2) *Sistemas de tempo compartilhado (time-sharing)* em que vários usuários podem utilizar a máquina simultaneamente. Cada um deles

trabalha num terminal de E/S e pode interagir com o sistema em cada fase de execução de sua tarefa. Normalmente os programas são preparados no próprio terminal através de um editor de texto para serem em seguida compilados e executados; podem ainda ser corrigidos através do editor, repetindo-se em seguida o mesmo processo. Nesses sistemas a taxa de tarefas processadas é normalmente baixa, porém o tempo de resposta por comando do usuário fica em geral dentro de limites aceitáveis.

- 3) *Sistemas de tempo real*, voltados para aplicações de medição ou controle de processos físicos, admitindo em geral a monitoração simultânea de várias atividades, com compartilhamento de recursos entre processos computacionais distintos.

A maioria dos sistemas operacionais reais pode ser incluída entre os três tipos já apresentados.

O estudo de sistemas operacionais modernos envolve, em geral, o conceito de *programação paralela*. A maioria dos computadores atuais opera em regime de *multiprogramação*, isto é, mais de um programa em execução "simultânea" na memória. Dá-se o nome de *processo* à execução de um programa que possa ser feita paralelamente a outras execuções (Madnick, 1974). Assim, num sistema com multiprogramação pode-se ter diversos processos em *estado de execução* (ou seja, em condições de ser executados). Isso não significa que os processos possam executar instruções num mesmo instante, pois o processador só pode atender a um processo de cada vez. O que acontece na realidade é que os processos em estado de execução são atendidos de acordo com regras de sincronização entre eles e tudo passa para o usuário como se as execuções fossem *simultâneas*.

As regras de sincronização servem para definir em cada instante as operações que devem ser realizadas pelo processador. Podem ser regras do tipo: *operação A deve ser executada antes da operação B, operações A e B nunca devem ser executadas ao mesmo tempo* e outras (Hansen, 1973a).

Durante a sua existência, um processo pode passar por diversos estados. Essencialmente existem dois estados (Guimarães, 1980):

- *ativo*, quando o processo está logicamente em execução;
- *bloqueado*, quando o processo está parado à espera de alguma informação ou sinal de um outro processo.

Um processo é bloqueado através de uma operação explícita como *espera por condição (wait)*, após o qual entra então em uma lista de espera da condição especificada.

Os processos ativos podem estar em duas situações distintas: *pronto para executar (ready)* e *em execução (running)*. Na primeira situação o processo está à espera de um processador para continuar a execução, enquanto na segunda o processo está de posse do processador.

Os processos no estado *pronto para executar* são também colocados numa lista. Quando um processo em execução é bloqueado, um outro processo é retirado dessa lista e passa a ser executado no processador recém-liberado.

Algumas vezes os processos não interagem. São independentes no sentido de que a atividade de um não depende e nem interfere na atividade do outro. Em muitos casos, porém, as atividades de processos paralelos não são inteiramente independentes, e dois ou mais processos precisam se comunicar entre si. Diz-se que os processos são *concorrentes*.

A comunicação entre processos concorrentes muitas vezes se faz através de *variáveis compartilhadas*, sendo necessário haver mecanismos de sincronização para controlar o acesso a essas variáveis, de forma que um processo possa realizar operações completas que as envolvam, sem a interferência dos outros processos. Esse problema é denominado *problema da exclusão mútua* entre processos.

Processos concorrentes podem fazer referência ou alterar variáveis comuns através das chamadas *regiões críticas* (ou seja, essas variáveis são usadas apenas dentro de regiões críticas). Uma região crítica é caracterizada pelo acesso a uma ou mais variáveis comuns conjuntamente manipuladas por diversos processos, de forma que ela seja executada de maneira exclusiva e seqüencial no tempo pelos processos.

As implementações de regiões críticas devem satisfazer aos requisitos desejáveis para uma solução geral do problema de exclusão mútua, que são os seguintes (Guimarães, 1980):

- 1) a solução não deve impor uma prioridade estática entre processos que queiram ter acesso à região crítica;
- 2) a única hipótese a ser feita a respeito da velocidade de execução de processos paralelos é de que ela não é nula; em particular um processo dentro de uma região crítica sempre abandona esta região após um tempo finito;
- 3) se um processo é bloqueado fora da região crítica, isto não deve impedir outros processos de entrarem na sua região crítica;
- 4) soluções que permitem situações de corrida que, mesmo improváveis, possam fazer com que dois ou mais processos se bloqueiem mutuamente na entrada de uma região crítica são inaceitáveis.

A implementação de exclusão mútua pode ser feita através de semáforos, variáveis inteiras controladas por operações primitivas que realizam o protocolo de entrada e saída da região crítica, em geral identificadas por  $P$  e  $V$ . Se um semáforo  $s$  é positivo, significa que a região crítica está livre, podendo portanto ser executada por algum processo (pode ser completada com sucesso a operação  $P(s)$ ). Se  $s$  é nulo a região crítica está ocupada e se um processo qualquer a requisitar ficará bloqueado sendo colocado numa fila de espera associada ao semáforo  $s$ . A operação  $V(s)$  incrementa  $s$  e, se um ou mais processos estiverem bloqueados na fila associada a ele, um deles será liberado.

Um dos problemas mais importantes a serem considerados em relação à utilização de semáforos é a possibilidade de ocorrência de "deadlock", um estado em que dois ou mais processos esperam indefinidamente por condições que nunca serão satisfeitas (Hansen, 1973a). Um "deadlock" envolve espera circular: cada processo apresenta-se esperando por uma condição que só pode ser satisfeita por um dos outros. Desde que cada processo espera condições de um dos outros para resolver o conflito, todos ficam impedidos de continuar.

O projeto de sistemas multiprogramados é baseado nos conceitos acima apresentados e visa satisfazer as seguintes necessidades (Madnick, 1974):

- 1) gerenciamento da memória;
- 2) gerenciamento do processador;
- 3) gerenciamento de dispositivo;
- 4) gerenciamento de informação.

De acordo com a aplicação do sistema operacional cada um desses itens será projetado para atender às necessidades do usuário.

## 1.2 - SISTEMAS DISTRIBUÍDOS

Num nível global pode-se identificar três categorias básicas de sistemas (Champine, 1980):

- 1) *sistema centralizado*, com todo processamento e armazenamento em um local geográfico, possivelmente com terminais não-inteligentes (interativos ou em "batch") em outros locais;
- 2) *sistema descentralizado*, com computadores independentes em locais geográficos diferentes que não se comunicam;
- 3) *sistema distribuído*, com computadores em diferentes locais geográficos que se comunicam e processam "software" aplicativo cooperativamente.

As definições baseadas apenas na distribuição física, como a apresentada acima, são questionadas por alguns autores. Segundo Enslow Júnior (1978) no mínimo quatro componentes de um sistema podem ser distribuídos: o "hardware", os dados, o processamento propriamente dito e o controle, podendo-se dizer que um sistema que tem qualquer um desses componentes distribuído é um *sistema de processamento de dados distribuído*. No entanto, Enslow Júnior concorda que não há distribuição de funções dissociada de uma distribuição do "hardware".

A definição de sistemas distribuídos não é rigorosa. Porém alguns parâmetros são normalmente aceitos para caracterizá-los. Uma característica considerada fundamental é a existência de muitos locais de controle. Além disso deve haver capacidade de processamento e armazenamento de dados em locais distintos. Os elementos de armazenamento e processamento dos vários locais podem ser ou não do mesmo tamanho, com compatibilidade e funcionalidade.

Um sistema distribuído é composto por dois ou mais elementos chamados *nós*. Cada nó é um sistema de computação completo numa certa localização. A ligação entre os nós é feita através de uma rede de interconexão, que pode ter várias topologias possíveis.

O projeto de sistemas distribuídos segue, em geral, as mesmas fases de projetos de sistemas centralizados, ou seja: definição de necessidades, especificação funcional, projeto, implementação e validação (Champine, 1980). A diferença mais significativa entre o projeto de sistemas centralizados e distribuídos é a partição do processamento e das funções. Nos sistemas centralizados uma partição é feita para alocar as funções para o pessoal de projeto e implementação. Em sistemas distribuídos a partição deve ser feita em primeiro lugar logicamente e depois fisicamente. Esta partição determina, em grande parte, o sucesso de um sistema distribuído. Como critério básico para a partição lógica tem-se a minimização da comunicação entre as partições.

O maior problema na implementação de sistemas distribuídos é o desenvolvimento do "software". Existe "hardware" adequado com topologias variadas, mas para cada nova aplicação necessita-se desenvol

ver ou adaptar o "software". Além disso, o "software" básico deve fornecer os recursos lógicos necessários à sincronização dos processos.

O Capítulo 3 apresenta diversas considerações relativas ao desenvolvimento de "software" distribuído voltado para facilitar a fase de testes.

## CAPÍTULO 2

### TÉCNICAS PARA DESENVOLVIMENTO DE "SOFTWARE"

Os conceitos relativos a técnicas e ferramentas para desenvolvimento de "software" têm hoje um tratamento homogêneo na literatura. Os diversos estudiosos do assunto já estão familiarizados com os tópicos da Engenharia de Software que são aqui abordados. Assim sendo, este capítulo vai se restringir a apresentar os conceitos fundamentais e a terminologia da área que sejam de interesse para o trabalho como um todo.

#### 2.1 - CONSIDERAÇÕES INICIAIS

O desenvolvimento de "software" tem ganho importância cada vez maior no projeto de sistemas baseados em computador, sobretudo devido à parcela que representa em relação ao custo total do projeto.

Os altos custos do "software", em relação ao "hardware", devem-se aos seguintes fatos (Shooman, 1983): (1) cada nova aplicação exige novos programas; (2) a substituição de computadores antigos por outros mais novos exige, em alguns casos, novo "software" ou modificações do existente; (3) programação é uma atividade valorizada no mercado de trabalho porque requer profissionais altamente especializados, principalmente para programas de grande porte.

Shooman (1983) apresenta um estudo realizado pela Força Aérea Americana sobre os gastos com "software" e "hardware", com projeções até 1985, mostrando uma relação de aproximadamente quatro para um entre os custos de "software" e "hardware" estimados para 1985, praticamente o inverso da relação existente trinta anos antes.

Outros estudos trazem conclusões semelhantes. Boehm (1973) apresentou uma estimativa para 1985, onde aproximadamente 90% dos custos totais de um sistema referem-se ao "software". Todos esses resul

tados percentuais, aplicados aos vultosos investimentos anuais em computação, dão uma idéia da importância dos custos de "software".

No Brasil a razão entre os custos de "software" e "hardware" é ainda maior devido aos baixos investimentos em desenvolvimento de novas máquinas e à necessidade, sempre presente, de adaptação dos computadores já existentes a novas aplicações.

O tempo investido e o número exigido de profissionais (os custos, em outras palavras) determinaram a necessidade de sistematizar os esforços voltados para o desenvolvimento de "software". Os resultados dos estudos e trabalhos práticos realizados constituem hoje a *Engenharia de Software*, área do conhecimento voltado para a produção de programas ao mesmo tempo economicamente viáveis e suficientemente confiáveis (Boehm, 1976). Esse duplo critério, aplicado racionalmente a cada projeto, orienta a metodologia a ser utilizada, bem como o investimento a ser feito em cada fase do desenvolvimento. Ambos os critérios devem ser utilizados com base em necessidades reais, determinadas pela aplicação prática a que o projeto se destina.

Gradualmente, a programação se transforma de uma arte para uma ciência de engenharia (Ramamoorthy, 1976).

Mas essa ciência ainda não conseguiu dominar a chamada *crise do software*, existente sobretudo pelo fato de as necessidades se desenvolverem com taxas superiores à capacidade de satisfazê-las com eficiência e métodos adequados. Essa crise pode ser resumida nos seguintes pontos (Booch, 1983): (1) sistemas baseados em computador frequentemente não vão ao encontro de todas as necessidades do usuário; (2) "software" falha com frequência; (3) custos de "software" são raramente previsíveis e frequentemente excessivos; (4) manutenção de "software" é uma atividade complexa, cara e sujeita a erros; (5) projetos de "software" estão frequentemente atrasados e apresentam resultados aquém do desejado; (6) dificilmente se utiliza "software" de um sistema em outro, mesmo quando funções similares são exigidas; (7) esfor

ços para desenvolvimento de "software" não otimizam o uso de recursos envolvidos, tais como tempo de processamento e área ocupada de memória.

Combater a crise do "software", seja através do desenvolvimento de técnicas e ferramentas para desenvolvimento e testes, seja através da conscientização quanto à necessidade de sua utilização, é a proposta da Engenharia de software. O treinamento de pessoal e a organização dos grupos de trabalho, realizados com base em seus fundamentos dão, ao desenvolvimento de "software", oportunidades maiores de sucesso.

Atribuem-se, geralmente, quatro objetivos principais à Engenharia de Software, a saber (Ross, 1975): (1) *alterabilidade (modifiability)*; (2) *eficiência*; (3) *confiabilidade (reliability)* e (4) *compreensibilidade*, os quais serão discutidos a seguir.

As alterações do "software" podem ser motivadas por possíveis modificações na especificação ou pela necessidade de correção de erros introduzidos durante o desenvolvimento. Se o "software" possui a característica de *alterabilidade*, ou seja, se sua estrutura é clara e consistente, além de bem documentada, a manutenção do "software" torna-se viável, se realizada também de forma adequada. As alterações são controladas de maneira a manter algumas partes ou aspectos e mudar outros, respeitando a estrutura já existente ou alterando-a de forma a manter a clareza e a consistência. Deve-se ressaltar que a alteração pode ocorrer em qualquer fase do desenvolvimento, exigindo-se sempre a revisão das fases anteriores.

Entende-se por *eficiência* a melhor utilização economicamente possível dos recursos disponíveis, sejam físicos (memória, periféricos, etc.) ou não (tempo de execução, por exemplo). Projetos que visam maior eficiência apoiam-se em estruturas de dados e nas rotinas de acesso a elas, ao lado de rotinas otimizadas para acesso a periféricos e para laços de programas (loops) considerados críticos. Também é possível aumentar a eficiência através do uso de paralelismo de programas, contanto que se utilizem adequadamente as características assíncronas de comunicação entre os processos.

O requisito básico para sistemas de "software" é a correção (Ramamoorthy, 1976); por isso a *confiabilidade* é um dos objetivos da Engenharia de Software. Os erros de software têm um grande peso nos custos do projeto (Shooman, 1983), sobretudo se sua detecção ocorrer em fases posteriores àquelas em que foram gerados (a correção de um erro de especificação, por exemplo, torna-se muito mais cara se sua detecção só ocorrer durante a fase de testes). Por isso, busca-se desenvolver "software" com geração de um número mínimo de erros e com recursos para verificação de correção nas diversas fases do desenvolvimento, com conseqüente eliminação dos erros detectados.

A *compreensibilidade* dos programas é também um dos objetivos da Engenharia de "software". Sendo o problema simples ou complexo, os objetivos de alterabilidade, eficiência e confiabilidade dificilmente podem ser atingidos sem que o sistema seja compreensível. A sua estrutura deve ser tal que permita a uma pessoa qualificada absorvê-la, o que envolve clareza na organização dos dados, na definição dos módulos (suas funções e características), na descrição dos algoritmos, em um nível mais alto, e na facilidade de leitura do código implementado, em um nível mais baixo.

As linguagens de programação estruturadas e de alto nível são as mais adequadas para atender ao objetivo da compreensibilidade. A documentação do sistema deve respeitar, em todos os níveis do projeto, ao objetivo da compreensibilidade, tornando-se o maior instrumento para atingi-la.

No âmbito desse trabalho a expressão Engenharia de Software será usada no sentido de Boehm (1976; 1981): "aplicação prática do conhecimento científico no projeto e construção de programas de computador e na documentação associada requerida para desenvolvê-los, operá-los e mantê-los".

## 2.2 - CICLO DE VIDA DO DESENVOLVIMENTO DE "SOFTWARE"

O desenvolvimento de "software" possui um ciclo de vida que pode ser dividido nas seguintes fases: (1) *análise do problema, levantamento de necessidades e especificação*; (2) *projeto*; (3) *codificação*; (4) *testes* e (5) *manutenção e operação*.

A identificação de cada uma das fases é fundamental para que se possa, por um lado, desenvolver o "software" com objetivos parciais bem definidos, e por outro, avaliar, em diversos passos, os resultados obtidos.

A Figura 2.1 mostra esquematicamente o ciclo de vida do desenvolvimento de "software". Apresentam-se, a seguir, as linhas gerais que orientam cada uma das fases desse ciclo.

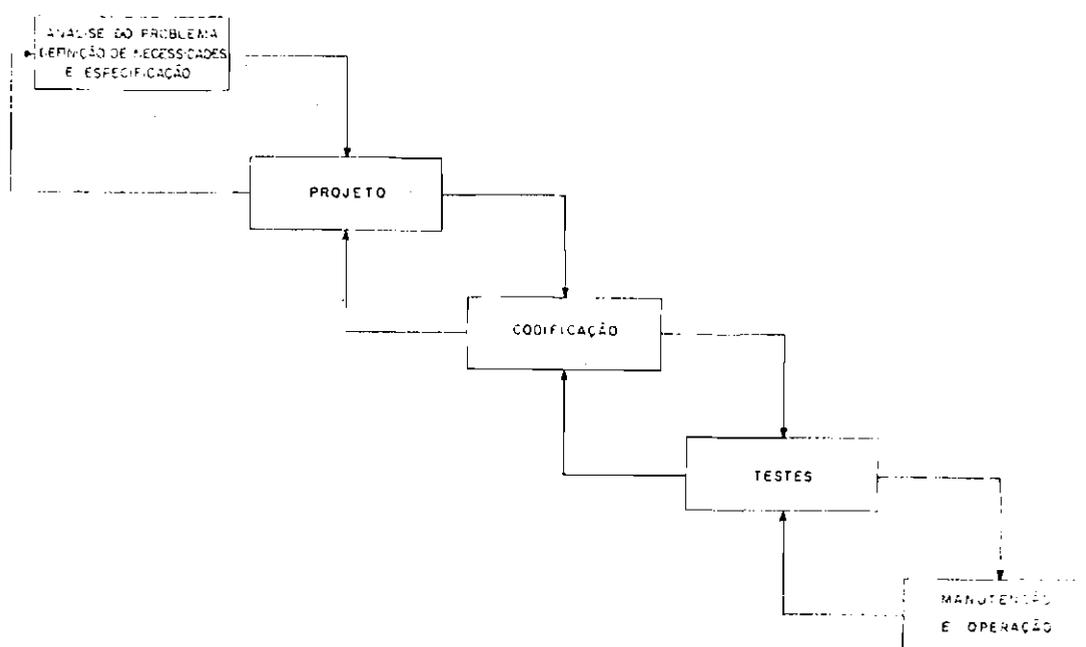


Fig. 2.1 - Ciclo de vida do desenvolvimento de "software".

### 2.2.1 - FASE 1: ANÁLISE DO PROBLEMA, DEFINIÇÃO DE NECESSIDADES E ESPECIFICAÇÃO

A primeira fase se inicia com a análise do problema a ser resolvido, visando esclarecer a sua natureza e a sua complexidade, e identificar os seus aspectos mais importantes. Essa análise não abrange apenas o "software" mas o sistema como um todo, pois leva à identificação do leque de alternativas para o projeto. Os primeiros estudos de viabilidade do sistema são realizados, juntamente com suas primeiras estimativas de custos.

Mesmo nos casos em que o computador e a linguagem a serem utilizados já estejam definidos a priori, a análise é realizada inicialmente abstraindo possíveis limitações e condicionamentos, visando identificar as características lógicas e as dificuldades técnicas ou administrativas inerentes ao problema a ser resolvido.

Essa fase se completa com a definição de necessidades do sistema (*o que deve ser feito*) e a especificação funcional do "software" (a estratégia a ser adotada). São duas atividades diferentes, mas estreitamente relacionadas entre si. Na definição de necessidades realiza-se um levantamento das características requeridas para o sistema, do ponto de vista do usuário final, sejam físicas, funcionais ou de desempenho.

Na definição de necessidades, o computador e a linguagem de programação ainda não são escolhidos, mas várias de suas características já são identificadas, o que restringe o universo de escolha e cria as condições necessárias para uma definição final adequada.

Os atributos essenciais do documento que contém a definição de necessidades são (Boehm, 1976): (1) *ser completo* em relação aos problemas e necessidades do usuário, particularmente sob o enfoque da interação homem-máquina; (2) *ser consistente* e (3) *não ser ambíguo*, evitando termos tais como "suficiente", "flexível", e apresentando as necessidades de maneira precisa.

Esse documento é elaborado freqüentemente em linguagem livre, não havendo nenhuma padronização de linguagem de descrição de necessidades. Entretanto é importante que essas necessidades sejam definidas de forma metódica, sobretudo porque, caso contrário, quaisquer correções a serem feitas em fases posteriores tornam-se mais difíceis e caras. Alguns sistemas foram desenvolvidos para orientar e facilitar a definição de necessidades, com metodologia "top-down". Os mais significativos são: (1) ISDOS (Teichroew, 1971), o primeiro sistema desenvolvido para tal finalidade, utilizando uma linguagem de proposição de problemas (PSA); (2) SADT (Ross, 1977), com uma metodologia para definição e análise de necessidades (incluindo uma linguagem gráfica auxiliar) e considerações sobre gerenciamento do "software" e (3) SREM (Alford, 1980), desenvolvido para gerar e validar necessidades de "software", para sistemas de defesa baseados em mísseis balísticos.

Após a definição das necessidades do sistema, o documento de especificação funcional do "software" é o próximo a ser elaborado nesta primeira fase do ciclo de vida do desenvolvimento de "software". Definidas as necessidades do usuário, é possível definir como cada uma delas será realizada pelo sistema. Com base em critérios técnicos e econômicos procede-se às escolhas de linguagem, sistema operacional, algoritmos, equações e funções de alto nível, e linha de evolução a ser seguida a partir do início do projeto propriamente dito. Algumas dessas escolhas podem ser preliminares, constituindo-se de uma apresentação das características escolhidas, em coerência com a estratégia de projeto adotada (no caso de linguagem de programação, por exemplo, transparente para o desenvolvimento da segunda fase, basta uma definição das suas potencialidades em termos de estruturação dos dados e programas, clareza e facilidades para documentação). As limitações dos sistemas são apresentadas em detalhes na especificação funcional.

### 2.2.2 - FASE 2: PROJETO

O projeto de "software" tem por base a sua especificação, que permite efetuar sua completa estruturação, de forma clara, não-ambígua e facilmente codificável através de linguagem de programação. O seu

desenvolvimento metódico foi, desde o início, uma das grandes propostas da Engenharia de Software, e ainda hoje ocupa lugar de destaque no ciclo de vida, durante a fase de desenvolvimento, apesar dos esforços em diminuir a sobrecarga desta fase. O resultado é um grande enriquecimento, em clareza e detalhes, da especificação funcional do "software". A maior dificuldade ainda é substituir o caráter artesanal e individual, que os projetos apresentam, por metodologias que explorem linguagens de representação padronizadas, bem como ferramentas auxiliares que ao mesmo tempo facilitem e disciplinem a aplicação de tais metodologias. Mais do que uma dificuldade técnica, esse problema ocorre sobretudo pela relutância do elemento humano à assimilação dos métodos sugeridos e, num sentido mais amplo, pela estrutura ineficiente de trabalho que ainda predomina entre os grupos de desenvolvimento de "software". Por isso, a fase de projeto apresenta-se, ainda, como uma importante fonte de inclusão de erros no "software".

Costuma-se dividir esta fase em duas: *projeto preliminar* e *projeto detalhado*. Na primeira define-se a estrutura do "software" (seus módulos, interconexões e estruturas de dados) e na segunda trabalha-se no detalhamento da estrutura lógica dos módulos, através de procedimentos que a realizem.

A partir dos estudos de Wirth (1971), a técnica de desenvolvimento de projeto chamada "*top down*" tem se firmado como a mais eficiente e a que traz os melhores resultados para o "software". Consiste em um processo de decomposição funcional com refinamento em passos sucessivos, quebrando-se em cada passo, os módulos em outros menores a eles subordinados. Todo passo de refinamento implica algumas decisões de projeto. Paralelamente ao refinamento das funções deve-se também realizar, sempre que possível, o refinamento das estruturas de dados (as decisões relativas a esses refinamentos são retardadas ao máximo, visando o maior confinamento possível dos dados e, conseqüentemente, maior independência entre os módulos). Tem-se, assim, um procedimento para organizar e desenvolver a estrutura de controle do projeto, dando prioridade às definições relativas a pontos críticos de integração e interface entre os módulos.

As vantagens dessa técnica podem ser resumidas em duas (Wilkes, 1976; Jensen, 1981): (1) o projeto, sendo naturalmente modular, torna-se fácil de compreender e verificar; (2) é fácil de manter e alterar.

Foram desenvolvidas várias técnicas para representação de projetos de "software". Destacam-se as mais importantes: (1) pseudocódigo (Caine, 1975; Fairley, 1976), uma forma de utilização de vocabulário de uma linguagem escrita (por exemplo, inglês) e sintaxe de outra (no caso, uma linguagem de programação estruturada, com as estruturas IF, DO WHILE, etc.); (2) HIPO (Hierarchy Plus Input Process Output) (Stay, 1976), que contém um diagrama hierárquico com as definições das funções e módulos, e diagramas denominados IPO, com apresentação, em diversos níveis, de listas de variáveis de entrada, funções e variáveis de saída; (3) diagrama estruturado (Nassi, 1973), com utilização de uma estrutura gráfica rígida, visando representação modular e estruturada.

Algumas ferramentas, associadas à fase de projeto, auxiliam na geração de documentos. A validação da documentação e dos resultados da fase de projeto permite que se inicie a codificação dos módulos.

### 2.2.3 - FASE 3: CODIFICAÇÃO

A codificação é a última etapa de processo de criação inicial do "software" e desenvolve-se na linha definida pela fase de projeto, mantendo a estrutura dos programas e dos dados estabelecida naquela fase. A escolha da linguagem de programação a ser utilizada deve considerar, em primeiro lugar, a necessidade de implementar aquelas estruturas, mas deve obedecer também a outros critérios importantes para a obtenção dos diversos objetivos de um programa, tais como: (1) possuir facilidades para implementação de programas estruturados, com estruturas para isolamento de blocos e para controle, próprios para isso; (2) possuir recursos para a representação estruturada dos dados; (3) possuir compiladores em diversas máquinas. A escolha de linguagem de alto nível é geralmente a mais adequada, pois satisfaz às exigências citadas

e permite a representação dos programas de forma compreensível, facilitando sua manutenção.

O grupo de codificação deve manter as mesmas preocupações da fase de projeto no desenvolvimento de seu trabalho, sobretudo no que se refere à compreensibilidade e alterabilidade, que podem ser obtidas mesmo fazendo uso de linguagens não-estruturadas e de baixo nível, já que a estrutura é imposta pelo programador e os comentários podem ser inseridos por ele, nos diversos pontos do programa. Deve-se evitar, por outro lado, artifícios de programação e expressões lógicas ou aritméticas complexas, substituindo-os, sempre que possível, por formas mais simples e claras de representação.

#### 2.2.4 - FASE 4: TESTES

A validação e a verificação de correção do "software" são realizadas freqüentemente através dos testes que constituem uma das mais longas fases do ciclo de vida do "software". Os conceitos, técnicas e ferramentas relativos a essa fase são apresentados em detalhes na Seção 2.3.

#### 2.2.5 - FASE 5: MANUTENÇÃO E OPERAÇÃO

Depois de desenvolvido e testado, o "software" pode ainda ser modificado, por três motivos: (1) são encontrados erros não detectados anteriormente; (2) adaptação do "software" a novo ambiente e (3) alteração na especificação.

Em qualquer caso, a eficácia das alterações será tanto maior quanto mais modulares, estruturados, claros e bem documentados forem o projeto e o programa codificado. Essas alterações não devem visar apenas o código final mas o projeto como um todo. Uma análise detalhada leva ao(s) ponto(s) de mais alto nível a partir do qual devem ser feitas alterações. O projeto e a codificação são efetuados seqüencialmente e os testes correspondentes aos módulos envolvidos são realizados

novamente. A documentação é alterada, devendo retratar o novo estado do "software" em seus diversos níveis.

## 2.3 - VERIFICAÇÃO, VALIDAÇÃO E TESTE DO "SOFTWARE"

### 2.3.1 - ALGUMAS DEFINIÇÕES

Desde que certos termos comuns utilizados ao longo deste texto aparecem na literatura com definições e usos diversos, é necessário defini-los para o âmbito do presente trabalho (Branstad, 1980):

- *Verificação*: em geral, a demonstração, para cada fase do ciclo de vida, de que o "software" é consistente, completo e correto. As análises estática e dinâmica são usadas durante a verificação.
- *Validação*: determinação da correção do programa ou "software" final produzido por projeto ou desenvolvimento, com relação às necessidades do usuário. É feita usualmente através de verificações em cada estágio do desenvolvimento do ciclo de vida do "software".
- *Certificação*: aceitação do "software" por pessoa autorizada, usualmente após o "software" ter sido validado por ela ou após sua validação ter sido demonstrada para tal pessoa.
- *Teste*: exame de comportamento de um programa através de sua execução com amostras de dados convenientemente escolhidas.
- *Prova de Correção*: uso de técnicas de lógica matemática para inferir se uma relação entre variáveis de um programa, considerada verdadeira na sua entrada, implica que uma outra relação entre variáveis do programa seja satisfeita na sua saída.
- *Depuração de Programas*: processo de correção de erros sintáticos e lógicos detectados durante os testes do programa.

### 2.3.2 - VERIFICAÇÃO DURANTE TODO O CICLO DE VIDA DO "SOFTWARE"

As diversas fases do desenvolvimento de "software" devem ser encerradas com a verificação de seus resultados para evitar que os problemas sejam identificados e resolvidos em fases posteriores, quando as correções se tornam mais difíceis e caras (Fujii, 1977).

Os documentos de definição de necessidades e de especificação devem ser verificados, analisando (Boehm, 1984): (1) *completeza* (ausência de decisões "a serem determinadas", de referências inexistentes etc.); (2) *consistência* (interna e externa) e (3) *praticabilidade* (aspectos humanos, recursos, testabilidade etc.),

A verificação da fase de projeto deve assegurar que o projeto está correto e satisfaz a especificação definida, eliminando erros para que não se propaguem durante a fase de codificação. É feita analisando os mesmos parâmetros citados, com atenção às estruturas de dados, funções e estrutura hierárquica dos módulos.

Na fase de codificação a verificação tem por objetivo assegurar a correção da implementação do projeto. Envolve a análise dos mesmos parâmetros citados para a definição de necessidades, voltando-se novamente para estrutura de dados, funções e estrutura hierárquica dos módulos. Esta fase inicia-se com análises estáticas do código gerado, verificando a estrutura lógica e dos dados, e desdobra-se através da fase de testes.

### 2.3.3 - A FASE DE TESTES

A fase de testes tem grande importância no ciclo de vida do "software", pois corresponde a uma análise completa e detalhada das características funcionais do programa, envolvendo tanto a verificação da correção quanto do desempenho. Por isso é grande o número de trabalhos, dirigidos para esse assunto, que podem ser encontrados na literatura.

Apesar da impossibilidade prática de testar o programa de forma exaustiva devido à excessiva quantidade de combinações de entradas e saídas (Huang, 1975), a eficácia da fase de testes tem progredido muito nos últimos anos com o desenvolvimento de métodos e técnicas mais precisos baseados em teoria sólida, além de ferramentas que facilitam e disciplinam o desenvolvimento dos testes (Miller Junior, 1981a). Os fundamentos teóricos dos testes de programas se encontram na análise do fluxo de controle e do fluxo de dados (baseados em teoria de grafos), nos estudos de linguagens de programação e nas contribuições da teoria de confiabilidade (Miller Junior, 1981b; Goodenough, 1975).

O princípio básico do teste é a execução controlada do programa, com entradas e saídas conhecidas. As entradas são submetidas ao programa e as saídas são examinadas. Com esse procedimento é possível descobrir a presença de erros no programa, para que sejam corrigidos. No entanto, se não forem encontrados erros no teste do programa, não se pode concluir pela inexistência de erros. As provas formais podem ser utilizadas para provar a correção (inexistência de erros) de programas (Basu, 1975; Birman, 1974; Hantler, 1976). Porém, a sua adoção para verificação de programas é normalmente inviável do ponto de vista prático, sobretudo devido à complexidade que essas provas apresentam. Para programas de grande porte essa complexidade é ainda maior. São formalmente provados programas de porte relativamente pequeno, sendo a prova de grandes programas praticamente inviável, atualmente.

A *depuração* do programa, ou seja, correção de seus erros, é o objetivo final dos testes, mas, para isso, é necessário antes detectar esses erros e diagnosticar as suas causas. Por isso, os testes estão sempre voltados para a descoberta de erros, em ambiente que permita também seu diagnóstico e depuração. O *diagnóstico* exige o levantamento das possíveis causas do erro, a partir da comparação dos resultados esperados. A partir deste levantamento parte-se para a identificação precisa da causa, possivelmente através de novos testes, voltados para esse fim. A depuração é feita em decorrência do diagnóstico, consistindo na correção do erro, partindo-se do nível mais alto do desenvolvimento

do programa até chegar à codificação do novo programa. A solução adotada, que não é única, em geral, deve levar em consideração as influências que trará às diversas funções do programa, particularmente àquelas que já foram validadas e que deverão ser submetidas a novos testes.

A fase de testes deve ser planejada de modo tal que a avaliação do programa seja feita em diversos níveis hierárquicos. São dois os tipos de testes que podem ser realizados (Howden, 1981): (1) *testes funcionais* e (2) *testes estruturais*.

Nos *testes funcionais* (Howden, 1980), também chamados testes de caixa preta, são desconsiderados os detalhes internos do programa ou módulo em teste. É realizado pela aplicação de entradas corretas e a análise das saídas geradas. Sua maior vantagem é a de permitir que os testes não sejam subordinados à estrutura interna do programa. Considerando a impossibilidade de realizar os testes para todas as combinações de entradas e saídas, os estudiosos tem desenvolvido critérios e ferramentas para definir, para cada programa, classes de dados de entrada e de saída que cubram os diversos casos a serem testados. Essa definição constitui-se numa das maiores dificuldades técnicas na realização dos testes funcionais. As classes de dados devem ser definidas com base na especificação da estrutura de dados de cada função em teste, abrangendo as diversas classes de dados de entrada e de saída, além de casos extremos e de casos especiais (Foster, 1980; Gabow, 1976; Goodenough, 1977; Howden, 1975).

Nos *testes estruturais*, também chamados testes de caixa branca, os dados são gerados com base na estrutura do programa ou do módulo em teste. Sua maior vantagem é a de levar a uma análise mais detalhada e segura da lógica adotada, podendo mesmo ser testado, com entradas adequadas, cada um dos ramos do fluxo de controle do programa ou módulo (o que garante também o teste de cada um dos comandos). Esse nível de detalhamento, no entanto, pode não ser viável em termos práticos, dada a grande quantidade de casos a serem verificados. A combinação de testes funcionais e estruturais é aplicada com frequência. A programação é feita de forma a abranger as funções especificadas, com a inclusão de testes detalhados de ramos ou caminhos importantes do programa.

Os testes devem ter um planejamento preciso, embora adaptável às necessidades e dificuldades de cada momento. Esse planejamento, que tem por objetivo a validação do programa como um todo, deve prever: (1) a seqüência de testes a ser adotada; (2) as equipes que irão trabalhar e (3) o ambiente de testes a ser utilizado. São feitas a seguir algumas considerações a respeito desses tópicos.

Quanto à seqüência de testes a ser adotada, pode-se identificar duas etapas distintas:

- *testes individuais dos módulos*, em que é verificado o funcionamento dinâmico de cada módulo isoladamente, observando sua interface, o comportamento da estrutura de dados e da estrutura lógica;
- *testes integrados*, em que se acoplam módulos já testados individualmente, para verificar o funcionamento dinâmico do conjunto.

O isolamento de um módulo para teste exige a criação de módulos de controle artificiais simplificados e voltados apenas para simular as interfaces do módulo em teste com os demais módulos. Esta prática, além de exaustiva, pode causar problemas adicionais, tais como a criação de módulos artificiais com erros. Por isso, é comum a tentativa de atenuar essa dificuldade programando-se os testes individuais de forma a utilizar módulos já testados durante a aplicação de testes a outro módulo.

Existem duas metodologias usuais para os testes integrados: "*top-down*" e "*bottom-up*". Na primeira, segue-se a linha de desenvolvimento do projeto, testando antes os módulos de mais alto nível. Na segunda, que pode ser utilizada mesmo para projetos desenvolvidos com metodologia "*top-down*", os módulos de níveis inferiores são testados antes. A maior vantagem da metodologia "*top-down*" para testes é a de dar prioridade à análise das interfaces entre os módulos, que podem ser feitas antes mesmo de os módulos subordinados terem sido codificados. A metodologia "*bottom-up*" apresenta como maior vantagem a possibilidade de

utilização de módulos já testados para a realização dos testes seguintes. A combinação das duas metodologias também pode ser utilizada.

A *equipe de testes*, que tem a função de executar os testes do programa, pode ser composta por pessoas que não pertençam às equipes de projeto e codificação. Essa prática traz benefícios para o desenvolvimento como um todo, pois garante a elaboração dos testes, independentemente dos erros sistemáticos e vícios que possam ocorrer nas fases de concepção. Havendo formas precisas de avaliação já definidas desde o início, cada grupo realiza o seu papel em clima de colaboração. Os erros detectados são discutidos e corrigidos com a participação das diversas equipes. Durante a realização dos testes, a equipe de testes pode ser dividida para realização das atividades programadas de algumas etapas.

O *ambiente de testes* é um dos pontos de maior peso na eficácia da fase de testes. É constituído por recursos físicos (computadores, periféricos etc) e recursos lógicos (programas auxiliares, simuladores, sistema operacional, arquivos de dados, ferramentas etc) utilizados nos testes.

O sistema operacional utilizado nos testes pode ser diferente daquele que será utilizado no produto final, mantendo a estrutura do núcleo e incluindo recursos para facilitar a detecção de erros e a depuração do programa, tais como verificações dinâmicas de consistência dos dados e parâmetros formais, envio de resultados parciais para arquivos de saída, etc. Deve-se levar em consideração que esses recursos adicionais tornam maior o tempo de execução do programa, no entanto, facilitam testes para os quais o fator tempo não é relevante (até mesmo os testes de programas para aplicação em tempo real incluem verificações funcionais ou estruturais nas quais a variação do tempo de execução não influi no resultado).

Muitas *ferramentas* foram desenvolvidas para auxiliar os testes, sobretudo no período 1972 - 1975. No entanto, poucas se firmaram no suporte adequado à manutenção, embora tenham servido, em geral,

como valiosos instrumentos de pesquisa (Miller Junior, 1981a). As reais necessidades em termos de técnicas e ferramentas também se delinearam nesse período. Os trabalhos mais significativos nesta área são os seguintes: (1) sistema de "pontas de prova" ("probes") nos programas fonte por inserção direta de comandos ou por chamadas a sub-rotinas, como PET (Stucki, 1977). Esses comandos e sub-rotinas adicionais realizam registros (em arquivos de saída de informações sobre fluxo de controle e o de dados durante a execução do programa modificado); (2) sistemas que interpretam asserções inseridas em comentários do programa e que fazem testes de consistência dinâmica dos dados, de acordo com as restrições impostas pelas asserções (Gerhart, 1976; Stucki, 1977); (3) sistemas para obtenção de valores através de execução simbólica dos programas (ou seja, desenvolvendo as expressões algébricas e lógicas), como DISSECT (Howden, 1977); (4) sistemas para realização de análises estáticas, como DAVE (Osterweil, 1976, 1983) e (5) sistemas para geração automática de casos de teste (Bird, 1983; Miller Junior, 1975; Ramammorthy, 1975).

As ferramentas existentes e disponíveis para uso apresentam muitas vezes dificuldades para adaptação ao ambiente de testes e à linguagem de programação em uso. Por isso, é comum o desenvolvimento de ferramentas voltadas para a aplicação específica de interesse.

A documentação dos testes é constituída por dois grupos de documentos, a saber:

- 1) Programação dos testes, com a definição das etapas a serem vencidas e o cronograma de testes a serem realizados. Elaborados com base na documentação, visam abranger as funções do programa através dos casos de testes. Entre esses documentos estão os planos de testes, constituídos por folhas com a descrição dos recursos e procedimentos necessários à realização dos testes.

2) Resultados dos testes, com a apresentação, para cada teste realizado, dos resultados obtidos, seguidos de observações necessárias à sua interpretação (tais como as condições em que foi realizado), comentários das equipes de testes, projeto e codificação, e conclusões gerais sobre o teste. Os comentários da equipe de testes constituem uma crítica dos resultados obtidos, em comparação com os resultados esperados. Ressaltam os pontos de contraste em termos qualitativos e quantitativos. No caso de haver erros, os comentários das equipes de projeto e codificação constituem apresentação do diagnóstico de sua causa e da solução para corrigi-lo (não sendo possível diagnosticar com base nas informações existentes, pode-se solicitar a realização de novos testes). As conclusões gerais, apresentadas pela equipe de testes, envolvem uma análise dos comentários e ainda decisões sobre os próximos passos a serem dados.

## CAPÍTULO 3

### TESTES DE "SOFTWARE" PARA SISTEMAS DE ARQUITETURA DISTRIBUÍDA

Os testes de "software" concorrente e distribuído possuem, em essência, o mesmo objetivo dos testes de programas seqüenciais: analisar o comportamento do programa codificado em relação a diversas entradas, visando a sua depuração através da detecção, diagnóstico e correção dos erros.

Os testes estão compreendidos no ciclo de vida do programa, do mesmo modo que ocorre no caso seqüencial, na Fase de Testes, cujas diretrizes estão apresentadas na Seção 2.3. Diferem, no entanto, com respeito às técnicas e ferramentas utilizadas, devido a uma série de características, inerentes aos sistemas distribuídos.

São apresentadas, nesse capítulo, as linhas gerais que devem nortear os testes de "software" de sistemas distribuídos, com atenção especial à concepção do ambiente de testes, das ferramentas a serem utilizadas e da metodologia a ser empregada. Partindo da identificação das dificuldades que esses testes apresentam, busca-se nas contribuições da literatura e na prática a definição das diretrizes para a realização de testes.

Os cuidados necessários para realizar o desenvolvimento do "software" visando os testes são também apresentados.

#### 3.1. - AS DIFICULDADES DE TESTAR "SOFTWARE" CONCORRENTE E DISTRIBUÍDO

Os testes e a depuração de "software" para máquinas de arquitetura distribuída apresentam dificuldades adicionais em relação aos de sistemas de estrutura lógica seqüencial. Essas dificuldades se devem tanto à característica do processamento quanto à arquitetura distribuída. A concorrência não só dificulta a detecção de muitos tipos de erros que também ocorrem em programas seqüenciais, como introduz a possibilidade de ocorrência de novas classes de erros (Taylor, 1980). Entre os

erros que podem ocorrer em ambos os casos pode ser citada a atribuição, a uma variável, de valor que esteja fora dos limites toleráveis. No caso seqüencial, é tarefa relativamente simples determinar a causa do erro e o ponto do programa no qual ocorre. Caso se tenha, no entanto, um programa concorrente, no qual aquela variável é compartilhada entre dois ou mais processos, a dificuldade para determinar a causa do erro é muito maior, pois nem sempre é possível indicar imediatamente qual foi o processo que determinou o erro e em que estado de tal processo tal erro ocorreu.

Entre os erros que ocorrem apenas em sistemas concorrentes tem-se os de sincronização entre processos. Tais erros podem causar "deadlock" ou resultados incorretos. É também comum encontrar erros nas trocas de informações entre processos.

São apresentadas a seguir as principais diferenças que trazem dificuldades adicionais aos testes de "software" para máquinas de arquitetura distribuída. Para facilidade de apresentação, estão divididas em dois grupos: (1) diferenças devidas à existência de múltiplos processos assíncronos e (2) diferenças devidas à existência de múltiplos processadores.

### 3.1.1 - MÚLTIPLOS PROCESSOS ASSÍNCRONOS

A concorrência entre processos (utilizando um ou mais processadores) traz diferenças no teste e depuração dos programas, em relação aos programas seqüenciais. Pode-se reuni-las em quatro grupos, a saber:

- 1) Existência de vários locais de controle. O fato de o programa ter vários locais de controle (cada processo tem o seu) torna-o de mais difícil compreensão e, portanto, mais vulnerável a erros. A compreensibilidade é afetada por não se poder absorver a lógica do programa pelo simples acompanhamento seqüencial de um processo, como se faria no caso de um programa seqüencial. O entendimento da dinâmica de trocas de informações entre os

processos é necessário para a análise do fluxo de controle de cada um. A alta suscetibilidade a erros ocorre pela necessidade de sincronização entre os processos que trocam informações ou compartilham variáveis. Os erros são muitas vezes sutis e esporádicos pois podem ocorrer em situações muito especiais a serem causados por sincronização imprópria dos processos ou por condições críticas de velocidade.

- 2) Dificuldade em reproduzir resultados. Em muitos casos os resultados observados no sistema são de difícil reprodução, pois não dependem apenas das suas entradas, mas também da evolução relativa do estado dos processos no tempo. A reprodução do erro exige, muitas vezes, além da aplicação das mesmas entradas, o conhecimento do estado instantâneo de todos os processos envolvidos, já que estes estados, combinados com os resultados da alocação dos recursos, por ocasião da competição entre os processos, vão determinar condições de temporização que afetarão os resultados. Os estados, no entanto, são dificilmente reconstituíveis. Às vezes é possível deduzir tais estados através da análise do conteúdo das estruturas de dados, observado na memória, após a ocorrência do evento caso os testes sejam realizados com monitoração ou se houver uma parada do processamento. No entanto, se o processamento continuar após a ocorrência do erro, pode-se tornar muito difícil, ou mesmo impossível, reconstituir aqueles estados.
- 3) Existência de variáveis compartilhadas. Os processos que utilizam áreas de dados comuns tornam-se interdependentes e sua execução concorrente pode trazer influências mútuas não previstas, de difícil diagnóstico e controle durante a aplicação dos testes. Essa situação ocorre até mesmo em casos em que são disponíveis mecanismos de controle do acesso à memória, os quais, se utilizados de forma incorreta, poderão não evitar erros tais como o acesso a uma mesma variável por dois ou mais processos simultaneamente. O uso excessivo e abusivo de variáveis globais traz dificuldades para os testes de quaisquer programas, sejam

seqüenciais ou paralelos, mas no caso desses últimos as dificuldades são, em geral, ainda maiores, porque tais variáveis são compartilhadas entre processos concorrentes.

- 4) Influência no tempo de execução. Em programas paralelos, o tempo de execução é uma variável importante. As variações no tempo de execução de um processo (que podem ocorrer, em vista da disputa pelos recursos) trazem arranjos diferentes de eventos, com efeitos globais variados nos estados dos processos e nas saídas. A utilização de mecanismos de sincronização entre os processos para a execução de certas funções é necessária, apesar das possíveis perdas que possam trazer, em matéria de eficiência.

### 3.1.2 - MÚLTIPLOS PROCESSADORES

Se os processos são executados em diferentes processadores físicos, algumas diferenças e dificuldades adicionais se apresentam para a realização dos seus testes. Essas dificuldades referem-se a:

- 1) Informação crítica. Havendo múltiplos processadores é difícil detectar a causa (ou seja, a informação crítica, causadora) de um possível erro, pois os processadores continuam em atividade, executando os processos, mesmo depois da ocorrência (excetuando-se, em muitos casos, o próprio processador onde houve o erro). Desta forma não é possível descobrir como o erro foi gerado, pois o próprio processador que gerou a informação crítica já tem o seu estado modificado em seguida. Não havendo recursos no sistema operacional para forçar uma parada de todos os processadores no instante da ocorrência de um erro detectável ou de uma parada de um processador, essa situação se torna frequente durante a aplicação dos testes, dificultando muito a depuração dos programas.
- 2) Falhas de processadores. Se o programa completo for executado em um único processador, as falhas de "hardware" (ou mesmo do sistema operacional) são percebidas rapidamente pelo programador. Com processadores múltiplos, no entanto, não existe a in

formação imediata de que a causa de um erro foi a falha de um processador. Como ocorre no diagnóstico de qualquer erro, abre-se um leque de possíveis causas e incia-se, então um processo, às vezes demorado, de análise dos estados dos processadores envolvidos, até se concluir pela verdadeira causa. Considere-se ainda que a probabilidade de ocorrer falha em algum processador cresce com o número de processadores do sistema distribuído.

- 3) Tempo de comunicação entre processadores. Se os processadores do sistema distribuído estão geograficamente distantes, ou se a aplicação exigir respostas em tempos curtos, o tempo de transmissão pode se constituir numa limitação à utilização de recursos para depuração que envolvam um aumento da carga de informações trocadas entre os processadores, sobretudo se a taxa de transmissão for baixa. Pode não ser viável, por exemplo, a inclusão de "traces" cujos dados devam ser transmitidos de um local para outro, através da mesma via utilizada na transmissão dos dados do processamento normal.
- 4) Porte do sistema. Sistemas distribuídos são geralmente de porte considerável uma vez que normalmente destinam-se à execução de diversas tarefas paralelas, exigindo, em geral, um grande número de processos e processadores em sua implementação. Como acontece no caso de programas sequenciais, também no caso distribuído as dificuldades para testar e depurar aumentam com o crescimento do programa. Considerando que essas tarefas são mais trabalhosas e demoradas no caso distribuído, o fato de se ter um "software" de grande porte contribui para tornar ainda mais crítica a tarefa de testar tal sistema.
- 5) Controle e monitoração da execução. Tanto em testes de programas sequenciais quanto no de programas paralelos, uma das técnicas utilizadas é a de monitorar a execução através do uso de uma interface, instalada entre o programa em teste e o programador. Por intermédio de tal ferramenta, o programador pode influir no processamento e ainda observar detalhes da execução. Nos testes que envolvem processos que são executados em processadores distin

tos não é possível, em geral, controlar sua execução de maneira síncrona através de ferramentas simples como, por exemplo, programas monitores cuja ação está restrita ao processador em que são executados. São necessárias ferramentas com mecanismos voltados para este tipo de atividade, que nem sempre são disponíveis e que, em muitos casos, não podem ser implantadas, devido a limitações do Sistema Operacional que gerencia as atividades dos processadores envolvidos.

### 3.2 - RESENHA DAS CONTRIBUIÇÕES DA LITERATURA

Em relação ao caso seqüencial, a literatura técnica é relativamente carente de trabalhos específicos voltados para a área de testes de "software" de sistemas concorrentes e distribuídos, sobretudo quanto à apresentação de técnicas, métodos e ferramentas próprias para o desenvolvimento de testes para tais sistemas: apesar de as contribuições existentes na área de testes serem abundantes e de boa qualidade, apresentando as bases teóricas e práticas para o seu desenvolvimento, tal literatura é polarizada, em geral, para aplicações aos sistemas seqüenciais. Fornecem as linhas gerais que norteiam a programação e a execução dos testes, apresentadas no Capítulo 2, mas não detalham a sua aplicação às necessidades próprias dos sistemas concorrentes e distribuídos. Os trabalhos com contribuições práticas específicas limitam-se, em geral, aos sistemas seqüenciais.

São relacionados aqui alguns trabalhos, de cunho teórico ou prático, que tratam das verificações de "software" concorrente e distribuído nas diversas fases do seu ciclo de vida. Entre os estudos teóricos, destacam-se modelamentos de sistemas e de efeitos de falhas bem como alguns trabalhos sobre prevenção, predição e verificação de existência de "deadlock". Entre os trabalhos práticos encontram-se propostas para a análise de desempenho, a análise de fluxo de dados, a análise funcional e a detecção e correção de "deadlock".

Os trabalhos teóricos baseiam-se em modelamentos dos sistemas e propõem técnicas para sua verificação e análise formal.

Kane (1975) utiliza um modelo de sistemas de "software" para estudar os efeitos de falhas de controle (falhas de "software" representadas por seqüências de execução incorretas). Apresenta uma teoria de detecção dessas falhas, identificando inicialmente as propriedades estruturais do programa que permitem tal detecção e propondo em seguida um algoritmo para realizá-la através de passos seqüenciais, durante as diversas fases do desenvolvimento do "software". Os mecanismos de detecção propostos devem ser implementados durante a fase de projeto do programa. Uma verificação empírica dos métodos formais apresentados é feita através da simulação de um sistema com multiprocessamento (três UCPS). O trabalho de Kane garante a viabilidade de eliminação de falhas de controle, desde que sejam respeitados certos critérios metodológicos na elaboração do programa.

Um outro estudo teórico é feito por Keller (1976) que propõe uma estrutura para verificação formal de programas paralelos, a partir de dois modelos formais para computação paralela: um modelo conceitual (suficientemente abstrato para "incluir quase todo modelo de programa paralelo existente") e um modelo específico de programa paralelo. Com o primeiro modelo, que vê um sistema como uma relação binária abstrata, Keller formula os conceitos de correção (correctness) de um programa e estabelece um princípio de indução ("um princípio chave para provar a correção"), base para toda verificação formal. Esse modelo contriui para a unificação de modelos de programas paralelos e das técnicas para sua verificação. Com o segundo modelo, que provê um meio de representar programas paralelos através de grafos, com estados e transições, realiza uma discussão relativa à prova de correção de sistemas em que um número arbitrário de processos executa o mesmo programa reentrante. Estabelece as condições para um programa paralelo ser considerado correto, entre as quais a necessidade de sempre terminar para certos valores iniciais e a ausência de "deadlock".

O modelo de programa proposto por Keller permite extensão para o caso de cada processo ter suas próprias variáveis locais e haver variáveis globais compartilhadas.

A verificação formal de sistemas distribuídos é estudada por Chen (1983) a partir de um modelo conceitual desses sistemas. O modelo proposto permite especificar formalmente o comportamento externo e a estrutura interna dos sistemas distribuídos. A verificação de correção do projeto pode ser feita, segundo Chen, provando a consistência entre a especificação formal e o projeto, antes da implementação.

Os sistemas com controle descentralizado de acesso à base de dados distribuída são o objeto de estudo de Voss (1980) que propõe o uso de redes predicado/transição ("uma extensão de primeira ordem das redes de Petri") para modelá-los e analisá-los. Mostra como essas redes podem ser construídas e interpretadas, e como os modelos podem ser analisados por métodos algébricos lineares.

Vários trabalhos teóricos são dedicados ao problema de "deadlock" (sua prevenção, predição e verificação de existência), sobretudo pelo interesse técnico desse tema no desenvolvimento de sistemas operacionais. Esses estudos têm aplicação natural a sistemas concorrentes e distribuídos de todos os tipos. Em relação à prevenção, um trabalho significativo é o de Habermann (1969), o primeiro a formular o problema de maneira precisa. Apresenta um modelo matemático para representar "deadlock" em sistemas operacionais e propõe algoritmos que examinam a possibilidade de ocorrência de "deadlock". Esses algoritmos foram usados com sucesso no sistema multiprogramado "THE" (Dijkstra, 1968).

Kameda (1980) apresenta um modelo para sistemas de computação baseado em teoria dos grafos e um algoritmo para testar a inexistência de "deadlock", demonstrando a validade desse algoritmo.

As contribuições para realização prática das verificações em sistemas distribuídos são encontradas sobretudo em literatura recente, notando-se, no entanto, uma carência de trabalhos com esse enfoque capazes de satisfazer às exigências que o desenvolvimento dos sistemas tem trazido.

Maples (1985) propõe critérios de análise e formas de aprimoramento do desempenho de sistemas com multiprocessamento e alto grau de paralelismo. Advoga que o desempenho é altamente dependente do grau de decomposição realizável para cada aplicação específica e da eficiência de utilização de uma arquitetura particular. Em relação à decomposição do código apresenta exemplo de implementação de alta eficiência, com uma fase de iniciações e outra de execuções paralelas. As estruturas "pipeline", com dois estágios, e mestre-escravo são examinadas. A análise de desempenho é feita em várias etapas, intercaladas com ajustes e correções. O alicerce do trabalho é o MIDAS (Modular Interactive Data Analysis System), operacional desde 1983.

A análise estática do fluxo de dados de programas concorrentes é utilizada por Taylor (1980) para verificar a estrutura de dados em programas que utilizam variáveis compartilhadas. Como já exposto na Seção 3.1 o gerenciamento da memória é uma das maiores dificuldades para o desenvolvimento e testes de programas concorrentes. O objetivo da análise é a detecção das seguintes classes de anomalias no fluxo de dados:

- 1) erro de iniciação (exemplo: variáveis não-iniciadas);
- 2) referência a variável enquanto está sendo alterada em processo paralelo;
- 3) definição de variável não-utilizada no programa;
- 4) referência a variável de valor não determinado.

Um ponto marcante do trabalho se refere à detecção de anomalias na sincronização dos processos, consideradas análogas, em sua forma, às anomalias de fluxo de dados. São apresentados algoritmos de detecção para os diversos casos estudados.

Os testes dinâmicos e a depuração dos programas de um sistema distribuído são pouco abordados, cabendo a Garcia-Molina (1984) um dos trabalhos mais detalhados nesta área. Baseando numa arquitetura básica do sistema, com multiprogramação e multiprocessamento, propõe um am

biente para realização dos testes, cujas ferramentas principais são programadas depuradores carregados em todos os nós (conjuntos que contêm processador, memória e interface de comunicação) acompanhados de um programa depurador principal, carregado em apenas um nó, para gerenciamento dos testes e interfaceamento com o programador. Propõe a metodologia "bottom-up" para realização dos testes, partindo de testes individuais dos processos para integrá-los em seguida.

Garcia - Molina analisa o papel dos "traces" para contornar a dificuldade em controlar a execução e repetir situações de erros já ocorridos. As ferramentas propostas para produzir e examinar "traces" dão recursos ao programador para encontrar os erros e ainda entender como e onde ocorreram. Rico em comentários carregados de experiência prática, o trabalho firma técnicas, conceitos e ferramentas para testes.

### 3.3 - DIRETRIZES PARA A REALIZAÇÃO DE TESTES

Os trabalhos da literatura, dedicados à verificação de sistemas concorrentes e distribuídos, em parte mencionados na Seção 3.2, podem servir de base para a escolha de diretrizes a adotar na realização dos testes desta classe de sistemas. As diferenças e dificuldades adicionais em relação aos sistemas sequenciais são enfrentados de maneiras diferentes pelos autores, de modo que um espectro de possíveis soluções, em seu conjunto, são, entretanto, insuficientes para determinar exatamente uma metodologia ideal para o desenvolvimento do "software" e para a realização dos testes. Isto se deve a dois motivos principais:

- 1) densidade de modelos e configurações apresentados para os sistemas distribuídos, geralmente concebidos tendo em vista a aplicação específica a que se destinam; e
- 2) carência de trabalhos voltados para a verificação dinâmica de sistemas distribuídos, já mencionada na Seção 3.1, que leva à dificuldade em analisar e comparar as técnicas, métodos e ferramentas utilizados pelos estudiosos do assunto.

São muitos, portanto, os caminhos que podem ser seguidos dentro de diretrizes gerais que a teoria e a prática sugerem no sentido de tornar mais eficiente a realização dos testes. São apresentadas aqui essas diretrizes, fruto de contribuições da literatura, aliadas a conclusões tiradas do trabalho prático de programação e realização dos testes.

As primeiras considerações são relativas à metodologia de desenvolvimento do "software", baseada em considerações que incluem, no próprio projeto, diretrizes para a obtenção de "software" facilmente testável, sem a qual podem tornar-se muitíssimo mais complexas e onerosas as atividades ligadas à fase de testes. A verificação do projeto nas diversas fases do seu desenvolvimento, com a exploração de técnicas de projeto que possam ser aplicadas com tal objetivo em mente, é feita com a finalidade de firmar a necessidade de analisar criteriosamente o "software" distribuído em seus diversos níveis, já nos seus primeiros estágios de desenvolvimento. Os testes propriamente ditos são também analisados, apresentando-se as diretrizes em relação ao ambiente necessário para a sua realização, às ferramentas disponíveis que podem ser utilizadas e à metodologia de testes a ser empregada.

### 3.3.1 - DESENVOLVIMENTO DE "SOFTWARE" VISANDO OS TESTES

A metodologia de desenvolvimento do "software" afeta a eficiência da fase de testes de sistemas distribuídos, tanto quanto acontece com sistemas seqüenciais.

São feitas aqui algumas considerações relativas ao desenvolvimento de "software" de sistemas distribuídos visando facilitar os testes.

### 3.3.1.1 - APLICAÇÃO DOS OBJETIVOS DA ENGENHARIA DE "SOFTWARE" A SISTEMAS DISTRIBUÍDOS

Os objetivos da Engenharia de "Software" aplicam-se, conforme foram apresentados na Seção 2.1, aos sistemas concorrentes e distribuídos, objetos deste estudo. Considera-se a que alterabilidade, a eficiência, a confiabilidade e a compreensibilidade são atributos obrigatórios dos programas, especialmente do ponto de vista do grupo de programadores encarregados de testá-los e depurá-los, sobretudo nos casos em que este grupo inclui pessoas que não participaram do projeto. Em relação a esses objetivos cabem algumas considerações específicas para o desenvolvimento do "software" de sistemas distribuídos em função das particularidades desses sistemas.

Como exposto na Seção 2.1, a alterabilidade de programas depende essencialmente da clareza e consistência de sua estrutura funcional. No caso de programas paralelos a clareza deve caracterizar desde a definição dos processos e de suas interconexões, realizadas na especificação funcional, até o detalhamento do projeto e da sua implementação. Está relacionada com o objetivo da compreensibilidade, analisado mais adiante.

Em relação à consistência da estrutura do "software", podem-se estabelecer alguns pontos básicos a serem considerados:

- 1) Os processos devem ter funções precisas, de forma que a alteração ou a eliminação de funções globais existentes, ou mesmo a criação de novas funções, sejam efetuadas necessariamente em harmonia com a arquitetura do programa, estabelecida a priori desde a sua fase do projeto. É desejável que nestes casos apenas alguns dos processos sejam alterados, exatamente aqueles cujas funções especificadas estejam relacionadas com as modificações desejadas. Para projetos bem feitos, este número deve ser sempre muito baixo. Se as funções dos processos não forem bem definidas desde sua criação, ou se perderem suas características originais após algumas modificações que não obedecem às diretrizes mencio

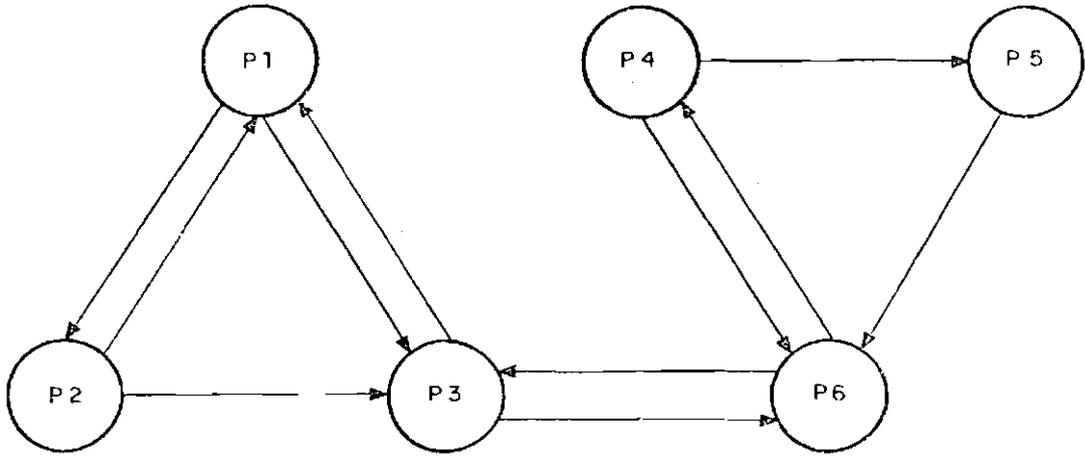
clusão e alteração de estruturas para a síntese de novas funções e até de novos processos que utilizem essas estruturas. As modificações devem evitar ao máximo mudanças estruturais profundas que muitas vezes podem ocorrer pela simples inclusão de variáveis globais, solução simples para resolver muitos problemas, mas que em geral introduz erros e altera a estruturação do "software" devido à conectividade que cria entre os processos, o que se reflete na dificuldade adicional que introduz na inclusão de novas alterações bem estruturadas. O projeto da estrutura de dados deve prever a possibilidades de alterações e inclusões de variáveis e de rotinas de acesso aos dados.

- 4) A comunicação entre processos, residentes ou não, em um mesmo módulo processador deve ser feita utilizando os mecanismos disponíveis de sincronização, devendo-se evitar projetos que se baseiam na hipótese de que certas seqüências de eventos nunca serão alteradas no programa. Além da falta de clareza que essas suposições introduzem no projeto, tem-se como consequência uma dificuldade maior de alteração do programa paralelo resultante, sem que tal alteração interfira na temporização prevista para os eventos na hipótese inicial.

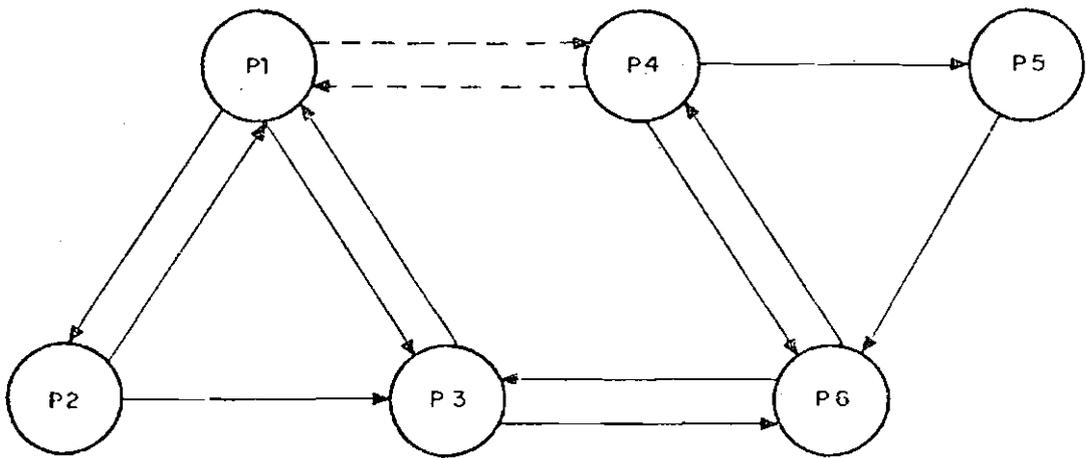
nadas, surgirão, em decorrência desta indisciplina, crescentes possibilidades de ocorrência de erros (sobretudo no que se refere à interconexão dos processos) reduzindo-se drasticamente as possibilidades de sucesso de eventuais alterações adicionais bem estruturadas. A Figura 3.1 apresenta um exemplo de modificação que, desrespeitando a estrutura original, afeta a clareza do programa, dificultando a implementação de novas modificações. O conjunto de processos P1, P2 e P3 originalmente se comunicava com o conjunto P4, P5 e P6 apenas através da ligação entre P3 e P6. Após a modificação, a conectividade entre os dois grupos passa a ser muito maior.

- 2) A hierarquia entre os processos deve ser estabelecida com precisão e respeitada durante a elaboração de todo o projeto, sob pena de desaparecer após algumas poucas alterações do programa. Cada modificação deve também respeitar essa hierarquia, o que só é possível se sua definição tiver sido feita prevendo maneiras de permitir adaptações e modificações, sem que a hierarquia original seja desrespeitada. A definição original da hierarquia entre os processos está vinculada à definição das funções desses processos. Por exemplo, processos dedicados a funções específicas de controle de entrada e saída através de periféricos podem estar subordinados a processos que gerenciam a utilização desses periféricos pelo sistema. Todo pedido de entrada e saída deve ser feito através desses gerenciadores. Por outro lado os processos subordinados só se comunicam com os demais processos através dos gerenciadores. Qualquer modificação posterior deve ser projetada respeitando esta estrutura. A Figura 3.2 apresenta um exemplo de modificação que desrespeita a hierarquia estabelecida anteriormente: na Figura 3.2b são introduzidas trocas de informações entre processos anteriormente subordinados a um outro e na Figura 3.2c são criadas trocas de informações entre P4 e um processo subordinado a P3.

- 3) A estrutura de dados de processos que utilizam áreas comuns de memória deve ser clara e flexível para permitir a inclusão, ex



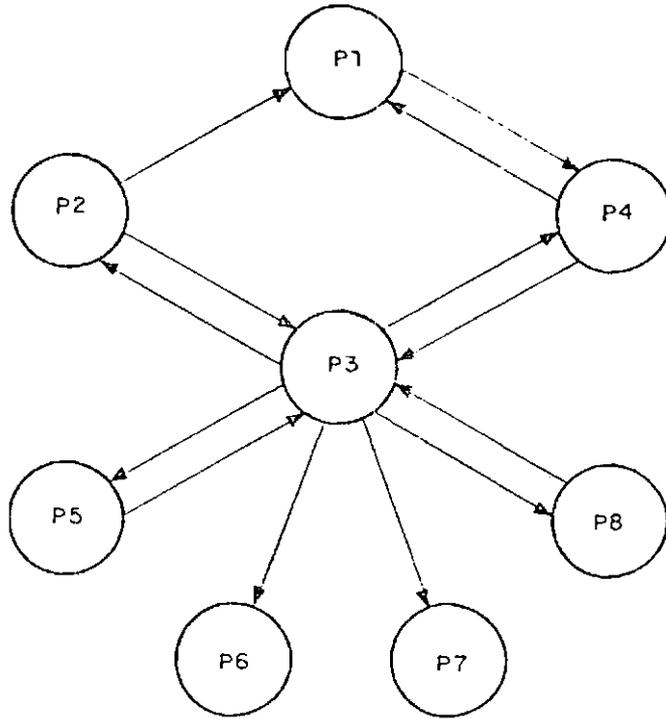
(a) estrutura original



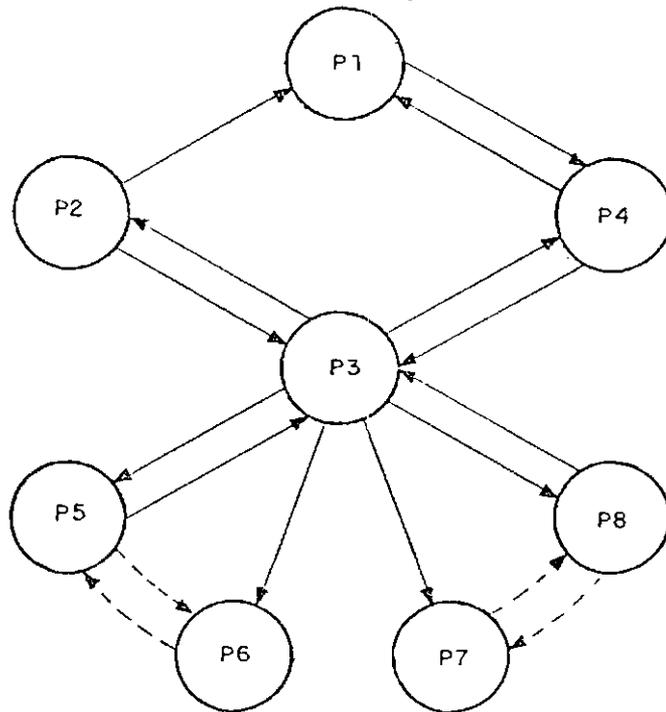
(b) estrutura modificada

Fig. 3.1 - Modificação de funções que desrespeita a estrutura original.

[inicial]



(a) estrutura original

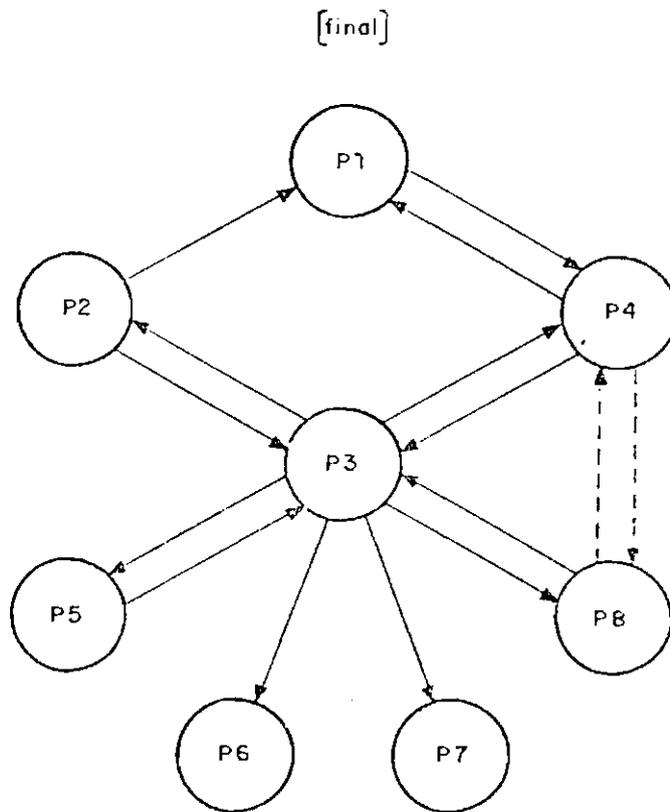


(b) inclusão de trocos de informações entre processo subordinados

Fig. 3.2 - Modificação de funções que desrespeita a hierarquia original.

(continua)

Figura 3.2 - CONCLUSÃO



(c) inclusão de trocas de informações entre um processo subordinado e um outro alheio à hierarquia

Geralmente a escolha de arquitetura distribuída para a solução de determinado problema está relacionada com requisitos relacionados com a eficiência do sistema, ou seja, com a utilização eficiente dos recursos disponíveis. No entanto, esse objetivo só é alcançado se o "software" utilizar adequadamente as características assíncronas de comunicação entre os processos, buscando-se fazer uso do paralelismo tanto quanto possível, sem ferir as regras de sincronização entre processos. Como diretriz de projeto, podem-se obter bons resultados nesse sentido impondo-se que cada um dos processos só bloqueie os demais processos com que se comunica durante um intervalo de tempo estritamente necessário para a execução segura das operações em que os processos estão envolvidos. Uma prática inadequada de programação, decorrente do hábito que apresentam os programadores de trabalhar em geral com programas seqüenciais, é o de liberar informações de um processo para os demais só depois de o processo em questão terminar de executar uma série de procedimentos (em geral não ligados às informações mencionadas), ao invés de liberá-las logo após estarem elas disponíveis. A Figura 3.3 apresenta um exemplo de trocas de informações entre processos em que se pode degradar a eficiência do programa caso este cuidado não seja observado. Seja P1 um processo que envia dados para serem tratados por P2, que por sua vez, verifica a consistência dos dados e realiza tratamentos adicionais, respondendo a P1 com uma mensagem indicativa do resultado do teste de consistência. Na Figura 3.3a está mostrada uma utilização pobre do paralelismo potencialmente disponível no sistema: P2 responde só depois de fazer todo o tratamento e P1 prepara novos dados só depois de receber a resposta. Em 3.3b há uma melhor utilização do paralelismo: P1 começa a preparar novos dados logo após enviar os primeiros e P2 responde logo após o final do teste de consistência.

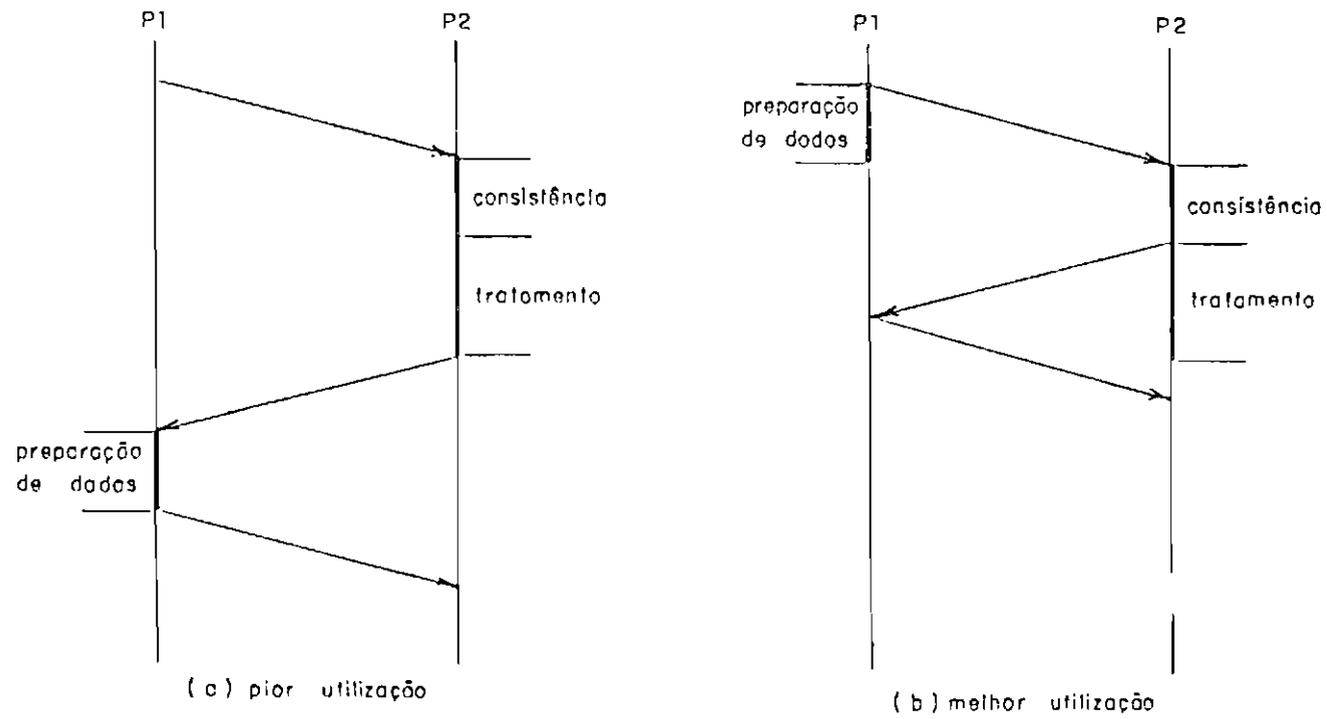


Fig. 3.3 - Utilização dos recursos do paralelismo visando eficiência do processamento.

O objetivo de atingir uma alta confiabilidade para o programa, quando aplicado a sistemas distribuídos, envolve todos os cuidados necessários à não-geração de erros e à criação de mecanismos que facilitem sua detecção quando existirem. Aos erros comuns a sistemas seqüenciais adicionam-se os erros próprios de sistemas distribuídos, nas quais o fator tempo influencia decisivamente. Como diretrizes básicas que visam atingir este objetivo podem-se destacar: (1) a desenvolvimento do "software" acompanhado de verificações em suas diversas fases, com o objetivo de analisar as trocas de informações (sua consistência e coerência nos diversos processos), o comportamento lógico do fluxo de dados e (2) a realização de testes, programados em função das características dos sistemas distribuídos (este assunto está detalhado na Seção 3.3.3).

Para o "software" de um sistema distribuído ser compreensível é necessário, em primeiro lugar, que haja clareza na definição dos grupos funcionais (grupos de processos, alocados geralmente em um mesmo módulo processador, os quais realizam funções especificadas para o sistema) e dos processos individuais. Esta definição envolve a estrutura lógica dos processos, suas interconexões, relações hierárquicas e estruturas de dados. Os conteúdos das informações trocadas entre processos devem ser definidos e documentados de forma clara e precisa, de forma que um técnico que tenha conhecimentos em "software" distribuído possa compreendê-las, avaliá-las e até alterá-las se necessário. Em vista da tendência que têm os sistemas distribuídos de serem de grande porte, um outro cuidado a ser tomado consiste em procurar garantir a homogeneidade do projeto em todos os seus níveis. Para sistemas compostos por vários processos e processadores é desejável que cada uma das suas estruturas funcionais sejam desenvolvidas aplicando métodos e critérios uniformes, o que facilita a compreensão do funcionamento de partes do sistema, com base no conhecimento de outras partes. Estruturas básicas de acesso aos dados, rotinas e algoritmos de uso geral devem ser idênticos nos diversos módulos.

### 3.3.1.2 - ESPECIFICAÇÃO, PROJETO E CODIFICAÇÃO

O desenvolvimento de "software" para sistemas de arquitetura distribuída se faz de acordo com os mesmos princípios, apresentados na Seção 2.2, para sistemas de computação quaisquer. As seguintes fases constituem o ciclo de vida completo do "software": (1) análise do problema, definição de necessidades e especificação; (2) projeto; (3) codificação; (4) testes e (5) manutenção e operação.

A preocupação com a fase de testes deve estar incluída na metodologia de desenvolvimento das três primeiras fases. São apresentados a seguir alguns pontos que devem ser considerados em cada uma delas para esse fim, em adição às considerações já apresentadas nesta mesma seção, que se aplicam a essas fases.

Na primeira fase é definida a estrutura funcional e de interconexão do "software". Os processos não são especificados desde o início, sendo antes necessário subdividir as funções globais do "software" em diversos grupos funcionais, encarregados de realizá-las. Especificam-se a seguir a hierarquia e as interconexões que devem existir entre os grupos funcionais. A seguir, para cada um dos grupos, definem-se os processos que deverão compô-los. Esse procedimento facilita a especificação de um "software" cujos módulos apresentam baixo grau de conectividade, e uma estrutura hierárquica bem definida, permitindo que, durante a fase de testes, os erros porventura existentes sejam detectados e corrigidos com maior facilidade. A Figura 3.4 apresenta um exemplo de estrutura funcional em que podem ser identificados os grupos funcionais, os processos e suas interconexões.

É muito conveniente que a técnica de desenvolvimento "top down" seja utilizada para o desenvolvimento do projeto. Na fase de projeto preliminar, definem-se as interfaces entre os processos e entre os grupos funcionais (estas interfaces são representadas pelas trocas de informações entre processos além das estruturas de dados a serem utilizadas pelos diversos grupos funcionais). Na fase de projeto detalhado, é desenvolvida a estrutura lógica dos módulos que compõem cada processo.

Cuidados especiais devem ser tomados na definição das mensagens trocadas entre os processos, uma vez que os erros cometidos nessa definição se rão transparentes à fase de codificação, manifestando-se apenas na oca sião da aplicação dos testes do programa. Na seção 3.3.2 são apresentadas algumas formas de verificar a consistência das mensagens definidas no projeto.

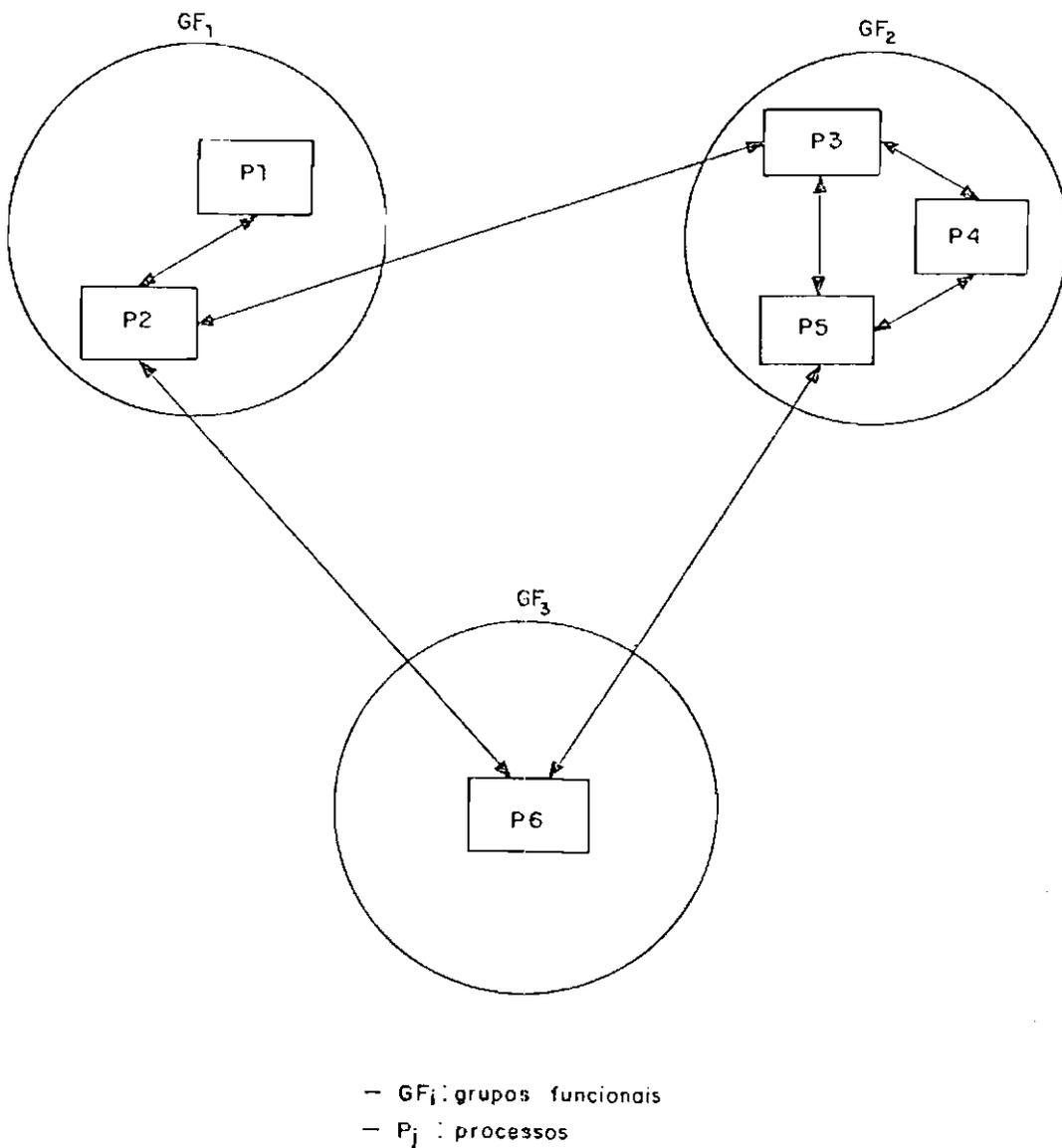
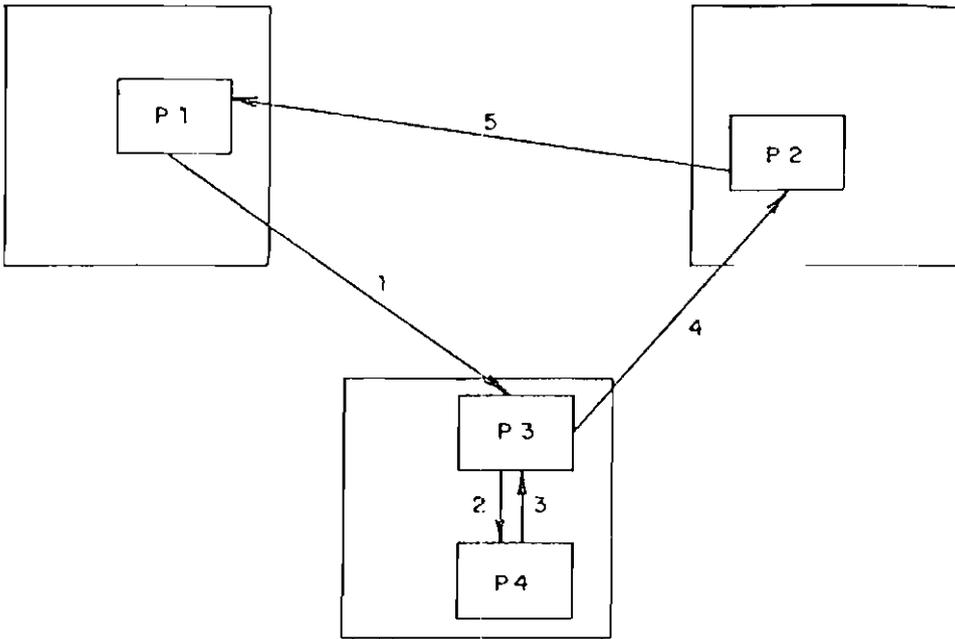


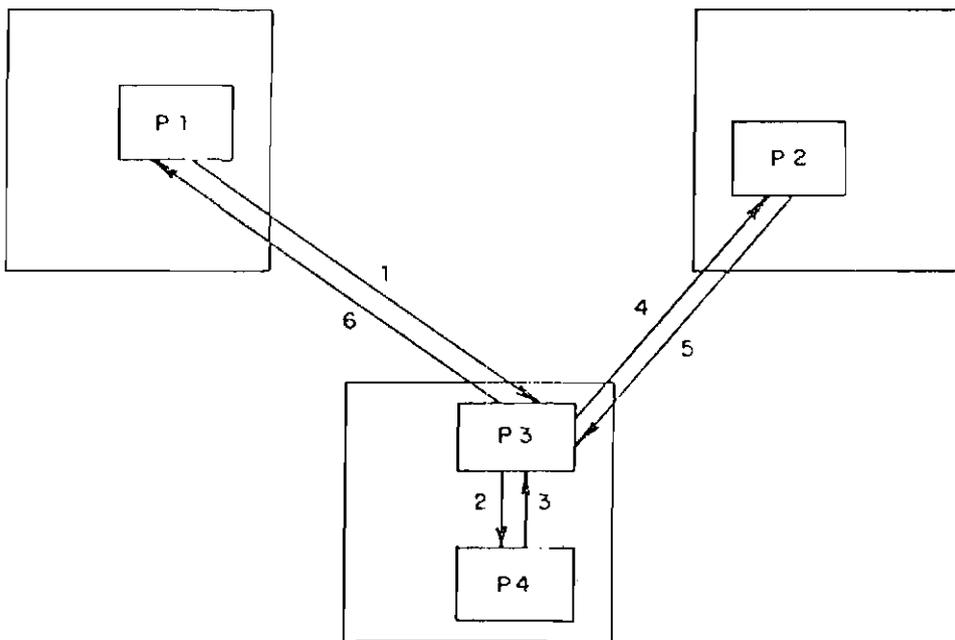
Fig. 3.4 - Exemplo de estrutura funcional.

A estrutura de sincronização entre os processos é definida na fase de projeto. Considerando que a maioria das dificuldades apresentadas na Seção 3.1 para se testar "software" concorrente e distribuído, são relacionadas com problemas de temporização crítica ou de sincronização, conclui-se que as decisões de projeto tomadas nesta fase tem grande influência na fase de testes. Algumas dificuldades, se não eliminadas, podem ser bastante reduzidas adotando-se os seguintes critérios:

- 1) Confinar os dados, o quanto possível, aos processos, evitando o uso excessivo de variáveis globais. Nos casos em que o S.O. permite a comunicação entre processos residentes em um mesmo nó através de mensagens, é possível, se necessário, garantir a inexistência de variáveis compartilhadas, fazendo com que os estados dos processos sejam totalmente independentes.
- 2) No detalhamento de funções do "software", evitar cadeias complexas de trocas de informações entre processos e sobretudo entre processadores. A melhor maneira de evitar erros de sincronismo é fazer com que a maioria das transações envolvam um número mínimo de processos (dois, se possível). A Figura 3.5 apresenta um caso em que uma transação que envolve três processadores pode ser implementada de tal modo que são criadas outras transações que envolvam apenas dois. Apesar de se notar na Figura 3.5b a existência de uma mensagem a mais e a sobrecarga de comunicações para P3, verifica-se a simplificação da estrutura de interconexão. Se o prejuízo obtido em relação à utilização menos eficiente do paralelismo não criar situações críticas, a segunda estrutura deve ser a adotada, pois é menos suscetível a erros e facilita, caso tais erros existam, a sua localização.
- 3) Desenvolver os módulos de "software" fazendo uso de técnicas próprias para representação de "software" concorrente e distribuído, tais como Redes de Petri (Agerwala, 1979; Nelson, 1983; Yau, 1983), de forma que possam ser representados e verificados os fluxos de controle e de trocas de informações entre os processos.



(a) estrutura original



(b) estrutura modificada

Fig. 3.5 - Simplificação de transação.

Na fase de codificação, deve-se considerar, além dos aspectos levantados na Seção 2.2, que é desejável que todos os programas sejam codificados utilizando uma única linguagem de programação. Pode-se fugir a essa regra apenas na codificação de rotinas consideradas como gargalos de velocidade que necessitam utilizar a eficiência total de máquina através do uso de uma linguagem de baixo nível. Por isso, a escolha da linguagem deve ser feita de maneira a satisfazer as necessidades do sistema como um todo (Viola, 1979).

Tomados os cuidados necessários nas fases de concepção inicial do "software" distribuído, este pode ser codificado, para em seguida apresentar-se em uma forma tal que possa ser finalmente submetido à fase de testes.

### 3.3.2 - VERIFICAÇÕES EM DIVERSAS FASES DO DESENVOLVIMENTO

As diversas fases do ciclo de vida do "software" de sistemas distribuídos devem ter seus resultados verificados, nos termos propostos na Seção 2.3.2, adotando-se, no entanto, técnicas próprias para análise das características desses sistemas. São feitas aqui algumas considerações relativas a essas técnicas, como atenção especial às verificações da fase de projeto, na qual se firmam as estruturas lógicas e de dados que dão sustentação ao processamento distribuído.

A verificação dos documentos de definição de necessidades e especificação deve ser feita em relação à completeza, consistência e praticabilidade, como em qualquer sistema de "software". Dessas características, a consistência é a que deve merecer análises mais exaustivas na especificação funcional. A estrutura de interconexão e as relações hierárquicas devem ser respeitadas na especificação de cada um dos grupos funcionais.

As verificações da fase de projeto devem concentrar-se sobretudo na estrutura de sincronização definida, cu seja, nas trocas de informações entre os processos. Duas atividades podem ser desenvolvidas em momentos distintos da fase de projeto:

- 1) Análise do protocolo de trocas de informações, realizada após a definição da estrutura de sincronização, a qual compõe-se das seguintes ações:
  - análise da utilização dos mecanismos de gerenciamento de memória para compartilhar variáveis;
  - análise dos documentos que definem o protocolo de trocas de mensagens entre processos.
- 2) Análise das estruturas internas e dos conteúdos das mensagens trocadas entre processos, realizada após a definição detalhada das mensagens. O detalhamento de cada um dos processos deve respeitar as características definidas para as mensagens que envia ou recebe.

No Capítulo 4 são apresentados alguns detalhes das verificações a serem realizadas na fase de projeto.

### 3.3.3 - TESTES

São levantadas aqui algumas diretrizes gerais para a elaboração e realização dos testes de "software" de sistemas distribuídos, que podem ser aplicadas a uma gama variada de sistemas distribuídos coerentes com um modelo que será apresentado. O ambiente de testes é o primeiro item a ser analisado com base nesse modelo, cuja estrutura serve como ponto de referência para comparações e adaptações das propostas apresentadas a outros sistemas distribuídos. As ferramentas que podem ser desenvolvidas e utilizadas constituem muitas vezes imprescindíveis e recebem atenção especial na apresentação das diretrizes. O ambiente de testes e as ferramentas para sistemas distribuídos são diferentes, em essência, daqueles próprios para sistemas seqüenciais. Esses recursos, colocados à disposição dos programadores, influem decisivamente na eficácia e confiabilidade da fase de testes.

A parte final deste item está dedicada à metodologia para os testes, com os pontos principais que devem nortear a sua elaboração para cada caso.

Não serão feitas considerações sobre testes do "software" básico, pois admite-se que esteja depurado. Assim, o Sistema Operacional (responsável pelo gerenciamento da utilização dos recursos e da comunicação entre os processos) e o "software" dedicado à comunicação entre processadores, são considerados corretos para efeito do presente estudo. As ferramentas propostas serão também consideradas depuradas, quando utilizadas na realização dos testes. A análise apresentada neste trabalho está concentrada, portanto, nos aspectos relativos aos testes do "software" de alto nível e do código aplicativo.

### 3.3.3.1 - AMBIENTE DE TESTES

Os testes são realizados em ambiente físico e lógico preparado especialmente para eles, chamado ambiente de testes, formado basicamente por equipamentos de trabalho (computadores, terminais de vídeo, consoles, impressoras etc.), programas que constituem o "software" básico para testes (ou seja, o próprio Sistema Operacional com possíveis modificações que visem facilitar os testes) e ferramentas que facilitam o trabalho de monitorar os testes, controlar a execução e depurar os programas. Entre essas ferramentas pode estar incluído um gerenciador de arquivos em disco, responsável pelo armazenamento de programas e tabelas e pela sua eventual carga nos módulos processadores, no momento da aplicação dos testes.

Considerando a grande variedade existente de arquiteturas e de estruturas de "software" distribuído, é mostrado um modelo de ambiente de testes para os quais as diretrizes apresentadas se aplicam. A Figura 3.6 apresenta esse modelo, baseado em outro análogo concebido por Garcia-Molina (1984). A cada conjunto formado por processador, memória e interface de comunicação dá-se o nome de nó. O sistema completo é formado, portanto, por vários nós (número, a princípio, ilimitado) ligados pela interface de comunicação com a Rede de Interconexão. Em cada nó reside uma cópia do "software" de baixo nível SO (responsável pelo gerenciamento da comunicação entre os processos); vários processos aplicativos, que constituem o "software" de alto nível, a ser testado; uma cópia do programa depurador local, dedicado aos testes e depurações; e uma de interface com a Rede de Interconexão.

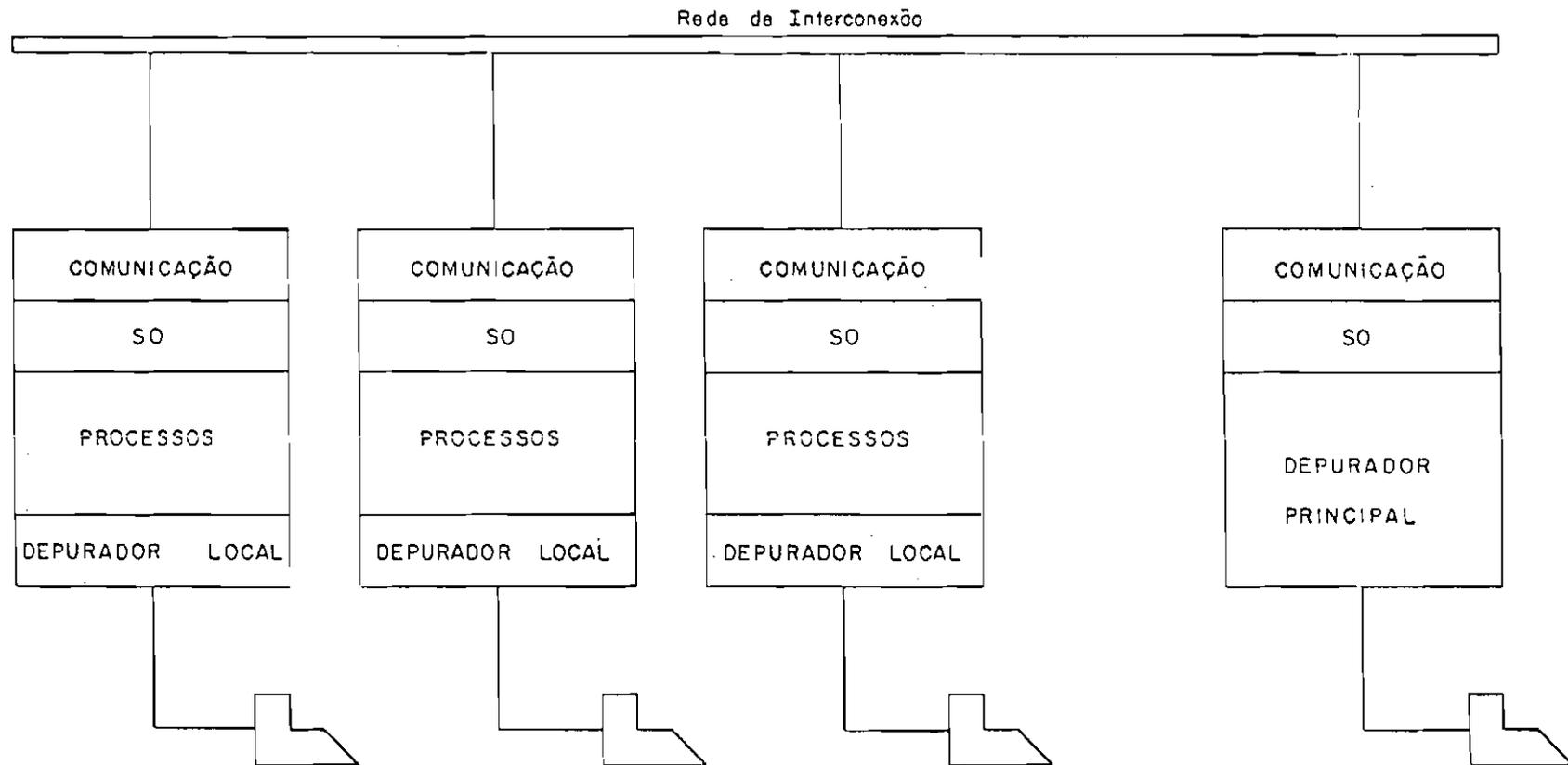


Fig. 3.6 - Modelo de sistema de arquitetura distribuída para testes de "software" distribuído.

O S0 para testes possui todos os recursos de S0 do sistema real que está sendo testado, tendo sido entretanto enriquecido com procedimentos adicionais, responsáveis pela realização de testes de consistência mais rigorosos das trocas de informações entre os processos, e ainda por outras atividades de interesse para os testes. Os recursos a serem incluídos no S0 para testes são discutidos na Seção 3.3.3.2. Não havendo interesse ou condições de desenvolver um novo S0, pode-se utilizar nos testes o próprio S0 do sistema final.

Os grupos de processos aplicativos (grupos funcionais, descritos na Seção 3.3.1) são carregados nos nós para serem testados, configurando-se de acordo com os interesses do teste a ser realizado.

O programa depurador local possui recursos para auxiliar na aplicação de testes e na depuração dos processos aplicativos locais. Suas características principais são: (1) oferece uma interface com o programador, que pode monitorar e observar a execução dos processos locais através de um terminal de vídeo e um teclado; e (2) permite a inclusão do nó em testes integrados, com gerenciamento central a partir de um nó carregado com um programa depurador principal. O programa depurador local contém ferramentas para depuração de programas seqüenciais e paralelos, devendo ser visto como um componente do sistema de depuração distribuído completo, que integra as facilidades trazidas pelos depuradores locais e pelo depurador principal. Isoladamente, cada programa daquele possui recursos para testar dinamicamente os processos que concorrem pelo processador e a memória de um nó.

A entrada e saída de informações de cada processador é feita através do sistema de comunicação, acessado através do programa de comunicação, residente em cada nó.

Com o programa depurador principal carregado em um dos nós, é possível gerenciar a execução de testes integrados entre os nós. Com o uso de um terminal de vídeo e um teclado o programador pode observar saídas da execução e enviar comandos aos nós executados através dos programas e dos S0. Não há processos aplicativos no nó depurador principal.

A troca de informações entre os processos é feita através de mensagens (conjuntos de dados enviados de um processo para outro com o auxílio do SO e, no caso de envolver processos residentes em nós diferentes, transmitidos através do sistema de comunicação): Os processos residentes em um mesmo nó podem também se comunicar através de variáveis compartilhadas.

Supõe-se que o ambiente físico ("hardware") em que os testes se desenvolvem possua as mesmas características exibidas pelo sistema onde o "software" final deverá ser implantado, podendo ser o próprio, se possível. Assim, só se utilizam simuladores para criar réplicas das condições externas do meio no qual o sistema final deverá ser inserido, na aplicação específica a que se destina (simuladores de eventos externos, a serem tratados pelo sistema distribuído).

### 3.3.3.2 - FERRAMENTAS

As ferramentas que compõem o ambiente de testes de sistemas distribuídos devem dar aos programadores condições para analisar o comportamento dos processos, além do comportamento da comunicação entre processos e entre processadores. Considerando que as ferramentas para testes de programas seqüenciais mencionadas no Capítulo 2 não possuem tais recursos, pode-se concluir pela necessidade de desenvolver novas ferramentas capazes de satisfazer às necessidades próprias dos sistemas distribuídos, em adição às ferramentas disponíveis, voltadas para o apoio aos testes de programas seqüenciais. Em geral, os programas que implementam o "software" de sistemas distribuídos possuem muitos trechos e rotinas seqüenciais, que devem ter seus testes incluídos na programação. Essas ferramentas, desenvolvidas ou adaptadas ao ambiente de testes, são imprescindíveis a um conjunto operacional de recursos para o apoio aos testes do "software" de sistemas distribuídos.

A utilização de qualquer ferramenta para testes de "software" de sistemas distribuídos obviamente interfere no comportamento do sistema, podendo impossibilitar a observação de erros existentes ou

provocar a manifestação de novos erros. A causa disso está no tempo de processamento adicional que a execução dessas ferramentas provoca, o que pode fazer com que erros sutis de sincronização se manifestem ou sejam mascarados. A execução do programa sem a utilização das ferramentas esclarecerá, em muitos casos, essas situações.

São apresentadas a seguir propostas de algumas ferramentas que demonstram a sua utilidade prática. As três primeiras são fruto da experiência profissional, as demais são propostas com base em contribuições de artigos técnicos dedicados ao assunto, indicando-se nesses casos as respectivas referências. Não sendo objetivo do trabalho apresentar detalhes estruturais e de implementação, mas sim as diretrizes gerais que devem nortear a realização dos testes, far-se-á, para cada ferramenta, a apresentação de suas características funcionais e de sua conexão lógica com o sistema completo.

As ferramentas propostas realizam as seguintes tarefas:

- 1) consistência de mensagens enviadas e recebidas pelos processos;
- 2) gerenciador de arquivos em disco;
- 3) interface com o SO;
- 4) geração de "traces";
- 5) exame de "traces";
- 6) execução controlada dos processos;
- 7) monitoração da execução

#### FERRAMENTA 1: Consistência das mensagens

A experiência mostra que, a despeito dos cuidados tomados pelo projetista através das verificações do "software" nas diversas fases do seu ciclo de vida, grande quantidade de erros pode ser nele encontrada. Um dos fatores causadores da manifestação de grande parte destes

erros durante a fase de testes é a existência da troca de mensagens in corretas entre os processos. Uma grande variedade de erros desta categoria pode ser detectada dotando-se o SO de recursos de baixa complexidade, destinados à realização de testes de consistência das mensagens enviadas e recebidas pelos diversos processos. O SO assim instrumentado pode ser carregado em todos ou apenas em alguns dos nós do sistema distribuído, dependendo da abrangência desejada para cada teste a ser realizado. Para cada mensagem enviada ou recebida (incluindo as mensagens entre processos internos ao nó), são analisadas as seguintes condições:

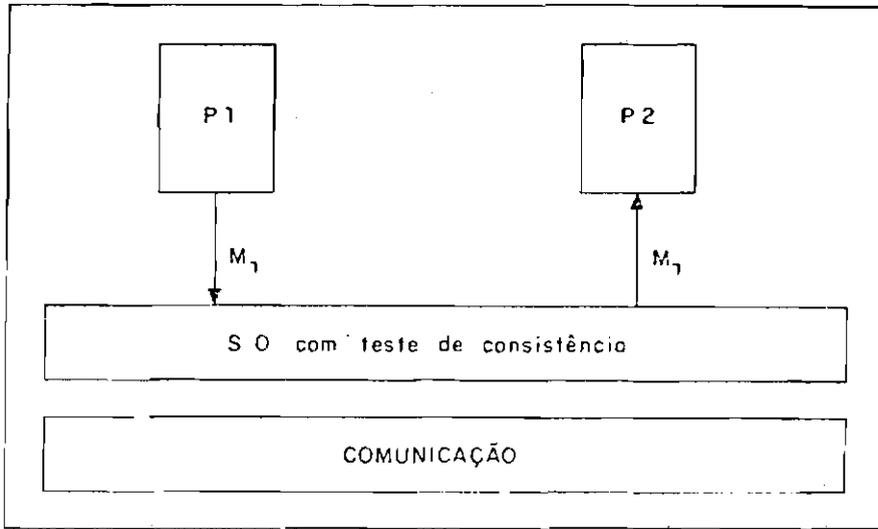
- 1) existência de indicação de informações sobre os processos de destino e de origem, na mensagem;
- 2) coerência entre o tamanho físico da mensagem e o tamanho índicado na própria mensagem; verificação de ultrapassagem do máximo permitido para o tamanho físico das mensagens.

A Figura 3.7 mostra o papel do SO com testes de consistência, no envio e no recebimento de mensagens. Em caso de mensagem com erro, força-se uma parada no processador, a qual permite uma análise do estado dos processos que nele são executados, no instante em que o erro foi detectado.

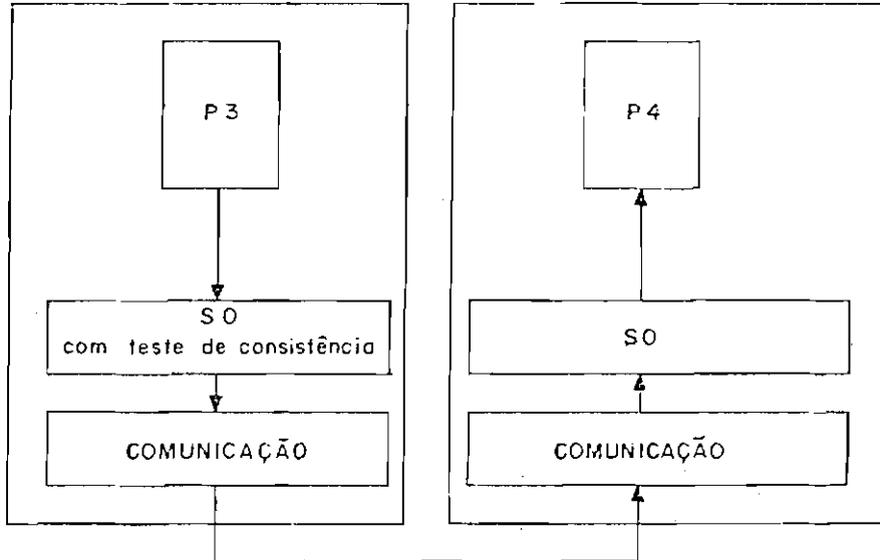
A modificação necessária no SO para permitir a realização dos testes de consistência é feita nas rotinas que implementam as primitivas responsáveis pelo envio e pela espera das mensagens que trafegam entre os processos. São apresentadas a seguir as modificações a serem feitas sem quebrar a estrutura original. Obtem-se as rotinas de envio e espera de mensagem, com consistência, ENVIASC e ESPERASC.

ENVIA: PROCEDIMENTO;  
realize envio de mensagem;

FIM



(a) mensagem entre processos internos a um nó



(b) mensagem entre processos executados em nós distintos.

Fig. 3.7 - Envio e recebimento de mensagem pelo S0 com testes de consistência de mensagens.

ESPERA: PROCEDIMENTO.  
realize espera de mensagem;

FIM

ENVIASC: PROCEDIMENTO:  
chame CONSISTÊNCIA DE MENSAGEM;  
se mensagem consistente  
então realize envio de mensagem  
senão pare;

FIM

ESPERASC: PROCEDIMENTO  
realize espera de mensagem;  
chame CONSISTÊNCIA DE MENSAGEM;  
se mensagem não consistente  
então pare;

FIM

## FERRAMENTA 2: Gerenciador de arquivos em disco

Com base na experiência pode-se afirmar que os testes ganham em agilidade se houver à disposição dos programadores uma unidade de disco para o armazenamento de programas e tabelas, com o respectivo gerenciador de arquivos. Esse gerenciador pode ser opcionalmente incluído no programa depurador principal, oferecendo ao programador os seguintes recursos principais:

- 1) diretório de arquivos em disco;
- 2) criação de arquivos com dados ou programas carregados em qualquer um dos nós ou na área de memória livre do nó gerenciador;
- 3) remoção de arquivos;

- 4) renomeação de arquivos;
- 5) cópias de arquivos a partir de arquivos armazenados em discos flexíveis;
- 6) carga de arquivos para a memória de qualquer nó.

Para utilizar essa ferramenta o ambiente de testes necessita módulos que realizam o interfaceamento com os periféricos envolvidos: o disco rígido e a unidade de discos flexíveis. A Figura 3.8 apresenta o Gerenciador de Arquivos em Disco e os nós que realizam essas interfaces: a Interface com Disco, para leitura e escrita, e a Interface com Unidade de Discos Flexíveis, para leitura. Esses nós são dedicados às tarefas de acesso aos periféricos, possuindo circuitos projetados com esta finalidade. Possuem, além disso, processos aplicativos para a comunicação com os demais nós, utilizando o próprio S0 desenvolvido para o sistema a ser testado. O seu projeto pode ser mais abrangente do que o exigido para o ambiente de testes, se for de interesse a sua utilização no próprio sistema a ser testado. De qualquer forma o seu desenvolvimento é fundamental para os testes.

O Gerenciador de Arquivos se utiliza de mensagens para enviar e receber dados dos nós. Cada comando de carga ou armazenamento executado pelo operador resulta em um "loop" de transferência de dados através das mensagens, até que não haja novos dados a enviar. Os tipos básicos de mensagens, necessários para realizar essas tarefas, são os seguintes:

- 1) Do Gerenciador de Arquivos em Disco (GAD) para a Interface com Disco (ID):
  - pedido de arquivo em disco (com sua identificação) para carga na memória do GAD;
  - dados para serem gravados em arquivo a partir da memória do GAD.

- 2) Do GAD para a Interface com Unidade de Discos Flexíveis (IDF):
  - pedido de arquivo em disquete (com sua identificação) para carga na memória do GAD.
- 3) Do GAD para um nã qualquer:
  - pedido de dados carregados no nã (com indicação da posição na memória) para carga na memória do GAD;
  - envio de dados da memória do GAD para serem carregados no nã.
- 4) Da ID para o GAD:
  - dados pedidos do arquivo;
  - pedido de mais dados para gravação.
- 5) Da IDF para o GAD:
  - dados pedidos do arquivo.

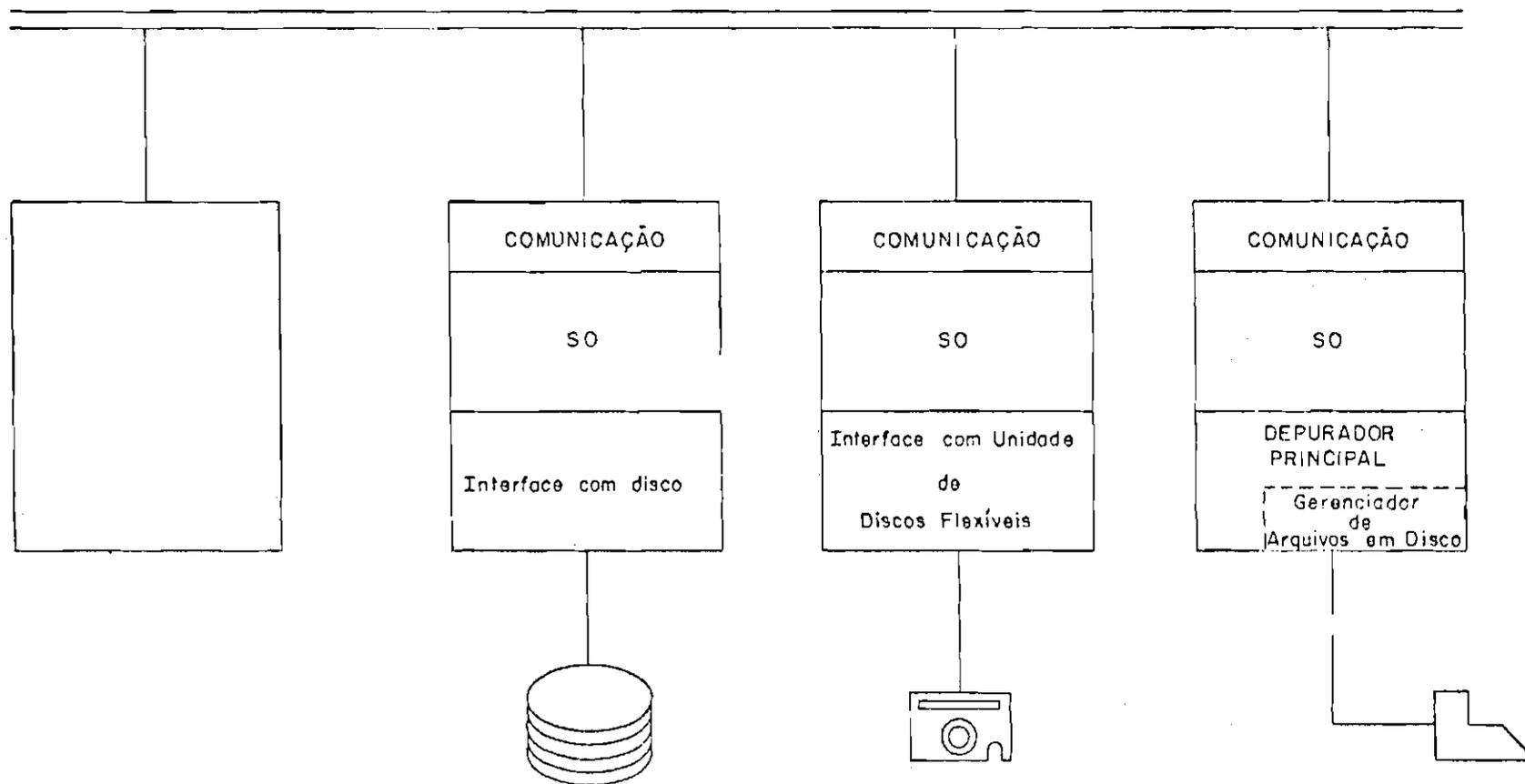


Fig. 3.8 - Gerenciador de arquivos em disco.

FERRAMENTA 3: Interface com o Sistema Operacional

As primeiras etapas dos testes dinâmicos de sistemas distribuídos envolvem análises detalhadas do comportamento dos processos (seus estados, mensagens enviadas e recebidas, etc.). A experiência mostra que essas análises são feitas com maior facilidade com o auxílio de uma Interface entre o programador e o SO, que permite monitorar a ocorrência de eventos relevantes e interferir na execução, seja alterando parâmetros, seja gerando eventos artificiais. Essa ferramenta está incluída no programa depurador local. A interface com o programador se faz através de um terminal de vídeo e de um teclado, ligados ao nó.

A Figura 3.9 mostra a relação entre o programador e a Interface com o SO no teste individual de um processo. Todas as mensagens enviadas pelo processo podem ser monitoradas, enquanto as mensagens a serem recebidas são criadas, via teclado, pelo próprio programador, com a ajuda de uma linguagem de entrada interativa.

Essa ferramenta pode ser usada também para testes de dois ou mais processos em conjunto, localizados ou não no mesmo nó. Detalhes sobre sua utilização são apresentados no Capítulo 4.

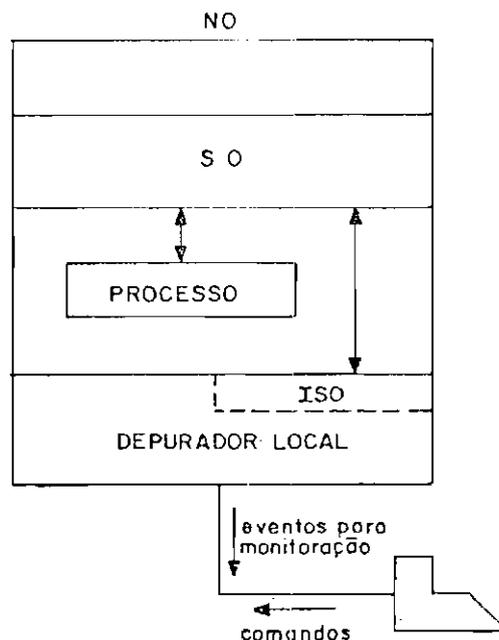


Fig. 3.9 - Utilização da interface com o SO para teste de um processo.

#### FERRAMENTA 4: Geração de "traces"

Muitos erros se manifestam em decorrência da interação entre vários processos aplicativos, tornando-se difícil sua reprodução em novos testes, como exposto na Seção 3.1.1. Através de "traces" (García-Molina, 1984; Hansen, 1973b) o programador pode analisar, em diversos processos, os eventos que antecederam determinado erro, registrados durante a execução.

Dois tipos de ferramentas são necessárias para depurar os programas dessa maneira:

- 1) ferramenta para gerar "traces" e
- 2) ferramenta para analisar "traces"

A geração de "traces" pode ser feita pelos próprios processos aplicativos, projetando, desde o início do desenvolvimento do "software", saídas em pontos apropriados do programa.

Como vantagem desse procedimento, tem-se a concisão, pois o projetista de cada processo conhece os eventos de maior importância para os testes, não sendo necessário, por exemplo, registrar cada mensagem enviada ou recebida. O desenvolvimento de um "software" aplicativo voltado para testes não prescinde desse recurso.

Como forma de enriquecer os "traces" gerados e superar parcialmente o problema da heterogeneidade de critérios adotados pelos projetistas para gerá-los, dota-se o S0 de uma ferramenta para gerar "traces" em condições variáveis de acordo com o interesse dos testes a realizar. Os eventos que ocorrem com frequência em sistemas distribuídos (como, por exemplo, o envio de uma mensagem) podem ser registrados. A saída pode ser feita para arquivos em disco (um por nó), ou para o terminal ligado ao nó.

Os eventos de maior interesse são geralmente os seguintes:

- 1) ativação de processo;
- 2) desativação de processo;
- 3) envio de mensagem;
- 4) recebimento de mensagem e
- 5) entrada em região crítica.

Além desses podem ser incluídos outros eventos cuja frequência é menor. Se o SO realiza a detecção de "deadlock", por exemplo, esse evento deve aparecer nos "traces".

Para cada um dos tipos de eventos acima citados, correspondem parâmetros diferentes que devem ser também registrados. Estão listados a seguir alguns dos parâmetros que podem ser definidos para cada caso:

- 1) ativação de processo: parâmetros de entrada;
- 2) desativação de processo: parâmetros de saída, "dump" de variáveis;
- 3) envio de mensagem: processador e processo origem, processador e processo destino, tipo de mensagem, tamanho, conteúdo;
- 4) recebimento de mensagem: processador e processo origem, processador e processo destino, tipo de mensagem, tamanho, conteúdo;
- 5) entrada em região crítica: processo, região;
- 6) saída de região crítica: processo, região.

Todos os registros devem possuir o horário de ocorrência do evento.

Enviando comandos através do no depurador principal, o programador define, para cada no, os tipos de eventos que devem aparecer no "trace", bem como os processos que devem ser supervisionados. Cada comando gera uma mensagem que é enviada ao S0 do no em questão. Este S0 se incumbe de executar a tarefa desejada. Os comandos do operador devem indicar:

- tipo de comando (ativação, desativação),
- identificação do no,
- identificação do processo,
- tipo do evento,
- arquivo de saída (terminal ou arquivo em disco).

Para evitar o registro indesejado de eventos em fases do processamento que não estão sendo analisados, o comando de ativação de "trace" pode ser executado depois da ativação do sistema.

Não ocorrendo o erro esperado ou tendo-se já realizado os registros desejados, utiliza-se o comando de desativação de "trace" para interromper o registro dos eventos sem interferir na continuidade do processamento.

#### FERRAMENTA 5: Exame de "Traces"

Os "traces" gerados pela ferramenta 4 são em grande quantidade para cada teste, exigindo busca exaustiva para serem identificados, para análise, os eventos relevantes. Após a sua geração, os arquivos de "traces" gravados em disco são pesquisados por uma ferramenta que realiza essa busca de acordo com critérios selecionados conforme a conveniência de cada teste (Garcia-Molina, 1984).

A Figura 3.10 esquematiza a seqüência de ações que esta ferramenta e a anterior estabelecem para análise dos eventos e busca de erros.

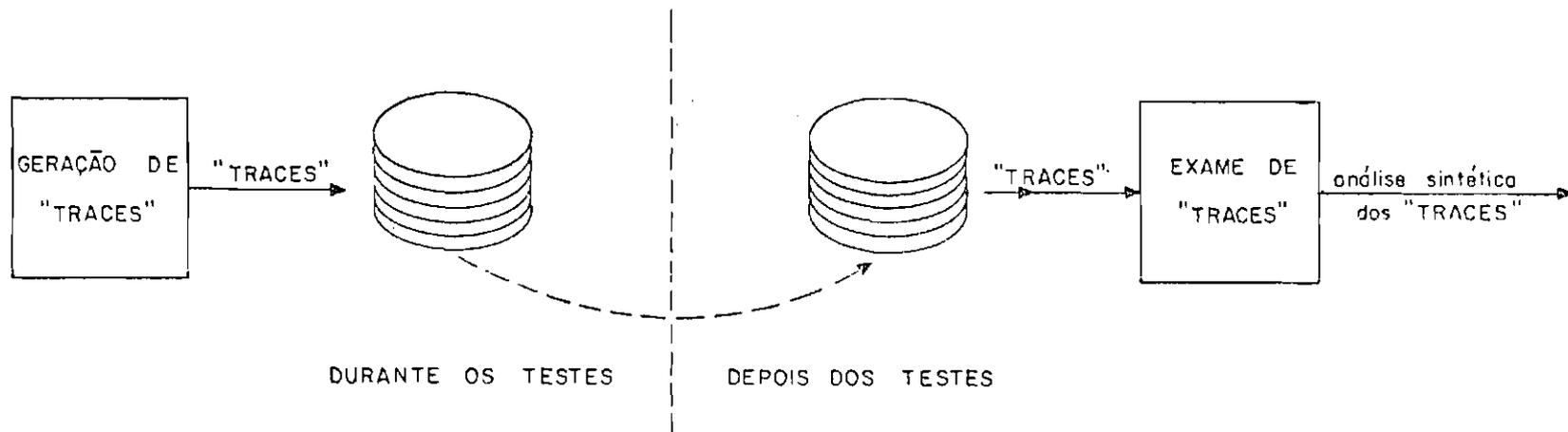


Fig. 3.10 - Geração e exame de "TRACES".

Para operar esta ferramenta, é conveniente que haja uma linguagem de comandos de busca, cuja execução resulte em listagens de eventos que satisfazem a um conjunto de condições.

Os seguintes tipos de comandos devem estar inseridos na linguagem:

- 1) seleção de evento de acordo com um ou mais parâmetros. Por exemplo, listar todos os envios de mensagens do tipo X efetuadas em um processo P1 para um processo P2;
- 2) contagem de eventos de acordo com um ou mais parâmetros. Por exemplo, listar o número de vezes que uma mensagem do tipo Y foi recebida por um processo P3 em determinado intervalo de tempo.

A implementação da ferramenta para examinar "traces" pode ser feita para que seja executada no próprio sistema distribuído em teste. Através de depurador principal, o programador distribui os arquivos entre os nós e envia os comandos de busca que cada um deles deve realizar. Os programas depuradores locais efetuam o exame e listam os resultados em arquivo especificado pelo programador (arquivo em disco ou terminal ligado ao nó). Os comandos para examinar os "traces" carregados nos nós devem indicar:

- identificação do nó,
- arquivo de saída (terminal ou arquivo em disco).

#### FERRAMENTA 6: Execução controlada

Como exposto na Seção 3.1, é desejável que as ativações e interrupções de execução dos processos sejam sempre sincronizadas de forma que o programador tenha controle sobre o estado dos processos. Por isso é necessário que os programas depuradores sejam dotados de recursos para realizar aquelas atividades como resposta a um comando do programador. Os seguintes comandos são necessários:

- 1) ativação de processo(s),
- 2) interrupção da execução de processo(s),
- 3) reativação de processo(s).

A ativação é realizada através do envio de uma mensagem do depurador principal para um nó qualquer. Os processos aplicativos daquele nó passam a ser executados a partir de seus pontos iniciais. O comando de parada causa, através de mensagem análoga, uma parada forçada da execução dos processos aplicativos do nó. Após uma parada, pode ser executado o comando de reativação, o qual coloca mais uma vez os processos em execução, a partir dos seus pontos de reativação. Esses pontos não coincidem necessariamente com os pontos de ativação (pode haver interesse, numa segunda ativação, em conservar os valores de algumas variáveis definidas durante a execução anterior, partindo-se, portanto, de um estado diferente do inicial).

Os pontos de ativação e reativação de cada processo devem estar indicados em tabelas que são utilizadas pelo depurador principal.

Os comandos de ativação, parada e reativação devem indicar:

- tipo de comando (ativação, parada, reativação),
- identificação do nó.

Com esses comandos à disposição, o programador pode realizar ativações, paradas e reativações dos processos de um ou mais nós, de acordo com a conveniência. No caso de haver uma parada espontânea de um dos processadores, perceptível pelo programador, este pode utilizar em seguida comando para parar todos os processos, evitando assim que os processos dos demais processadores mantenham-se ativos indefinidamente. Com essas paradas é possível, em muitos casos, recuperar o estado em que o erro ocorreu. Porém, além de não haver a garantia de que os estados dos processadores não foram alterados antes de serem paralisados (pois transcorrem-se, no mínimo, alguns segundos para o programador perceber o erro

e acionar o comando), esse procedimento tem o inconveniente de ser incômodo para o programador. Uma solução para esse problema está em desenvolver uma ferramenta que pare todos os processadores automaticamente, logo que um erro ocorra.

Garcia - Molina (1984) propõe a geração de uma mensagem "broadcasting" de parada sempre que um comportamento inesperado ocorre em algum processador.

#### FERRAMENTA 7: Monitoração de execução

As ferramentas apresentadas anteriormente são utilizadas para a detecção de erros e a depuração do "software" distribuído. Após testar e depurar o programa por algum tempo, chega-se a um estágio em que o sistema pode ser executado continuamente, tornando-se mais esporádica a manifestação de erros. Além da busca e correção de erros, que deve ainda prosseguir, o estado do sistema permite a realização de outra atividade, em que se analisa o comportamento do sistema em regime. Para isso, é útil o desenvolvimento de uma ferramenta para monitoração, que gera, em tempo real, informações diversas sobre a execução nos nós e apresenta-as no terminal principal (Garcia - Molina, 1984). As informações de maior interesse são geralmente as seguintes:

- 1) dados relativos ao desempenho do sistema (utilização de UCP, tamanho médio, mínimo e máximo das filas, estados dos processos);
- 2) número total de mensagem e taxa média de mensagens por segundo.

Através do teclado do terminal principal o programador envia comandos para os processadores, definindo os tipos de informações que devem ser coletadas e enviadas ao depurador principal pelos diversos nós.

Essa ferramenta completa o quadro de propostas para o ambiente de testes de "software" de sistemas distribuídos. A Figura 3.11 resume esquematicamente esse quadro.

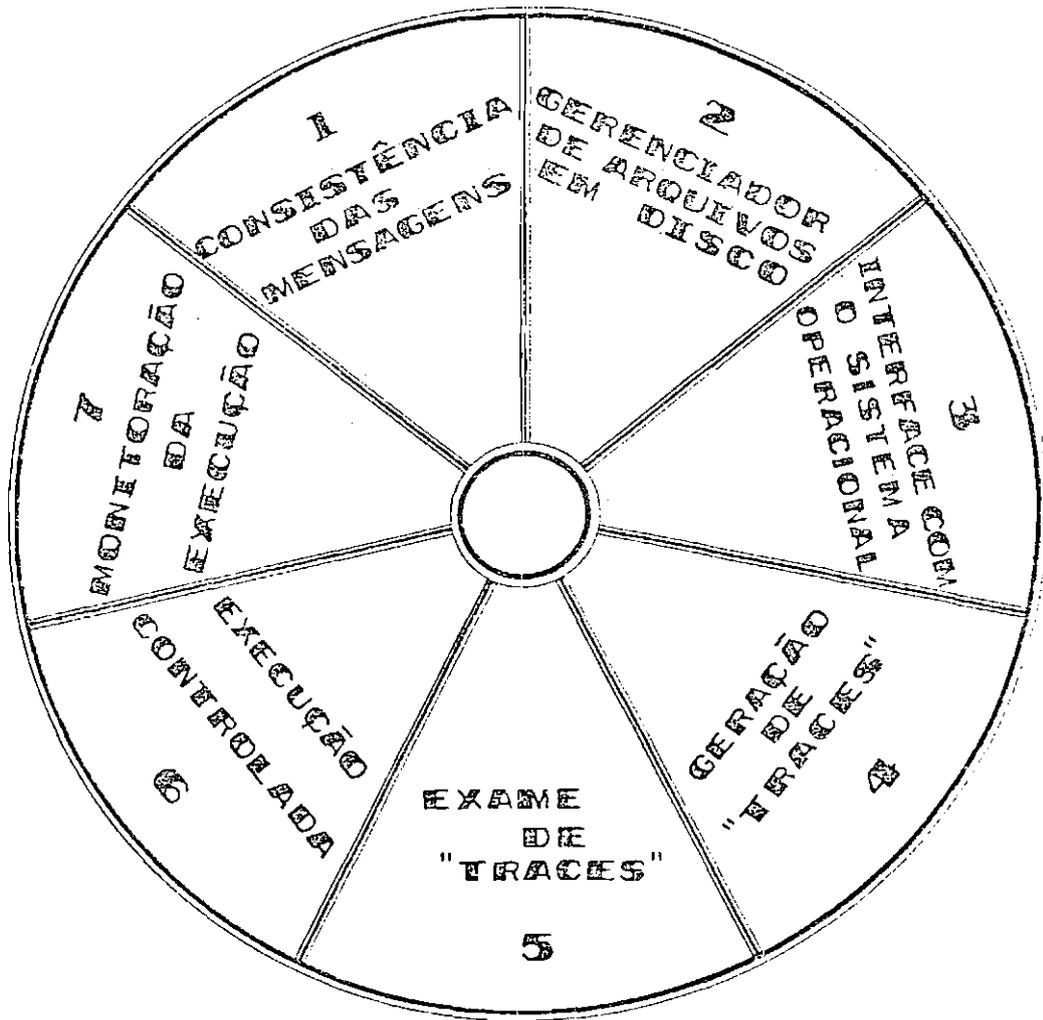


Fig. 3.11 - Ferramentas para o ambiente de teste.

### 3.3.3.3 - METODOLOGIA

Os testes de "software" de sistemas distribuídos possuem duas etapas:

- 1) testes individuais dos processos,
- 2) testes integrados, com dois ou mais processos envolvidos.

Os testes individuais são necessários para fazer com que cada processo esteja trabalhando propriamente ao ser integrado com outros. Considerando que as trocas de informações fazem parte das interfaces entre os processos, podendo ser vistas como entradas e saídas, conclui-se que a integração só é possível após terem sido verificadas as trocas de informações de cada processo individualmente.

Na primeira etapa os processos são isolados, e com o uso de Interface com o Sistema Operacional podem ser realizados testes das interfaces e funções internas de cada um. Cabe ao próprio programador gerar as informações requeridas pelo processo em teste e observar as informações colocadas à disposição dos demais processos.

Os testes integrados podem ser realizados utilizando a técnica "top-down" ou a técnica "bottom-up". Para diferenciar essas duas técnicas nos testes de sistemas distribuídos leva-se em consideração a hierarquia definida pela estrutura de interconexões entre os processos. Essa hierarquia foi discutida na Seção 3.3.1 ao apresentar os objetivos da Engenharia de Software aplicados aos sistemas distribuídos. A diferença básica entre as técnicas está na seqüência de integração dos processos:

- se as interfaces dos processos e grupos funcionais de mais alto nível devem ser verificadas com antecedência, a técnica "top-down" é a mais adequada;

- se as interfaces dos processos, e grupos funcionais de mais baixo nível devem ser verificadas com antecedência, a técnica "bottom-up" é a mais adequada.

A Figura 3.12 apresenta um exemplo que visa esclarecer a diferença entre os procedimentos adotados ao escolher cada uma das duas técnicas acima apresentadas. Os processos A, B, C e D trocam informações entre si, não sendo possível diferenciar níveis hierárquicos na comunicação entre eles. Já os processos 1 e 2 são subordinados, em termos de comunicação, ao processo A. O mesmo tipo de subordinação existe do processo 3 em relação ao processo B, do processo 4 em relação ao processo C e dos processos 5, 6 e 7 em relação ao processo D.

Aplicando a técnica "top-down", integram-se anteriormente os processos A, B, C e D, para depois integrarem-se os processos 1, 2, 3, 4, 5, 6 e 7. Uma possível seqüência para essa integração é apresentada a seguir (separam-se por barras os processos participantes de um mesmo teste):

A/B, A/C, A/D, B/D, C/D,  
A/B/C, A/B/D, B/C/D,  
A/B/C/D,  
A/B/C/D/1, A/B/C/D/2, A/B/C/D/3, A/B/C/D/4, A/B/C/D/5, A/B/C/D/6,  
A/B/C/D/1/2, A/B/C/D/5/6/7,  
A/B/C/D/1/2/3/4/5/6/7.

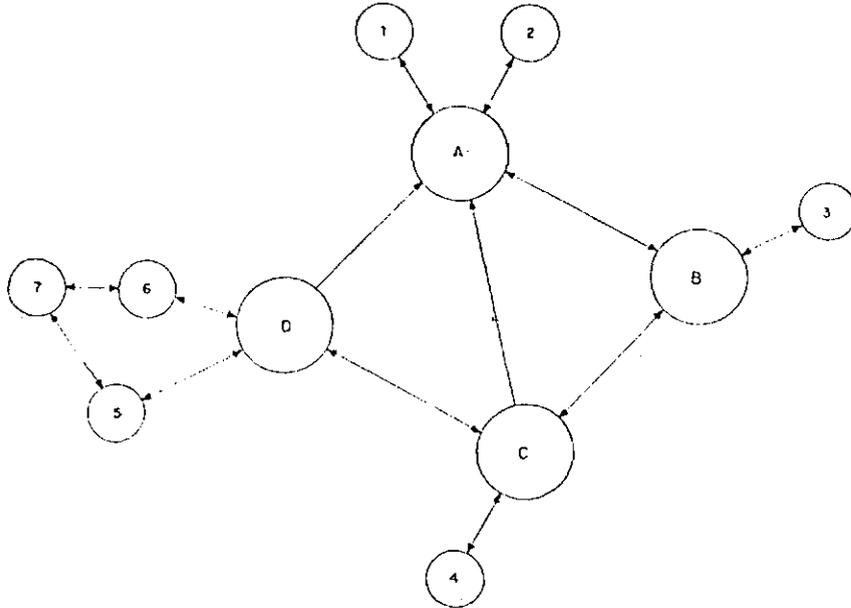


Fig. 3.12 - Estrutura de interconexão de processos a serem testados.

Aplicando a técnica "bottom-up", integram-se anteriormente os processos de nível hierárquico inferior com os processos que com eles se comunicam, para depois integrarem-se os processos de alto nível entre si. Uma possível seqüência para essa integração é apresentada a seguir:

5/7, 6/7,

A/1, A/2, B/3, C/4, D/5/7, D/6/7.

A/1/2, D/5/6/7,

A/B/1/2/3, A/C/1/2/4, A/D/1/2/5/6/7, B/C/3/4, C/D/4/5/6/7,

A/B/C/1/2/3/4, A/B/D/1/2/3/5/6/7, B/C/D/3/4/5/6/7,

A/B/C/D/1/2/3/4/5/6/7

As relações hierárquicas que servem de base para a programação da seqüência de testes a ser adotada não são sempre claras e precisas.

sas, o que faz com que as técnicas "top-down" e "bottom-up" não possam, muitas vezes, ser aplicadas. Nesses casos os testes integrados devem ser programados de acordo com outros critérios estruturais ou funcionais. Como possíveis critérios estruturais têm-se, entre outros:

- 1) complexidade crescente das transações,
- 2) maior prioridade à integração de processos residentes em um mesmo nó.

Como possíveis critérios funcionais têm-se, entre outros:

- 1) maior prioridade à integração dos processos que disparem um maior número de transações distintas;
- 2) definição da seqüência de testes com base em funções especificadas do "software", integrando os processos envolvidos em cada uma delas.

Qualquer que seja a técnica adotada, as ferramentas para geração e exame de "traces" podem ser aplicadas desde o início, podendo ser mais úteis, no entanto, nos testes integrados entre vários processos, registrando os eventos de maior interesse, de acordo com as escolhas do programador.

A atividade de depuração de erros integrados pode ser realizada com auxílio das ferramentas para controle da execução, as quais permitem ativar, parar ou reativar os processos sempre que necessário. Quando um erro ocorre, é fundamental que todos os processos sejam interrompidos, permitindo assim uma análise de seus estados.

### 3.4 - CONCLUSÃO

A concepção do ambiente de testes e da metodologia a ser adotada para os testes de "software" distribuído depende das características particulares do sistema a ser testado. No entanto as diretrizes

apresentadas podem ser adaptadas aos testes de uma gama variada de sistemas vistos de acordo com o modelo apresentado.

A literatura e a experiência demonstram que os testes começam a ser preparados nas primeiras fases do desenvolvimento, através de uma metodologia precisa e de verificações ao final de cada passo. Alguns aspectos do desenvolvimento que afetam os resultados da fase de testes foram levantados, servindo de subsídio para uma elaboração daquela metodologia.

As propostas apresentadas para a fase de testes dão ênfase às ferramentas que podem ser utilizadas, não sendo necessário, no entanto, que todo o conjunto seja implementado. A apresentação dessas propostas tem por objetivo abrir caminhos no sentido de tornar mais completo o ambiente de testes.

## CAPÍTULO 4

### TESTES DE "SOFTWARE" DO NÚCLEO DE PROCESSAMENTO DISTRIBUÍDO

A metodologia e as técnicas utilizadas para os testes do Núcleo de Processamento Distribuído (NPD) são apresentadas neste capítulo, cujo objetivo, no contexto geral do trabalho, é mostrar uma implementação real dos conceitos e propostas apresentados no Capítulo 3. As limitações impostas pelas características do NPD e pelo seu ambiente de testes influem sobretudo na definição das técnicas a adotar em cada atividade, mas não impedem que a metodologia proposta seja aplicada em suas linhas gerais. Detalhes, dificuldades e problemas práticos que essa aplicação envolve são discutidos, bem como as soluções adotadas em cada caso.

Na Seção 4.1 são mostradas as características do NPD e do ambiente de testes. Na 4.2, são apresentadas as atividades do projeto relacionadas com a detecção, o diagnóstico e a correção dos erros. Na 4.3 as bases da metodologia utilizada são firmadas. Em 4.4, 4.5 e 4.6 são detalhadas as etapas a serem vencidas na realização dos testes. A organização da documentação envolvida, com exemplos, é mostrada na Seção 4.7. Para finalizar, uma proposta de aprimoramento do ambiente de testes e da metodologia é apresentada na Seção 4.8.

#### 4.1 - CARACTERÍSTICAS DO NPD E DO AMBIENTE DE TESTES

O Núcleo de Processamento Distribuído (NPD) é uma Máquina de Arquitetura Distribuída com capacidade para multiprogramação e multiprocessamento. A propriedade de execução concorrente coloca este tipo de sistema na classe de máquinas paralelas, mais especialmente, na classe de máquinas Multiple Instruction Stream - Multiple Data Stream (MIMD) (Flynn, 1972). Sua estrutura não é guiada pelo modelo de Von Neumann, utilizado com sucesso para orientar a concepção das máquinas Single Instruction Stream - Single Data Stream (SISD), ou seja, sistemas de computação tradicionais.

Os diversos processos residentes em cada processador se comunicam entre si através de mensagens, enviadas de um para outro, gerenciadas por um programa chamado Sistema de Comunicações (SC), que constitui o "software" básico que opera como do núcleo sistema operacional no NPD.

As estruturas física e lógica do NPD estão resumidas no Apêndice A, onde são delineados o "hardware", o "software" básico e um "software" de alto nível, desenvolvido para aplicação ao controle de tráfego ferroviário.

No Apêndice B é apresentado o ambiente de testes do NPD, enfatizando-se as ferramentas de "software" utilizadas. As principais ferramentas já descritas são as seguintes:

- um SC modificado, com recursos para depuração;
- um programa Controlador do Disco (CD), responsável pelo gerenciamento de arquivos de disco;
- um Sistema de Depuração de Programas Paralelos (SDPP) para auxiliar a realização de testes monitorados;
- um programa monitor com recursos de acesso à memória.

O próprio NPD é utilizado como ambiente físico dos testes.

#### 4.2 - ATIVIDADES VOLTADAS PARA DETECÇÃO, DIAGNÓSTICO E DEPURAÇÃO DE ERROS

As atividades ligadas ao desenvolvimento de um "software" aplicativo de grande porte impõem a execução de diversas tarefas voltadas para a detecção o diagnóstico e a depuração de erros, durante todo o projeto, desde as primeiras etapas do desenvolvimento dos programas até sua aprovação final. Destacam-se as seguintes (Melnikoff, 1984):

- 1) desenvolvimento do "software" a partir da especificação, utilizando técnicas de estruturação, codificação e documentação adequadas de programas, visando facilitar sua análise e possíveis correções posteriores;

- 2) análise das funções do programa, em relação à especificação, identificando as transações (sequências de trocas de mensagens entre processos) que realizam tais funções;
- 3) acompanhamento manual das transações, verificando o fluxo das mensagens, os tipos de mensagens e as suas estruturas internas;
- 4) testes das funções especificadas, através da execução parcial ou total dos programas envolvidos.

As três primeiras atividades são desenvolvidas de forma a permitir uma realização adequada da última, ou seja, os testes funcionais. A programação e a execução destes testes baseiam-se, por um lado, na documentação e nos resultados das três primeiras atividades e, por outro, na especificação funcional, seu ponto de apoio.

Na atividade de desenvolvimento, a modularidade e a estruturação dos programas são os critérios principais, levando em consideração também as demais idéias levantadas na Seções 2.2 e 3.3.1. A técnica "top-down" é aplicada na fase de projeto do "software", permitindo o seu desenvolvimento em níveis de detalhamento cada vez maiores, até a implementação, com geração simultânea da documentação correspondente a cada nível. Essa documentação define as funções, as características e a estruturação lógica dos módulos do "software". Para isso faz uso de diagramas de Nassi-Schneiderman (Nassi, 1973) para representação de programas estruturados e de redes de Petri (Agerwala, 1979; Nelson, 1983) para representação de programas paralelos. Essas técnicas gráficas são adotadas por todos os projetistas com o objetivo de uniformizar a documentação, podendo ser substituídas por outras em novos projetos se for conveniente. A Figura 4.1 apresenta um exemplo de utilização de redes de Petri e diagramas Nassi-Schneiderman. As primeiras mostram os estados, transições, envios e esperas de mensagens. Os últimos são utilizados para expandir trechos sequenciais do programa (as transições), podendo ser apresentados em vários níveis de detalhamento.

As atividades 2, 3 e 4 realizam verificações em fases distintas do desenvolvimento.

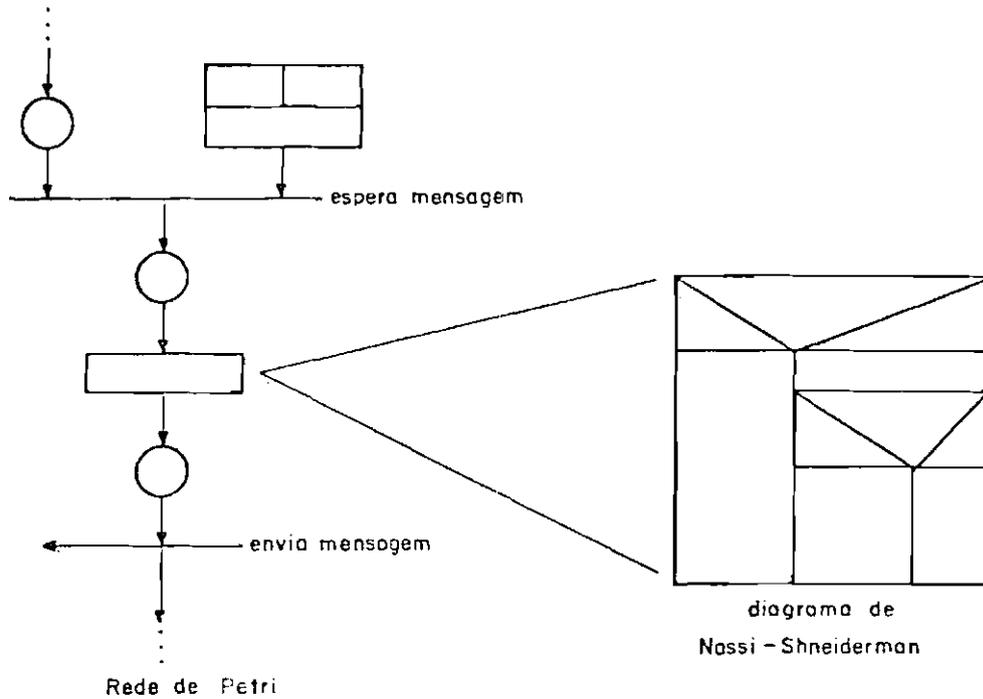


Fig. 4.1 - Representação gráfica de programas.

Na atividade de identificação das transações que realizam as funções especificadas, é feita uma verificação exaustiva para cada função. A documentação necessária a essa atividade corresponde à definição, para cada função, das transações que constam no protocolo estabelecido entre os processos envolvidos.

A Figura 4.2 apresenta um exemplo de representação das transações que compõem uma função lógica de pedido de data e hora certa em dois níveis de detalhamento. Através dessa função o sistema fornece, a pedido do operador, as informações solicitadas, com data (dias, mês e ano) e horário (horas, minutos e segundos). Para a execução deste comando, a entrada e a saída são feitas através de uma console ligada a um nó Concentrador de Terminais (CT), que se comunica com o nó Gerenciador de Entrada e Saída (GES). Cada um desses processadores possui um grupo funcional de processos. Na Figura 4.2.a estão representadas as transações entre os processadores e na Figura 4.2.b detalham-se as transações entre os processos.

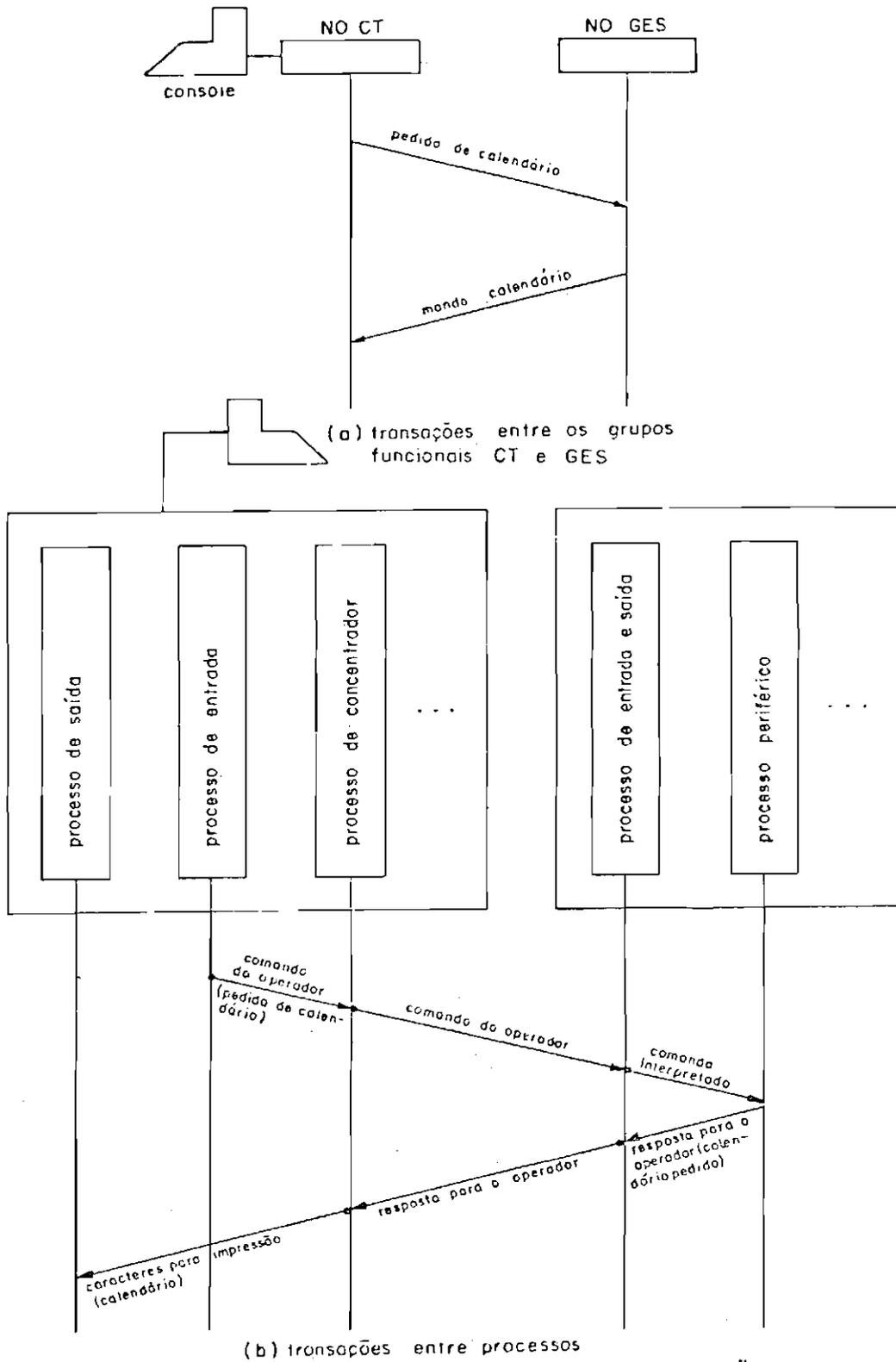


Fig. 4.2 - Representação das transações que compõem a função "pedido de calendário",

A atividade (2) é acompanhada de verificações de coerência entre as transações definidas no protocolo e aquelas integrantes da especificação de cada grupo funcional e cada processo.

A atividade de acompanhamento manual das transações é feita com base nas redes de Petri dos processos. Para cada função realiza-se uma verificação das transações entre os processos envolvidos; analisando a coerência de tipos e estruturas internas das mensagens enviadas por uns e recebidas por outros. Nesta atividade busca-se identificar as possíveis ocorrências de "deadlock" que podem ter causa em erros das transações ou em seqüências não previstas (Cunha, 1980).

Este trabalho não se propõe a detalhar a metodologia de desenvolvimento das três primeiras atividades. A atenção está voltada em especial para os testes, sua metodologia, análise dos resultados e documentação. Traçados os aspectos fundamentais para cada atividade, sem os quais os testes não poderiam ser realizados com eficiência, far-se-á a apresentação da metodologia de testes.

#### 4.3 - DIRETRIZES BÁSICAS PARA O DESENVOLVIMENTO DOS TESTES

A metodologia de testes de "software" do NPD baseia-se em três regras gerais, a saber:

- 1) técnica "bottom-up", partindo-se sempre do particular para o geral, da parte para o todo;
- 2) testes funcionais, tendo por critério para sua programação a identificação das funções especificadas e das funções definidas nos diversos níveis de detalhamento do projeto;
- 3) testes concebidos de forma a levar em conta a característica não seqüencial dos programas.

As duas primeiras regras representam escolhas entre opções possíveis, nada possuindo de absolutas. No entanto, sua escolha se baseia em critérios práticos, buscando combinar os objetivos dos testes com

facilidades para sua execução. São apresentados a seguir esses critérios, juntamente com alguns detalhes técnicos que as três regras envolvem.

Nos testes que exploram a técnica "bottom-up", os módulos de níveis hierárquicos inferiores são testados antes, seguindo-se os de níveis superiores, tais como exposto na Seção 2.3.3. Uma vantagem muito importante dessa técnica é a de não ser necessário criar ambientes para chamadas artificiais (para os módulos chamados e ainda não testados). O desenvolvimento e o teste de eventuais módulos artificiais a serem usados seriam muito trabalhosos no caso de um sistema distribuído de grande porte. Com a técnica "bottom-up" é possível acoplar os próprios módulos já testados para criar ambientes para testar módulos de nível superior. Além disso, conservam-se os resultados já obtidos, os quais podem ser reutilizados ao ser programado e aplicado cada novo teste. Os módulos que contêm processos também podem ser testados através dessa técnica: inicialmente testam-se os processos individualmente, e depois estes são integrados para permitir a aplicação de novos testes. Na Seção 4.4 são apresentadas as etapas a serem executadas nos testes dos processos assíncronos.

A realização de testes funcionais, descritos na Seção 2.3.3, visa direcionar os testes para a verificação do programa quanto à execução correta das funções especificadas e projetadas. Tendo sido a modularização do "software" definida de forma a atribuir funções aos módulos, os testes dessas funções seguem a seqüência estabelecida pela regra (1).

Em relação à terceira e última regra, os seguintes pontos devem ser considerados na elaboração dos testes: os trechos seqüenciais, as transações, o fator tempo na seqüência dos eventos e ainda os estados dos processos envolvidos.

Convém que os trechos seqüenciais de um módulo, ou seja, aqueles que não apresentam transações, sejam testados com antecedência (antes de serem envolvidos no teste as transações que o módulo possui).

Esses testes têm como principal objetivo analisar as influências da execução daqueles trechos do programa sobre o estado da base de dados e devem ser aplicados mediante a utilização de técnicas tradicionais de testes de programas seqüenciais, sucintamente resumidos na Seção 2.3.3 (Huang, 1975; Howden, 1981a, 1981b). Assim sendo, neste capítulo não são apresentadas as metodologias para realização dos testes de tais trechos, no entanto visando manter uma coerência metodológica, as restrições impostas pelas três regras propostas devem ser respeitadas também nestas partes do programa.

Efetuados os testes dos trechos seqüenciais, o passo seguinte é o teste das transações, as quais devem ser estimuladas dinamicamente depois de realizadas e validadas as atividades (2) e (3) (Seção 4.2) para aquelas que estiverem em foco. Na Seção 4.4 tecem-se as considerações necessárias para efetuar a análise das transações.

O tempo é um fator importante na execução de processos concorrentes, pois deve ser considerado como uma variável de grande influência na execução e no comportamento das funções; por isso, a programação de cada teste deve considerar os possíveis efeitos de sua variação e também a influência da presença do próprio programa de teste no comportamento dinâmico do programa em execução.

O estado de cada processo está em contínua mudança durante a execução do programa. Dependendo do estado em que o processo se encontra (cujo retrato está no conteúdo da sua base de dados) a ocorrência de um evento no sistema pode ocasionar efeitos variados no seu comportamento. Conhecer e considerar o estado inicial e a dinâmica de estados de cada processo é condição necessária para o sucesso dos testes. Esta preocupação existe também para testes de programas seqüenciais; no entanto, torna-se mais crítica no caso de programas paralelos, com vistas nas dificuldades de manter sob controle o estado dos processos durante todo o período em que o teste é aplicado.

A Figura 4.3 apresenta as bases para a realização dos testes do NPD. O detalhamento das etapas que compõem os testes deve-se apoiar nas três regras acima apresentadas e será feito a seguir.

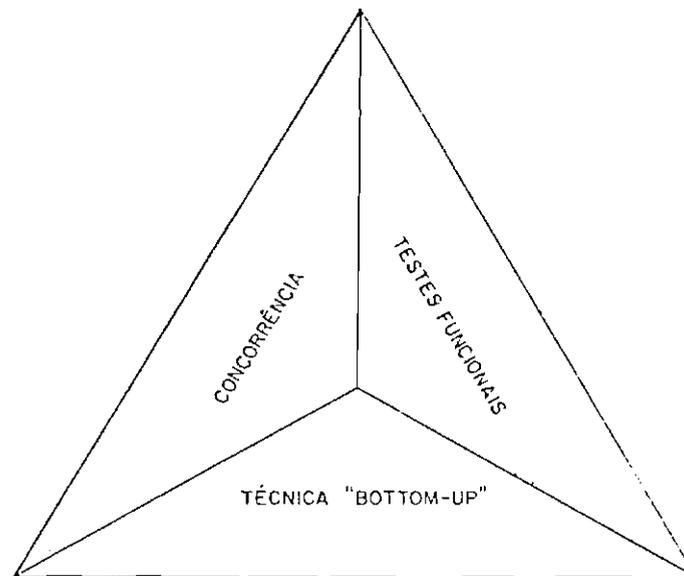


Fig. 4.3 - Bases para a realização dos testes do NPD.

#### 4.4 - ETAPAS DO DESENVOLVIMENTO DOS TESTES

Os testes do "software" completo compreendem basicamente três etapas:

- 1) testes individuais dos processos, em que cada processo é isolado dos demais para testar as funções a que se destina;
- 2) testes parcialmente integrados, em que os processos que trocam mensagens ou que utilizam áreas de dados comuns são agrupados para testar funções que os envolvem;
- 3) testes totalmente integrados, em que todos os processos são incluídos para testar as funções globais.

É feito a seguir um detalhamento de cada uma dessas etapas.

### 1ª Etapa: Testes Individuais dos Processos

Os testes individuais de cada processo constituem a primeira etapa na seqüência de testes do "software" completo.

O objetivo desta etapa é avaliar o comportamento externo do processo na execução das funções das quais participa. Para cada função a ser executada, procura-se analisar as mensagens que implementam as transações que o processo envia ou recebe enquanto executa a função em questão.

Esta etapa requer uma participação intensa no programador, pois depende de um alto grau de acompanhamento na execução do programa por parte do programador, durante a execução dos testes. Isto se deve ao fato de não haver, fisicamente, processos-destino para as mensagens enviadas, nem processos-origem para as mensagens esperadas, cabendo ao programador analisar os conteúdos daquelas e sintetizar os conteúdos destas. Para efetivar os testes desta fase é intensamente utilizado o Sistema de Depuração de Programas Paralelos (SDPP) (Melnikoff, 1982), que permite o isolamento do processo e a simulação de suas interfaces, através da participação do programador, por meio de um terminal de vídeo e de um teclado.

A Figura 4.4 apresenta esquematicamente a relação entre o programador, o SDPP e o processo em teste (P1). Na Figura 4.4a, aparecem os processos na sua configuração real (que, no caso das funções em teste, envolve três nós), com a indicação das trocas de mensagens existentes entre eles e, na Figura 4.4.b, a configuração para teste de P1, sem a presença dos processos P2, P3 e P4 e com o SDPP instalado no nó I. As mensagens a serem recebidas por P1 são geradas pelo programador e enviadas a P1 através do SDPP. As mensagens enviadas por P1 a qualquer outro processo são absorvidas pelo SDPP e apresentadas no terminal para serem observadas pelo programador.

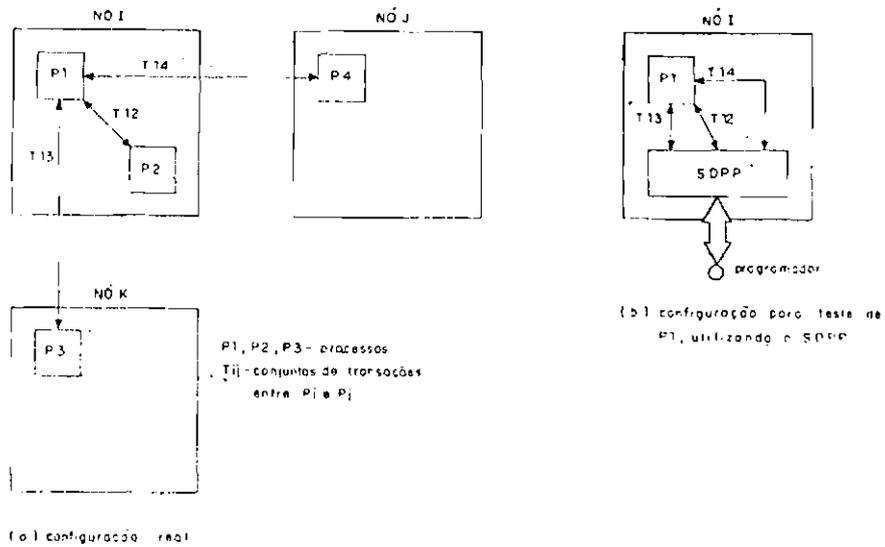


Fig. 4.4 - Uso do SDPP no teste de um processo.

## 2ª Etapa: Testes Parcialmente Integrados

Processos já testados individualmente são submetidos a testes integrados das funções que os envolvem (para cada função participam do teste processos que utilizam áreas de dados comuns ou que trocam mensagens entre si). O objetivo dessa fase é analisar o comportamento das interfaces entre os processos durante a execução das funções em exame.

Para cada grupo de processos a ser testado identificam-se as funções que os envolvem e estabelecem-se prioridades para os respectivos testes, obedecendo ao critério de complexidade crescente. Aos poucos, sequencial ou paralelamente, vão sendo testadas todas as funções e todos os grupos de processos.

O SDPP permite, também, nos testes parcialmente integrados, a monitoração das trocas de mensagens pelo programador. As transações entre os processos em testes são observadas e analisadas, podendo ser alteradas para observar os efeitos de tais modificações. As transações que envolvem processos não ativados no teste devem ser testadas de maneira análoga ao que foi descrito para os testes individuais, ou seja, explorando a presença do programador para gerar as mensagens que seriam enviadas por tais processos e para observar e acompanhar o conteúdo das mensagens que seriam por eles recebidas.

Na Figura 4.5 apresentam-se três situações que ilustram o uso do SDPP no teste de processos integrados. Na Figura 4.5.a tem-se a configuração real; na Figura 4.5.b o teste de dois processos de um mesmo nó, com o SDPP; na Figura 4.5.c o teste de dois processos residentes em nós distintos, (neste caso, os programas do SDPP são carregados nos dois nós envolvidos, podendo ser monitorados paralelamente por dois programadores), na Figura 4.5.d. o teste de processos residentes em nós distintos, com uso do SDPP em apenas um deles.

Um ponto que não pode ser ignorado nos testes que envolvem o SDPP é que a realação entre os tempos de execução dos processos é radicalmente alterada, o que, em geral, introduz importantes diferenças no comportamento do programa em teste.

A sequência de testes a serem efetuados baseia-se na complexidade, no número de processadores, no número de processos, no número de transações envolvidas nas funções visadas e na conectividade entre os processos. Partindo de funções simples e básicas, que envolvem apenas dois processos com poucas transações, aos poucos o âmbito dos testes se amplia para englobar funções mais complexas, sempre fazendo uso dos resultados de testes já realizados.

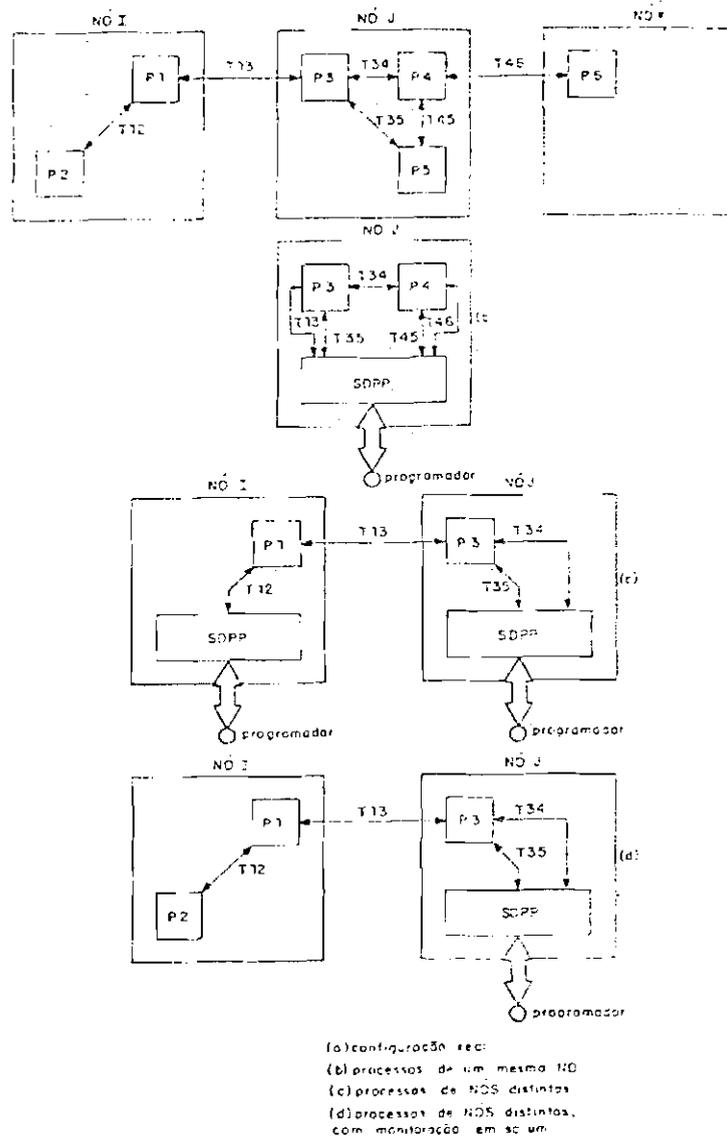


Fig. 4.5 - Uso do SDPP em testes integrados.

Nos primeiros testes parcialmente integrados participam, em geral, apenas processos de um mesmo n $\bar{o}$ , devido à conectividade freqüentemente existente entre eles, em função da utilização de uma base de dados comum. Nesses casos os processos são interdependentes, pois as variáveis acessadas por uma função que envolva um deles poderão ser utilizada (ou modificada) por outra função que envolva o outro. A execução paralela dessas funções traz influências mútuas que seriam, se não analisadas com antecedência, de difícil diagnóstico. É necessário, portanto, levar em consideração a existência desse tipo de conectividade na programação dos testes.

A Figura 4.6.a apresenta um caso para análise. Supondo que os dois processos do n $\bar{o}$  I utilizam variáveis comuns, enquanto os do n $\bar{o}$  J não possuam esse tipo de vínculo, uma sequência possível para os testes daqueles processos seria, após os testes individuais:

- 1) P1/P2, P1/P3,  
P4/P5, P4/P6;
- 2) P1/P2/P3,  
P4/P5/P6.

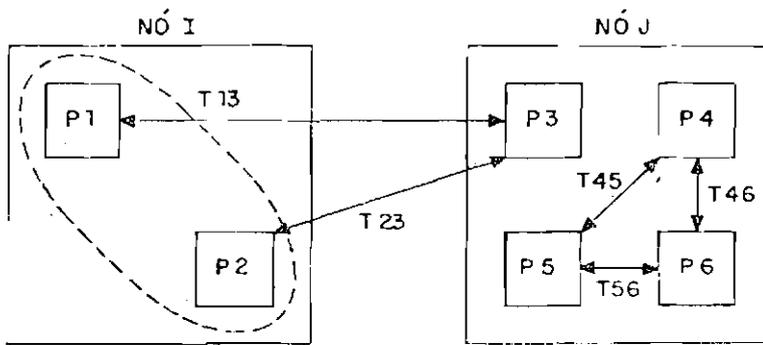
Os primeiros testes integrados, neste exemplo, reuniram P1 e P2, apesar de esses processos não trocarem mensagens entre si. Esse teste também é feito com uso do SDPP, como esquematizado na Figura 4.6.b.

Considerando que a situação apresentada na Figura 4.6 é comum, ou seja, os processos residentes em um mesmo n $\bar{o}$  utilizam freqüentemente áreas de dados comuns, pode-se estabelecer como critério que os primeiros testes parcialmente integrados sejam entre processos de um mesmo n $\bar{o}$ . Desta forma, a segunda etapa seria cronologicamente subdividida em duas partes, a saber:

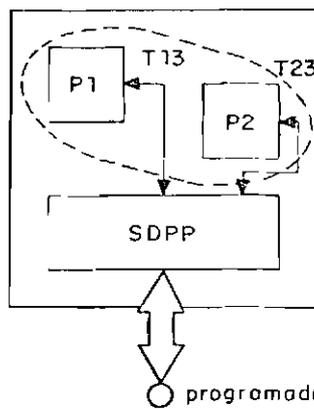
- 1) testes parcialmente integrados envolvendo processos residentes em um único n $\bar{o}$ , e

2) testes parcialmente integrados envolvendo processos residentes em nós distintos.

Não se incluem na primeira parte, os processos residentes em um mesmo nó, que não possuam variáveis compartilhadas, nem troquem mensagens. Por outro lado, as interfaces já testadas na primeira parte podem ser utilizadas com segurança nos testes da segunda.



(a) configuração real



P1 - P6 - processos  
Tij - conjunto de transações  
entre Pi e Pj

(b) configuração para teste de P1 e P2

Fig. 4.6 - Teste de processos com variáveis compartilhadas.

A Figura 4.7 apresenta um exemplo que esclarece a utilização do critério acima estabelecido.

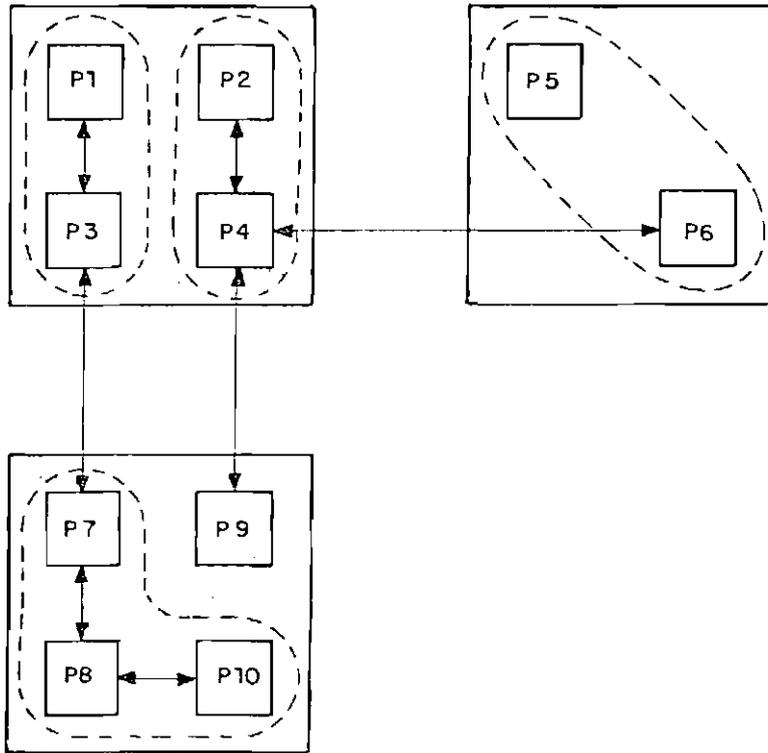


Fig. 4.7 - Testes parcialmente integrados.

Os seguintes conjuntos de processos possuem variáveis comuns entre si: P1/P3, P2/P4, P5/P6 e P7/P8/P9. Assim sendo, uma seqüência possível para os testes parcialmente integrados, após completados os testes individuais, seria:

- 1) P1/P3, P2/P4,  
P5/P6,  
P7/P8, P8/P10;
- 2) P7/P8/P10;

- 3) P1/P3/P7/P8/P10,  
P2/P4/P5/P6/P9.

Os processos que não participem de funções comuns e não usem base de dados comum não são testados em conjunto na fase de testes parcialmente integrados. Apesar disso, essa fase é a mais longa e trabalhosa das três.

### 3ª Etapa: Testes Totalmente Integrados

A fase final dos testes consiste em testar as funções globais, com todos os processos, dos diversos nós, ativados. É analisado nesta etapa, o comportamento dinâmico do sistema, com as diversas funções trabalhando concomitantemente. Para isso observam-se:

- 1) As funções globais do sistema, ou seja, as funções constantes da especificação funcional. A programação dos testes integrados é feita com base na identificação das funções especificadas, que, em geral, envolvem vários processos e vários nós. A execução correta dessas funções é observada para uma gama variada de dados de entrada, definidos de forma a cobrir as situações de interesse, do ponto de vista do usuário final.
- 2) Os efeitos das execuções dessas funções, ou seja, sua influência na base de dados e na utilização dos recursos. Verificada a consistência do conteúdo das estruturas de dados do sistema, esta pode ser considerada coerente em relação às atividades relacionadas com a manutenção de tais estruturas.
- 3) A recuperação do sistema em caso de falhas e reconfiguração. A recuperação do sistema após a detecção de falhas em periféricos e processadores dedicados, já testada com a participação dos nós envolvidos em cada caso, recebe atenção específica na fase dos testes de integração final, verificando-se a correção do software em relação ao cumprimento de todos os requisitos referentes à recuperação de falhas. A reconfiguração automática do sistema, em

caso de falha de um processador, é testada para diversas situações simuladas, provocando-se falhas simuladas de vários tipos, e estudando-se o comportamento do sistema do ponto de vista de sua recuperação em relação a esta falha.

- 4) Os "gargalos" do sistema, ou seja, processos, processadores ou periféricos cuja carga de trabalho é excessiva, fazendo com que seja degradada a velocidade de execução normal das diversas funções do sistema.
- 5) Os efeitos da aplicação dos limites de carga de trabalho estabelecidos na especificação do sistema. Aplicando a carga máxima especificada para os diversos parâmetros do sistema, pode-se observar no comportamento do sistema os efeitos, individuais ou conjuntos, de serem tais limites atingidos.

Levando em consideração a grande variedade de funções a serem testadas nesta etapa, torna-se conveniente estabelecer uma programação de testes, baseada na sua decomposição em níveis funcionais hierárquicos. Essa programação é discutida na Seção 4.5.

A realização dos testes integrados constitui o último passo no processo de testes do "software", mas não se mostra eficaz a não ser que as fases anteriores tenham sido cuidadosamente elaboradas, com as necessárias precauções e dentro de um rígida sistemática. As trocas de mensagens entre processos, as conexões entre os módulos, o fluxo e a estrutura de armazenamento dos dados devem ter sido detalhadamente analisados de acordo com a proposta apresentada.

#### 4.5 - TESTES EM NÍVEIS FUNCIONAIS

Embora a metodologia proposta para os testes de "software" do NPD esteja baseada em testes funcionais, de acordo com o que foi estabelecido na Seção 4.4, a estrutura interna dos programas tem forte influência na programação dos testes, já que as funções internas (definidas no projeto) são também testadas nesta ocasião. Nos primeiros tes

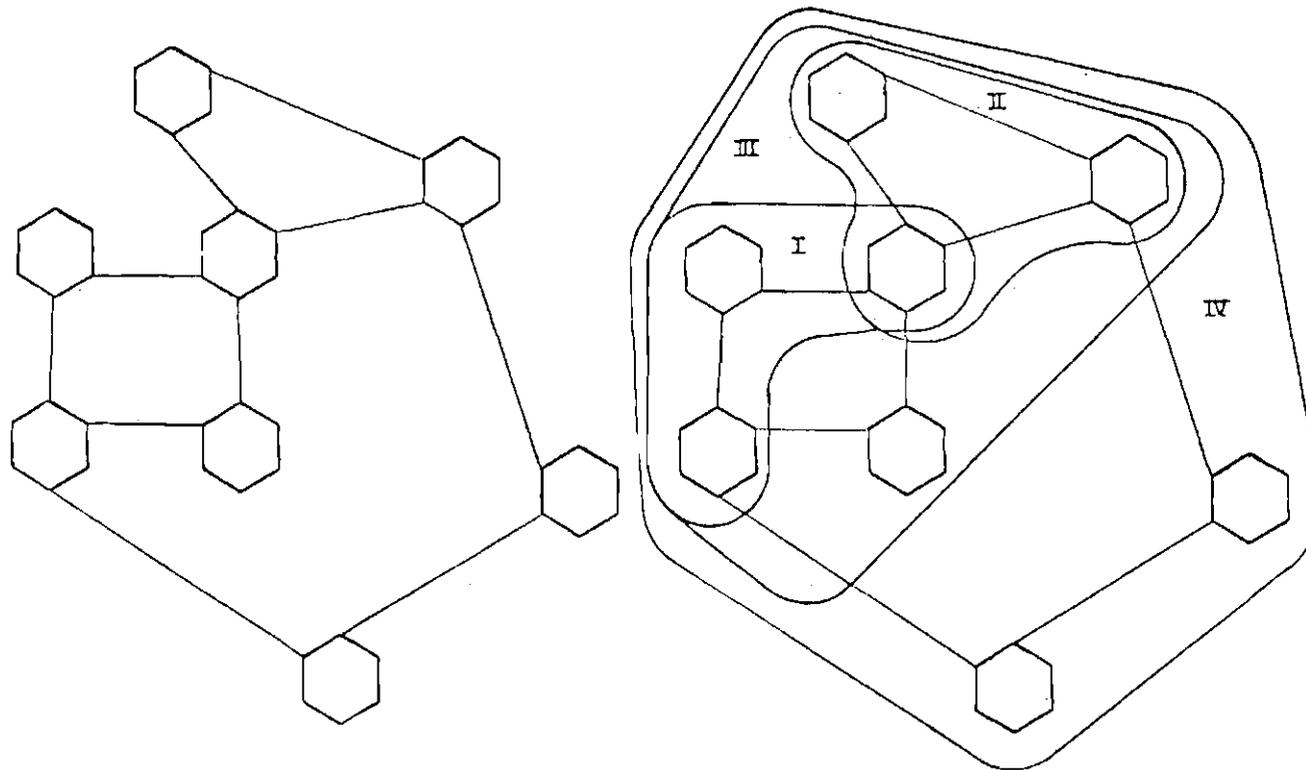
tes, principalmente, realizados com módulos de baixo nível hierárquico, as funções testadas apresentam ampla relação com a estrutura dos programas. Os testes nesta ocasião, são baseados em partições estruturais do "software", como mostra o exemplo da Figura 4.8.

À medida que se desenvolvem os testes parcialmente integrados, as funções de alto nível (normalmente aquelas que constam da especificação funcional) passam a ser analisadas. Nos testes totalmente integrados toda a programação de testes é feita com base nessas funções. O sistema pode então ser visto como uma caixa preta, onde as entradas aplicadas determinam as saídas.

Ao ser atingido este ponto, em que a estrutura básica do "software" pode ser considerada satisfatoriamente correta, e as interfaces entre os processos relativamente consistentes, ou seja, na ocasião em que o sistema começa a operar como um todo, estabelece-se uma programação de testes das funções especificadas, divididas em níveis funcionais conforme a especificação de operação, vista sob o ângulo do usuário final. Estes níveis são escolhidos de tal maneira que cada um adicione novas funções às apresentadas nos níveis anteriores, como mostra o exemplo da Figura 4.9.

A Figura 4.10 apresenta esquematicamente uma possível relação hierárquica entre níveis funcionais. Neste exemplo, as funções do nível 1 dão suporte às funções dos níveis 2 e 3 que, por sua vez, dão base para os testes de nível 4, e assim por diante.

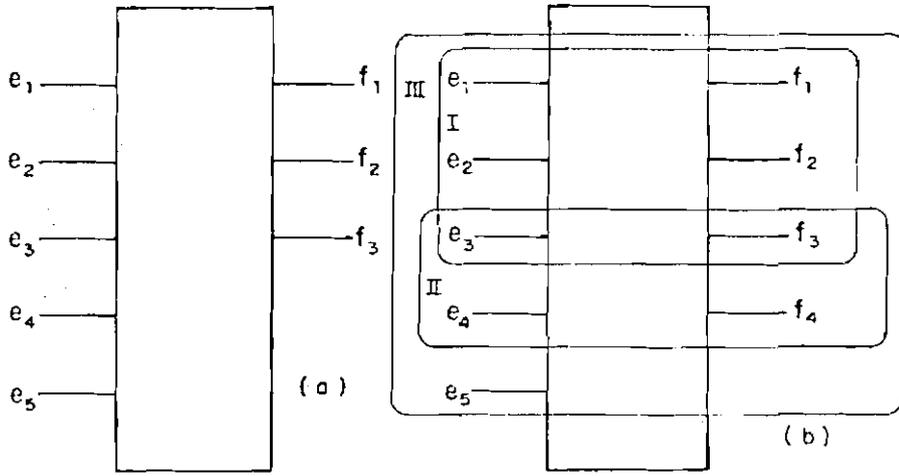
Como vantagens desse tipo de programação tem-se, além da independência dos testes em relação à estrutura do "software", a definição de passos com objetivos precisos e unificados a serem alcançados pelos programadores que realizam os testes e a depuração dos programas.



(a) Estrutura do software

(b) Partição do software em níveis estruturais

Fig. 4.8 - Níveis estruturais.



(a) funções do software (b) partição do "software" em níveis funcionais.

Fig. 4.9 - Níveis funcionais.

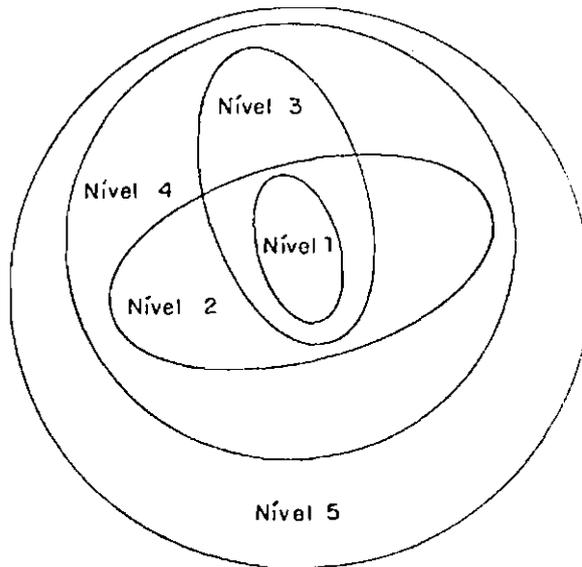


Fig. 4.10 - Relação hierárquica entre os níveis funcionais.

São apresentados a seguir os critérios gerais adotados para a efetivação da programação em níveis. Os testes devem adotar a seguinte ordem:

- 1) testes da interface homem-máquina com apresentação de dados estáticos através dos periféricos;
- 2) testes dinâmicos, com dados externos simulados, apresentando os resultados através de periféricos, sem a execução de atualizações das informações dinâmicas do sistema.
- 3) atualização dinâmica de tabelas e relatórios e sua apresentação através de periféricos;
- 4) teste de execução de operações de atuação sobre o processo a ser controlado pelo sistema;
- 5) testes do tratamento automático de falhas de periféricos e de processadores dedicados a funções específicas;
- 6) testes da execução automática das operações de reconfiguração (ou seja, de substituição de um processador por outro em caso de detecção de falha irreversível).

#### 4.6 - DEPURAÇÃO DE ERROS

Nos testes de "software" distribuído de grande porte a depuração de erros é uma prática contínua e necessária, já que em grande quantidade de casos a busca e correção de novos erros só é possível se um erro encontrado for previamente corrigido. A dificuldade em reproduzir erros é também um forte motivo para que os testes sejam sempre acompanhados da atividade da depuração dos erros detectados.

Manifestado um erro, deve-se encontrar a sua causa, e corrigi-lo. Como instrumentos importantes para a realização dessas duas atividades, o programador dispõe de análise dos estados dos nós envolvidos e da documentação do "software". São feitas a seguir algumas considerações a respeito dessas atividades.

A busca da causa de um erro é mais simples nas primeiras etapas da fase de testes, tornando-se gradualmente mais complexa à medida que cresce a integração entre os módulos. Na primeira etapa, por exemplo, em que um único processo é testado e na qual toda a execução é monitorada, através do uso do SDPP, a maioria dos erros que ocorrem pode ser imediatamente notada pelo programador. Através da análise de uma listagem do programa que implementa o processo em teste, deve-se localizar o ponto em que o erro foi detectado e, em seguida, identificar a sua causa que, em geral, pode ser inferida como resultado da análise dos trechos vizinhos do programa, e de verificações, diretamente na memória, do último valor assumido pelas diversas variáveis envolvidas. Se o erro não for detectado rapidamente, ou se houver dificuldades em determinar a sua causa, o teste pode ser repetido, reproduzidas as mesmas condições do teste anterior, e procurando-se localizar detalhes que possam levar ao diagnóstico do erro em questão. Essa repetição de situação do teste é sempre possível nos testes da primeira etapa, em que os dados são todos estáticos.

Na segunda etapa, a causa dos erros nem sempre é encontrada com facilidade devido à interação existente entre vários processos. Nos testes em que a execução de todos os processos esteja sendo monitorada, procede-se como foi feito na primeira fase, analisando, para cada um deles, o ponto em que o processo parou (com auxílio de listagem) e os estados das variáveis envolvidas, com atenção especial às variáveis compartilhadas. A análise se faz, em geral, procurando respostas para as seguintes questões: qual é o erro?, a partir de que ponto a execução está incorreta?, que processo causou o primeiro erro? e qual a causa do primeiro erro? Muitos erros não podem ser localizados em um único processo, pois são causados por problemas de sincronizações incorretas ou pela ocorrência de mensagens incoerentes (por exemplo: um processo A espera mensagem do tipo X, enquanto um processo B envia para ele uma mensagem do tipo Y).

Nos testes da segunda etapa, em que um ou mais processos não estejam sendo monitorados (inclui-se o caso em que nenhum processo é monitorado), a busca da causa do erro geralmente exige uma análise dos

estados dos processos envolvidos, realizada combinando informações tiradas das bases de dados dos processos e do Sistema de Comunicações (SC). Detectado um erro, interrompe-se o processamento, através de um comando do operador, de todos os nós ativos e inicia-se a análise a partir da base de dados do SC de cada nó, através do qual se definem os estados dos processos no momento da parada (em execução, parado e esperando qualquer mensagem, parado e esperando mensagem de tipo específico, etc.) e os endereços de retorno constantes das pilhas dos processos (chamadas de sub-rotina cuja execução não foi completada). Definida a situação em que o sistema teve sua execução interrompida, situação esta que deve ser a mais próxima possível daquela na qual ocorreu o erro, inicia-se a análise das bases de dados dos processos que possam tê-lo causado, até que a causa seja encontrada. A Figura 4.11 exemplifica uma situação típica na atividade de depuração de erros. Supõe-se que a pilha de um dos processos analisados indica que ele está bloqueado, esperando uma mensagem de tipo específico e seu último endereço de retorno é C857H. Utilizando a lista de relações cruzadas dos endereços com variáveis e comandos do processo (Figura 4.11.a), descobre-se rapidamente a posição em que o processo está bloqueado. Pela Figura 4.11.b, nota-se que o processo está bloqueado, esperando uma mensagem de tipo específico. Inicia-se, então, uma análise da listagem e da base de dados do processo.

Muitas vezes não é possível deduzir a causa do erro pela técnica exposta acima. Isso acontece sobretudo nos casos em que os estados dos processos tenham sofrido mudanças significativas após o erro, não sendo mais possível reconstituí-los a partir da simples análise das bases de dados. Neste caso, é necessário realizar novas execuções, adicionando instrumentos para monitoração (SDPP) ou simplesmente fazendo uso de "breakpoints". Esta última opção possui a vantagem de permitir testes mais rápidos, sem a necessidade de participação intensa do programador. Além disso, evita a influência que o SDPP traz ao tempo de execução e à sincronização dos processos. Como a maior desvantagem dos "breakpoints" tem-se a impossibilidade de acompanhar todas as transações que ocorrem (muitas vezes o erro não está localizado onde o programador supõe inicialmente). A cada nova parada, volta-se à análise dos estados dos processos, até que a causa do erro seja descoberta e o erro eliminado.

LIN	256	C613H	LIN	257	C616H	LIN
LIN	259	C626H	LIN	260	C638H	LIN
LIN	262	C642H	LIN	263	C64AH	LIN
LIN	265	C65AH	LIN	267	C667H	LIN
LIN	269	C681H	LIN	270	C681H	LIN
LIN	272	C686H	LIN	273	C695H	LIN
LIN	275	C6ACH	LIN	276	C682H	LIN
LIN	278	C68DH	LIN	279	C6C3H	LIN
LIN	281	C6D5H	LIN	282	C6DAH	LIN
LIN	284	C713H	LIN	285	C718H	LIN
LIN	287	C720H	LIN	288	C721H	LIN
LIN	290	C72EH	LIN	291	C72FH	LIN
LIN	293	C736H	LIN	295	C743H	LIN
LIN	297	C758H	LIN	298	C763H	LIN
LIN	300	C773H	LIN	301	C77DH	LIN
LIN	303	C795H	LIN	304	C798H	LIN
LIN	306	C7A3H	LIN	307	C7A3H	LIN
LIN	309	C7A9H	LIN	310	C7ABH	LIN
LIN	312	C7C6H	LIN	313	C7C9H	LIN
LIN	315	C7DAH	LIN	316	C70EH	LIN
LIN	318	C7E6H	LIN	319	C7EBH	LIN
LIN	321	C7FEH	LIN	322	C807H	LIN
LIN	324	C815H	LIN	325	C816H	LIN
LIN	327	C826H	LIN	328	C82BH	LIN
LIN	330	C83BH	LIN	331	C842H	LIN
LIN	333	C857H	LIN	334	C860H	LIN
LIN	337	C866H	LIN	338	C866H	LIN
LIN	340	C897H	LIN	342	C89CH	LIN
LIN	344	C8C0H	LIN	345	C8D1H	LIN
LIN	347	C8E1H	LIN	348	C8ECH	LIN
LIN	350	C8F3H	LIN	351	C8F4H	LIN
LIN	353	C8FEH	LIN	354	C909H	LIN



(a) lista de relações cruzadas dos endereços com variáveis e comandos do processo.

Fig. 4.11 - Depuração de erros.

(Continua)

Fig. 4.11 - CONCLUSÃO - (b) Listagem do processo.

```

325 2      END;

          /*****
          * PEDESARQUIVOSFLOPPY: PEDE AO OPERADOR O NOME DO PROXIMO ARQUIVO *
          *                               NO 'FLOPPY'. *
          *****/

326 1      PEDESARQUIVOSFLOPPY: PROCEDURE;

327 2      INICIO:

          CALL TRANSFERE(.M2, .MI(1), .MCENV$PERFPES(6));
328 2      MI(0) = PERF;
329 2      MCENV$PERFPES(4) = MG2$PERFPES; /* .PEDE NOVO NOME DO ARQUIVO NO
330 2      CALL ENVIA(PM1, .MCENV$PERFPES); /* 'FLOPPY' P/ OPERADOR

331 2      MCSPRS$PESPRF(7) = TZENLI * 4 + PERF; /* .ESPERA RESPOSTA DO OPERADOR
332 2      CALL TEMPO (MCSPRS$PESPRF (7), (ZENLI));
333 2      CALL SPTIF(.PM1, .MCSPRS$PESPRF);

334 2      IF MI(0) = ABORTO /* .SE OCORREU ABORTO
          THEN /* ENTÃO
335 2          DO; /* .SINALIZA
336 3          NERRU = ABORTO; /* .RETORNA
337 3          RETURN;
338 3          END;

339 2      IF MI(1) = ':' AND MI(2) = 'F' AND /* .SE OPERADOR RESPONDEU C/ ':F'
          MCSPRS$PESPRF(6) = 3
          THEN /* ENTÃO
340 2          FIM = TRUE; /* .SINALIZA FIM DE OPERACAO
          ELSE /* CASO CONTRARIO

```

Nos testes da terceira etapa, a busca da causa de um erro é ainda mais complexa. É grande o número de nós e o número de processos envolvidos. Não é possível, em geral, utilizar o SDPP (pela interferência que traz à velocidade da execução), restando as bases de dados para análise e, em casos de se poder repetir a situação de erro, o uso de "breakpoints" para controlar a execução. Comandos de ativação, reativação e suspensão dos processos podem também ser utilizados. O procedimento adotado em caso de erro é o seguinte:

- 1) interrompe-se a execução de todos os nós através do comando de suspensão da execução dos processos;
- 2) analisam-se os estados dos processos verificando quais deles estão envolvidos no erro ocorrido;
- 3) analisam-se as bases de dados do SC e desses processos, visando descobrir a causa do erro, sempre com apoio das listagens.

Se novas execuções forem necessárias é fundamental reconstituir as condições em que o erro ocorreu. Se isso não for possível, preparam-se apenas as condições para uma análise posterior (quando o erro voltar a acontecer em novos testes). Para isso, podem mais uma vez ser utilizados os "breakpoints" em pontos de interesse para a análise.

Determinada a causa de um erro, ele deve ser corrigido. Em muitos casos essa correção é simples e direta, afetando apenas o código objeto (erros de codificação). No entanto existem erros cuja correção afeta as estruturas do projeto (lógica e de dados) e até mesmo a especificação funcional (inconsistências). Como já exposto na Seção 2.1, esses são os erros mais caros pois representam um retrocesso a estágios do desenvolvimento correspondentes a passos já vencidos do projeto do "software". Qualquer que seja o caso, a correção deve ser feita, a documentação alterada e os testes de funções afetadas que já tenham sido realizados devem ser reaplicados. Todo esse processo é trabalhoso e longo, mesmo para corrigir pequenos erros, sobretudo porque não é possível desenvolver o "software" utilizando o próprio equipamento que o

"software" está sendo implantado, sendo necessário, no caso, recorrer a outros equipamentos, nem sempre disponíveis prontamente.

Para evitar que os testes sejam interrompidos após a localização de cada erro, adotam-se muitas vezes correções provisórias alterando o código de máquina na própria memória do no (e, se for de interesse, salvando o novo código em arquivo em disco para uso em testes subseqüentes). Esta prática deve ser realizada com extremo cuidado pois em geral acarreta sérios problemas. Por exemplo, se não for imediatamente documentada a correção que deve ser feita, ela não será implantada nas novas versões do programa, e o erro voltará a ocorrer. Assim sendo, ao fazer uma correção diretamente no código objeto, prática desaconselhável, deve-se:

- 1) em documento próprio anotar a correção feita, para uso nos testes, bem como a situação anterior para um possível retorno caso a correção não tenha sucesso;
- 2) em documento para uso no desenvolvimento, anotar a solução adotada para o erro, visando sua inclusão na geração de novas versões e a alteração da documentação (mesmo no caso de não haver ainda uma solução definitiva, a anotação da solução adotada deve ser feita).

Um outro problema que pode ocorrer é a introdução de novos erros na ocasião da correção direta do código objeto. Correções de grande porte devem ser evitadas com maior empenho, pois são as mais sujeitas a erros e as que causam maiores problemas subseqüentes. Eis alguns dos casos em que aquelas correções podem ser aplicadas com mínimo risco:

- 1) correções de constantes e de valores iniciais de variáveis;
- 2) pequenas correções em matrizes constantes ou em valores iniciais de matrizes variáveis;

- 3) pequenas correções no código, tais como alterações de números, inclusão de instruções e exclusão de instruções existentes.

Após cada correção os testes devem ser reaplicados para verificar a validade da alteração e a eventual introdução de efeitos colaterais.

#### 4.7 - DOCUMENTAÇÃO DOS TESTES

A fase de testes de um "software" distribuído de grande porte é longa e trabalhosa. São muitos os procedimentos de testes que devem ser executados, muitos erros são detectados e corrigidos, muitas pessoas são envolvidas.

A documentação desta fase tem como objetivo disciplinar a realização dos testes e facilitar a absorção dos seus resultados por parte de todos os elementos envolvidos nos testes e no desenvolvimento do "software".

Os documentos podem ser divididos em dois grupos, a saber:

- 1) documentos de programação e controle dos testes;
- 2) documentos de acompanhamento de testes.

Os documentos do primeiro grupo são mantidos por um coordenador dos testes. Registram os diversos testes programados para cada etapa e, para cada teste, o objetivo a atingir, a data prevista para têrmino, a(s) pessoa(s) responsável(eis) e os recursos necessários. Constam também desses documentos quaisquer alterações na programação, jã que todo o controle deve ser feito com base neles.

Os documentos de acompanhamento de testes servem de subsídios para o programador realizar os testes e a depuração, e ainda registrar os resultados obtidos. São os seguintes:

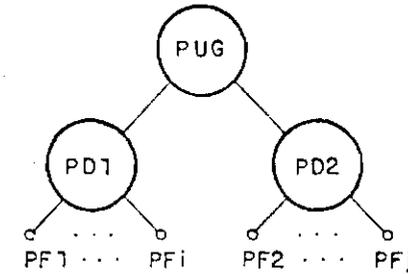
- plano de teste, com os procedimentos que se deve executar na realização de determinado conjunto de testes, os resultados obtidos e observações adicionais;
- correções para uso durante a aplicação dos testes, com a indicação das alterações efetuadas diretamente no código gerado;
- correções para uso por parte da equipe de desenvolvimento, com a indicação das alterações propostas para novas versões dos programas. Correspondem às alterações anotadas no documento anterior;
- testes de aceitação, com a programação de testes funcionais a serem executados eventualmente com a participação do cliente.

Os planos de testes são preparados para prever e organizar a realização de testes em todas as etapas. Podem ser testes estruturais (funções básicas) ou funcionais (funções especificadas). A Figura 4.12 apresenta um exemplo de testes funcionais, que abrange tanto a utilização em regime de consoles e impressoras quanto a recuperação do sistema em caso de falhas desses periféricos ou dos processadores periféricos que os controlam. Os documentos com correções são gerados durante a atividade de depuração dos programas, para uso durante a execução dos testes, devendo ser elaborados de acordo com os critérios apresentados na Seção 4.6. As Figuras 4.13 e 4.14 apresentam exemplos desses documentos.

Os testes de aceitação são procedimentos para verificação de um conjunto de funções especificadas do "software". A Figura 4.15 apresenta um exemplo.

Teste de falhas do NPD

PUG:GES DATA \_\_\_/\_\_\_/\_\_\_  
 PDI:CT1,CT2  
 PERIFÉRICOS (PFI): TTY1,IMP1,TTY2,IMP2



ESTADO	ELEMENTO (S)	ALARME	TESTE	OBSERVAÇÕES
D	CT1	FALHA:TCPP CT1 "data"		
L	CT1			
D	CT2	FALHA:TCPP CT2 "data"		
L	CT2			
D	CT1,CT2	FALHA:TCPP CT1 "data"		
		FALHA:TCPP CT2 "data"		
L	CT1			
L	CT2			
O	TTY1	FALHA:TCPF TTY1 "data"		
L	TTY1			
D	IMP1	FALHA:TCPF IMP1 "data"		
L	IMP1			
D	TTY2	FALHA:TCPF TTY2 "data"		
L	TTY2			
D	IMP2	FALHA:TCPF IMP2 "data"		
L	IMP2			

Obs.: D- Desligar

L- Ligar (ao ligar, teclar na console: AC TCPP PDi ON e/ou )  
 AC TCPF PDi ON

Fig. 4.12 - Plano de teste.

TESTE DE SOFTWARE DO NPD  
CORREÇÃO NO CÓDIGO OBJETO

DATA : 08/10/85  
FUNÇÃO : GD  
VERSÃO : 24/09/85

PG	ENDEREÇO	NOVO CONTEUDO	CONTEUDO ANTERIOR	OBS
1	855E 8560	00 00	00 01	teste só com IDW1
1	855E 8560	01 01	00 01	teste só com IDW2
2	DF3A DF4C FF20	00 00 00 C3 20 FF 0E 18 CD 06 08 01 00 00 C3 4F DF	CD 06 08 01 00 00	para GDID temporização IDW2
3	FF80    856C	21 BC E3 36 00 23 36/00 CD B5 94 C9 CD 80 FF		enviar estação 00 para trem especial.

Fig. 4.13 - Correções para uso durante a aplicação dos testes.



do de carga de disquete. Verificar a impressão da mensagem de erro indicando que o CDF não responde e do alarme de sistema: TCPP - CDF, ambas na console 1. Digitar o comando LC TCPP e verificar o estado do CDF;

PASSO 14 - Ligar o CDF e digitar outro comando de carga de disquete e verificar a impressão da mensagem de erro indicando que o CDF está desligado;

PASSO 15 - Digitar o comando AC TCPP CDF ON. Digitar o comando LC TCPP e verificar o estado do CDF. Digitar um comando de carga de disquete e verificar sua execução.

#### D) Tratamento de Falha Dupla

PASSO 1 - Desligar, ou colocar "off-line", as duas impressoras. Solicitar através do TV, a impressão de uma tabela ou relatório: Verificar a impressão, na console 1, dos alarmes de sistema: TCPF - IMP1 e IMP2. Reconhecer os alarmes no painel;

Fig. 4.15 - Teste de Aceitação.

#### 4.3 - PROPOSTA PARA APERFEIÇOAMENTO DOS TESTES DE "SOFTWARE" DO NPD

O ambiente de testes e depuração de "software" do NPD oferece recursos úteis para a realização dos testes. No entanto, alguns recursos adicionais são necessários para aperfeiçoar este ambiente, trazendo maiores facilidades aos programadores e oferecendo recursos que permitam detectar e corrigir erros mais rapidamente.

Uma das limitações do ambiente de testes disponível é a impossibilidade de desenvolver e corrigir programas na própria máquina, paralelamente à realização dos testes. A cada correção necessária, necessita-se de outros equipamentos para realizá-la, nem sempre disponíveis imediatamente, o que atrasa significativamente o projeto. O "hardware" do NPD fornece todos os recursos para execução paralela de testes e desenvolvimento, sendo necessária apenas a adaptação do SC através de uma extensão que dê capacidade de executar em qualquer número de comandos para gerenciamento dos arquivos da unidade de discos flexíveis e geração de código objeto. A Figura 4.16 apresenta esquematicamente a configuração do NPD com essa ferramenta.

Um outro aspecto operacional merece destaque: nos testes realizados no NPD, são muitos os erros que ocorrem e que não podem ser repetidos, devendo o programador, como único recurso nesses casos, tentar entender o que ocorreu através da análise das bases de dados do SC e do programa, uma tarefa geralmente árdua e muitas vezes sem resultados positivos. A utilização do SDPP pode não ser útil nesses casos, pois não se sabe quando um erro semelhante ocorrerá novamente. A geração automática de "traces", da forma proposta pela ferramenta "geração de traces", apresentada na Seção 3.3.3.2 seria útil para o NPD. Sua utilização poderia ser feita sempre que se inicia um teste em que determinado(s) erro(s) pode(m) acontecer. Esta ferramenta aplica-se ainda melhor nos casos em que a repetição do erro é prevista com certeza. Ocorrido o erro, obtêm-se os arquivos gravados em disco e analisam-se os eventos que antecederam a ocorrência do erro. Com essa ferramenta a depuração de erros ganha em rapidez e exige menos esforços dos programadores.

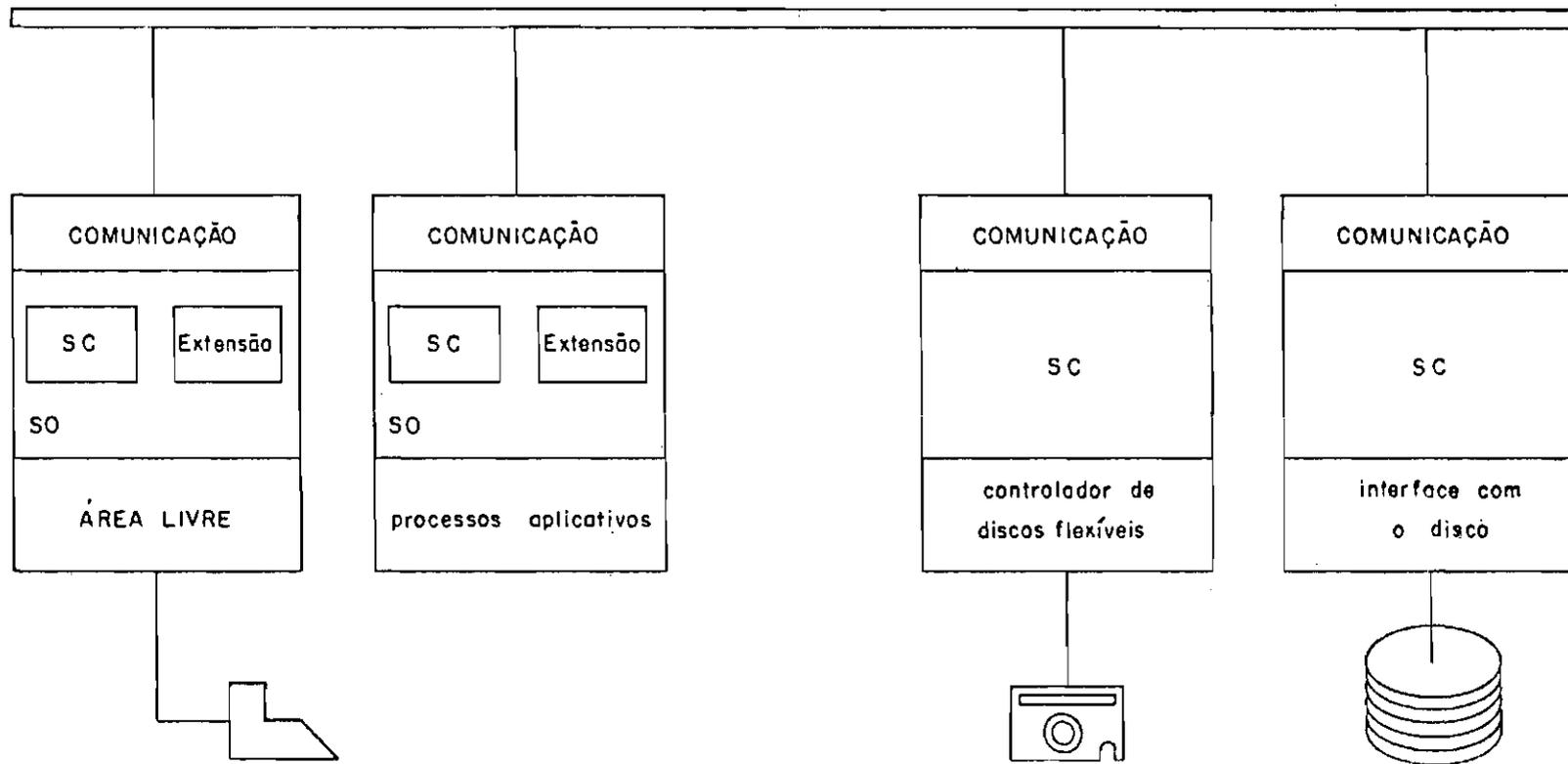


Fig. 4.16 - NPD com recursos para desenvolvimento de "software".

#### 4.9 - CONCLUSÃO

Os testes do "software" para controle de tráfego ferroviário realizados trazem contribuições práticas relevantes. São elas:

- 1) o ambiente de testes concebido com base no próprio "hardware" do NPD, com a inclusão de ferramentas de "software" para controle e monitoração da execução, trouxe facilidades para a realização dos testes e das depurações de erros, notando-se no entanto a necessidade de um investimento maior na criação de ferramentas voltadas para o desenvolvimento de "software" na própria máquina, e para a monitoração dos eventos ocorridos;
- 2) a metodologia com técnica "bottom-up" faz com que muitos erros funcionais sejam encontrados e corrigidos na primeira etapa dos testes, quando é ainda elevado o grau de monitoração da execução por parte do programador. A integração gradativa dos processos permite identificar detalhadamente os erros de interconexão.

Os testes de "software" distribuído, desenvolvidos para sistemas como o NPD, devem manter a mesma linha metodológica, contando, se possível, com as melhorias para o ambiente de testes propostas neste trabalho.

## CAPÍTULO 5

### CONCLUSÃO

Este estudo se caracteriza por apresentar um registro crítico de um trabalho de equipe, documentando os métodos e técnicas utilizados, contrastando-os com aqueles apresentados na literatura. Não é, portanto, fruto de um trabalho individual, pois retrata a experiência prática dessa equipe.

As contribuições da literatura relativas a sistemas de "software" distribuído, bem como a experiência profissional adquirida na realização prática dos seus testes, conduziram à idealização de uma concepção geral de um ambiente de testes e de uma metodologia aplicáveis a sistemas desta natureza. Sua aplicação não está restrita a estruturas particulares de "software" e "hardware", sendo necessário apenas adaptar a cada sistema (ou classe de sistemas) particular as propostas aqui apresentadas.

A experiência vivida com os testes de "software" do NPD mostra a importância do ambiente de testes na detecção e depuração de erros. As atividades realizadas com o auxílio de ferramentas adequadas são mais ágeis e permitem a obtenção de resultados mais rápidos e precisos. A depuração dos programas, durante a fase dos testes totalmente integrados, também pode ser aperfeiçoada através do uso de ferramentas para monitoração.

Quanto à metodologia adotada nos testes do NPD, coerente com a proposta feita em linhas gerais para sistemas distribuídos quaisquer, verifica-se que a detecção e a depuração de erros podem se desenvolver com passos seguros através da divisão da fase de testes em três etapas.

Considerando a perspectiva de crescimento cada vez maior da aplicação dos sistemas distribuídos para realização de atividades di

versas, sobretudo para controle de processos físicos em tempo real, conclui-se que os ambientes de testes a serem desenvolvidos ou aperfeiçoados devem visar a flexibilidade, pois é desejável que num mesmo ambiente possam ser testados vários sistemas distribuídos com "software" aplicativo diferenciado e configuração variável em termos do número de módulos e dos dispositivos utilizados. As ferramentas propostas poderão ser implementadas com base nas propostas do Capítulo 3, nas quais elas são descritas funcionalmente.

O desenvolvimento das técnicas e ferramentas deve tender a dar aos programadores cada vez mais facilidades para realizar os testes e a depuração dos erros. Essas facilidades não devem estar relacionadas apenas com as atividades mecânicas de controle e monitoração, mas também com as tomadas de decisões no encaminhamento dos testes e na depuração de erros. O desenvolvimento de sistemas especialistas para assessorar essas tomadas de decisões é um caminho natural a ser seguido.

## REFERÊNCIAS BIBLIOGRÁFICAS

- AGERWALA, T. Putting Petri nets to work. *IEEE Computer*, 12(12): 85-94, Dec. 1979.
- ALFORD, M.W. Software requirements engineering methodology (SREM) at the age of four. In: COMPUTER SOFTWARE AND APLICATIONS CONFERENCE, 4., Chicago, IL, Oct. 27-31, 1980. *Proceedings*. New York, NY, IEEE, 1980. p.866-874.
- BASU, S.S.; MISRA, J. Proving loop programs. *IEEE Transactions on Software Engineering*, 1(1): 76-86, Mar. 1975.
- BIRMAN, A. On proving correctness of microprograms. *IBM Journal of Research and Development*, 18(3): 250-266, May 1974.
- BIRD, D.L.; MUNOZ, C.U. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3): 229-245, 1983.
- BOEHM, B.W. Software and its impact: a quantitative assessment. *Datamation*, 19(5): 48-59, May 1973.
- BOEHM, B.W. Software engineering. *IEEE Transactions on Computers*, 25(12): 1226-1241, Dec. 1976.
- BOEHM, B.W. *Software Engineering Economics*. Englewood Cliffs, NJ, Prentice-Hall, 1981.
- BOEHM, B.W. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1): 75-88, Jan. 1984.
- BOOCH, G. *Software engineering with Ada*. Menlo Park, CA, Benjamin/Cummings, May 1983.
- BRANSTAD, M.A.; CHERNIAVSKY, J.C.; ADRION, W.R. Validation, verification and testing for individual programmer. *IEEE Computer*, 13(12): 24-30, Dec. 1980.
- CAINE, S.H.; GORDON, E.K. PDL - a tool for software design. In: AFIPS NATIONAL COMPUTER CONFERENCE, 44., Anaheim, CA, May 19-22, 1975. *Proceedings*. Montvale, NJ, AFIPS Press, 1975.
- CHAMPINE, G.A.; COOP, R.D.; HEINSELMAN, R.C. *Distributed computer systems: impact on management, design, and analysis*. New York, NY, North-Holland, 1980.

- CHEN, B.C.; YEH, R.T. Formal specification and verification of distributed systems. *IEEE Transactions on Software Engineering*, 9(6): 710-721, Nov. 1983.
- CUNHA, P.R.F.; LUCENA, C.J.; MAIBAUM, T.S.E. On the design and specification of message oriented programs. *International Journal of Computer and Information Science*, 9(3): 161-192, Jun. 1980.
- DJKSTRA, E.W. The structure of the "THE" multiprogramming system. *Communications of the ACM*, 11(5): 341-346, May 1968.
- ENSLow JUNIOR, P.H. What is a "distributed data processing system"? *IEEE Computer*, 11(1): 13-21, Jan. 1978.
- FAIRLEY, R.E. Modern software design techniques. In: SYMPOSIUM COMPUTER SOFTWARE ENGINEERING, New York, NY, Apr. 20-22, 1976. *Proceedings*. Brooklyn, NY, Polytechnic Press, 1976. p. 11-13 (Microwave Research Institute Symposium Series, 24).
- FLYNN, M. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9): 948-960, Sept. 1972.
- FOSTER, K.A. Error sensitive test case analysis (ESTCA). *IEEE Transactions on Software Engineering*, 6(3): 258-264, May 1980.
- FUJII, M.S. Independent verification of highly reliable programs. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 1., Chicago, IL Nov. 8-11, 1977. *Proceedings*. New York, NY, IEEE, 1977, p.38-44.
- GABOW, H.N.; MAHESHWARI, S.N.; OSTERWEIL, L.J. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, 2(3): 227-265, Sept. 1976.
- GARCIA-MOLINA, H.; GERMANO JUNIOR, F.; KOHLER, W.H. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, 10(2): 210-219, Mar. 1984.
- GERHART, S.L.; YELOWITZ, L. Observations of fallibility in applications of modern programming methodologies. *IEEE Transactions on Software Engineering*, 2(3): 195-207, Sept. 1976.
- GOODENOUGH, J.; GERHART, S. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2): 156-173, June 1975.

- GOOBENOUGH, J.B. Toward a theory of testing: data selection criteria. In: YEH, R.T. ed. *Program validation*. Englewood Cliffs, NJ, Prentice Hall, 1977. p.44-79 (Current trends in programming methodology, 2).
- GUIMARÃES, C.C. *Princípios de sistemas operacionais*. Rio de Janeiro, Campus, 1980.
- HABERMANN, A. N. Prevention of system deadlocks. *Communications of the ACM*, 12(7): 373-377, 385, July 1969.
- HANSEN, P.B. *Operating system principles*. Englewood Cliffs, NJ, Prentice-Hall, 1973a.
- HANSEN, P.B. Test a multiprogramming system. *Software-Practice and Experience*, 3(2):145-150, Apr./Jun. 1973b.
- HANTLER, S.L.; KING, J.C. An introduction to proving the correctness of programs. *ACM Computing Survey*, 8(3): 331-353, Sept. 1976.
- HOWDEN, W.E. Methodology for the generation of program test data. *IEEE Transactions on Computers*, 24(5): 554-559, May 1975.
- HOWDEN, W.E. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4): 266-278, July 1977.
- HOWDEN, W.E. Functional testing and design abstractions. *Journal of Systems and Software*, 1(2): 307-313, 1980.
- HOWDEN, W.E. A survey of static analysis methods. In: MILLER, E.; HOWDEN, W.E. *Tutorial: Software Testing and Validation Techniques*. 2.ed. New York, NY, IEEE, 1981a. p.101-115.
- HOWDEN, W.E. A survey of dynamic analysis methods. In: MILLER, E.; HOWDEN, W.E. *Tutorial: Software Testing and Validation Techniques*. 2.ed. New York, NY, IEEE, 1981b. p.209-231.
- HUANG, J.C. An approach to program testing. *ACM Computing Surveys* 7(3): 113-128, Sept. 1975.
- JENSEN, R.W. Structured programming. *IEEE Computer*, 14(3): 31-48, Mar. 1981.

- KAMEDA, T. Testing deadlock - freedom of computer systems. *Journal of the ACM*, 27(2): 270-280, Apr. 1980.
- KANE, J.R.; YAU, S.S. Concurrent software fault detection, *IEEE Transactions on Software Engineering*, 1(1): 87-99, Mar. 1975.
- KELLER, R.M. Formal verification of parallel programs. *Communications of the ACM*, 19(7): 371-384, July 1976.
- MADNICK, S.E.; DONOVAN, J.J. *Operating systems*. Tokyo, McGraw-Hill, 1974.
- MAPLES, C. Analysing software performance in a multiprocessor environment. *IEEE Software*, 2(4): 50-63, July 1985.
- MELNIKOFF, S.S.S. *Sistema AP<sup>2</sup>: um ambiente para a programação paralela*. Tese de doutoramento em Engenharia. São Paulo, Escola Politécnica da USP, 1982.
- MELNIKOFF, S.S.S.; LUCENA, C.J. Programação paralela: uma experiência na depuração de programas na MAD. In: SIMPOSIO BRASILEIRO SOBRE REDES DE COMPUTADORES, 2., Campina Grande, PB, Abr. 16-18, 1984. *Anais*. Campina Grande, PB, Universidade Federal da Paraíba, 1984, p.14.1 - 14.15.
- MILLER JUNIOR, E.F.; MELTON JUNIOR, R.A. Automated generation of testcase datasets. *SIGPLAN Notices*, 10(6): 51-58, June 1975.
- MILLER JUNIOR, E.F. Programming testing technology in the 1980's. In: MILLER, E.; HOWDEN, W.E. *Tutorial: Software Testing and Validation Techniques*. 2.ed. New York, NY, IEEE, 1981a. p.467-474.
- MILLER JUNIOR, E.F. Introduction to software testing technology. In: MILLER, E.; HOWDEN, W.E. *Tutorial: Software Testing and Validation Techniques*. 2.ed. New York, NY, IEEE, 1981b. p.4-16.
- NASSI, I.; SCHNEIDERMAN, B. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8): 12-26, Aug. 1973.
- NELSON, R.A.; HAIBT, L.M.; SHERIDAN, P.B. Casting Petri nets into programs. *IEEE Transactions on Software Engineering*, 9(5): 590-602, Sept. 1983.

- OSTERWEIL, L.J.; FOSDICK, L.D. DAVE - a validation error detection and documentation system for FORTRAN programs. *Software - Practice and Experience*, 6(4): 473-486, Oct./Dec. 1976.
- OSTERWEIL, L.J. Toolpack - an experimental software development environment research project. *IEEE Transactions on Software Engineering*, 9(6): 673-685, Nov. 1983.
- RAMAMOORTHY, C.V.; HO, S.F. Testing large software with automated software evaluation systems. *IEEE Transactions on Software Engineering*, 1(2): 46-58, Mar. 1975.
- RAMAMOORTHY, C.V.; HO, S.F.; CHEN, W.T. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4): 293-300, Dec. 1976.
- ROSS, D.T.; GOODENOUGH, J.B.; IRVINE, C.A. Software engineering: process, principles and goals. *IEEE Computer*, 8(5): 17-27, May 1975.
- ROSS, D.T.; SCHOMAN JUNIOR, K.E. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, 3(1): 6-15, Jan. 1977.
- RUGGIERO, W.V.; MELNIKOFF, S.S.S.; LUCENA, C.J. Implementação de um sistema de comunicação entre processos através de uma máquina de arquitetura distribuída. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 7., Campinas, 1980. *Anais*. s.n.t. p.1-9.
- SHIMIZU, E.Y.; RUGGIERO, W.V.; MOSCATO, L.A. Componentes básicos de uma máquina de arquitetura distribuída: identificação e implementação. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 7., Campinas, 1980. *Anais*. s.n.t. p.77-87.
- SHOUMAN, M.L. *Software engineering*. Tokyo, MacGraw-Hill, 1983.
- SPINOLA, M.M.; KOBAYASHI, G. Sistema de controle de tráfego ferroviário: testes de hardware e software. In: CONGRESSO NACIONAL DE AUTOMAÇÃO INDUSTRIAL, 2., São Paulo, SP, Nov. 25-29, 1985. *Anais*. s.n.t. p.147-161.

- STAY, J.R. HIPO and integrated program design, *IBM Systems Journal*, 15(2): 143-154, 1976.
- STUCKI, L.G. New directions in automated tools for improving software quality. In: YEH, R.T., ed. *Program validation*. Englewood Cliffs, NJ, Prentice-Hall, 1977. p.80-111. (Current trends in programming methodology, 2).
- TAYLOR, R.N.; OSTERWEIL, L.J. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, 6(3): 265-278, May 1980.
- TEICHROEW, D.; SAYANI, H. Automation of system building. *Datamation*, 17(8): 25-30, Aug. 1971.
- VIOLA, F.E.C. *Lanac - uma linguagem "alto nível" para aquisição de dados e controle de processos por minicomputador*. Dissertação de Mestrado em Computação Aplicada. São José dos Campos, INPE, 1979.
- VOSS, K. Using predicate/transition - nets to model and analysis distributed database systems. *IEEE Transactions on Software Engineering*, 6(6): 539-544, Nov. 1980.
- WILKES, M.V. Software Engineering and structured programming. *IEEE Transactions on Software Engineering*, 2(4): 274-276, Dec. 1976.
- WIRTH, N. Program development by stepwise refinement. *Communications of the ACM*, 14(4): 221-227, Apr. 1971.
- YAU, S.S.; GAVLAYAN, M.U. Distributed software design representation using modified-Petri nets. *IEEE Transactions on Software Engineering*, 9(6): 733-745, Nov. 1983.

## BIBLIOGRAFIA COMPLEMENTAR

- ALLEN, J.R.; KENNEDY, K. A parallel programming environment. *IEEE Software*, 2(4): 21-29, July 1985.
- ANDREWS, D.M.; BENSON, J.P. An automated program testing methodology and its implementation. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, San Diego, CA, Mar. 9-12, 1981. *Proceedings*. New York, NY, IEEE, 1981, p.254-261.
- ANSAWA, M.; MINORU, I. Debugging methods in recursive structured FORTRAN. *Software - Practice and Experience*, 10(1): 29-43, Jan. 1980.
- BACKUS, J. Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8): 613-641, Aug. 1978.
- BECK, L.L.; PERKINS, T.E. A survey of software engineering practice: tools, methods and results. *IEEE Transactions on Software Engineering*, 9(5): 541-561, Sept. 1983.
- BERGLAND, G.D. A guided tour of program design methodologies. *IEEE Computer*, 14(10): 13-37, Oct. 1981.
- BURSTALL, R.M. Proving properties of programs by structural induction. *The Computer Journal*, 12(1): 41-48, Feb. 1969.
- CHARLTON, C.C.; LENG, P.H. Aids for pragmatic error detection. *Software - Practice and Experience*, 13(1): 59-66, Jan. 1983.
- CLARKE, L.A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3): 215-222, Sept. 1976.
- COX, B.J. Message/object programming: an evolutionary change in programming technology. *IEEE Software*, 1(1): 50-61, Jan. 1984.
- CRISTIAN, F. A rigorous approach to fault - tolerant programming. *IEEE Transactions on Software Engineering*, 11(1): 23-31, Jan. 1985.

- CURTIS, B. Software metrics: guest editor's introduction. *IEEE Transactions on Software Engineering*, 9(6): 637-638, Nov. 1983.
- DANNENBERG, R.B.; ERNST, G.W. Formal program verification using symbolic execution. *IEEE Transactions on Software Engineering*, 8(1): 43-52, Jan. 1982.
- DURAN, J.W.; NTAPOS, S.C. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4): 438-444, July 1984.
- GERMAN, S.M.; WEGBREIT, B. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1): 68-75, Mar. 1975.
- GLIGOR, V.D.; SHATTUCK, S.H. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, 6(5): 435-440, Sept. 1980.
- GOLD, E.M. Deadlock prediction: easy and difficult cases. *SIAM Journal of Computing*, 7(3): 320-336, Aug. 1978.
- GOMAA, H. A software design method for real-time systems. *Communications of the ACM*, 27(9): 938-949, Sept. 1984.
- GOOD, D.I.; LONDON, R.L.; BLEDSOE, W.W. An interactive program verification system. *IEEE Transactions on Software Engineering*, 1(1): 59-67, Mar. 1975.
- GOURLAY, J.S. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6): 686-709, Nov. 1983.
- HABERMANN, A.N. System deadlocks. In: CHANDY, K.M.; YEH, R.T. ed. *Software modeling*. Englewood Cliffs, NJ, Prentice-Hall, 1978. p.256-297 (Current trends in programming methodology, 3).
- HANSEN, P.B. The programming language concurrent PASCAL. *IEEE Transactions on Software Engineering*, 1(2): 199-207, June 1975.
- HAVENDER, J.W. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2): 74-84, 1968.
- HOARE, C.A.R. The quality of software - guest editorial. *Software - Practice and Experience*, 2(2): 103-105, Apr./June 1972.

- HOARE, C.A.R. Communicating sequential processes. *Communications of the ACM*, 21(8): 666-677, Aug. 1978.
- HOWDEN, W.E. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3): 208-214, Sept. 1977.
- HOWDEN, W.E. An evaluation of the effectiveness of symbolic testing. *Software - Practice and Experience*, 8(4): 381-397, July 1978.
- HUANG, J.C. Detection on data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, 5(3): 226-236, May 1979.
- HUANG, J.C.; VALDES, P. and YEH, R.T. A tool-based approach for software testing and validation. In: AFIPS NATIONAL COMPUTER CONFERENCE, 53., Las Vegas, NV, July 9-12, 1984. *Proceedings*. Reston, VA, AFIPS Press, 1984, p.411-421.
- JACHNER, J.; AGARWAL, V.K. Data flow anomaly detection. *IEEE Transactions on Software Engineering*, 10(4): 432-437, July 1984.
- KAVI, K.M.; KRISHNAMOHAN, K. Architecture quality. *Operating Systems Review*, 18(1): 11-19, Jan. 1984.
- KING, J.C. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385-394, July 1976.
- KUCK, D.J. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1): 29-59, Mar. 1977.
- MILLER, W.; SPOONER, D. Automatic generation of floating point test data. *IEEE Transactions on Software Engineering*, 2(3): 223-226, Sept. 1976.
- MILLS, H.D. Software development. *IEEE Transactions on Software Engineering*, 2(4): 265-273, Dec. 1976.
- OLSON, R. Parallel processing in a message-based operating system. *IEEE Software*, 2(4): 39-49, July 1985.
- OUSTERHOUT, J.K.; SCELZA, D.A.; PRADEEP, S.S. Medusa: an experiment in distributed operating system structure. *Communications of the ACM*, 23(2): 92-105, Feb. 1980.

- PARNAS, D.L. A technique for the specification of software modules with examples. *Communications of the ACM*, 15(5): 330-336, May 1972.
- PRATT, T.W. Pisces: an environment for parallel scientific computation. *IEEE Software*, 2(4): 7-20, July 1985.
- PROBERT, R.L. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering*, 8(1): 34-42, Jan. 1982.
- RAMAMRITHAM, K.; KELLER, R.M. Specification of synchronizing processes. *IEEE Transactions on Software Engineering*, 9(6): 723-732, Nov. 1983.
- SARRAGA, R.F. Static data flow analysis of PL/I programs with the PROBE system. *IEEE Transactions on Software Engineering*, 10(4): 451-459, July 1984.
- SOUNDARARAJAN, N. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Language System*, 6(4): 647-662, Oct. 1984.
- STUCKI, L.G. Guest editorial: a case for software testing. *IEEE Transactions on Software Engineering*, 2(3): 194, Sept. 1976.
- TAI, K.C. Program testing complexity and test criteria. *IEEE Transactions on Software Engineering*, 6(6): 531-538, Nov. 1980.
- VAN HORN, E.C. Three criteria for designing computing systems to facilitate debugging. *Communications of the ACM*, 11(5):360-544, May 1968.
- VOGES, U.; GMEINER, L.; VON MAYRHAUSER, A.A. SADAT - an automated testing tool. *IEEE Transactions on Software Engineering*, 6(3): 285-290, May 1980.
- WEYUKER, E.J.; OSTRAND, T.J. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3): 236-246, May 1980.
- WHITE, L.J.; COHEN, E.I. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3): 247-257, May 1980.

- WIRTH, N. On the composition of well-structured programs. *ACM Computing Surveys*, 6(4): 247-259, Dec. 1974.
- WOODWAR, M.R.; HEDLEY, D. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, 6(3): 278-286, May 1980.
- ZELKOWITZ, M.V. Perspectives on Software Engineering. *ACM Computing Surveys*, 10(2): 197-216, June 1978.

## APÊNDICE A

### DESCRIÇÃO DO NÚCLEO DE PROCESSAMENTO DISTRIBUÍDO

O Núcleo de Processamento Distribuído (NPD) é uma Máquina de Arquitetura Distribuída (Shimizu, 1980; Ruggiero, 1980) projetada e desenvolvida no Laboratório de Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, em convênio com a Fundação para o Desenvolvimento Tecnológico da Engenharia (FDTE), sob o patrocínio da FEPASA - Ferrovia Paulista S.A..

Uma grande equipe de engenheiros participou do seu desenvolvimento, incluindo "hardware", "software" básico e "software" aplicativo. O "software" aplicativo desenvolvido destinou-se inicialmente à supervisão e controle de tráfego ferroviário para trens de subúrbio de São Paulo (projeto FEPASA). Visando uma aplicação distinta foi desenvolvida a partir do final de 1983 um "software" aplicativo destinado à supervisão e controle de tráfego ferroviário do metrô de Porto Alegre (projeto TRENSURB).

São apresentados a seguir resumidamente as estruturas do "hardware" e do "software" do NDP.

#### A.1 - "HARDWARE"

O NPD é um sistema distribuído confinado, ou seja, as distâncias entre seus componentes básicos não ultrapassam alguns poucos metros. Compõe-se de um agregado de processadores independentes, que podem processar informações paralela e cooperativamente.

A estrutura do NPD foi concebida de forma a garantir as seguintes características (Shimizu, 1980):

- 1) Confiabilidade: Tolerância a falhas simples, isto é, as falhas desta classe devem ser detectadas e isoladas, não bloqueando o funcionamento do sistema, e garantindo um alto grau de disponi

bilidade do equipamento. Isso é implementado através da duplicação de elementos-chave e da manutenção de elementos de reserva.

- 2) Modularidade: Possibilidade de expansões da máquina para abranger uma maior gama de aplicações e utilização em larga escala de um conjunto pequeno de componentes básicos para a realização da máquina (isso simplifica o projeto e a manutenção do equipamento).
- 3) Concorrência: Controle de concorrência e adoção de um esquema que permite que o compartilhamento de memória seja evitado. Cada processador do NPD é um sistema completo, baseado no microprocessador 8085 da Intel, possuindo sua própria memória RAM além de circuitos integrados auxiliares. Esta unidade processadora constitui um "nó" do NPD; estes nós estão ligados entre si através de dois anéis de comunicação paralela. Os nós trabalham independentemente entre si, porém de uma forma cooperativa.

A Figura A.1 mostra a arquitetura do NPD. Podem ser identificados dois grupos de processadores ou nós:

- 1) Processadores de Uso Geral (PUG) que possuem apenas recursos locais de processamento e armazenamento;
- 2) Processador Dedicado (PD) que possui, além dos recursos locais de processamento e armazenamento, pelo menos um recurso adicional específico (como por exemplo interface, com dispositivos de entrada e saída).

Apesar das diferenças funcionais entre os elementos das categorias PUG e PD, as suas estruturas apresentam características comuns, pois são constituídas por um conjunto de componentes básicos comuns e por um outro, de componentes específicos. Os componentes comuns são responsáveis pelas funções de processamento e de gerenciamento da comunicação entre módulos processadores, através dos anéis.

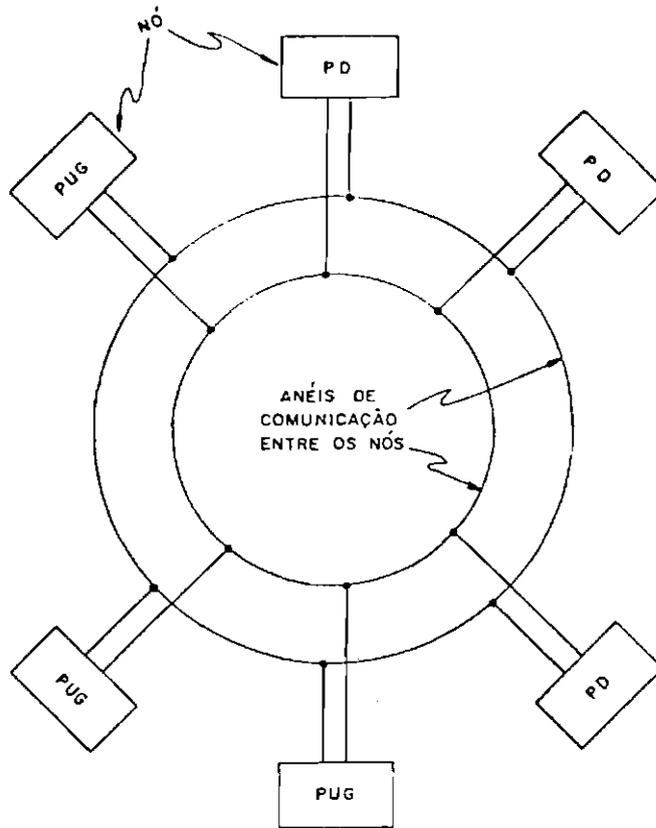


Fig. A.1 - Arquitetura do NPD.

Os anéis de comunicação que interligam todos os nós (tanto PUGs quanto PDs) permitem que os processadores troquem mensagens entre si, tendo como consequência uma interação profunda entre as tarefas executadas nos nós. A existência de dois anéis garante uma maior confiabilidade na comunicação. É apresentado a seguir um resumo das principais características físicas do NPD:

1) nó PUG:

- memória EPROM: 10 Kbytes (monitor + sistema de comunicação),
- memória RAM: 112 Kbytes (para carga dos programas aplicativos);

2) nó PD:

- memória EPROM: 22 Kbytes (monitor + sistema de comunicação + programa aplicativo específico),

- memória RAM: 12 Kbytes;

3) anel de comunicação;

- total de dois anéis, permitindo cada um a transferência de dados a uma taxa de 100 Kbytes por segundos.

O "hardware" dos nós PUG e PD foi feito de maneira modular, padronizando as partes comuns, como mostrado na Figura A.2.

Um nó é uma composição de quatro placas (cartões) de circuito impresso:

- 1) uma placa processadora, denominada Módulo Básico (MB), que contém o microprocessador e os demais circuitos integrados de apoio;
- 2) duas placas de interface com o anel (IA), que controlam a comunicação entre a placa processadora e os anéis; e
- 3) uma placa de expansão, que é diferente para cada tipo de nó. Para um nó PUG a placa de expansão é uma expansora de memória RAM (placa de Memória Dinâmica Paginada - DP), enquanto, para um nó do tipo PD, a placa de expansão depende da tarefa a ser executada por ele: poderá ser uma Unidade Assíncrona (UA), capaz de realizar as comunicações seriais e assíncronas uma placa de Interface com Disco Winchester (IDW), uma placa com programa Concentrador de Discos Flexíveis (CDF), que controla a unidade de discos flexíveis, ou ainda qualquer outra placa, que pode ser desenvolvida em função de alguma característica particular do sistema. Desta forma, em termos de placas de circuito impresso, o NPD poderá ser montado a partir de três tipos de placas: MB, IA e uma placa dedicada, comumente uma DP (nó PUG). Esta modularidade facilita os trabalhos de manutenção pela uniformidade dos componentes do NPD e pela disponibilidade de placas de reposição.

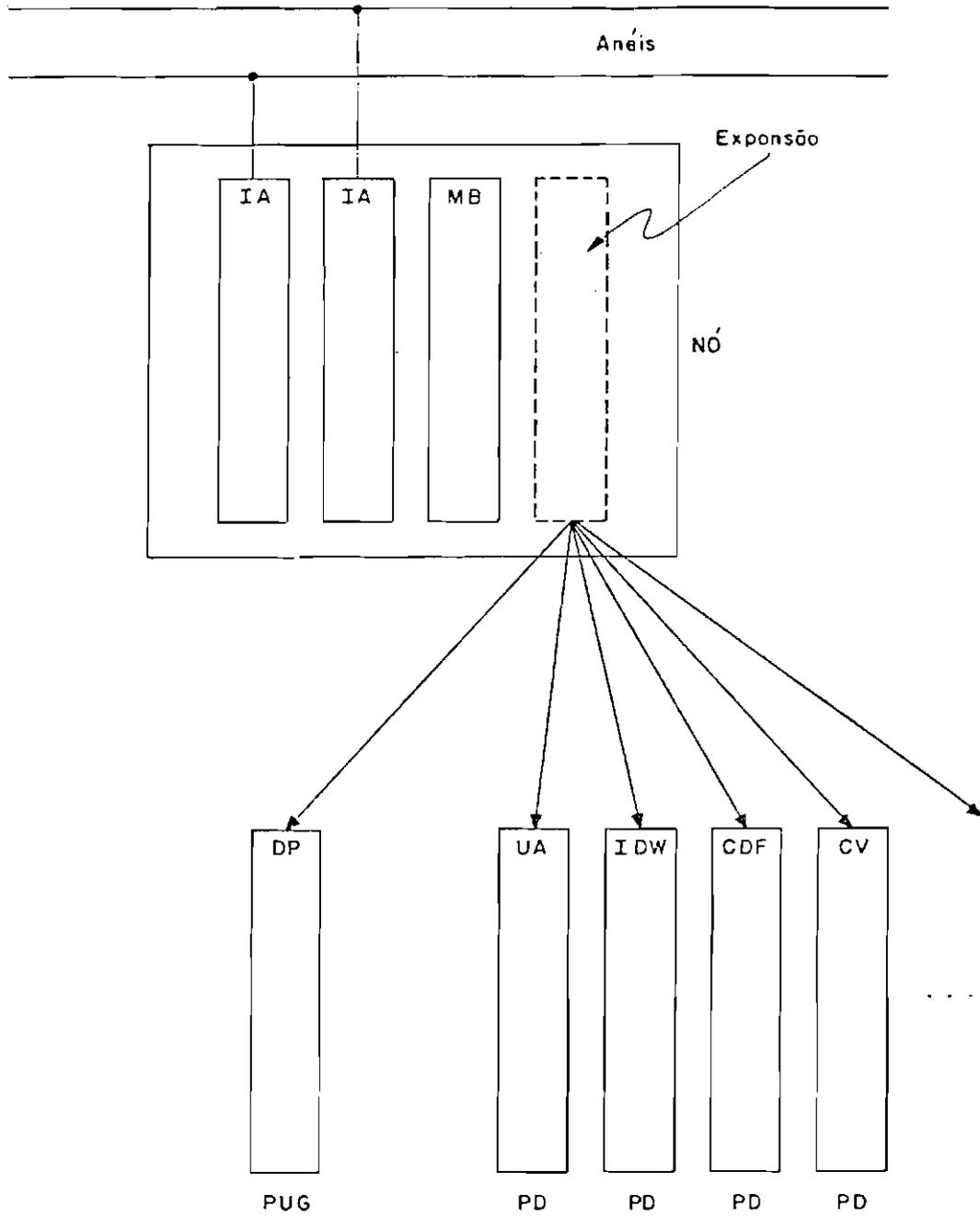


Fig. A.2 - Estrutura física dos nós PUG e PD.

Não existe nenhuma restrição, inerente à estrutura do NPD, que impeça o crescimento do número de nós do sistema. O NPD é composto por 28 nós, sendo 17 PUGs e 11 PDs. Todos os PDs realizam atividades dedicadas de interfaceamento com periféricos do sistema. São os seguintes:

- dois nós ISTDs (Interface com o Sistema de Transmissão de Dados), que tenham a comunicação com o sistema de transmissão de dados, utilizando placas de expansão do tipo UA;
- dois nós CVs (Concentrador de Terminais de Vídeo Colorido), que tratam a comunicação com até dois terminais de vídeo coloridos;
- dois nós IPSs (Interface com o Painel Sinótico), que tratam da comunicação com o controlador de painéis, responsável por enviar as informações do campo para um painel sinótico e os alarmes de sistemas para um painel de alarmes. Estes nós utilizam placas de expansão do tipo UA;
- dois nós IDWs (Interface com o Disco Winchester), que controlam dois discos rígidos, do tipo Winchester, com 10 Kbytes de capacidade de armazenamento cada um;
- dois nós CTs (Concentrador de Terminais), que tratam da comunicação com duas consoles e duas impressoras, utilizando placas de expansão do tipo UA;
- um nó CDF (Concentrador de Discos Flexíveis), que controla uma unidade de discos flexíveis.

A duplicidade dos PDS nesta configuração tem por objetivo aumentar o grau de confiabilidade global da máquina, podendo sempre um módulo substituir totalmente o outro da mesma categoria. O CDF foge a essa regra pois é destinado à carga inicial de dados e programas nos discos, um evento de baixa frequência. A Figura A.3 mostra a configuração do NPD definida para o projeto do CTC da TRENSURB, incluindo os equipamentos periféricos.

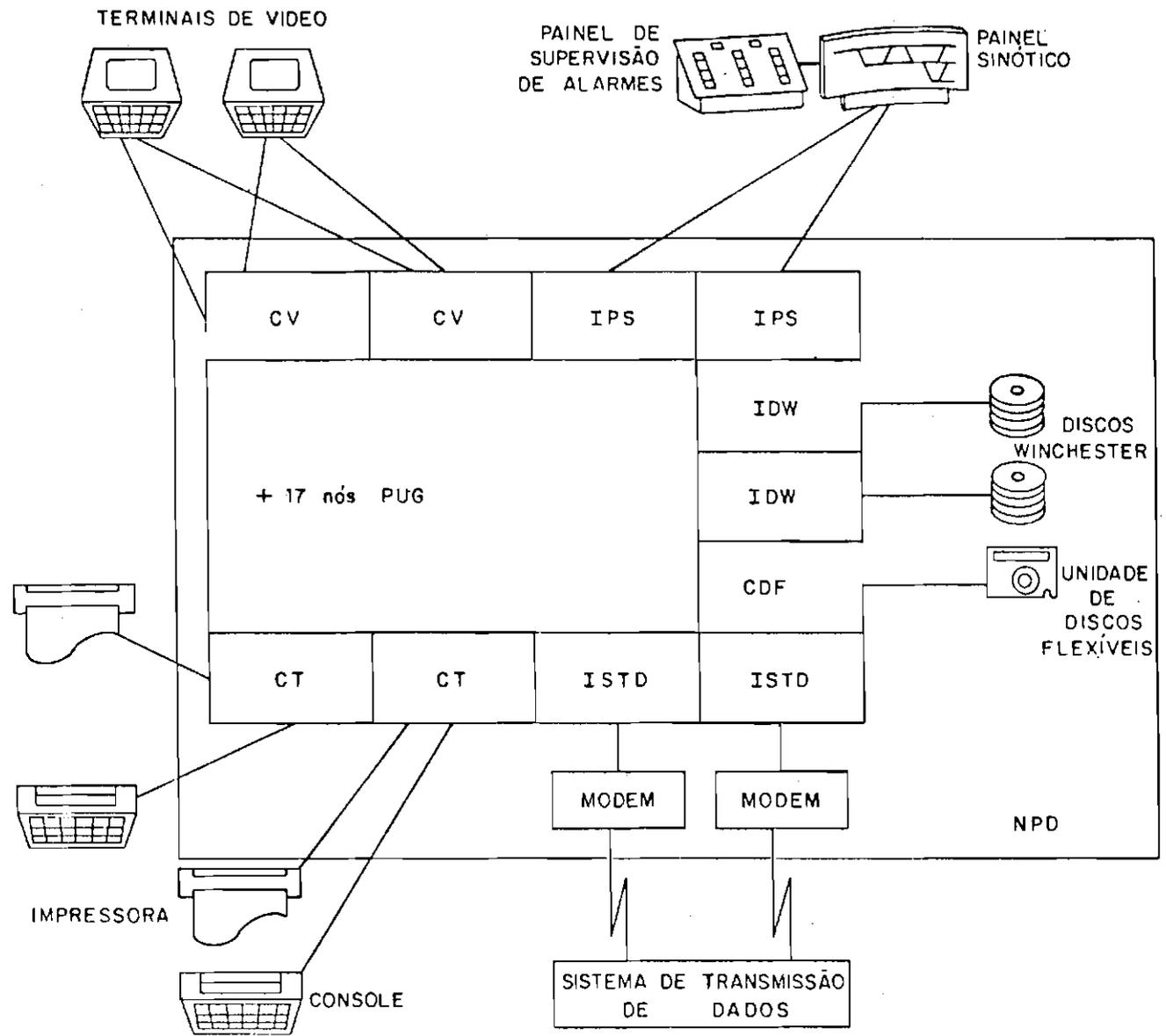


Fig. A.3 - NPD e periférico.

## A.2 - "SOFTWARE"

O "software" desenvolvido para o NPD é dedicado às funções de controle de circulação dos trens, podendo ser dividido em duas partes:

- 1) um "software" básico que oferece recursos para a comunicação entre os processos; e
- 2) um "software" aplicativo, composto por diversos processos que executam as funções especificadas.

O tamanho total do "software" é de aproximadamente 1 Mbyte.

São apresentadas a seguir as características desses dois conjuntos de programas, tomando por opção o "software" aplicativo de desenvolvido para o projeto TRENSURB.

### A.2.1 - "SOFTWARE" BÁSICO

O "software" básico, chamado Sistema de Comunicação (SC), gerencia e executa a comunicação entre os processos usuários (residentes ou não num mesmo nô). É um conjunto de programas cujas cópias residem em cada um dos nôs do sistema. Foi desenvolvido em linguagem Assembly-80.

Além da comunicação entre os processos, o SC coordena o partilhamento do processador do nô através de mecanismos que permitam ativar e suspender processos, chavear o uso do processador, gerenciar os processos conforme sua prioridade, etc. (Ruggiero, 1980).

A comunicação entre os processos é feita através de mensagens enviadas de um para outro. As mensagens são classificadas por tipo e se constituem em conjuntos de bytes que possuem as informações necesárias à continuidade do processamento e à sincronização dos proces

tos. Um processo fica bloqueado quando necessita de uma mensagem para continuar a execução (diz-se que o processo "espera mensagem"). Quando recebe a mensagem o processo retorna o processamento, já com as novas informações à disposição. Nos bloqueios em que o processo esteja capacitado a tratar apenas algum tipo específico de mensagem, essa espera pode ser seletiva (ou seja, a execução só tem continuidade após a chegada de um determinado tipo de mensagem, e as mensagens não tratadas entram em uma fila para tratamento posterior).

As operações básicas que realizam as funções de comunicação entre os processos (envio de mensagem, espera de mensagem e outras) são disponíveis para utilização pelos processos usuários através das rotinas primitivas do SC.

Dois outros recursos são ainda oferecidos pelos "software" básico: a temporização de mensagens e a reconfiguração. Através da temporização é possível acionar atividades periódicas ou então desbloquear um processo após um tempo programado de espera. Já a reconfiguração, ativada sempre que constatada a falha de um PUG, permite a substituição automática do módulo em falha e a reativação de todos os processos. É acionada em geral após um estouro de temporização na troca de mensagens entre processos residentes em PUGs distintos.

#### A.2.2. - "SOFTWARE" APLICATIVO

O "software" aplicativo desenvolvido para implantação no CTC-TRENSURB realiza as funções de supervisão, controle manual, administração da operação e recuperação de falhas. Essas funções foram desenvolvidas para atender às necessidades do Sistema de Sinalização, esquematizado na Figura A.4. Podem ser identificadas neste sistema:

- o subsistema Intertravamento que atua diretamente nos elementos de sinalização e controle da ferrovia, tais como sinaleiros, máquinas de chave, circuitos de via, etc.;

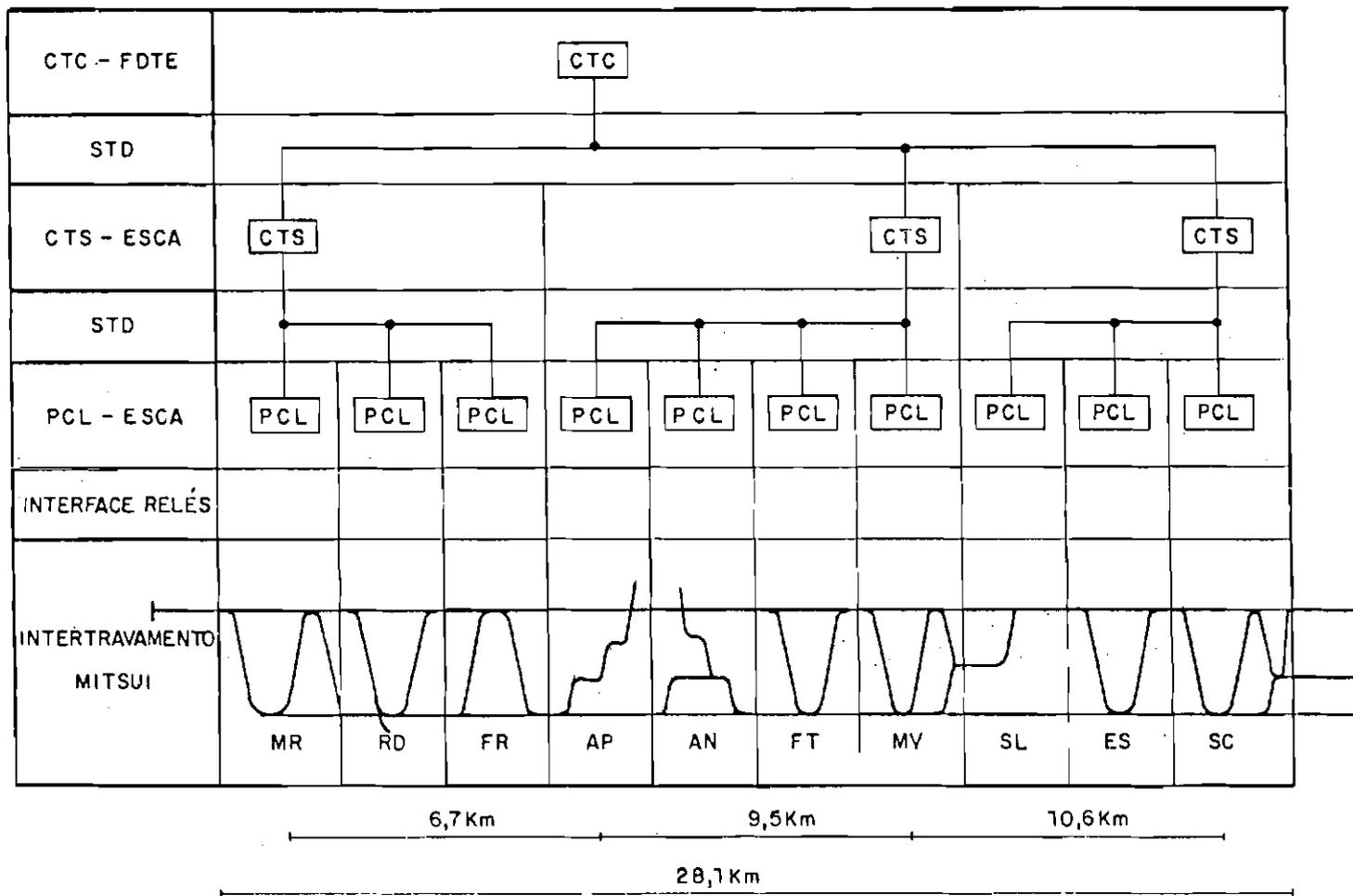
- A.10 -

- os subsistemas PCL (Painel de Controle Local) que, através de um painel, supervisionam e controlam, cada um, os elementos do intertravamento de uma determinada região da via. São dez PCLs ao todo;
- os subsistemas CTS (Controle de Tráfego Setorial) que realizam, cada um, varreduras dos PCLs sob seu controle, colhendo informações e enviando comandos. São três CTs ao todo;
- o subsistema CTC (Controle de Tráfego Centralizado), nível superior do Sistema de sinalização.

As funções realizadas pelo "software" aplicativo estão descritas a seguir (Spinola, 1985):

- a) Supervisão, que compreende a aquisição automática de informações sobre o estado da ferrovia e a apresentação aos operadores das informações necessárias ao controle do tráfego. Os seguintes parâmetros são observados pelo sistema e indicados no Painel Sinótico e no TV:
  - posição dos trens na via;
  - situação das rotas (livre, ocupada, alinhada ou requisitada);
  - situações dos elementos da via (posição dos aparelhos de mudança de via, estado dos sinais, sentido de tráfego);
  - movimento em frota por PCL e por via;
  - modo de controle em cada região, que pode ser central, setorial ou local;
  - modo de operação do intertravamento por PCL (manual ou automatico);
  - acompanhamento de entrada e saída dos pátios;
  - estados de elementos da via, que não são fornecidos pelos CTs mas podem ser introduzidos manualmente pelo operador no TV (seções bloqueadas, liberadas, em manutenção ou com manutenção terminada, aparelhos de mudança de via ativos, desativados, bloqueados pelo operador ou em automático, sinais ativos ou desativados).

- b) Controle, que consiste no envio de comandos de controle aos equipamentos de campo e na verificação da efetivação dos comandos da via. As seguintes funções de comando podem ser efetuadas, sempre pelo TV:
- alinhamento de rotas através das indicações de origem e destino;
  - cancelamento de rotas através das indicações de seus pontos de origem;
  - seleção e cancelamento de movimento em frota;
  - seleção do modo de controle, que pode ser central, setorial ou local;
  - colocação da via em modo frota e cancelamento do modo frota;
  - seleção do modo de intertravamento (manual ou automático);
  - pedido de autorização para entrada no pátio de oficinas.
- c) Administração da operação, que compreende as funções de auxílio à programação dos trens. São elas:
- identificação dos trens por números de missão atribuídos quando entraram na área controlada pelo CTC;
  - operações sobre números de missões (mudança do número de missão de um trem, fusão de dois números em um só, separação de um número de missão em duas missões distintas);
  - atualização dinâmica e apresentação de tabelas e relatórios aos operadores, através dos TVs, das impressoras e consoles.
- d) Recuperação de falhas, que se compõe dos mecanismos de recuperação do sistema em caso de falhas, nas seguintes situações:
- falhas de periféricos;
  - falhas de PDs;
  - falhas de PUGs (tratamento de reconfiguração).



- A.12 -

Fig. A.4 - Sistema de sinalização - TRENSURB.

Para realizar essas funções operacionais foram definidas funções lógicas e distribuídas entre os nós. Coube ao "software" instalado nos PUGs fazer todo o gerenciamento de estados e recursos disponíveis, sustentando a maior carga de processamento do sistema. Aos PDs coube realizar tarefas específicas e especializadas, já descritas anteriormente em A.1.

Além dos PDs, os terminais de vídeo em cores (TV) possuem também um módulo de programa para o processamento das informações relativas à montagem das telas e interpretação dos comandos.

As funções lógicas carregadas nos PUGs são os seguintes:

- a) GD (Gerenciador de Discos): gerencia a utilização dos discos pelos processos do sistema, através das IDWs;
- b) GE (Gerenciador de Estado da Ferrovia): faz o tratamento das indicações recebidas do campo, atualizando as bases de dados do sistema;
- c) GES (Gerenciador de Entrada e Saída): gerencia a utilização das impressoras e das consoles através dos CTs, e da unidade de disco flexível, através do CDF;
- d) IO (Interface com os Operadores): executa os comandos requisitados pelos controladores do sistema através dos CVs;
- e) GSTD (Gerenciador do Sistema de Transmissão de Dados). gerencia a utilização do Sistema de Transmissão de Dados, através das ISTDs;
- f) GV (Gerenciador de Via): gerencia a atuação sobre a via;
- g) GP (Gerenciador de Painéis): apresenta o estado da via no painel sinótico e os alarmes do CTC e dos CTSS no painel de supervisão de alarmes;

- h) GA (Gerenciador de Alarmes). formata convenientemente os alarmes da via gerados por outras funções;
- i) GBU (Gerenciador de Back-up): armazena dados para serem utilizados após a reconfiguração.

A Figura A.5 apresenta esquematicamente as inter-relações entre as funções do "software". Pode-se observar as posições hierárquicas e a descentralização do fluxo de informações.

A interface com os operadores é gerenciada pelas funções GES (impressoras e consoles), IO (terminais de vídeo) e GP (painéis).

O GSTD gerencia a interface com o campo. Cabe o GE e ao GV gerenciar os estados dos trens e da via. O GD gerencia os acessos a disco e o GA coloca em formato adequado os alarmes da via.

Os processos que constituem as funções acima descritas foram codificados em linguagem PLM80.

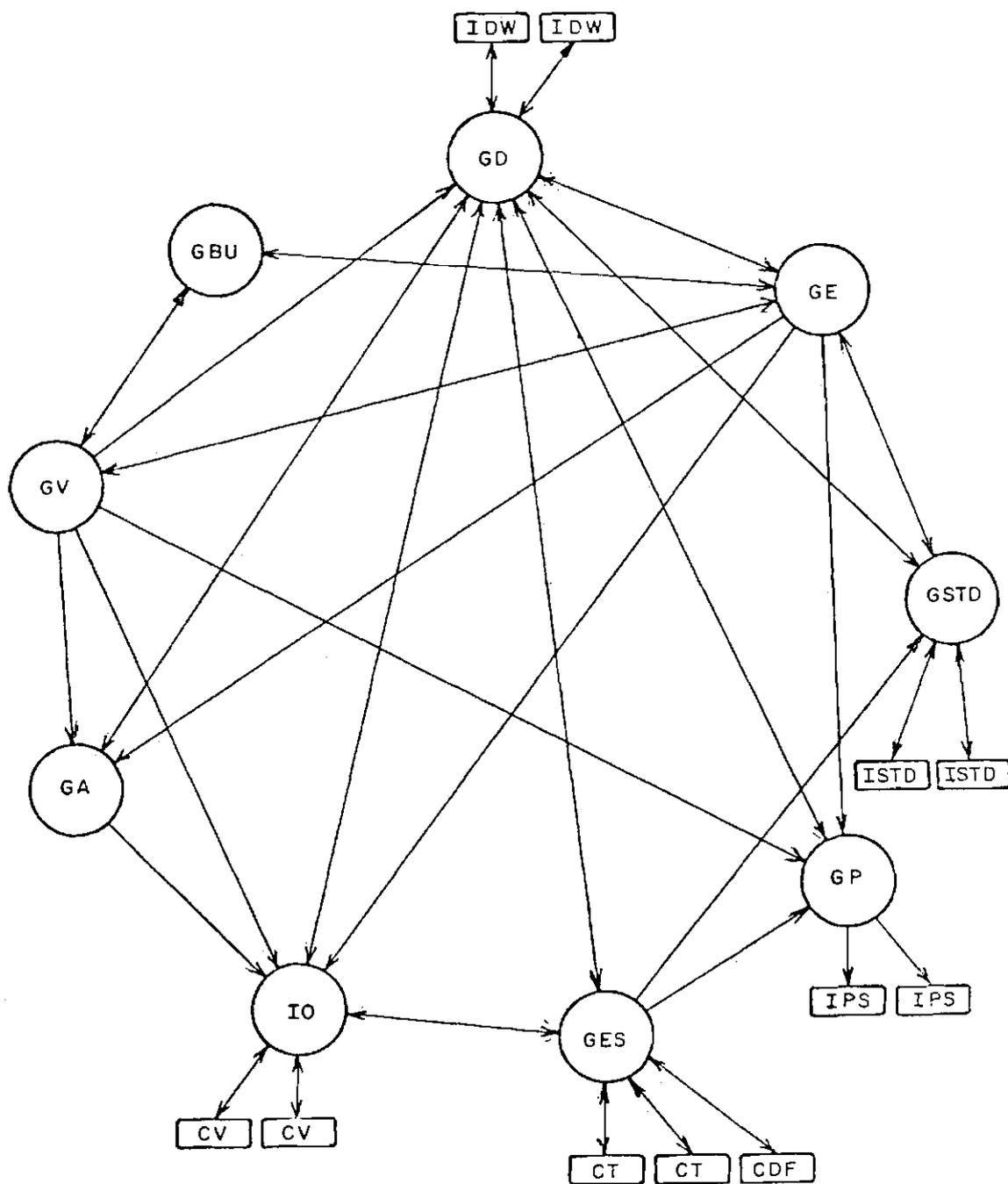


Fig. A.5 - Funções do "software" aplicativo.

## APÊNDICE B

### O AMBIENTE DE TESTES NO NPD

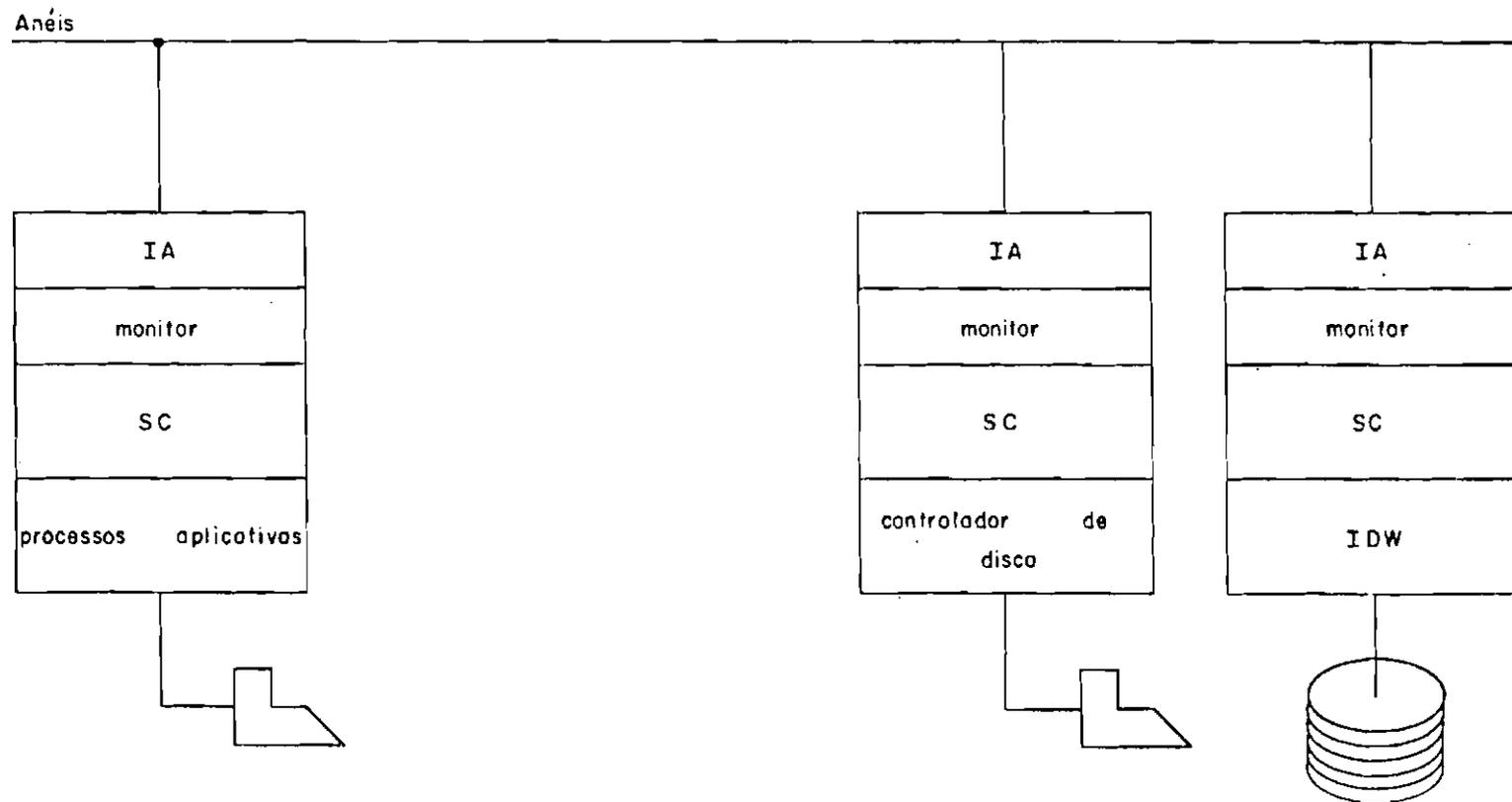
Os testes de "software" do NPD são realizados no próprio "hardware" do computador, com a inclusão de ferramentas de "software" que facilitam esses testes e a depuração. Os testes preliminares são realizados em protótipos com características físicas e lógicas análogas às dos produtos finais, a menos do número de nós disponíveis. Os testes de implantação são desenvolvidos no "hardware" dos produtos definitivos. Paralelamente aos testes de implantação, prosseguem os testes de apoio com uso dos protótipos.

Com vistas na igualdade os recursos físicos do ambiente de testes e o próprio NPD, cujas características físicas estão apresentadas no Apêndice A, serão apresentadas apenas as características lógicas do ambiente de testes.

A Figura B.1 apresenta esquematicamente o ambiente de testes do NPD. Pode-se notar a similaridade com o modelo proposto na Seção 3.3.3.1.

Em cada nó identificam-se:

- uma cópia do programa de interface com o anel (IA) dedicados à comunicação com outros nós, e residente nas placas IA;
- uma cópia do programa monitor, que fornece subsídios para controlar a execução dos processos do nó ou observar o estado do nó; e
- uma cópia do Sistema de Comunicações (SC), que gerencia a utilização de recursos do nó e a troca de mensagens entre processos.



Fog. B.1 - Ambiente de testes do NPD.

O SC utilizado nos testes não é o mesmo do produto final, pois possui recursos adicionais voltados para a atividade de depuração dos programas. Esses recursos realizam a consistência dos parâmetros nas chamadas de rotinas primitivas do SC feitas pelos processos aplicativos. A ocorrência de erros na utilização da área de trocas de mensagens ocasiona uma parada na execução do nó.

A estrutura de dados do SC realiza um papel importante na depuração de programas, já que permite analisar o estado dos processos e do processador. Entre as informações que podem ser obtidas, pode-se citar (Melnikoff, 1984):

- estado dos processos, obtido através das suas pilhas, podendo-se deduzir em que ponto se encontra cada um dos processos bloqueados;
- última mensagem recebida ou enviada pelos processos;
- conteúdo das filas de mensagens de cada processo;
- número de blocos livres na área de troca de mensagens;
- conteúdo dos blocos utilizados na área de troca de mensagens;
- variáveis relativas à transmissão das mensagens pelos nós.

Os nós do tipo PUG são carregados com os processos aplicativos, a serem testados, enquanto os nós do tipo PD possuem residentes os programas IDW, CV, CT, ISTD, CDF e IPS. Além desses programas, a Figura B.2 mostra o programa Controlador do Disco (CD), residente em um dos nós, de grande importância na realização dos testes. São feitas a seguir algumas considerações relativas ao CD e ao programa monitor.

Através do nó CD (cujas funções são análogas às da ferramenta "gerenciador de arquivos em disco" proposta na Seção 3.3.3.2) o programador tem acesso aos arquivos em disco, onde estão gravados programas do "software" aplicativo e programas utilitários. Os seguintes comandos podem ser executados pelo CD:

- B.4 -

- COPY/COPYA,  
para cópias de arquivos que admite as seguintes possibilidades:
  - . de arquivo em disco para:
    - .. outro arquivo em disco,
    - .. memória do CD,
    - .. memória de um nō PUG qualquer,
  - . da memória do CD para um arquivo em disco,
  - . da memória de um nō PUG qualquer para um arquivo em disco,
  - . de arquivo gravado em disco flexível para arquivo em disco;
  
- DIR,  
para observar os arquivos gravados, seus atributos, tamanho e localização;
  
- ATTRIB,  
para mudar atributos de arquivos;
  
- DELETE,  
para remover arquivos;
  
- RENAME,  
para alterar o nome de arquivos;
  
- MON/MONA,  
para poder utilizar o programa monitor residente no nō CD;
  
- SUBMIT,  
para submeter à execução uma sērie de comandos gravados em determinado arquivo;
  
- LES,  
para carregar na memória do CD áreas do disco endereçadas diretamente pelo seu setor inicial;
  
- GRAVA,  
para armazenar em disco flexível dados ou programas carregados na memória do CD;

- ATIVA,  
para ativar os processos carregados em determinado n<sup>o</sup> (ou em todos os n<sup>os</sup>, se desejado);
- PARA,  
para interromper a execu<sup>ç</sup>o de processos carregados em determinado n<sup>o</sup> (ou em todos os n<sup>os</sup>, se desejado);
- REATIVA;  
para reativar os processos carregados em determinado n<sup>o</sup> (ou em todos os n<sup>os</sup>, se desejado) a partir dos endere<sup>ç</sup>os de reativa<sup>ç</sup>o;
- HELP,  
para obter informa<sup>ç</sup>o<sup>es</sup> sobre os comandos do CD.

Os comandos ATIVA, PARA e REATIVA s<sup>ao</sup> an<sup>al</sup>og<sup>os</sup> aos prop<sup>os</sup>tos na ferramenta "execu<sup>ç</sup>o controlada dos processos", apresentada na Se<sup>ç</sup>o 3.3.3.2.

Atrav<sup>es</sup> da execu<sup>ç</sup>o dos comandos do CD o programador realiza a maioria das a<sup>ç</sup>o<sup>es</sup> necess<sup>ari</sup>as aos testes e <sup>a</sup> depura<sup>ç</sup>o: copia os programas nos n<sup>os</sup>, ativa-os, desativa-os, efetua corre<sup>ç</sup>o<sup>es</sup> e grava novamente no disco. As demais a<sup>ç</sup>o<sup>es</sup> necess<sup>ari</sup>as s<sup>ao</sup> realizadas, em geral, atrav<sup>es</sup> de consoles, ligadas aos pr<sup>op</sup>rios n<sup>os</sup> em testes- .

O programa monitor carregado em cada n<sup>o</sup> permite ao programador realizar nos PUGs opera<sup>ç</sup>o<sup>es</sup> locais para observar os estados dos processos e controlar a execu<sup>ç</sup>o dos programas. S<sup>ao</sup> permitidas, entre outras, as seguintes a<sup>ç</sup>o<sup>es</sup>:

- observar o conte<sup>u</sup>do de posi<sup>ç</sup>o<sup>es</sup> de mem<sup>o</sup>ria;
- alterar o conte<sup>u</sup>do de posi<sup>ç</sup>o<sup>es</sup> da mem<sup>o</sup>ria;
- observar o conte<sup>u</sup>do de registradores;
- listar uma <sup>area</sup> de programa na tela;

- mover um conjunto de bytes de uma área da memória para outra;
- executar os programas totalmente ou atingir certos endereços ("breakpoints").

Dois outros recursos lógicos constituem o ambiente de testes do NPD: o SDPP e um simulador da via, descritos a seguir.

O SDPP (Sistema de Depuração de Programas Paralelos) (Melnikoff, 1982) é um dos instrumentos mais importantes na efetivação da metodologia de testes do NPD. Fornece recursos para monitorar a execução do programa paralelo no nível das rotinas de interface com o SC. O seu princípio básico é o seguinte: o processo aplicativo a ser testado é isolado e todas as suas interfaces com outros processos (trocas de mensagens) passam a ser feitas com o próprio SDPP. Através deste o operador pode observar as mensagens enviadas e gerar as mensagens esperadas pelo processo. Podem também ser monitorados dois ou mais processos ao mesmo tempo. A Figura B.2 mostra a relação entre o SDPP, o programador e os processos em teste.

Para utilizar o SDPP, o programador deve carregá-lo na memória dos nós, do tipo PUG, onde estão instalados os processos cuja execução deve ser monitorada (pode ser utilizada, se necessário, a área de memória ocupada normalmente por processos que não participarão do teste). Depois, utilizando terminais ligados aos nós, realiza-se a execução dos processos sob controle do SDPP. A operação compreende duas etapas:

- 1) Configuração do sistema, em que o programador fornece informações relativas à parte do programa que se quer testar, a saber:
  - tipo de teste a ser executado: com um ou com mais nós;
  - dimensionamento da área de troca de mensagens;
  - identificação dos processos que serão ativados;
  - identificação dos processos que serão observados;
  - identificação dos processos que serão simulados pelo operador;

- tipo de monitoração;
  - identificação das comunicações que serão monitoradas, etc.
- 2) Execução do programa, em que é feito o acompanhamento da execução dos processos com intensa interação com programador que observa os eventos monitorados e aciona comandos para controle da execução e da monitoração.

- envio de uma mensagem, registrando seu conteúdo;
- espera de uma mensagem (qualquer ou de tipo específico);
- recepção de uma mensagem, registrando seu conteúdo.

Durante a execução, os seguintes comandos, entre outros, podem ser acionados pelo programador:

- impressão do conteúdo de um trecho da memória;
- preenchimento de um trecho de memória com constantes;
- retorno ao programa monitor;
- reiniciação do programa;
- pedido do número de blocos disponíveis na área de mensagens;
- pedido do conjunto de processos observados;
- pedido do conjunto de processos não observados;
- pedido das variáveis de controle dos processos;
- alterações dos valores das variáveis de controle dos processos;
- inclusão de um processo no conjunto de processos não observados;
- retirada de um processo do conjunto de processos não observados;
- envio de mensagem via operador;
- continuação da execução.

No Capítulo 5 são apresentadas várias situações de utilização do SDPP.

Para a realização dos testes de funções de supervisão e controle de tráfego ferroviário foi desenvolvido um programa simulador da via férrea. Esse simulador pode ser carregado e executado em qualquer no PUG, e seu papel é o de simular o comportamento dinâmico da

via férrea e todos os equipamentos de campo, sob o ponto de vista do "software" do NPD. A criação do SV teve os seguintes objetivos:

- 1) assegurar a independência entre o desenvolvimento do "software" e a implantação da interligação com o campo;
- 2) dar subsídio para a verificação das funções lógicas do NPD antes de sua ligação com o campo, o que garante o isolamento entre os erros lógicos e os erros de comunicação; e
- 3) permite a realização de testes em ambientes desprovidos de interligação com o campo.

Apesar de ter sido desenvolvido para simular dinamicamente o Sistema de Transmissão de Dados (STD), o SV possui limitações na execução dessas tarefas que devem ser consideradas na realização dos testes:

- 1) o comportamento do SV é uniforme, não possibilitando testes de falhas do STD;
- 2) o SV apresenta algumas simplificações em relação a situações reais;

Os testes finais das funções de supervisão e controle devem ser feitos com o NPD ligado ao campo. No entanto, é possível, com o SV, dar passos largos nos testes e depuração do "software".

