AUTOMATIZAÇÃO DAS TAREFAS DA MANUTENÇÃO DE SISTEMAS

Alberto Courrège Gomide

DISSERTAÇÃO APRESENTADA AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

ORIENTADOR
Prof. Dr. VALDEMAR W. SETZER

São Paulo, junho de 1981

AGRADEÇO

Ao Prof. Dr. Valdemar W. Setzer, pela valiosa orientação e incentivo;

À minha esposa e filhos, pela paciência e auxílio;

A meus pais, por tudo e mais o serviço gráfico;

Ao Ademir José Rodrigues, pelo excelente trabalho de datilografia.

"Fie then, why sit we musing-
Youth's sweet delights refusing?"

Thomas Morley

ERRATA

pag.	local	onde se lê	leia-se
1-4	penúltima linha do segundo parágrafo	os autores	esses autores
1-5	primeira linha	/D-V 77/	/T-H 77/
2-1	sexta linha	relacões	relações
2-18	primeira linha	integrão	integração
3-1	terceira linha do segundo parágrafo	disponível	disponíveis
A-8	quinto parágrafo	eles:	eles.
A-11	primeira linha	WFL	de WFL
A-11	terceira linha	pelo	por um
A-11	sétima linha	nomes,	nomes

ÍNDICE

INTRODUÇÃO

Este trabalho apresenta uma forma sistemática para a descrição de um sistema de "software", de forma a possibilitar um suporte automático à sua manutenção. Um sistema, ao longo de sua vida útil em uma organização, sofre várias mudanças e adaptações a novas condições e exigências que surgem continuamente. O suporte automático a esta manutenção se destina a integrar as alterações no sistema de forma segura, garantindo a obtenção de um sistema sempre consistente; manter um histórico de todas as alterações que incidiram sobre o sistema; recuperar um sistema tal como era em determinada época anterior; efetuar de maneira automática a maior parte das tarefas puramente mecânicas exigidas por essa manutenção. Dessa sistemática resultou uma implementação para o computador B-6700, que está em uso e tem demonstrado uma grande eficácia, reduzindo em muito os custos da manutenção de sistemas de "software" em geral, tal como o próprio sistema operacional da máquina, seu conjunto de utilitários, sistemas comerciais como o sistema de cobrança do uso do computador, etc.

No Capítulo 1 apresentamos alguns conceitos usados no desenvolvimento e construção de sistemas de "software", discutimos a terminologia utilizada, e apresentamos um modelo existente para a representação abstrata de um sistema.

No Capítulo 2 discutimos os problemas existentes na manutenção de sistemas, e desenvolvemos um modelo de representação abstrata de um sistema orientado ao suporte da manutenção e solução dos problemas citados.

No Capítulo 3 discutimos os problemas da implementação a partir do modelo, notadamente os de portabilidade, e damos uma descrição sucinta da implementação existente.

No Capítulo 4 comentamos o resultado obtido, e sugerimos uma extensão ao modelo para incluir o suporte à opera-

ção de sistemas.

O Apêndice descreve em detalhes a implementação existente no computador B-6700, e constitui o "manual do usuário".

CAPÍTULO 1

CONCEITOS EM "ENGENHARIA DE SOFTWARE"

Neste capítulo apresentamos alguns conceitos utilizados na metodologia de construção de sistemas, que têm sido usados por vários autores para desenvolvimento da automatização da construção e sua documentação. Discutiremos com certo detalhe um dos modelos de evolução de um sistema, dando ênfase apenas aos pontos relevantes ao escopo deste trabalho: manutenção do resultado do projeto. Apresentamos a terminologia adotada, discutindo a interpretação de alguns conceitos mais pertinentes a este trabalho. A matéria deste capítulo está bem esclarecida na bibliografia, e é apresentada apenas como requisito para a identificação dos problemas existentes na manutenção de sistemas.

1.1 - Terminologia

Até o momento não há, ainda, uma terminologia que seja consistente, não ambígua e universalmente aceita. Hesse /HES 80/ ressalta o fato de que termos comuns, tais como "sistema", "processo", "especificação", "implementação", "verificação", "planejamento", "organização", etc, possuem interpretações diversas, e muitos conceitos são descritos por termos diferentes em diferentes autores. Vamos, então, escolher e adotar uma interpretação própria para os termos utilizados neste trabalho. Para alguns termos, daremos uma definição precisa. Outros, utilizaremos conforme a interpretação do autor referenciado. Outros, ainda, aceitaremos como claros e sem necessidade de explicação, bastando exemplificar.

A maioria dos termos que utilizaremos descrevem conceitos exclusivos da ciência da computação, e possuem um significado diferente dos usuais em nossa língua. Um exemplo típico é "compilar". Isto decorre do fato de terem sido escolhidos por semelhança às palavras da língua inglesa que introduziram esses

conceitos, e normalmente nos deparamos com grandes dificuldades na tradução. A primeira delas consiste em traduzir os conceitos descritos em inglês por "project" e "design".A palavra "design" se traduz corretamente para "projeto", contendo a seqüência de conceitos "planejar, detalhar e dar forma a uma idéia abstrata", não contendo, em geral, a idéia da execução material daquilo que foi projetado. A palavra "project", que tem sido traduzida de maneira laxa por "projeto", na verdade engloba "analisar, projetar e construir". À falta de termo melhor, adotamos "construção" como tradução de "project", tentando implicar que a uma idéia demos uma forma, e a materializamos; deixamos assim "projeto" como tradução precisa de "design". Certos termos da língua inglesa são, a nosso ver, impossíveis de traduzir com uma única palavra, a exemplo de "hardware","software","scheduling"; estes utilizaremos no original, entre aspas.

O objetivo deste trabalho é manter de maneira automatizada "sistemas" existentes em um "computador". Segundo Teichroew e Hershey /T-H 77/ um computador é a união de uma máquina física, seu "sistema operacional" ("hard-software") e seus "programas utilitários" ("software"). O conjunto de programas utilitários, tais como compiladores, pode ser considerado como parte integrante do sistema operacional, e os separamos apenas por conveniência. Preferimos denominar de "contexto ('environment') operacional" à união do sistema operacional e seus utilitários. Uma máquina física é capaz de armazenar informações de forma permanente em dispositivos dedicados,alguns com capacidade de recuperação das informações exclusivamente seqüencial (tal como fitas magnéticas), outros com capacidade adicional de recuperação aleatória (tal como discos magnéticos). A estes dispositivos denominaremos "meios de armazenamento". A organização das informações e o modo de acesso a elas são determinados e controlados pelo sistema operacional, que agrupa as informações em entidades denominadas "arquivos". O conteúdo de um arquivo pode ter um significado, em um computador específico, que o classifica como "arquivo de dados", "programa", "programa objeto",etc. Uma manipulação ou transformação de arquivos consti-

tui um "processo", que conceituaremos como sendo: uma atividade autônoma executada em um computador, iniciada por um pedido ao, e controlada pelo, sistema operacional; uma atividade que sempre termina; uma atividade cujo término pode ser espontâneo (normal) ou forçado pelo sistema operacional (anormal). O modo de seu término pode, ou não, ser registrado pelo sistema operacional para consulta posterior. Um conjunto de processos executados em um computador constitui um "job", que é descrito através de uma linguagem particular que denominaremos "linguagem de comandos" do computador.

Podemos agora conceituar "sistema". Segundo Teichroew e Hershey /T-H 77/ um sistema existe inicialmente como um conceito ou uma proposta em um nível alto de abstração. No instante em que ele se torna operacional, ele existe como um conjunto de "regras" e programas objeto executáveis em um computador específico. Entre estas duas fases, o sistema existe em várias formas intermediárias. Assim, a "construção" de um sistema é sua evolução metodizada entre as duas fases. Como parêntese, citamos que "documentação" é o registro das descrições do sistema em cada uma de suas formas; seu escopo é facilitar a transição de uma forma a outra e a manutenção futura.Para Hesse /HES 80/, "sistema" é usado no sentido estrito de um conjunto de programas ou união programas-computador, sem incluir seus aspectos técnicos, de organização ou administrativos. Para a finalidade deste trabalho, definiremos "sistema" como um conjunto de objetos, que são: 1) o resultado de uma seqüência finita de processos executados em um computador, desde que esta seqüência seja reprodutível fornecendo sempre o mesmo resultado, ou 2) uma coleção de informações armazenada em um computador, com acesso seqüencial, como resultado de uma operação manual. Na prática,nossa definição se aproximará mais de /HES 80/, embora alguns objetos não serão programas. As regras citadas em /T-H 77/ incluem fluxos manuais de informações, e apenas a descrição desse fluxo poderia constituir um "objeto" de um sistema.

Outros termos serão introduzidos no decorrer do trabalho, não cabendo aqui sua definição, devido a necessitarem de

conceituações e discussões prévias extensas.

1.2 - Metodologia de construção de sistemas

A metodologia de construção de sistemas preocupa-se em determinar seus componentes e fases, documentá-los, implementá-los, e verificar a implementação ou o comportamento do sistema. Vários autores têm proposto modelos consistentes, que resultaram na confecção de sistemas automáticos de suporte à construção /T-H 77/ /I-B 77/ /BBD 77/ /ALF 77/ /D-V 77/. Estes trabalhos diferem consideravelmente em escopo e enfoque, e estão em contínua evolução.

Segundo Teichroew e Hershey /T-H 77/, um suporte automático de construção de sistemas deve ter as seguintes capacidades:

1) descrever sistemas de informação
2) armazenar essa descrição em um banco de dados
3) adicionar, modificar ou eliminar informações nas descrições contidas no banco de dados
4) produzir documentação para uso dos analistas ou usuários do sistema.

O aspecto inicial do problema consiste em descrever sistemas de informação. Para isto, os autores definem uma linguagem de descrição de sistemas, PSL (Problem Statement Language).

A descrição de um sistema baseia-se no conceito de "especificação", ou "especificação de requisitos". Davis e Vick /D-V 77/ citam que os requisitos têm sido especificados em uma variedade de linguagens, desde textos em linguagens naturais, equações, expressões lógicas, até formas processáveis em computadores tais como PSL. Os conceitos básicos para a definição de requisitos estão contidos em quatro primitivos: elementos, atributos, relações e estruturas. Com isto, esses autores definem sua linguagem RSL (Requirements Statement Language). Como em

/D-V 77/, temos um banco de dados do tipo relacional. A formalização de "especificação" está bem fundamentada por Liskov e Zilles /L-Z 75/, que introduzem o conceito de "data abstraction" e não nos cabe aqui aprofundar o assunto.Esta formalização constitui a base para o desenvolvimento da automatização das especificações nos modelos recentes.

O trabalho de Davis e Vick /D-V 77/ estende-se bem além de Teichroew e Hershey /T-H 77/. Davis e Vick incluem como necessárias ao suporte automático da construção as seguintes capacidades adicionais: facilidades para a construção de processos que automaticamente coletam módulos especificados e os ligam em um processo executável; compilação independente de módulos; recompilação automática de módulos de nível inferior afetados por uma mudança em um módulo de nível superior; recompilação automática de todos os módulos dependentes afetados pela modificação de um módulo particular. Estas capacidades são importantes para nosso trabalho, e são uma das bases da automatização da manutenção. Como dizem Irvine e Brackett /I-B 77/,"o sistema PSL/PSA (/T-H 77/) suporta principalmente a fase de análise da construção de um sistema"; em /D-V 77/ encontramos uma extensão à fase de manutenção.

A construção de um sistema evolui através de várias "fases". Lembrando Teichroew e Hershey /T-H 77/ podemos dizer que "fase" é cada uma das formas em que o projeto existe entre seu estado conceitual e seu estado operacional. No entanto,a decomposição da construção em fases difere entre autores, e adotaremos a decomposição de Hesse /HES 80/, cujo modelo apresentaremos com algum detalhe no sub-título seguinte. Usamos nesta exposição muitos termos sem apresentarmos um conceito preciso. Por exemplo, "módulo" é utilizado com sentido muito amplo em Davis e Vick /D-V 77/. Vamos então definir os conceitos representados por cada termo de maneira mais ordenada, à medida que apresentamos as fases de decomposição da construção do sistema no modelo de Hesse.

Antes de passarmos a essa apresentação, entretanto,de

vemos tecer algumas considerações sobre "documentação". Segundo Teichroew e Hershey /T-H 77/, é universalmente aceito que a documentação é o elo fraco na construção de sistemas em geral. O computador deve ser utilizado para gerar o máximo de documentação possível, correlacionando as informações armazenadas no banco de dados que contém a descrição do sistema. Daqui surge a necessidade de descrever o sistema em uma linguagem processável pelo computador, pois "linguagens naturais não são suficientemente precisas para descrever um sistema e diferentes leitores podem interpretar uma sentença de diferentes maneiras". No entanto, concordam que uma informação narrativa é necessária para legibilidade. Esta narração deve ser armazenada no banco de dados, embora não seja analisada pelo computador, pois o fato de incluí-la em conjunto com a descrição final auxilia a detecção de discrepâncias e inconsistências. Consideramos fundamental este ponto, pois mesmo a ausência da narração ao lado de uma descrição pode traduzir uma informação importante: aquele pode ser um ponto fraco do projeto. Já Davis e Vick /D-V 77/ julgam pouco interessante a inclusão de texto narrativo, devido à morosidade do processo manual de leitura e revisão, e preferem estabelecer um compromisso entre um texto natural e uma linguagem processável pelo computador, traduzido na estrutura de sua linguagem RSL: com um pós-processamento esta se torna virtualmente indistinguível do inglês corrente. Este objetivo, porém, implica em uma complexidade que está acima do escopo de nosso trabalho, e preferimos uma linguagem mais próxima da máquina, associada a um texto narrativo.

Como conclusão desta análise breve, vemos que o núcleo do suporte automático de construção de sistemas é o gerenciamento de um banco de dados que contém a descrição do sistema, suas especificações e requisitos, e textos associados. Este gerenciamento permite a produção de vários tipos de relatórios, sumários e análises das informações já existentes sobre o sistema, de modo a facilitar, coordenar e orientar os esforços das pessoas envolvidas na sua construção. Segundo Teichroew e Hershey /T-H 77/, um aspecto importante do suporte automático é

que toda a documentação que realmente existe está contida no banco de dados. Consequentemente, as falhas e omissões são bastante aparentes, e uma organização não necessitará depender de pessoas (que poderão não estar disponíveis) quando quiser informações sobre um ítem específico.

1.3 - Um modelo de construção de sistemas

O modelo de Hesse /HES 80/ descreve nove fases na evolução de um sistema:

- análise
- definição
- projeto do sistema
- projeto dos componentes
- implementação dos módulos
- integração dos sub-sistemas
- integração do sistema
- instalação
- operação e manutenção

Estas fases representam uma estrutura temporal do desenvolvimento do sistema, embora as atividades de várias fases normalmente se sobreponham no tempo. Desta forma, as fases são principalmente um meio conveniente de agrupar classes distintas de atividades.

A fase de análise consiste na investigação do problema abstrato que se propõe resolver com uma solução a ser executada em um computador específico. O resultado da análise é denominado "análise do problema", e deve coletar as informações necessárias para a especificação dos requisitos.

A fase de definição consiste na descrição dos requisitos do sistema tendo como enfoque principal suas funções e comportamento desejados. "Requisito" é utilizado no sentido estrito de "exigência". Estes requisitos são descritos em alguma lin

guagem apropriada (PSL, RSL, etc) e armazenada em um banco de dados, denominado "catálogo de requisitos". Os requisitos do sistema formam a base para o início das fases de projeto.

O núcleo das fases de projeto é a "modularização", ou seja, a decomposição do sistema em unidades menores, e a descrição de todas as unidades. Irvine e Brackett /I-B 77/ citam a definição de modularização como "estruturação orientada aos objetivos". Davis e Vick /D-V 77/ citam que é necessário haver uma técnica de estruturação que permita descer a níveis mais baixos de detalhe de maneira ordenada. Para a decomposição Hesse utiliza uma estrutura de árvore, definindo a "árvore de decomposição do sistema" (figura 1.3.1).

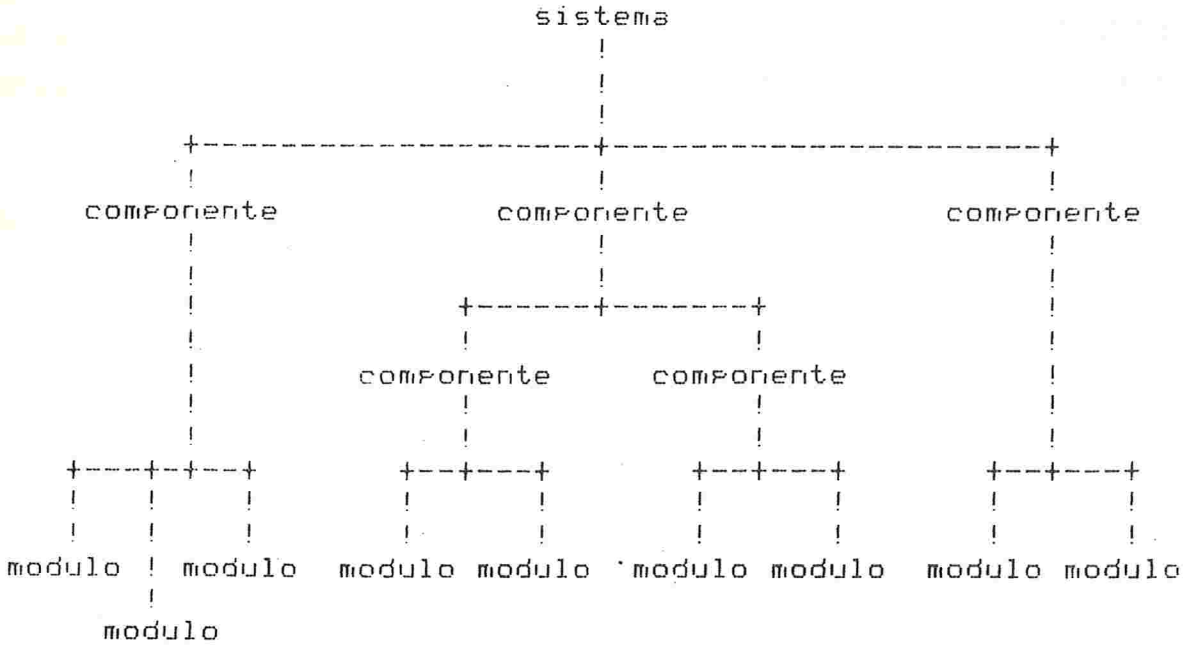

Fig. 1.3.1: A arvore de decomposicao

O nó raiz da árvore é o "sistema", suas folhas são os "módulos", e todos os outros nós são "componentes". Hesse não define completamente "módulo", mas estabelece como conceitos básicos que:

- cada módulo é confeccionado por uma única pessoa.

- módulos não são decomponíveis em unidades menores.
- módulos são definidos como "data abstractions" conforme a formalização de /L-Z 75/, ou seja: grupos de dados associados a operações, onde a representação dos dados pode ser alterada apenas através das operações associadas.

O conceito de módulo como não decomponível não é utilizado por Davis e Vick /D-V 77/ ou Irvine e Brackett /I-B 77/, que definem módulo de maneira ampla o suficiente para englobar "módulos que são compostos de outros módulos". Irvine e Brackett /I-B 77/ citam que um módulo é uma entidade que pode ser manipulada (alterada, eliminada) como um todo. Isto reflete o fato de que duas sub-árvores disjuntas podem ser manipuladas separadamente sem que uma altere a estrutura da outra. No entando, o uso da palavra "componente" para uma composição de módulos evita ambigüidades, e neste trabalho a palavra "módulo" será utilizada apenas para entidades não decomponíveis; sua definição será apresentada no próximo capítulo.

O projeto de cada parte começa com seus requisitos. Os requisitos do sistema derivam de sua definição; os requisitos de um componente derivam do projeto de seu ancestral na árvore de decomposição. Dos requisitos de cada ítem (sistema ou componente) derivam os passos necessários à sua montagem ou execução, e que descrevem uma solução técnica adotada ou um conjunto de possíveis soluções equivalentes.

A fase de implementação dos módulos começa como um projeto: cada módulo é especificado a partir de seus requisitos, utilizando a formalização de Liskov e Zilles /L-Z 75/. A implementação de um módulo consiste em preparar um programa para ele, documentar e verificar o programa. Lembrando a definição de sistema dada por Hesse, mencionada em 1.1, vemos que seus módulos são programas ou segmentos de programas, tais como procedimentos ("procedures"). A verificação de um módulo (programa) é feita por três meios: o teste de seu comportamento, utilizando um conjunto de dados para teste preparados em conjunto com o programa ("teste de caixa preta"); a prova formal de sua corre-

ção baseada nas especificações para o módulo; e a leitura atenta do programa em confronto com a solução técnica adotada. Este último meio de verificação é muito pouco confiável, embora bastante utilizado. Para incorporarmos a verificação através de provas formais de correção, é necessário que o programa tenha sido preparado com base em técnicas formais, a exemplo de transformações de programas.

Segundo Manna e Waldinger /M-W 78/, "as técnicas derivadas da lógica matemática prometem uma alternativa para a metodologia convencional de construir, corrigir e otimizar programas; estas técnicas devem levar à automação de muitas das facetas do processo de programação". A aplicação da lógica matemática à programação é um campo recente, objeto de inúmeras pesquisas /L-Z 75/ /M-W 78/ /M-W 79/ /BAL 81/ /B-P 81/ /DEA 81/ /WIL 81/ /A-R 81/. No entanto, estas técnicas são ainda muito limitadas e pouco utilizadas, e podemos aceitar a posição de Teickroew e Hershey /T-H 77/: o projeto é essencialmente um processo criativo e não pode ser automatizado. Na prática, teremos apenas verificações por teste e leitura, e uma boa parte dos erros de um programa serão detectados apenas quando em operação.

As fases de integração são o inverso das fases de projeto. A árvore de decomposição do sistema tem sua correspondência na árvore de integração, onde os módulos são as folhas, o sistema é o nó raiz e todos os outros nós são subsistemas. No caso ideal, as duas árvores possuem a mesma estrutura,e os subsistemas são congruentes aos componentes. A integração se faz partindo-se das folhas e chegando à raiz, unindo os módulos em subsistemas, e estes em subsistemas de nível mais baixo. A verificação de um subsistema utiliza um conjunto de dados de teste preparados durante o projeto do componente que lhe corresponde. A integração do sistema consiste na ligação de todos os subsistemas de nível 1(o nível imediatamente superior ao da raiz). Esta ligação pode não existir, no caso em que os subsistemas de nível 1 sejam completos e independentes.

A fase de instalação consiste apenas em inserir o sis

tema dentro do contexto operacional de destino. Os processos necessários para a instalação serão discutidos com detalhe no capítulo 2.

A fase de operação e manutenção encerra o ciclo de desenvolvimento do sistema. Estritamente falando, a construção do sistema está terminada, e passa-se ao desenvolvimento de algum outro sistema. Vemos aqui implícita a afirmação de Liu /LIU 76/: podemos assumir com segurança que o grupo de pessoas encarregadas da manutenção do sistema não é o mesmo grupo que originalmente o projetou. Deste fato surgem os principais problemas existentes na manutenção de sistemas.

CAPÍTULO 2

A MANUTENÇÃO DE SISTEMAS

Neste capítulo apresentamos os problemas encontrados na manutenção de sistemas, e discutimos as capacidades que devem existir em um suporte automático à fase de manutenção. Apresentamos um modelo de descrição de sistemas capaz de incluir as capacidades necessárias a esse suporte, e discutimos com detalhe os elementos do modelo, suas relações com os problemas de manutenção, e a descrição de cada elemento.

2.1 - O problema da manutenção

Segundo Liu /LIU 76/, manutenção de um sistema significa, tradicionalmente, modificar programas para emitir novos relatórios, alterar sua lógica para incorporar novos objetivos, expandir arquivos para incluir novas informações, gerar novos arquivos, etc. De maneira mais geral, é o processo de adaptação à contínua evolução dos objetivos de uma organização. Para Roze e Nyman /R-N 78/, além dessa adaptação, a manutenção consiste na correção dos erros do sistema e no ciclo perpétuo de injetar novos erros a cada nova correção.

O custo da manutenção de sistemas, comparado ao custo de seu projeto, tem se revelado extremamente alto. O grupo de suporte de "software" de uma organização dedica 75% /COO 78/ a 95% /LIU 76/ de seu tempo à manutenção, produzindo um custo até 50 vezes maior que o custo do projeto /R-N 78/. Apesar disso, Liu /LIU 76/ cita que um curso típico em desenvolvimento de sistemas devota menos de 5% do tempo à fase de manutenção. Em /C-S 78/ encontramos uma discussão extensa sobre os principais pontos da metodologia de projetos que afetam o custo da manutenção de um sistema.

Segundo Cooper /COO 78/, a ciência da computação não está preparada para enfrentar o mundo real dos problemas não

acadêmicos. Apesar de todos os esforços realizados no sentido de formalizar o desenvolvimento de um sistema, criando normas e convenções (ao menos internas à organização), sempre se apresentam excusas comuns que levam ao não cumprimento das normas. Como conseqüência, qualquer mudança futura em um sistema implica em um esforço equivalente a um novo projeto, e as fases iniciais do desenvolvimento do sistema deverão ser repetidas. Sem dúvida, quando mais de uma pessoa participa da construção de um sistema, é necessário que todos obedeçam ao mesmo conjunto de normas, pois esta uniformidade reduz drasticamente o trabalho de manutenção. No entanto, esta padronização é vista por muitos analistas como limitação à sua liberdade e criatividade. Isto não é verdade quando as normas são aplicadas inteligentemente, e as barreiras existentes são de caráter psicológico apenas. Embora desagradável, a coerção se torna necessária.

Um dos obstáculos sempre presentes é a utilização de linguagens de montagem ("assembly-level languages"), cujo impacto no custo da manutenção é sempre subestimado. Segundo Cooper /COO 78/, quase todas as razões apresentadas para a utilização das linguagens de montagem são inválidas, e uma organização deve considerar tal utilização como um pecado capital. Não obstante, embora as linguagens de alto nível sejam capazes de gerar códigos tão ou mais eficientes, ainda é necessário recorrer à coerção para evitar o uso indiscriminado das linguagens de montagem.

Uma das desculpas para não se utilizar as normas /COO 78/ consiste na alegação de que um programa será utilizado apenas umas poucas vezes e logo abandonado. Isto acontece principalmente quando da confecção de um programa de "emergência" para a emissão de um novo tipo de relatório. A prática demonstra que tais programas tendem a continuar em uso por anos, necessitando contínuas alterações e complicando o problema da manutenção, principalmente lembrando que normalmente o grupo de manutenção não é o mesmo que projetou o sistema /LIU 76/.Assim, o grupo de manutenção tem dificuldade em entender os aspectos mais gerais dos propósitos do sistema, quase nunca cons

tantes de maneira clara na documentação existente, e tendem a tomar decisões sob um ponto de vista estreito, focalizando às vezes o problema imediato causado por erros de um programa. Estes erros criam, em geral, uma situação com caráter de urgência, o que também impede o bom cumprimento das normas e convenções. Segundo Liu /LIU 76/ os analistas de manutenção sentem que estão continuamente "apagando incêndios", e isto necessariamente distorce sua visão do sistema.

O caráter de urgência da manutenção produz naturalmente a deficiência na documentação das mudanças efetuadas, tendendo a tornar a documentação existente progressivamente obsoleta. Embora julguemos que alguma documentação, mesmo obscura, sempre é melhor que nenhuma documentação, também é verdade que uma documentação que não reflete a realidade devido à completa obsolescência pode ser mais danosa que sua ausência completa.

A situação ideal, na manutenção, seria a de um sistema projetado dentro da própria organização, seguindo as técnicas apresentadas no capítulo 1, com toda a documentação organizada, automatizada e disponível ao grupo de manutenção. Toda mudança feita poderia ser integrada ao banco de dados contendo a descrição do sistema, e a documentação atualizada estritamente dentro das normas da organização. No caso de alterações de urgência, essa mudança poderia ser integrada a posteriori, após passado o período de crise. Esta situação ideal é ainda inatingível na maioria das organizações atuais. As situações de crise tendem a surgir ininterruptamente, os analistas de suporte possuem pouco incentivo para retomar alterações anteriores e integrá-las corretamente ao sistema, e uma organização possui poucos mecanismos de controle direto capazes de forçar a aderência às normas /COO 78/. Segundo Cooper /COO 78/ o meio de controle mais eficaz é a chamada "regra de ouro": aquele que tem o ouro faz as regras. Do ponto de vista econômico, a aderência às normas durante a manutenção reduz os custos, e embora o trabalho de policiamento seja pouco popular, é necessário. Em nossa opinião, deve-se dedicar um grupo de analistas à auditoria constante do processo de manutenção.

A documentação das alterações efetuadas durante a manutenção também tende a se fixar apenas naquilo que foi feito, e como foi feito, sem se importar em descrever o porquê da alteração /LIU 76/. Se automatizarmos a documentação de quais alterações existem, facilitaremos à auditoria extrair os porquês.

Paralelamente ao problema da documentação na manutenção, surge o problema da recuperação do sistema em algum estágio anterior, notadamente para se determinar em que ponto um erro foi injetado devido à correção de outro erro. A utilização de mantenedores de programas fonte ("librarians") /M-O 76/ /B-J 76/ tende a fazer com que se perca o histórico das alterações do sistema, obrigando a manter nos meios de armazenamento do computador várias versões dos programas fonte, ou colocá-las em microfilme /M-O 76/; além do alto custo de armazenamento, torna-se difícil reconstituir o sistema tal como era em alguma data anterior. A manutenção do histórico do sistema e a capacidade de recuperação de versões anteriores com o armazenamento de quantidades mínimas de informação constitui uma característica altamente desejável.

Outro problema crítico na manutenção é a decisão de como manter sistemas adquiridos de terceiros. Segundo Cooper /COO 78/ é importante não se deixar ficar na dependência completa do suporte dado pelo originador do sistema, devendo-se criar mecanismos próprios de manutenção. No entanto, a manutenção deste tipo de sistemas é de grande dificuldade, dada a ausência de documentação minuciosa. Podemos apenas reduzir um pouco este problema aliviando o analista de suporte das tarefas puramente mecânicas da manutenção. Podemos considerar o problema da manutenção de sistemas adquiridos de terceiros como um problema de "implantação": um caso particular de construção de sistemas onde existem apenas as fases de "adaptação", integração dos sub-sistemas, integração do sistema, instalação e operação e manutenção. A fase de adaptação consiste em analisar o sistema, projetar e implementar componentes novos ou

implementar de maneira diferente alguns módulos. Quando o sistema foi projetado para a máquina física a ser utilizada, a adaptação pode ser necessária para conformar o sistema às limitações e normas existentes na organização. Quando projetado para máquina diferente da disponível, implica em converter manualmente um grande número de comandos de linguagens de programação, uma vez que até o momento não há, ainda, perspectivas de uma portabilidade absoluta. Em ambos os casos, temos o problema da verificação, e necessitamos preparar arquivos de dados para teste de caixa preta. A fase de adaptação gera novas descrições do sistema, que deverão ser sobrepostas à descrição original e incluídas no suporte de manutenção.

Uma grande parte das tarefas mecânicas na manutenção de sistemas provém da fase de instalação. Ao término da fase de integração do sistema, temos um certo número de programas objeto e de arquivos distribuídos em meios de armazenamento diferentes (fitas ou discos) ou em volumes diferentes do mesmo meio. O sistema, após instalado, deverá estar acessível a seus usuários com nomes de arquivo específicos e em volumes específicos. Isto requer a movimentação de arquivos entre meios ou volumes, e a manipulação de seus nomes, ou "rótulos". Normalmente dispõe-se de um "job" que descreve a seqüência de processos necessários à movimentação de todos os arquivos do sistema, e quando este é muito grande é necessário extrair e executar uma parte apenas desse "job" por razões de economia. Desta forma, uma alteração no sistema exige do analista a confecção de um "job" por um processo manual, e muitas vezes a montagem de outro "job" para executar a fase de integração, isto é, compilação e ligação dos componentes. Neste ponto é muito comum o esquecimento de alguns arquivos atingidos pela alteração, resultando em um sistema conflitante e não utilizável.

A necessidade de instalação do sistema pode surgir também devido à destruição de volumes de armazenamento. Um dos modos correntes de se precaver contra esta destruição é executar periodicamente a duplicação dos volumes de acesso aleatório (discos magnéticos), criando os chamados "backups".Além de

ser um processo de custo muito alto, ainda restará o trabalho manual de verificação e confronto das versões existentes na duplicata com as versões mais atuais de todos os sistemas, e reinstalar o que for necessário. A automatização da instalação pode eliminar o custo da duplicação dos volumes de disco, bem como eliminar o tempo perdido no confronto de versões.

Outro problema correlato surge quando, por razões administrativas ou técnicas, decide-se alterar os rótulos dos arquivos do sistema, ou torná-lo acessível em outro volume de armazenamento; isto costuma exigir alterações extensas no "job" dedicado à instalação.

Após esta exposição dos problemas usualmente encontrados na manutenção de sistemas, vamos apresentar uma forma de descrever um sistema orientada à solução dos problemas aqui citados. Lembrando que muitas vezes a manutenção do sistema exige a repetição de algumas partes das fases de projeto e implementação /COO 78/, vemos que ao descrever o sistema temos que nos preocupar em oferecer também uma ferramenta de suporte ao projeto, com algumas das capacidades mencionadas em 1.2 /T-H 77/ /D-V 77/.

Esta parte de suporte ao projeto poderá ser utilizada para projetos novos dentro da própria organização, ou utilizar uma ferramenta que já esteja disponível na instalação. Na implementação feita por nós incluímos um suporte próprio para o projeto, que pode ser utilizado de modo exclusivo ou em conjunto com qualquer outro suporte que venha a existir.

Tomaremos como capacidades necessárias ao suporte da manutenção as seguintes:

- descrever sistemas, mesmo quando desenvolvidos por terceiros.
- armazenar essa descrição em um banco de dados, associada a documentação narrativa.
- adicionar, modificar ou eliminar informações contidas no banco de dados.

- produzir documentação para uso dos analistas ou usuários do sistema.
- gerar componentes de maneira independente.
- gerar automaticamente todas as partes dependentes afetadas pela modificação de um módulo ou componente específico.
- integrar o sistema ou partes do sistema.
- instalar automaticamente o sistema completo ou apenas partes do sistema que tenham sido modificadas.
- manter e documentar o histórico das alterações do sistema através de seus vários estados ou versões durante sua fase operacional.
- recuperar o sistema em qualquer versão anterior.
- efetuar a verificação automática do sistema instalado em confronto com a versão mais recente, eliminando discrepâncias.

Estas capacidades devem ser implementadas em um conjunto de ferramentas de "software", ao qual denominaremos "sistema mantenedor". No entanto, a implementação completa de todas essas capacidades é um objetivo por demais ambicioso.Vamos adotar um enfoque que produza um modelo abrangendo todas as capacidades citadas, embora a implementação existente no momento possua algumas dessas capacidades em forma apenas embrionária.

2.2 - A árvore de geração de um sistema

Segundo Teichroew e Hershey /T-H 77/ uma linguagem de descrição de um sistema deve ser do tipo relacional, onde uma descrição consiste em identificar e rotular elementos com suas interrelações, com uma implementação orientada para a utilização de banco de dados. Segundo Davis e Vick /D-V 77/ os conceitos básicos de uma descrição estão contidos em quatro primitivos: elementos, atributos, relações e estruturas. Com esta base desenvolveremos nossa linguagem de descrição de sistemas, identificando os pontos necessários à solução dos problemas expostos em 2.1.

Segundo nossa definição, dada em 1.1, um sistema se constitui de objetos, cada objeto sendo ou o resultado de uma seqüência de processos executados em um computador, ou um arquivo nele armazenado por um processo manual. Estes objetos serão parte dos elementos que descreveremos.

Retomemos o modelo de Hesse /HES 80/ descrito em 1.3 e tomemos o caso ideal em que a árvore de integração é congruente à árvore de decomposição (figura 1.3.1). Assim, sub-sistemas são equivalentes a componentes, e podemos usar esses termos sem distinção. A fase de integração percorre a árvore das folhas à raiz, ligando módulos em componentes, estes entre si, até a integração final do sistema. Temos então, associada a cada nó "componente", uma seqüência de processos que efetuam a integração desse nó. Ao nó raiz, "sistema", temos associada uma seqüência de processos adicionais necessários à sua instalação. Se em cada um desses nós incluirmos a seqüência de processos necessários à sua composição, obteremos os elementos da representação do sistema, e adotamos a mesma estrutura de árvore, que denominaremos "árvore de geração do sistema" (figura 2.2.1).

O nó raiz é o "sistema", as folhas são "módulos", e todos os outros nós são "unidades". No caso ideal, a árvore de geração é congruente à árvore de decomposição, e um nó "unidade" corresponde a um nó "componente".

De um modo geral cada elemento do sistema, dado por um nó da árvore (unidade ou sistema) é formado pela união do componente (ou sistema) que lhe corresponde com a seqüência de processos que o geram, e a descrição desse elemento deverá conter a especificação da seqüência de processos. Os módulos não exigem processos para sua montagem, uma vez que são indivisíveis em partes menores, e sua descrição será baseada na própria definição.

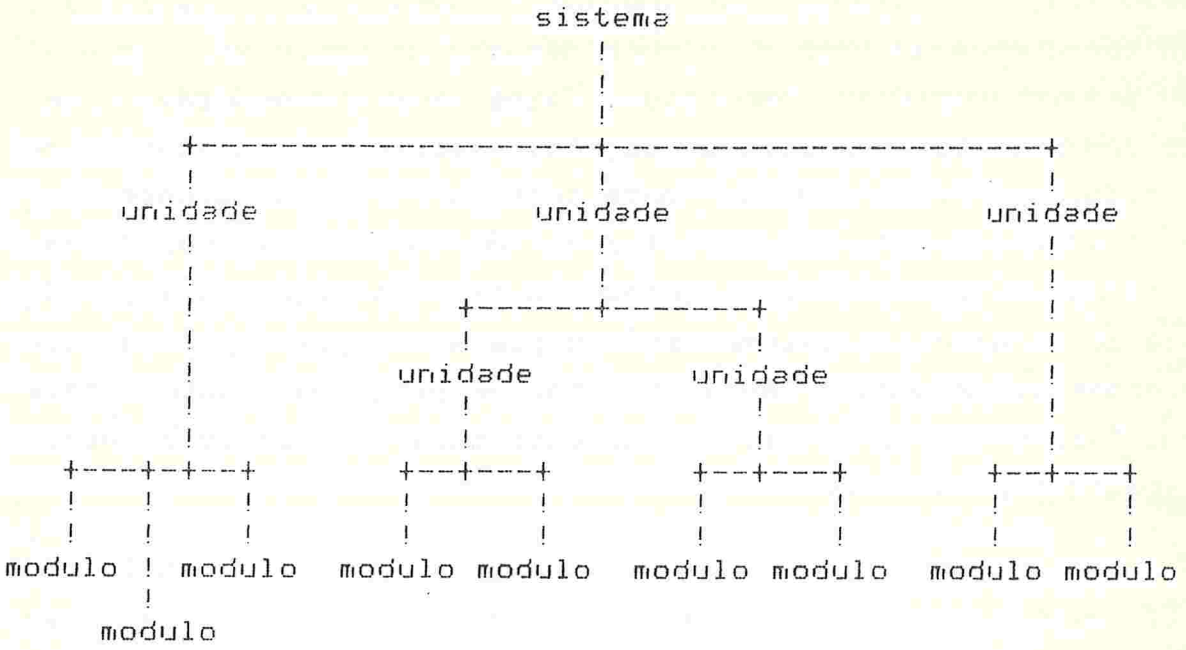

Fig. 2.2.1: A arvore de geracao

2.3 - O elemento "módulo"

Um módulo, como utilizado neste trabalho, será definido por três conceitos básicos:

- módulos não são decomponíveis em entidades menores.
- módulos são arquivos armazenados em um computador através de um processo manual; são, portanto, objetos da classe 2) na definição de "sistema" dada em 1.1.
- um módulo, do ponto de vista da manutenção, é inalterável a partir do instante em que se encerrou a fase de projeto e implementação.

Estes conceitos devem ser analisados sob o ponto de vista do grupo de manutenção. Quando um sistema é adquirido de terceiros, normalmente não poderemos nos beneficiar da estruturação e modularização de seu projeto, e em geral não teremos

acesso sequer à documentação interna. Neste caso, os módulos serão arquivos compostos de módulos através de processos que não tentaremos adivinhar, e consideramos um processo manual seu armazenamento no computador. Estes arquivos poderão ser programas fonte ou programas objeto, e serão manipulados como um todo; se houver arquivos para teste preparados durante a fase de adaptação, estes constituirão módulos do sistema. No caso em que o sistema foi projetado na própria organização, a árvore de geração do sistema está completa, em princípio, ao fim do projeto. Os módulos serão trechos de programas fonte (tais como "procedures", descrições de estruturas de arquivos, etc.) e arquivos para teste.

O terceiro conceito de módulo, como sendo inalterável, requer uma discussão mais profunda. Ele constitui a base para a manutenção e documentação da história das alterações do sistema, e possibilita a recuperação do sistema em uma versão anterior. Uma vez que um módulo é inalterável, necessitamos um mecanismo para introduzir alterações no sistema. Utilizamos então o conceito de "patch": um "patch" constitui uma mudança específica em um módulo do sistema, e é formado por um grupo de alterações que se relacionam a essa mudança nesse módulo. Possui como atributos uma data e uma razão da alteração. A data é mantida automaticamente e fornecida pelo computador no instante de sua introdução no sistema mantenedor, e sua razão é descrita como texto narrativo, sem verificação automática. Alterações em um módulo ao longo do tempo constituem uma seqüência de "patches", e esta seqüência fica armazenada em um "arquivo de 'patches' do módulo", juntamente com seus atributos.Uma vez armazenado um "patch" nesse arquivo, ele não deve mais ser retirado. Cada vez que um módulo vai ser utilizado na composição de uma unidade, executaremos antes um processo que aplica o arquivo de "patches" sobre o módulo, tendo como resultado um "módulo atualizado". Este, após ser utilizado na geração de um componente do sistema, deve ser removido deste, sendo estritamente temporário; isto significa que ao utilizarmos um módulo aplicaremos sobre ele todas as alterações já havidas,mesmo que

uma alteração mais recente se sobreponha a uma anterior. Como resultado, temos um registro fiel e permanente da história das alterações do sistema; podemos recuperar cada módulo, e consequentemente o sistema, em qualquer forma anterior, bastando selecionar "patches" com datas anteriores à desejada; estamos armazenando uma quantidade mínima de alterações, uma vez que o custo do armazenamento do arquivo de "patches" é menor que o de armazenar módulos completos em versões consecutivas; o custo do processo de geração do "módulo atualizado", embora em alguns casos possa ser superior ao custo de armazenamento de algumas versões do módulo, é fartamente compensado pela capacidade resultante de manutenção e documentação da história, produzindo um menor custo final de manutenção. Notemos ainda que tanto o arquivo de "patches" do módulo como o "módulo atualizado" são objetos que satisfazem nossa definição de sistema, embora não possam ser denominados de módulos.

Os atributos de um módulo são alguns dos atributos normais de arquivos: um nome, ou "rótulo", o meio e volume específicos em que está armazenado, sua estrutura, ou modo de acesso, seu tipo (programa fonte, programa objeto, arquivo de dados, etc.), sua data de criação. A estes atributos adicionaremos mais um: a existência ou não de alterações. Estes atributos devem constituir a descrição do módulo.

A data de criação normalmente é fornecida pelo computador, e pode ser armazenada e verificada automaticamente. Tanto o tipo como o modo de acesso são mantidos pelo sistema operacional de alguns computadores, embora a maioria não possua a capacidade de manter o atributo "tipo", não permitindo a distinção entre um programa fonte e um arquivo de dados. Estes dois atributos, no entanto, estão determinados pelo processo que for utilizar o módulo; além disso, o mesmo arquivo pode ser utilizado ora como programa fonte, ora como arquivo de dados por processos diferentes. Assim, deixam de ser relevantes e os retiramos da descrição do módulo, embora julguemos inte-

ressante sua inclusão como comentário sempre que possam ser fornecidos pelo sistema operacional.

Restam então, na descrição do módulo, apenas o rótulo, o meio de armazenamento, o volume desse meio e a existência de alterações. A descrição dos três primeiros atributos caracteriza completamente o módulo, pois: ao definirmos o módulo como inalterável, tanto o rótulo como o local de armazenamento são fixados e imutáveis, e a existência de alterações pode ser constatada automaticamente, dada uma convenção apropriada que ligue o rótulo do módulo ao rótulo do arquivo de "patches" do módulo. Assim, podemos descrever um módulo de modo conciso através de seu "título", definido como:

```
<título> ::= <rótulo> <meio> .
<meio>   ::= ON <especificação do volume de acesso
                 aleatório> |
             IN <especificação do volume de acesso
                 seqüencial> |
             Ø
```

Quando o meio for a cadeia vazia Ø, significará o volume padrão de trabalho do computador.

Um título, como aqui definido, possui uma marca de término, o ponto (.). A escolha desta marca de término depende da escolha do modo de descrever <rótulo>, <especificação do volume de acesso aleatório> e <especificação do volume de acesso seqüencial>, que depende de considerações de compromisso entre portabilidade e simplicidade na programação do sistema mantenedor. Basicamente temos duas estratégias possíveis: adotarmos as construções usuais da linguagem de comandos do computador que utilizamos, ou definir uma linguagem a ser traduzida para diversas máquinas. Vamos deixar as considerações sobre esta escolha para o próximo capítulo.

Uma vez que a utilização de um módulo por um processo implica obrigatoriamente na execução do processo de sobreposição com o arquivo de "patches", deveríamos dizer de maneira

mais exata que um processo utiliza um "módulo atualizado". No entanto, por simplicidade, diremos que um processo utiliza um módulo para gerar um componente, ficando implícito que esse uso exigiu a criação do módulo atualizado com caráter puramente transitório.

2.4 - O elemento "unidade"

Uma unidade consiste na união de um componente com a especificação da seqüência finita de processos que, quando executados, resultam nesse componente (figura 2.4.1).

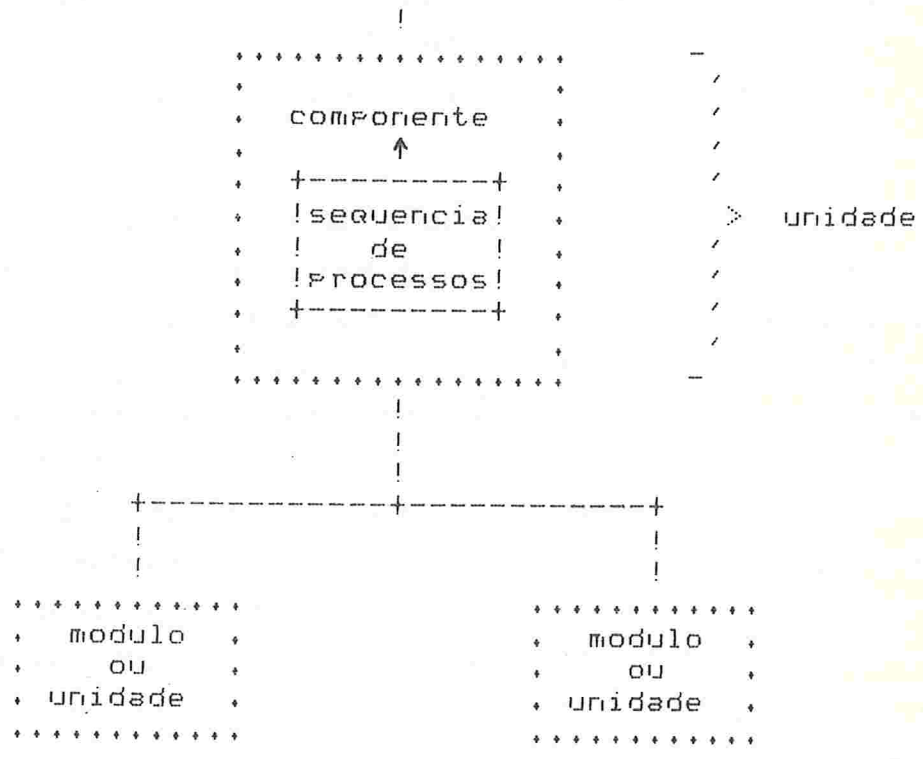

Fig. 2.4.1 - O nó "unidade" na arvore de geracao

Escolhemos o nome "unidade" para esse elemento devido ao fato de que ele constitui, basicamente, uma unidade de trabalho na geração do sistema. O componente associado à "unidade" é uma parte do sistema descrita e identificada quando de

sua decomposição na fase de projeto, e existe em caráter permanente. Esse componente é o resultado da execução de uma seqüência finita de processos em um computador, e sempre que necessitamos criá-lo novamente devemos executar a mesma seqüência desde seu início, formando assim a idéia de unidade de trabalho.

Notemos, do que foi dito acima, que um componente é um objeto de classe 1) na definição de sistema, como dada em 1.1, e uma "unidade" não é um objeto do sistema, mas apenas uma estrutura contendo a descrição de um objeto de classe 1) do sistema, o componente. Podemos dizer que o componente é o resultado da unidade, sendo o único resultado com caráter permanente. Ao executarmos a seqüência de processos teremos,em geral, outros resultados intermediários, mas estes serão estritamente temporários e removidos do sistema assim que for possível.

A seqüência de processos de uma unidade utiliza módulos e unidades na composição do componente, seguindo os ramos da árvore de geração. Ao dizermos que utilizamos uma unidade, estamos dizendo na verdade que utilizamos o componente que é seu resultado: não necessitamos executar seus processos, uma vez que um componente existe em caráter permanente.Notemos ainda que o primeiro processo da seqüência não utiliza, necessariamente, todos os módulos e unidades determinados pelos ramos da árvore, alguns sendo utilizados apenas em processos de ordem mais alta na seqüência. Esta informação, que não está explícita na estrutura da árvore de geração, fará parte da especificação dos processos.

A descrição de uma unidade consiste, então, na descrição do componente, na descrição das especificações dos processos da seqüência (que forma parte da estrutura da unidade), e na descrição de suas relações com outros módulos e unidades. Estas relações são as arestas da árvore, e indicam quais módulos são necessários para produzir seu componente. Às arestas associamos uma orientação, no sentido das folhas à raiz, e es-

tas arestas orientadas definem uma relação entre elementos que possui a propriedade transitiva: se o nó (ou folha) A é usado na composição do nó B, e o nó B é usado na composição do nó C, então A é usado na composição do nó C. Desta forma, temos um caminho único orientado, partindo de qualquer nó ou folha e chegando à raiz; uma alteração em algum nó ou folha propaga-se através desse caminho até a raiz, o sistema. Denominaremos "trajetória hierárquica" ou simplesmente "trajetória" de uma alteração a esse caminho único, que parte do elemento sobre o qual incidiu a alteração e percorre as arestas orientadas no sentido destas, terminando na raiz.

A relação definida pelas arestas orientadas é uma relação não reflexiva, pois uma unidade não pode ser parte da composição de si mesma. É uma relação não simétrica, pois se a unidade A é usada na composição da unidade B, é evidente que B não pode ser usada na composição de A. Logo, temos uma relação de ordem estrita, e as propriedades de reflexibilidade e simetria podem ser testadas de forma automática: sua presença indica um erro na descrição do sistema.

Os atributos de uma unidade são: um nome que a caracteriza, os atributos do componente que é seu resultado, e os atributos da seqüência de processos que quando executada teve como resultado esse componente.

Os atributos do componente são os atributos normais de arquivos, já mencionados em 2.3, e deles tem relevância apenas a data de sua criação; esta é fornecida pelo computador e mantida de forma automática, e será relevante para a verificação da instalação do sistema. O atributo rótulo pode ser formado através de uma regra arbitrária que o derive do nome da unidade. Assim, ao descrever o nome da unidade estamos implicitamente descrevendo o rótulo do componente.

Da seqüência de processos tiramos como atributo o modo do término de cada processo. Notemos que cada processo não faz parte, como um todo, da estrutura da unidade, mas apenas

sua especificação. Uma mudança em algum ponto do sistema propaga-se através da trajetória até a raiz, "ativando" diversas unidades. Ativar uma unidade significa coletar as especificações dos processos nela descritos e efetuar os pedidos ao sistema operacional para iniciar e executar esses processos. O modo de término dos processos, após executados, é coletado e agregado à descrição da unidade. Dizemos que se todos os processos tiveram um término normal, então o componente existe em estado normal, ou em sua versão mais recente. Caso algum processo tenha um término anormal, então o componente existe em estado anormal, e não deve ser utilizado. O atributo "modo do término de cada processo" reduz-se, assim, ao estado normal ou anormal do componente.

A descrição da especificação de um processo depende das mesmas considerações que mencionamos em 2.3 para a descrição de "título", e serão feitas no próximo capítulo.

A especificação da seqüência de processos implica em uma ordem temporal: um processo de ordem i somente deve ser iniciado após o término do processo de ordem i-1, e cabe ao sistema mantenedor cuidar para que os pedidos ao sistema operacional obedeçam a essa ordem temporal. Ao mesmo tempo, deve ter a capacidade de interromper a execução da seqüência quando um processo teve término anormal, uma vez que isto já caracteriza o estado do componente como anormal, e os processos restantes são supérfluos.

A relação de ordem definida pelas arestas orientadas também implica em uma ordem temporal. Ao percorrermos a trajetória de uma alteração até a raiz, estaremos ativando unidades de maneira seqüencial: os processos de uma certa unidade somente devem ter início após o término dos processos da unidade que a antecede na trajetória; temos uma seqüência temporal de seqüências temporais de processos. Analogamente à seqüência dentro de uma unidade, devemos ter a capacidade de interromper a ativação das unidades posteriores quando alguma unidade teve como resultado um componente anormal. Neste caso, os processos

das unidades subseqüentes não são iniciados e o atributo "componente anormal" propaga-se por essa trajetória até o nó anterior à raiz, constando da descrição de cada uma dessas unidades. Notemos ainda que, ao percorrermos uma trajetória, uma unidade pode ficar em estado anormal por duas razões: ou um de seus processos termina anormalmente, ou necessita para sua composição de uma unidade fora da trajetória cujo componente esteja em estado anormal. A segunda razão fica implícita na primeira, se considerarmos que um componente em estado anormal efetivamente não existe, e qualquer processo que deva utilizar esse componente forçosamente terá um término anormal.

2.5 - O elemento "sistema"

O sistema, como nó raiz, pode ser encarado como composto apenas dos nós "unidade" que estão em nível imediatamente superior, ou nível 1 como dado em 1.3. Associamos ao nó "sistema" o conjunto de processos necessários à instalação do sistema após sua integração, e seus atributos devem possibilitar a instalação automática e a verificação automática dessa instalação. Adotamos como atributos deste nó um estado, um nome que o caracteriza, e um destino.

O estado do sistema pode ser normal ou anormal, e deriva dos estados das unidades de nível 1: o estado do sistema é normal somente se todos os componentes dessas unidades estiverem em estado normal.

O nome do sistema o caracteriza no sentido de diferenciá-lo dos outros sistemas mantidos no mesmo computador. Dado o que foi exposto até agora, vemos que um sistema se constitui de um conjunto de arquivos. Durante a integração do sistema estamos gerando arquivos, e desejamos manter a independência de rótulos de arquivos entre sistemas. Como vimos em 2.4, o rótulo de um componente deriva do nome característico de sua unidade. Da mesma forma, incluímos nessa regra de derivação o nome do sistema, e com isto garantimos a unicidade dos rótulos

de arquivos durante a integrão do sistema. Devemos agora distinguir esse conjunto de arquivos em dois instantes diferentes: antes e após a instalação do sistema. Ao término da integração do sistema cada arquivo tem um rótulo conhecido pelo sistema mantenedor, derivado diretamente dos elementos de descrição do sistema; segue-se então a instalação. Devido a normas administrativas da organização, ou mesmo razões técnicas dadas por limitações do computador utilizado, pode ser impossível derivar o rótulo final a partir dos nomes do sistema e das unidades. Assim, somos obrigados a incluir na descrição do sistema os rótulos desejados de cada componente após a instalação.

O destino do sistema consiste na especificação dos processos necessários à sua instalação. Como vimos em 2.1, a instalação do sistema consiste em distribuir seus arquivos em alguns volumes específicos com rótulos específicos. Os processos necessários para isto são apenas os de movimentação de arquivos entre meios e volumes, e os de manipulação de atributos de arquivos. Estes processos não necessitam ser especificados, uma vez que existem como parte integrante do contexto operacional, e sua especificação pode ser gerada pelo sistema mantenedor. Uma vez que os rótulos dos componentes antes da instalação são conhecidos pelo sistema mantenedor, resta-nos apenas descrever o rótulo, meio e volume desejados para cada componente de nível 1, o que fazemos de maneira concisa através do "título" definido em 2.3. Notemos, em primeiro lugar,que pode ser necessário instalar um componente em mais de um volume ou meio de armazenamento, com o mesmo rótulo ou não. Nesse caso, teremos para cada componente de nível 1 um conjunto de títulos, cada um deles determinando um processo de manipulação de atributos e movimentação de arquivos. Em segundo lugar, um novo atributo de arquivo é importante após a instalação: a classe de segurança, ou permissão seletiva de acesso a usuários diferentes. Este atributo deve ser associado a cada título de destino de um componente de nível 1, e a descrição da instalação se constitui de títulos associados à classe de segurança.

Uma mudança em um sistema produz, em geral, alterações em vários componentes, determinando várias trajetórias. Estas trajetórias podem possuir em comum apenas o nó raiz, sendo completamente disjuntas (a menos do nó raiz), ou possuir em comum vários nós "unidade", sendo parcialmente disjuntas. Fica claro, no segundo caso, que a parte comum de duas trajetórias deve ser ativada apenas uma vez, e sua ativação precisa esperar a conclusão das partes disjuntas delas. Podemos dizer que duas sub-trajetórias se unem em uma única trajetória, e identificamos assim três seqüências distintas de ativação de unidades. Notemos aqui que duas unidades pertencentes a sub-trajetórias disjuntas não possuem nenhuma relação, e portanto a relação de ordem definida em 2.4 não é total. Neste caso, não temos restrição temporal entre essas unidades, e a ordem em que são ativadas é irrelevante. Em nossa implementação adotamos uma ativação seqüencial de unidades, com ordem arbitrária quando não há restrição temporal. Nada impede que neste caso se adote ativações em paralelo, se desejado. A experiência mostra que a ativação em paralelo tende a reduzir a produção útil do computador, desde que ele não esteja sub-utilizado.

Os processos de instalação devem ser iniciados apenas após o término de todas as seqüências de todas as unidades ativadas, e devem ser inibidos quando o sistema fica em estado anormal, ou seja, algum componente de nível 1 em estado anormal. Neste caso, algumas trajetórias podem ter sido executadas normalmente, e o sistema resultante desta integração incompleta é quase sempre inconsistente e não utilizável. Sempre existirão casos, na prática, em que será desejável a instalação de um componente assim que esteja disponível, e isto poderia constituir mais um atributo do sistema. Em nossa implementação, optamos pela inibição incondicional, e deixamos a criação desse atributo como uma extensão futura.

A instalação deve ser seletiva: devemos mover apenas os arquivos atingidos pela alteração do sistema, com o intuito de reduzir o custo da manutenção. Ainda com o mesmo intuito de

reduzir custos, quando um sistema fica em estado anormal,devemos retomar apenas as trajetórias (ou suas partes superiores) que produziram unidades em estado anormal, e não refazer as sub-trajetórias ou trajetórias que foram executadas corretamente. Desta forma, a cada vez que chegarmos a um sistema normal, devemos confrontar esta nova versão com o sistema anteriormente instalado para determinar quais arquivos deverão ser movidos; isto significa proceder à verificação da instalação. Esta verificação se faz pela comparação da data de criação do componente com a data de criação do arquivo instalado. Embora a capacidade de verificação da instalação seja um requisito de um sistema mantenedor, e esteja presente nele, efetuar a verificação a cada alteração do sistema pode gerar um custo excessivo. Adotamos, então, mais um atributo dentro da descrição do destino do sistema, que indica o estado de instalação de cada componente de nível 1: inibida ou efetuada. Deixamos, assim, a verificação da instalação como um processo dentro do sistema mantenedor executado apenas através de um pedido específico.

Notemos aqui que a verificação da instalação exige como requisito que o sistema operacional possua a capacidade de manter e informar a data de criação de um arquivo armazenado no computador. A maioria das máquinas físicas possue sistemas operacionais com essa capacidade, embora em alguns casos sejam necessários alguns procedimentos não usuais e obscuros na criação de arquivos, que necessitarão ser incorporados ao sistema mantenedor. Quando o sistema operacional não possuir tal capacidade, sempre será possível criar um banco de dados gerenciado pelo sistema mantenedor que torne possível a verificação da instalação. Este caso não será discutido neste trabalho.

Temos ainda a considerar dois casos particulares do modelo de Hesse. Em primeiro lugar, está implícito em seu modelo o caso em que o nó "sistema" contém um processo de ligação de todos os componentes de nível 1 gerando um único arquivo, o sistema. Neste caso, criamos uma unidade única em nível 1 con-

tendo a especificação desse processo, elevando o nível de todos os outros nós. Em segundo lugar, seu modelo admite a ligação de um módulo diretamente ao nó "sistema", sem nenhum nó "componente" intermediário. Neste caso, criamos um nó "unidade" contendo apenas o processo de duplicação do módulo atualizado, transformando-o em um componente não temporário. Estas duas transformações são bastante simples, e não exigem maiores discussões. Citemos apenas que o segundo caso é muito comum na manutenção de sistemas adquiridos de terceiros.

2.6 - O grafo de geração de um sistema

A representação do sistema em estrutura de árvore é, na grande maioria dos casos, redundante. Um módulo pode entrar diretamente na composição de várias unidades, e da mesma forma uma certa unidade pode ser necessária na composição de várias outras. Assim, muitos nós da árvore de geração são idênticos, possuindo a mesma descrição e atributos. A eliminação dos nós duplicados transforma a árvore orientada em um grafo orientado, ao qual denominaremos "grafo de geração" do sistema (figura 2.6.1). A orientação das arestas é a mesma orientação das arestas da árvore, seguindo a relação de ordem estrita definida em 2.4.

O grafo de geração, além de formar uma estrutura mais compacta, permite visualizar melhor a estrutura interna do sistema e a trajetória gerada por uma alteração. A trajetória de uma alteração, no grafo, pode ser formada pela união de várias trajetórias na árvore, e duas unidades na mesma trajetória podem não possuir relação de ordem, ao contrário do que ocorre na trajetória da árvore. Tomando como exemplo o grafo da figura 2.6.1, uma alteração no módulo A gera uma trajetória (A, C, D, F, G, H), e é a união de três trajetórias da árvore correspondente. Vemos aqui que os pares F,G e C,D não possuem relação de ordem, sendo irrelevante a ordem de sua ativação. No entanto, notemos que a ativação de F exige a conclusão dos pro

cessos C e D; a notação usada acima para a trajetória indica a ordem de ativações das unidades, permitindo transposições quando a relação de ordem não está definida. Assim, são equivalentes as notações (A, D, C, G, F, H) e (A, D, G, C, F, H). Cabem aqui as mesmas considerações feitas em 2.5 sobre trajetórias disjuntas e parcialmente disjuntas.

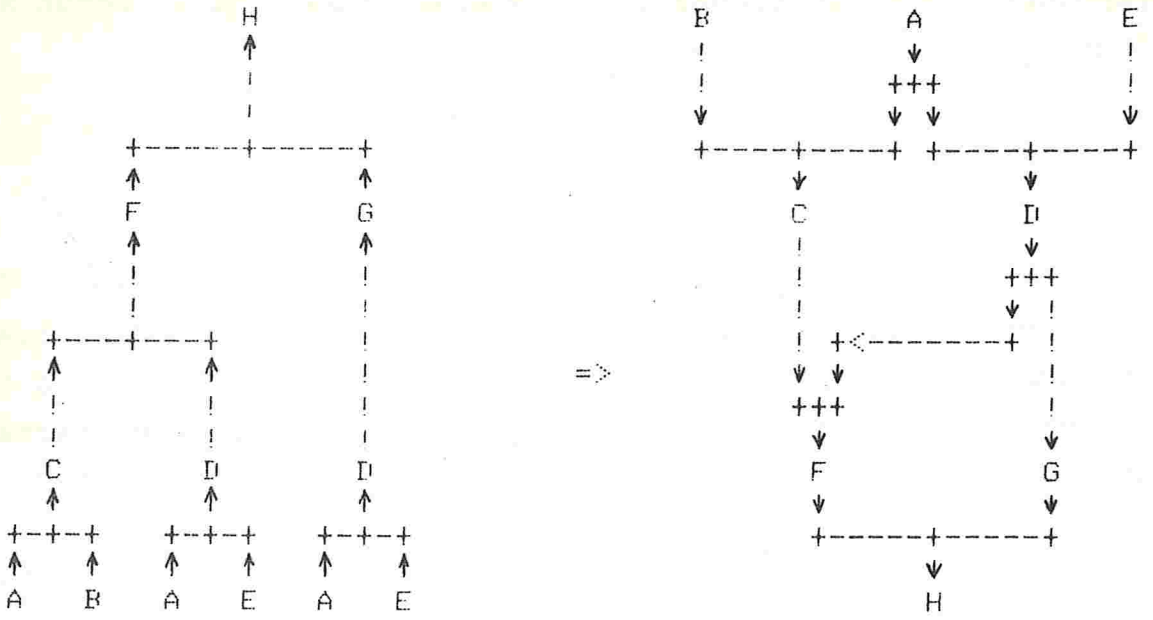

Fig 2.6.1 - Um trecho de arvore e seu grafo associado

A orientação do grafo de geração sugere de maneira natural a nomenclatura de "nó sucessor" e "nó antecessor",oposta à nomenclatura usual de árvores. Assim, o nó "sistema" não possui sucessores, e os nós "módulo" não possuem antecessores. À relação de ordem já definida daremos o nome "usado em",e utilizaremos a notação $\vec{R}$. Esta relação admite uma inversa unívoca, apenas invertendo o sentido da relação. À relação inversa daremos o nome "necessita de", e utilizaremos a notação $\overleftarrow{R}$. Assim A $\vec{R}$ C <=> C $\overleftarrow{R}$ A. Estas duas relações determinam o significado de antecessor e sucessor, e podemos definir o conjunto de antecessores de U_i como $\mathbb{A}(U_i) = \{U_j |\ U_i \overleftarrow{R} U_j\}$. Definimos o conjunto dos sucessores de U_i como $\mathbb{P}(U_i) = \{U_j |\ U_i \vec{R} U_j\}$. Estas

duas relações são necessárias para o sistema mantenedor: a trajetória de uma alteração é determinada através dos conjuntos $\mathbb{P}$ e o atributo estado de uma unidade U_i é determinado pela inspeção das unidades pertencentes a $\mathbb{A}(U_i)$. Notemos aqui que $\mathbb{A}(U_i)$ possui elementos que não pertencem à trajetória da alteração, no caso mais geral.

É necessário descrever apenas uma destas relações, ficando assim determinada a outra. Para nosso sistema mantenedor escolhemos descrever apenas a relação "necessita de", e incluímos essa descrição em cada unidade. Uma mudança no sistema pode, por exemplo, introduzir uma nova unidade, U_k, e ao montarmos a descrição de U_k estaremos descrevendo o conjunto $\mathbb{A}(U_k)$. A descrição de todo $U_i \in \mathbb{A}(U_k)$ não se altera, e deveremos alterar a descrição apenas das unidades $U_j \in \mathbb{P}(U_k)$. A experiência mostra que a cardinalidade de $\mathbb{P}(U_k)$ é quase sempre bem menor que a cardinalidade de $\mathbb{A}(U_k)$, e a escolha que fizemos minimiza o número de alterações a serem feitas na descrição do sistema.

O grafo de geração, após a eliminação dos nós duplicados da árvore de geração, ainda reflete bastante bem a estrutura da árvore, embora sub-árvores não correspondam a partições do grafo, no caso mais geral. No entanto, por razões de ordem técnica ou econômica, poderemos aplicar novas transformações ao grafo, diminuindo sua semelhança com a árvore de geração. Isto pode obscurecer a estrutura interna do sistema, e os compromissos de ordem econômica devem ser consideradas judiciosamente. Uma transformação elementar que pode-se aplicar ao grafo é a "retirada" de um certo nó unidade, fazendo com que o componente que é seu resultado passe a ter caráter estritamente temporário. Para executarmos esta transformação basta concatenarmos a seqüência de processos dessa unidade à frente da seqüência de cada um de seus sucessores, e ligarmos cada um de seus antecessores como antecessores de cada um de seus sucessores. Usando a notação S_{U_i} para a ênupla que é a seqüência de processos de U_i, então para eliminarmos U_i transformamos todo U_j tal que $U_i \in \mathbb{A}(U_j)$ em U_j' através das seguintes regras, onde

o ponto (.) indica concatenação:

$$S'_{U_j} = S_{U_i} \cdot S_{U_j}$$

$$\mathbb{A}(U'_j) = \mathbb{A}(U_j) \cup \mathbb{A}(U_i)$$

Tomemos como exemplo o grafo da figura 2.6.1 e eliminemos as unidades F e G, obtendo o grafo da figura 2.6.2:

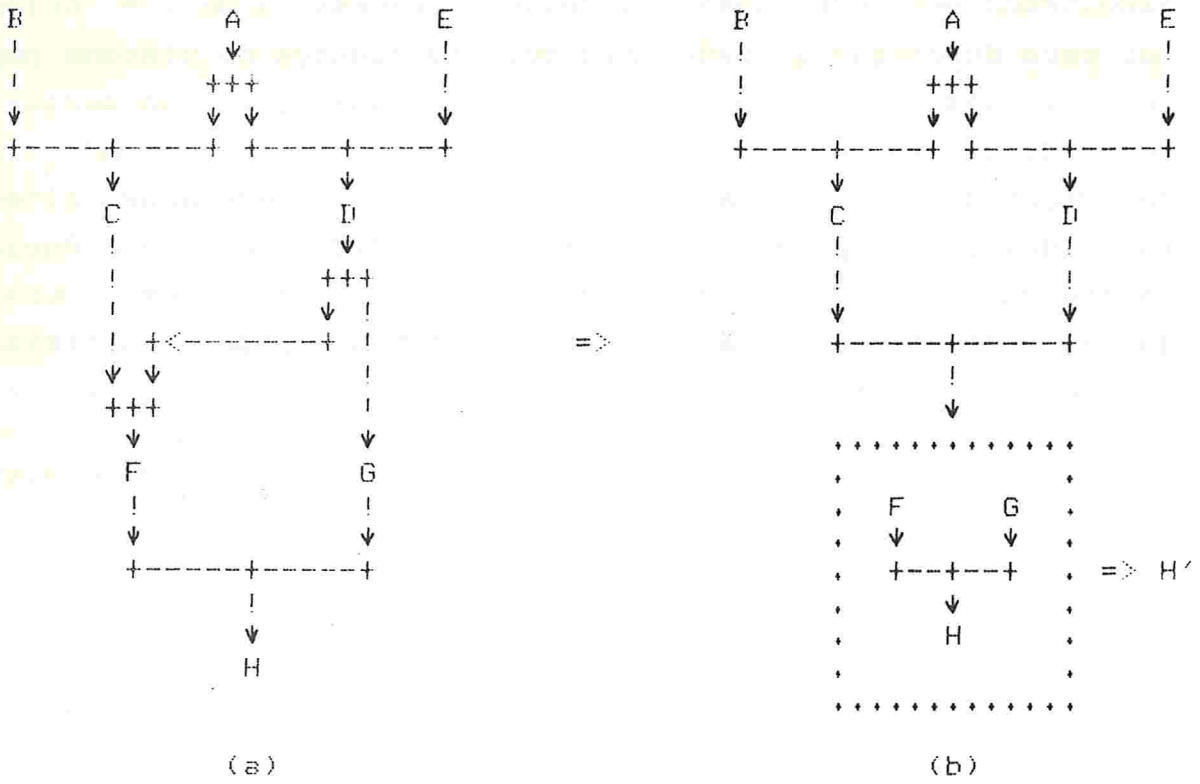

Fig 2.6.2 - Um exemplo da transformacao "retirada"

No caso da figura 2.6.2(a), uma alteração no módulo B gera uma trajetória que obriga a execução dos processos de C, F e H. Já na figura 2.6.2(b), a mesma alteração obriga a execução dos processos de C, F, G e H. Neste caso, no entanto, os arquivos- ou componentes - F e G são removidos do sistema após sua utilização, e eliminamos o custo de seu armazenamento através do custo adicional da execução dos processos de G, que pode ser bem menor. A documentação da existência dos arquivos F e G foi

removida da descrição do sistema, e essa existência somente pode ser inferida através da inspeção da descrição da unidade H; o grupo de manutenção pode ser levado a decisões errôneas pela ausência desse tipo de documentação, onerando muito o custo da manutenção.

Devemos ter em mente, no entanto, que considerações de ordem técnica podem obrigar a transformações no grafo, e em certos casos considerações de ordem econômica podem se tornar imperiosas, devido a custos proibitivos. Para resolver uma grande parte dos problemas que surgem aqui, vamos definir uma nova relação entre os nós "unidade" do grafo, completamente independente de tudo que foi exposto até agora, e destinada exclusivamente a eliminar a necessidade de certas transformações no grafo, não apenas evitando obscurecer a estrutura interna do sistema como também introduzindo uma documentação adicional que evidencia a existência desses problemas.

A esta nova relação daremos o nome de "propaga-se para"; possui apenas a propriedade de transitividade, e em geral não admite inversa. Nada impede que seja reflexiva, embora a reflexibilidade seja uma redundância desnecessária. Esta relação introduz novas arestas orientadas no grafo, e uma vez que pode ser simétrica permite a criação de ciclos no grafo. Seu significado pode ser descrito como: se A "propaga-se para" B, ao ativarmos a unidade A devemos ativar em seguida a unidade B, mesmo que sobre esta não incida nenhuma alteração. Isto significa que se uma alteração determina uma trajetória que contém A e não contém B, a unidade B será ativada determinando através de $\mathbb{P}(B)$ uma nova trajetória com início em B, e que não teve como origem uma alteração nas unidades desta nova trajetória. Efetivamente, uma alteração se propaga através das relações "usado em" e "propaga-se para" (donde este nome), e a determinação da trajetória se faz pela união de $\mathbb{P}(U_i)$ com $\{U_j \mid U_i$ "propaga-se para" $U_j\}$.

Retomemos o exemplo da transformação dado na figura 2.6.2. Suponhamos que seja economicamente inviável armazenar

os arquivos F e G, e não desejamos retirar as unidades F e G. Basta incluirmos as relações F "propaga-se para" G e H "propaga-se para" F, obtendo o grafo da figura 2.6.3.

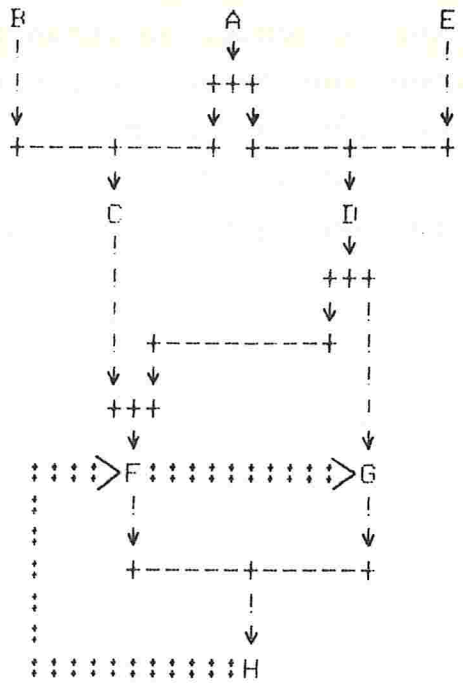

Fig 2.6.3 - A relacao "propaga-se para"

Desta forma, uma alteração em qualquer nó implicará na execução dos processos das unidades F e G, produzindo os componentes - ou arquivos - F e G. Notemos em particular que a relação H "propaga-se para" F cria um ciclo fictício com a relação F "usado em" H, criando uma trajetória aparente (H,F,G,H). É necessário reordenar a trajetória através da relação "necessita de", eliminando ocorrências duplicadas. Isto se faz de maneira mais simples eliminando-se as primeiras ocorrências das unidades duplicadas, e o resultado já está ordenado.

Podemos citar uma razão de ordem técnica que impeça o armazenamento dos arquivos F e G. Imaginemos que os arquivos

F e G sejam parte de um banco de dados, gerados pelos processos de F e G com conteúdo vazio, e a unidade H teste a transação "incluir a informação" a nos arquivos F e G, com a condição de "não há duplicatas". Fica claro que toda vez que ativarmos H será necessário executar antes os processos de F e G para gerar novamente o banco de dados em seu estado inicial, não contendo a informação a, caso contrário a transação em H terá um término anormal.

Muitos outros problemas análogos ocorrem na prática, e a relação "propaga-se para" revela-se extremamente útil.

CAPÍTULO 3

O SISTEMA MANTENEDOR

Neste capítulo apresentamos a implementação de um sistema mantenedor, baseado no modelo apresentado no capítulo 2. Discutimos os problemas relacionados à implementação, notadamente os referentes à portabilidade, identificando-os com as capacidades necessárias a um sistema mantenedor, e indicando as decisões tomadas em nossa implementação para o sistema B-6700. Apresentamos de maneira resumida o fluxo de informações e o esquema de funcionamento de nossa implementação; os detalhes da implementação e da linguagem de comandos do sistema mantenedor são de interesse apenas dos seus usuários, e serão apresentados como apêndice.

3.1 - Considerações sobre problemas da implementação

O sistema mantenedor é uma ferramenta de "software" destinada ao suporte da manutenção de sistemas, e como tal deve tornar disponível informações claras e completas que facilitem ao grupo de manutenção tomar decisões sobre as alterações necessárias, além de aliviá-lo de tarefas puramente mecânicas. Em 2.1 relacionamos as capacidades que julgamos necessárias a um sistema mantenedor. Vamos aqui repetí-las, dividindo-as em três classes de problemas correlatos, e discutir a implementação de cada classe com base no modelo de descrição de sistemas apresentado no capítulo 2.

A primeira classe inclui as capacidades:

- descrever sistemas, mesmo quando desenvolvidos por terceiros.
- armazenar essa descrição em um banco de dados, associada a documentação narrativa.
- adicionar, modificar ou eliminar informações contidas no

banco de dados.

- produzir documentação para uso dos analistas ou usuários do sistema.
- manter e documentar o histórico das alterações do sistema através de seus vários estados ou versões durante sua fase operacional.

A primeira dessas capacidades, descrever sistemas, vem do próprio modelo: um sistema é descrito através de seus elementos "módulo", "unidade" e "sistema" com seus atributos, relações e estruturas. Resta apenas definir uma linguagem de comandos adequada a essa descrição, o que faremos mais abaixo neste capítulo. Notemos que uma parte da descrição do sistema é coletada automaticamente pelo sistema mantenedor, a exemplo de datas de criação, tal como exposto no capítulo 2. A parte da descrição do sistema gerada manualmente evolui de forma natural juntamente com a fase de projeto, ou a fase de adaptação quando o sistema é adquirido de terceiros.

A capacidade de manter e documentar o histórico das alterações do sistema decorre naturalmente do conceito de módulo inalterável associado ao seu arquivo de "patches". No entanto, a descrição do sistema também evolui através de sua vida operacional. Novas unidades podem ser acrescentadas, o grafo de geração do sistema pode ser alterado por transformações tais como "retirada" de nós, pode haver modificações em rótulos ou volumes do sistema instalado, etc. Desta forma, a manutenção do histórico das alterações implica no armazenamento de todas as transações efetuadas sobre o banco de dados que conterá a descrição do sistema, associando a cada uma sua data.

As capacidades restantes desta primeira classe constituem um problema simples de implementação de banco de dados. No entanto, tanto a estrutura do banco de dados como seu gerenciamento necessários ao nosso propósito são extremamente simples, como veremos mais tarde; a utilização de um sistema de banco de dados comercial introduziria no sistema mantenedor uma complexidade desnecessária, além de eliminar a portabilida

de; os modelos fornecidos pela teoria não incluem de maneira simples o retrocesso do banco de dados no tempo, ou seja, recuperar seu conteúdo tal como era em alguma data anterior. Devido a isto, optamos em nosso sistema mantenedor por uma implementação simples, guiando-nos apenas pelos conceitos gerais da teoria de banco de dados. Não utilizamos, por exemplo, implementação de arquivos com acesso direto a seus elementos, nem a manutenção simultânea de uma relação e sua inversa. Com isto sacrificamos um pouco a eficiência do sistema mantenedor, pois muitas vezes será necessário percorrer seqüencialmente um arquivo para determinar o conjunto de elementos de uma determinada relação; uma referência de um elemento de um arquivo para um elemento de outro arquivo será sempre descrito por um nome simbólico, e devido a isso várias tabelas deverão ser montadas a cada execução do sistema mantenedor. No entanto este sacrifício não é crítico, e é compensado pela simplicidade e portabilidade da implementação. Ao mesmo tempo, todos os arquivos do "banco de dados" da descrição do sistema são independentes, e nos valemos do mecanismo simples que é o conceito de "patch" para manter o histórico das transações efetuadas sobre ele: a cada arquivo temos associado um arquivo de "patches", e exatamente como descrito para o "módulo" em 2.3, a utilização desse arquivo exige antes um processo de sobreposição do arquivo de "patches", mantendo a descrição original do sistema inalterada. Com isto, temos disponíveis o histórico das alterações da descrição do sistema, e a capacidade de recuperar essa descrição tal como era em qualquer época passada.

A descrição de um "patch" merece algumas considerações. O arquivo de "patches" contém uma seqüência com caráter temporal, ordenada pela data de introdução do "patch" no sistema mantenedor, e a sobreposição ao arquivo correspondente deve obedecer a essa ordem. É possível, assim, que estejamos alterando uma parte do arquivo já alterada por um "patch" anterior. A maioria das instalações possui mantenedores de arquivos fonte (ou arquivos em geral), e estes devem ser invocados pelo sistema mantenedor, sempre que possível. No entanto pode

acontecer que uma seqüência de "patches" forme comandos conflitantes para o mantenedor de arquivos, o que exigiria mais de uma execução do mesmo. Quando isto acontecer, será aconselhável incluir no sistema mantenedor a facilidade de sobrepor o arquivo de "patches" em um único passo. Em nossa implementação preferimos adotar a norma de que todos os arquivos possuem em cada registro um número de seqüência que o identifica; as alterações nos arquivos se fazem através desse número de seqüência, e um "patch" se reduz a um bloco de registros numerados. A implementação da sobreposição dos "patches" se torna assim bastante simples.

Naturalmente, o processo de sobreposição dos arquivos de "patches" gera um custo adicional, que neste caso pode ser eliminado: após a sobreposição dos "patches" temos, analogamente ao "módulo atualizado", um "banco de dados atualizado". Este, no entanto, é permanente, sendo armazenado junto com o sistema, e o processo de sobreposição será efetuado apenas em raras ocasiões. Isto é possível devido ao fato de que o banco de dados é, na prática, pequeno, e o número de transações sobre ele muito reduzido. Devemos esclarecer aqui que as partes da descrição do sistema coletadas automaticamente pelo sistema mantenedor (a exemplo de data de criação de arquivos), são alteradas por ele sem que isto constitua uma transação, e são irrelevantes para o histórico das alterações na descrição do sistema.

A segunda classe inclui as capacidades:

- gerar componentes de maneira independente.
- gerar automaticamente todas as partes dependentes afetadas pela modificação de um módulo ou componente específico.
- integrar o sistema ou partes do sistema.
- recuperar o sistema em qualquer versão anterior.

Estas capacidades se reduzem em ativar unidades,quer através de alterações em algum nó do grafo, percorrendo suas

trajetórias, quer através de pedidos de ativação de unidades selecionadas, mesmo sem nenhuma alteração. Como vimos em 2.4, ativar uma unidade significa coletar as especificações dos processos nela descritos e efetuar os pedidos ao sistema operacional para iniciar e executar esses processos. Ao ativarmos uma unidade selecionada, ou unidades determinadas por trajetórias, estaremos formando uma seqüência de processos a serem executados em um computador, através de um sistema operacional. Como vimos em 1.1, esta seqüência de processos constitui um "job", e é descrita para o sistema operacional através da linguagem de comandos do computador. O sistema mantenedor, então, deve ser capaz de produzir um "job" e entregá-lo ao sistema operacional para ser executado. A produção deste "job" é exatamente o trabalho manual normalmente feito pelo grupo de manutenção. No entanto, limitar o sistema mantenedor a isto traz uma série de desvantagens. Em primeiro lugar, o sistema mantenedor não mantém o controle do fluxo de execução dos processos. Como vimos em 2.4, alguns processos dependem do término normal de outros, e é necessário poder interromper a execução de processos de uma unidade ou de uma trajetória quando houver um término anormal. Nem todas as linguagens de comando possuem a capacidade de iniciar um processo de maneira condicional, utilizando o modo de término de outros; de qualquer maneira, nos casos em que um "job" é interrompido por uma parada da máquina, e reiniciado a partir do processo interrompido, todas as informações internas ao "job" sobre o modo de término dos processos anteriores são perdidas. Em segundo lugar, o sistema mantenedor terá dificuldades em coletar as informações necessárias à atualização da descrição do sistema, principalmente nos casos em que a execução do "job" é terminada prematuramente, não completando a seqüência de processos. Para eliminar essas desvantagens adotamos o método de entregar ao sistema operacional a especificação de um processo apenas por vez, coletar todas as informações referentes à execução desse processo e imediatamente atualizar a descrição do sistema antes de passar ao próximo processo. Desta forma mantemos controle completo sobre o fluxo de execução de processos, podemos determinar de antemão se

existem todos os pré-requisitos necessários à execução de um processo antes de entregá-lo ao sistema operacional, e conseguimos a recuperação completa em casos de reinício após uma parada da máquina.

Devemos agora tecer considerações sobre a escolha do modo de descrevermos a especificação de processos. Como vimos em 2.3 e 2.4 essa escolha depende de um compromisso entre a simplicidade do sistema mantenedor e sua portabilidade, havendo duas estratégias possíveis: adotarmos as construções usuais da linguagem de comandos do computador, ou definir uma linguagem independente a ser traduzida para diversas máquinas. A segunda estratégia é nitidamente inferior, pois estaríamos sacrificando a portabilidade do sistema mantenedor em favor da portabilidade da descrição do sistema. A portabilidade da descrição do sistema, principalmente no que se refere à seqüência de processos necessária para gerar um componente de uma unidade, está ainda mais remota que a portabilidade de uma linguagem de programação. Em geral, ao passarmos a outra máquina, seremos obrigados a alterar até mesmo a estrutura do grafo de geração do sistema. Como conseqüência, adotamos para especificar um processo a própria linguagem de comandos do computador, obtendo uma série de vantagens. Em primeiro lugar, o sistema mantenedor se destina ao suporte da manutenção de todos os sistemas em um computador, que será utilizado durante um período de tempo bastante longo. O grupo de analistas está acostumado a pensar e escrever na linguagem de comandos do computador, e ao montar a descrição de uma unidade utilizará essa linguagem de maneira natural para especificar seus processos, e esta especificação constituirá um trecho de um "job" da maneira habitual. Em segundo lugar, o sistema mantenedor não necessitará inspecionar a especificação do processo, entregando-a praticamente sem modificações ao sistema operacional, o que favorece sua portabilidade. Como adendo, isto implica em que escolheremos para a descrição de "título", como definido em 2.3, a sintaxe da linguagem de comandos do computador.

Notemos aqui que a implementação desta segunda classe de capacidades possui pouca portabilidade, uma vez que depende do intercâmbio de informações entre um sistema operacional e um programa. Fica claro que ao projetarmos o sistema mantenedor não desejamos modificar o sistema operacional existente, e devemos nos contentar com as capacidades que ele contenha. Em geral teremos a capacidade de entregar uma especificação de um processo ao sistema operacional para ser executado, mas não teremos controle sobre o instante de seu início. Em apenas alguns casos seremos informados do instante de seu término,e quase nunca receberemos a informação do modo de seu término. A implementação mais viável será adicionarmos mais um processo que coleta o modo do término e cria um arquivo com um rótulo determinado cujo conteúdo é essa informação. O sistema mantenedor, então, entrega ao sistema operacional a especificação conjunta de dois processos e aguarda a existência daquele arquivo, obtendo daí o modo de término. Há que considerar o caso em que a especificação do processo é rejeitada pelo sistema operacional devido a uma construção inválida; neste caso o arquivo nunca chegará a existir, e o sistema mantenedor ficaria aguardando indefinidamente. É necessário, então, que o sistema operacional forneça a informação de "processo rejeitado". Caso esta capacidade não exista, uma implementação alternativa seria interceptar o "log" do sistema para obter as informações desejadas; esta implementação seria de custo bastante elevado, e na maioria das máquinas de grande porte não é necessária.

A terceira classe inclui as capacidades:

- instalar automaticamente o sistema completo ou apenas partes do sistema que tenham sido modificadas.
- efetuar a verificação automática do sistema instalado em confronto com a versão mais recente, eliminando discrepâncias.

Como vimos em 2.5, a instalação consiste em mover arquivos para outros meios ou volumes, alterando seus rótulos e assoçiando a esses arquivos certas classes de segurança; esta movimentação de arquivos somente pode ser iniciada quando o

sistema resulta em estado normal, caso contrário teríamos em geral um sistema incoerente, mistura de duas versões.Isto quer dizer que deveremos cuidar para que o sistema mantenedor, ao gerar arquivos, não destrua algum arquivo já instalado.

Como dito em 2.5, o atributo "nome" do sistema o distingue de todos os outros sistemas, e a regra de derivação do rótulo do componente a partir do nome do sistema e do nome da unidade garante a independência dos arquivos durante a integração do sistema. A maioria das linguagens de comando de computadores admite a construção de rótulos com vários níveis hierárquicos, e em nossa implementação adotamos dois níveis para o rótulo do componente, sendo o primeiro o nome do sistema e o segundo o nome da unidade.

O processo de movimentação de arquivos faz parte de todos os sistemas operacionais, e sua especificação pode ser gerada automaticamente pelo sistema mantenedor, uma vez que o rótulo do componente é definido internamente e o destino é dado por um título na descrição do nó "sistema". Esta também é uma capacidade com pouca portabilidade, uma vez que depende diretamente da linguagem de comandos do computador; no entanto é de implementação bastante simples.

A verificação da instalação consiste apenas em confrontar as datas de criação do componente com o arquivo descrito pelo título de destino. Quase sem exceção, os sistemas operacionais existentes possuem a facilidade de informar a data de criação de um arquivo sendo conhecido seu rótulo.Existe uma exceção quando usamos arquivos com estrutura "particionada",ou seja, o arquivo em questão é um "membro" de um arquivo particionado. Como regra geral, o sistema operacional não mantém a data de criação de membros, mas apenas do arquivo particionado. Neste caso, se desejarmos a capacidade de movimentar membros independentemente, devemos confiar ao sistema mantenedor a tarefa de associar uma data a cada membro. Isto poderia ser feito, por exemplo, adicionando ao arquivo particionado um membro que descreva a data dos outros membros. No entanto isto não

oferece segurança alguma quanto a alterações indevidas,sem utilizar o sistema mantenedor. Como regra geral, o uso de arquivos particionados não fornece nenhuma segurança na verificação da instalação, e devemos adotar a política de recompor todo o arquivo particionado sempre que houver dúvidas quanto ao estado da instalação.

3.2 - Um exemplo de descrição de sistemas

Para darmos uma idéia melhor do que se espera de um sistema mantenedor, como identificamos seus elementos com o modelo do capítulo 2 e como os descrevemos, vamos mostrar um exemplo simples.

Suponhamos que o sistema que estamos mantendo seja um conjunto de programas utilitários, e imaginemos que um desses utilitários seja um compilador que gere código para uma máquina "A" diferente da que estamos utilizando. O programa fonte do compilador poderia ter sido desenvolvido como um arquivo único, e teríamos então um ramo do grafo do sistema como na figura 3.2.1.

Fig 3.2.1 - Exemplo 1 de descricao de sistemas

Temos um módulo, o arquivo fonte, descrito por um título apenas. Temos uma unidade, que se constitui de um único processo, uma compilação, e um arquivo objeto, o compilador, resultado desse processo. Este arquivo objeto é um componente de nível 1, e portanto consta na descrição do sistema, possuindo um rótulo de destino. A descrição da unidade se faz descrevendo suas relações, seus processos e os atributos do componente. No caso, temos uma relação: a unidade "necessita de" o arquivo fonte. Temos um processo, que especificamos de acordo com a sintaxe da linguagem de comandos do computador que estamos utilizando. Temos um componente que é seu resultado, e descrevemos seus atributos, como por exemplo o fato de ser um arquivo executável, ou objeto.

Quando desejarmos efetuar uma alteração no compilador, montamos um "patch" para o módulo e o entregamos ao sistema mantenedor. Este agrega o "patch" ao arquivo de "patches", e efetua a sobreposição deste ao módulo, criando um arquivo fonte atualizado. Lembramos que neste ponto todas as alterações passadas são refeitas. Em seguida, através da relação "necessita de" determina que o módulo é "usado em" essa unidade, e a "ativa". Coleta a especificação do processo de compilação e a entrega ao sistema operacional; fica então aguardando sua execução. Após o término da compilação, remove o arquivo fonte atualizado, coleta o modo de seu término e a data de criação do arquivo objeto, atualizando imediatamente a descrição da unidade. Caso a compilação tenha terminado normalmente (sem "erros de sintaxe"), o componente é descrito como em estado normal, e por ser componente de nível 1 propaga esse estado para o sistema, iniciando o processo de instalação, que consiste em copiar o arquivo objeto com um rótulo escolhido para o volume onde ficará disponível aos usuários, alterando o atributo "permissão de acesso" do arquivo instalado para, por exemplo, "execute apenas". O sistema mantenedor, então, emite um ou mais relatórios que documentem o sistema, e termina. Caso a compilação tenha terminado anormalmente, o componente é descrito como em estado anormal, e propaga esse atributo para o sis-

tema, inibindo o processo de instalação.

O programa fonte do compilador poderia ter sido desenvolvido em três arquivos, dada a estrutura natural de um compilador: o analisador léxico, o analisador sintático e o gerador de código, ou rotinas semânticas. Teríamos então um grafo como na figura 3.2.2.

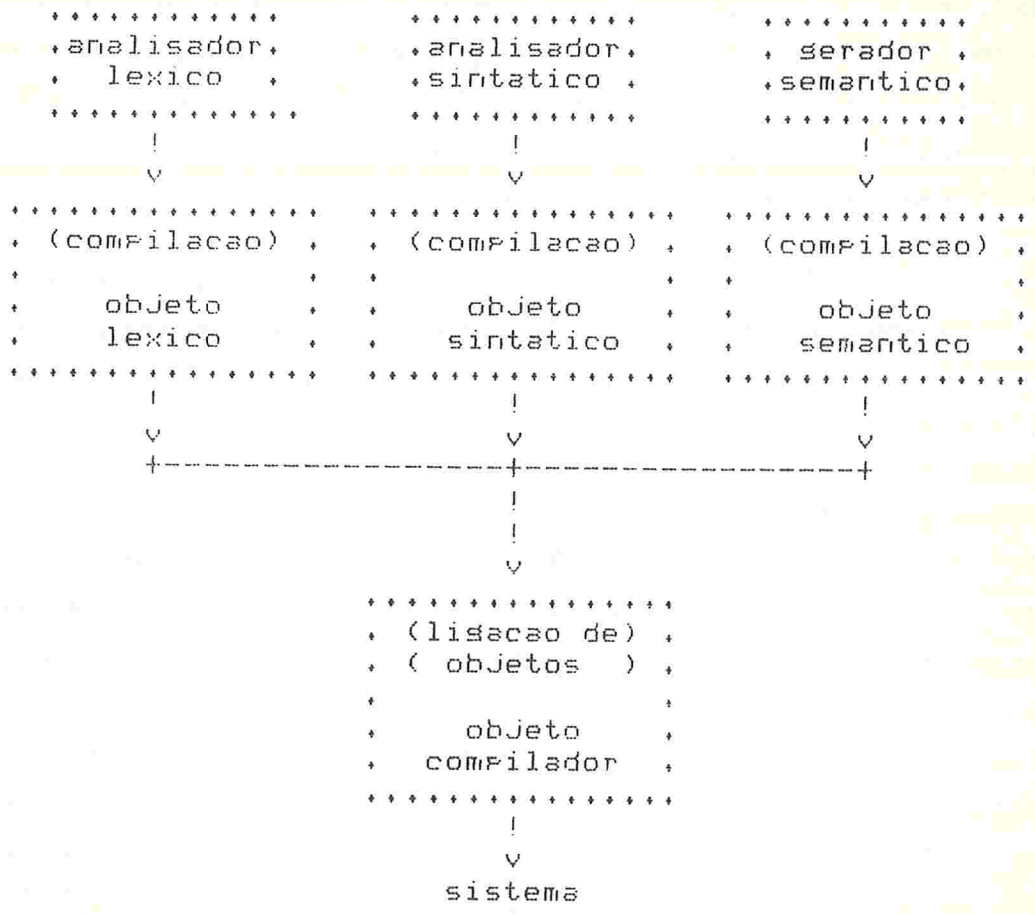

Fig 3.2.2 - Exemplo 2 de descricao de sistemas

Temos agora quatro unidades e três módulos. Se alterarmos o analisador sintático, para corrigir um erro, por exemplo, o sistema mantenedor ativará as unidades do analisador sintático e do compilador, causando a execução de uma compilação, que produz um novo objeto do analisador sintático,e a exe

cução de uma "ligação de objetos", produzindo um novo objeto do compilador, que será instalado. Notemos que os objetos do analisador léxico e gerador de código existem em caráter permanente, não necessitando compilação. Em relação ao exemplo da figura 3.2.1, reduzimos o custo da compilação, e acrescentamos o custo do armazenamento de mais três arquivos objeto, o que em geral resulta em um custo global menor. Devemos mencionar aqui que a unidade do compilador possui três relações "necessita de", e ao ativá-la o sistema mantenedor deve verificar o estado dos componentes dessas outras três unidades: caso um deles, mesmo estando fora da trajetória, esteja em estado anormal, o processo de ligação não será executado e o componente "compilador" ficará em estado anormal.

Poderíamos utilizar a mesma linguagem para gerar código para uma outra máquina "B", bastando acrescentar um novo módulo de rotinas semânticas. Teríamos então um grafo como na figura 3.2.3.

Uma alteração no analisador sintático, agora, se propaga por três unidades, causando uma compilação, duas ligações de objetos e dois processos de instalação. Notemos apenas aqui que ao acrescentarmos estas duas novas unidades não alteramos em nada a descrição das unidades do analisador léxico e do analisador sintático, embora o conjunto $\mathbb{P}$ destas unidades tenha sido alterado. Devido ao fato de havermos escolhido a relação "necessita de" na descrição de unidades, o sistema mantenedor gera os conjuntos $\mathbb{P}$ pela inspeção de todos os conjuntos $\mathbb{A}$.

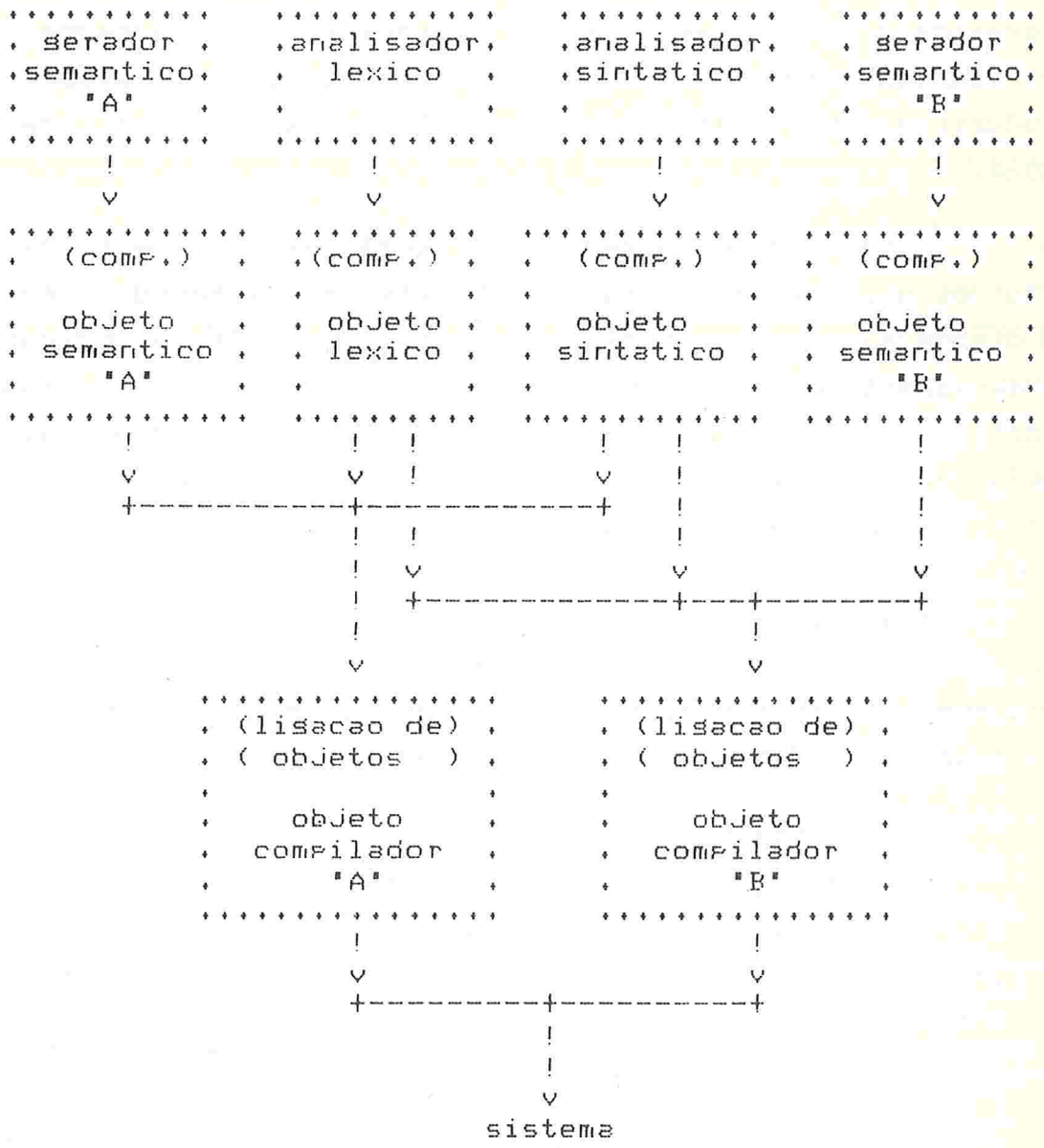

Fig 3.2.3 - Exemplo 3 de descricao de sistemas

3.3 - A implementação do sistema mantenedor

Vamos aqui apresentar a implementação existente para o sistema B-6700, sem descer a detalhes. Sendo uma implementação para uma máquina específica, naturalmente utilizaremos os conceitos usuais de seu sistema operacional, e suporemos que são bem conhecidos. Como toda implementação, é necessário introduzir algumas restrições ao modelo, e estas serão ressaltadas.

Como primeira restrição, será exigido que todos os arquivos utilizados e gerados pelo sistema mantenedor existam em discos magnéticos; qualquer arquivo cujo título indique um armazenamento em fita implicará em um processo de transferência de ou para um disco. Existe no sistema operacional do B-6700 um processo padrão de carga de arquivos de fita para disco, ou descarga de arquivos de disco para fita, denominado "library-maintenance". Os arquivos descarregados para fita não são utilizáveis a não ser após serem carregados para disco, e este processo de transferência preserva a data de criação de arquivos. O "library-maintenance" possui seu correspondente em praticamente todas as outras máquinas.

Como segunda restrição, imporemos que cada unidade utilize não mais que um módulo que possua "patches", de modo que a cada unidade corresponderá um único processo de sobreposição do arquivo de "patches". Como veremos, esta restrição não traz limitações ao uso do sistema mantenedor, e permite a introdução de convenções que facilitam sua utilização.

Como vimos em 2.3, um módulo é descrito unicamente através de seu título. No entanto, um módulo possui, em geral, várias relações "usado em". Uma vez que adotamos a descrição apenas da relação "necessita de", cada unidade que utilizar o módulo conterá uma referência a esse módulo. Colocamos, então, o título do módulo como referência da relação "necessita de" dentro de unidades, e da descrição das unidades extraímos a descrição dos módulos.

Como vimos em 2.5, os atributos do nó "sistema" incluem a descrição dos títulos de destino de seus componentes de nível 1. Todos esses atributos podem ser agregados à descrição das unidades de nível 1, e da descrição das unidades extraímos a descrição do sistema.

Necessitamos, assim, apenas de um arquivo contendo a descrição das unidades. A este arquivo denominaremos "arquivo units". Adotando a idéia de que toda informação passível de alteração não deve constar em vários lugares, criamos um arquivo inversor para conter essas informações, ao qual denominaremos "arquivo intnames". Qualquer referência a um elemento do arquivo intnames será feita através de um nome simbólico, ao qual denominaremos "nome interno" do sistema, e utilizaremos este termo indiferentemente tanto para a referência ao elemento como para o conteúdo do elemento no arquivo intnames.

Para a estrutura desses dois arquivos escolhemos a estrutura habitual de arquivos fonte: cada registro possui um campo útil e um campo contendo um número de seqüência associado. Desta forma alterações nesses arquivos são feitas através de "patches" de maneira idêntica à de "patches" em módulos como descrito em 3.1, preservando o histórico das alterações no sistema.

Um nome interno pode ser uma simples substituição de cadeia de caracteres, um arquivo completo ou uma referência a um arquivo. Desta forma podemos nos valer desta facilidade para colocarmos no arquivo intnames módulos pequenos que sejam usados em muitas unidades, evitando assim uma multiplicidade grande de arquivos. Um exemplo típico seria a descrição de uma estrutura, a ser incluída em vários programas COBOL. Com isto, eliminamos também a necessidade de informar ao sistema mantenedor a relação "necessita de" módulos que são nomes internos. A simples referência ao nome interno, mesmo dentro de um arquivo fonte, é captada pelo sistema mantenedor e adicionada como uma relação na descrição da unidade.

A descrição de uma unidade se faz através de comandos em formato fixo, não tendo sido escolhido o formato livre por julgarmos desnecessário. Necessitamos de comandos de três tipos apenas: que descrevem relações, que descrevem processos e que descrevem a instalação.

Notemos que a escolha dos comandos da linguagem do sistema mantenedor é feita de forma a não confundí-los com a linguagem de comandos do computador, e possuem caracteres especiais que os iniciam e que não constam de nenhuma construção da sintaxe dessa linguagem. Por exemplo, conforme 2.3, escolhemos o ponto (.) como terminador de um título devido ao fato de que no sistema B-6700 que utilizamos, tanto <título> como <meio> não podem conter um ponto. Esta escolha deverá ser adaptada para cada máquina.

Os comandos que descrevem relações identificam se a relação é com um módulo ou uma unidade, e são três: .SOURCE, .NEEDS e .PROPAGATE . O comando .PROPAGATE descreve as relações "propaga-se para" unidades. O comando .NEEDS descreve as relações "necessita de" unidades. O conteúdo de ambos, então, é um conjunto de nomes característicos de unidades. O comando .SOURCE descreve as relações "necessita de" módulos, e colocamos em seu conteúdo os títulos dos módulos, aproveitando para descrever simultaneamente a relação e o módulo.

Devido à segunda restrição que impusemos, uma unidade possui um único módulo sobre o qual incidem "patches". Convencionamos que esse é o módulo que foi relacionado em primeiro lugar no comando .SOURCE . Associamos, desta forma, o rótulo do arquivo de "patches" do módulo com o nome da unidade; um "patch" mencionará o nome da unidade, ativando-a diretamente. Na prática, os módulos que são utilizados em várias unidades pertencem ao arquivo intnames, e seus "patches" fazem parte do arquivo de "patches" de nomes internos. Muito raramente teremos um arquivo usado em mais de uma unidade. Neste caso, basta criar uma unidade a mais cujo resultado é uma cópia do "módulo atualizado", com as relações "necessita de" e "propa-

ga-se para" apropriadas. Esta limitação, portanto, não traz sérias restrições ao uso do sistema mantenedor, e traz como vantagem a simplicidade do rótulo do arquivo de "patches", derivado do nome da unidade. Notemos aqui que o rótulo de um módulo, principalmente quando adquirido de terceiros, pode não se amoldar a nenhuma convenção.

Mencionamos em 2.5 que o rótulo do componente de uma unidade deve ser derivado do nome do sistema e do nome da unidade. Como um rótulo é descrito em níveis hierárquicos de diretório, adotamos como primeiro nível do rótulo de um componente, arquivo de "patches", arquivo units ou arquivo intnames, o nome do sistema, e desta forma todos os arquivos permanentes do sistema constituem um único diretório para o sistema operacional. Para um componente, adotamos como segundo nível o nome da unidade.

Os comandos que especificam processos são: .WFL , .RUN , .COMPILE e .BIND . O primeiro é usado para descrever processos cujo modo de término não é informado pelo sistema operacional, os outros três para processos cujo modo de término pode ser coletado. O conteúdo de cada um desses comandos é uma especificação de um processo, feita na linguagem de comandos do computador (no caso do B-6700, a linguagem WFL). As especificações são coletadas pelo sistema mantenedor e entregues ao sistema operacional praticamente sem modificações, além da substituição de nomes internos. Notemos aqui que a substituição de um nome interno pode exigir do sistema mantenedor a criação de um arquivo auxiliar com caráter temporário, principalmente no caso em que o nome interno constitui um módulo.

O fato de que o sistema mantenedor não necessita inspecionar a sintaxe de uma especificação de um processo favorece sua portabilidade. No entanto, os processos de compilação e ligação de objetos são os mais comuns, e constituem a maioria dos processos necessários à geração de sistemas. Devido a isso os comandos .COMPILE e .BIND são fornecidos ao sistema mante-

nedor com uma especificação parcial, sendo completada automaticamente por ele. Isto diminui um pouco a portabilidade; no entanto simplifica em muito a descrição do sistema. Além disso as convenções necessárias são muito simples, e a portabilidade destes dois comandos não apresenta maiores dificuldades.

Os comandos que descrevem a instalação existem apenas nas unidades de nível 1, e sua própria existência caracteriza esse fato. O principal deles é o comando .DESTINATIONS , sendo o primeiro registro desse comando a classe de segurança desejada, e cada registro subseqüente um título. A cada título corresponderá um processo de movimentação de arquivos, especificado e executado automaticamente pelo sistema mantenedor.

A execução do sistema mantenedor para uma ou mais alterações pode ser descrita resumidamente pelos seguintes passos:

- o sistema mantenedor lê um conjunto de "patches". Cada um menciona o nome de uma unidade, podendo haver mais de um para a mesma unidade.
- os conjuntos $\mathbb{P}$ de cada unidade são determinados através da inspeção dos conjuntos $\mathbb{A}$ de todas as unidades.
- cada unidade sobre a qual incidiu um "patch" propaga uma trajetória através de seu conjunto $\mathbb{P}$ mais suas relações "propaga-se para". As trajetórias serão, em geral, parcialmente disjuntas.
- as trajetórias são concatenadas formando uma seqüência; esta é ordenada pela relação "necessita de" e são retiradas as ocorrências duplicadas de unidades. Temos assim a seqüência de ativação de unidades.
- o sistema mantenedor carrega para disco todos os arquivos necessários, mencionados nos comandos .SOURCE das unidades a serem ativadas, e que estejam em fita.
- o sistema mantenedor passa a ativar as unidades na seqüência de ativação dada acima, coletando os resultados dos processos e atualizando a descrição do sistema. Ao mesmo tempo cria um mecanismo de recuperação, permitindo retomar

a seqüência no ponto em que foi interrompida, em caso de parada da máquina.

- antes de ativar uma unidade, verifica se todas as condições necessárias estão satisfeitas, em particular se existem todos os arquivos mencionados em .SOURCE .
- após ativar a última unidade, caso o sistema tenha resultado normal, efetua a instalação de todos os arquivos alterados.

A linguagem de comandos do sistema mantenedor associada ao arquivo inversor com a capacidade de manter módulos também fornece um bom suporte ao projeto de sistemas. A estrutura do sistema surge naturalmente com o grafo de geração, e a modularização do sistema é facilitada pela definição dos nomes internos, gerando módulos a serem utilizados em vários programas fonte, homogeneizando e diminuindo o trabalho de programação. A utilização das facilidades do sistema mantenedor durante o projeto é feita através de seu modo interativo, que pressupõe um ambiente de processamento remoto e interativo, suportado por uma parte específica do sistema operacional. Este suporte inclui, em quase todos os sistemas operacionais, capacidades de edição de arquivos, execução de processos, manipulação de atributos de arquivos e utilização de arquivos de comando indireto.

O modo interativo não constitui uma restrição ao modelo, nem uma extensão a ele, mas apenas do uso exclusivo das capacidades de gerenciamento do pequeno banco de dados formado pelos arquivos units e intnames, substituindo nomes internos e criando arquivos auxiliares. Neste modo, não existem "patches", e o sistema mantenedor limita-se a coletar as especificações dos processos de uma única unidade, transformando-os para a forma de um arquivo de comando indireto do suporte de processamento remoto interativo. Notemos que esta transformação das especificações dos processos exige a manipulação da sintaxe de WFL, e portanto possui pouca portabilidade, talvez até com problemas sérios de adaptação a outras máquinas. No

entanto constitui uma capacidade necessária ao sistema mantenedor, mesmo para a manutenção. Como dito em 2.1, uma alteração no sistema pode exigir a repetição das fases iniciais da construção do sistema. O grupo de manutenção pode, então, obter todos os módulos atualizados através do sistema mantenedor, e utilizá-lo juntamente com o suporte interativo para determinar a forma correta de efetuar uma alteração. Em seguida, as alterações resultantes são colocadas em forma de "patches" e entregues ao sistema mantenedor para que este as integre convenientemente ao sistema.

CAPÍTULO 4

CONCLUSÕES

A implementação do sistema mantenedor tem-se revelado uma ferramenta poderosa na manutenção de sistemas em geral, reduzindo drasticamente os custos da manutenção, não só pela redução do tempo a ela dedicado pelo grupo de manutenção, mas principalmente eliminando a possibilidade de ocorrência de situações catastróficas que ocorrem em sistemas mal integrados devido a erros ou esquecimentos nas rotinas manuais usuais na manutenção.

O sistema mantenedor não possui uma segurança absoluta contra alterações indevidas, ou seja, alterações que não deixam traços, e acreditamos que esta segurança seja impossível de obter. No entanto, dada a facilidade de sua utilização e o alívio de muitas das tarefas que recaíam sobre o grupo de manutenção, existe um incentivo natural ao seu uso, progressivamente eliminando a ocorrência de alterações indevidas. Não obstante, a documentação através de textos narrativos continua sendo pouco utilizada, e cabe à auditoria da manutenção preocupar-se em exigí-la.

Como observação de ordem prática, queremos mencionar que muitas vezes, ao descrevermos um sistema, somos tentados a criar uma descrição incompleta, por exemplo deixando de especificar algumas relações "necessita de"; com isto introduzimos deliberadamente algumas rotinas manuais na manutenção, tentando reduzir seus custos imediatos. A experiência já comprovou que nesses casos fatalmente chegaremos a uma situação catastrófica: a rotina manual será esquecida.

A implementação existente, como está, já constitui um suporte eficaz ao projeto de sistemas, possuindo como ponto fraco a pouca documentação gerada. Uma extensão ao modelo deverá ser feita em futuro próximo, na tentativa de formalizar uma

descrição da documentação, complementando o suporte ao projeto.

Como sugestão a pesquisas futuras, indicamos que o modelo poderá ser extendido para suporte da operação de sistemas. A operação de um sistema possui atributos e relações tais como a freqüência de execução de componentes, sua ordem de execução dentro de um "job", o fluxo de informações através de vários arquivos, o destino das informações, seu local de armazenamento, regras de recuperação por erros ou paradas de máquina, etc; com estes elementos poderíamos definir um "grafo de operação" do sistema, permitindo ao sistema mantenedor, por exemplo, gerenciar a execução de uma seqüência de "jobs" de alguns ou todos os sistemas. A ativação da seqüência de "jobs" poderia ser feita automaticamente em datas pré-determinadas,ou através da emissão de programações mensais, semanais ou diárias para o grupo de operação.

BIBLIOGRAFIA

/A-R 81/ J. Arthur e J. Ramanathan,
"Design of Analyzers for Selective Program Analysis",
IEEE Transactions on Software Engineering,
Vol. SE-7, No. 1, January 1981, pp 39-52

/ALF 77/ M. W. Alford,
"A Requirements Engineering Methodology for Real-Time Processing Requirements",
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 1, January 1977, pp 60-69

/B-J 76/ R. A. Baker e J. C. Jefferies,
"A Computerized Librarian",
DATAMATION, December 1976, pp 61-64

/B-P 81/ M. Broy e P. Pepper,
"Program Development as a Formal Activity",
IEEE Transactions on Software Engineering,
Vol. SE-7, No. 1, January 1981, pp 14-23

/BAL 81/ R. Balzer,
"Transformational Implementation: An Example",
IEEE Transactions on Software Engineering,
Vol. SE-7, No. 1, January 1981, pp 3-14

/BBD 77/ T. E. Bell, D. C. Bixler e M. E. Dyer,
"An Extendable Approach to Computer-Aided Software Requirements Engineering",
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 1, January 1977, pp 49-60

/C-S 78/ W. C. Cave e A. B. Salisbury,
"Controlling the Software Life Cycle - The Project Management Task",
IEEE Transactions on Software Engineering,
Vol. SE-4, No. 4, July 1978, pp 326-334

/COO 78/ J. D. Cooper,
"Corporate Level Software Management",
IEEE Transactions on Software Engineering,
Vol. SE-4, No. 4, July 1978, pp 319-326

/D-V 77/ C. G. Davis e C. R. Vick,
"The Software Development System",
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 1, January 1977, pp 69-84

/DEA 81/ E. Deak,
"A Transformational Derivation of a Parsing Algorithm in a High-Level Language",
IEEE Transactions on Software Engineering,
Vol. SE-7, No. 1, January 1981, pp 23-32

/HES 80/ W. Hesse,
"A Project Model for Software Engineering",
SOFTLAB GmbH, Muenchen, Germany,
Course Schedule, 1980

/I-B 77/ C. A. Irvine e J. W. Brackett,
"Automated Software Engineering Through Structured Data Managements",
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 1, January 1977, pp 34-40

/L-Z 75/ B. H. Liskov e S. N. Zilles,
"Specification Techniques for Data Abstractions",
IEEE Transactions on Software Engineering,
Vol. SE-1, No. 1, March 1975, pp 7-19

/LIU 76/ C. C. Liu,
"A Look at Softaware Maintenance",
DATAMATION, November 1976, pp 51-56

/M-O 76/ L. K. Miller e D. Ostrom,
"Source Prosram Manasement",
DATAMATION, December 1976, pp 67-70

/M-W 78/ Z. Manna e R. Waldinser,
"The Losic of Computer Prosrammins",
IEEE Transactions on Software Engineering,
Vol. SE-4, No. 3, May 1978, pp 199-229

/M-W 79/ Z. Manna e R. Waldinser,
"Synthesis: Dreams => Prosrams",
IEEE Transactions on Software Engineering,
Vol. SE-5, No. 4, July 1979, pp 294-328

/R-N 78/ B. C. de Roze e T. H. Nyman,
"The Sofware Life Cycle - A Management and Technolosical Challense in the Department of Defense",
IEEE Transactions on Software Engineering,
Vol. SE-4, No. 4, July 1978, pp 309-318

/T-H 77/ D. Teichroew e E. A. Hershey, III,
"PSL/PSA: A Computer-Aided Technique
for Structured Analysis of Information
Processing Systems",
IEEE Transactions on Software Engineering,
Vol. SE-3, No. 1, January 1977, pp 41-48

/WIL 81/ D. S. Wile,
"Type Transformations",
IEEE Transactions on Software Engineering,
Vol. SE-7, No. 1, January 1981, pp 32-39

APÊNDICE

MANUAL DO USUÁRIO

O programa MACYS implementado no B-6700 é um sistema mantenedor de sistemas. Destina-se a executar e documentar automaticamente todos os passos necessários à geração e instalação do sistema. Para isto, o sistema é descrito através de uma linguagem de comandos própria, e qualquer alteração efetuada no sistema é incorporada através do MACYS à descrição do sistema, o que permite manter a documentação do histórico da vida do sistema. Toma-se como norma que nenhuma alteração no sistema deverá ser feita a não ser através do MACYS.

Além das capacidades de manutenção o MACYS incorpora um suporte ao projeto de sistemas e à documentação de sistemas. Para isso possui dois modos de execução: o modo batch, destinado à manutenção, e o modo interativo, destinado ao projeto;possui ainda uma rotina de conversão da descrição do sistema no modo interativo para a descrição do sistema no modo batch.

A manutenção automática de um sistema baseia-se nos seguintes conceitos:

- O sistema possui um nome que o caracteriza, e esse nome constitui o primeiro nível do rótulo de qualquer arquivo do sistema. Esse sistema constitui assim um único diretório.

- O diretório do sistema fica armazenado em fitas library-maintenance. A cada execução do MACYS para introduzir uma alteração no sistema, este se encarrega de carregar o diretório para o disco, e após executar todos os processos determinados pela alteração, descarrega o diretório para outra fita. Basta fornecer ao MACYS o número de série (serialno) das fitas para armazenamento do sistema, e o rodízio das fitas é automático.

- Quando um programa ou arquivo de dados passa para o grupo de manutenção, existe um arquivo original em uma fita

library-maintenance. Este arquivo original não deve ser alterado, e quaisquer correções necessárias serão feitas através de patches. O MACYS mantém na descrição do sistema a data de criação desse arquivo, recusando arquivos com data diferente. Isto garante que o arquivo original não será alterado, eliminando o problema de distinguir qual é sua versão mais recente. Toda vez que um arquivo for utilizado, incidirão sobre ele todos os patches já feitos, o que mantém documentado o histórico de suas alterações.

- A descrição do sistema, uma vez montada, raramente muda, e contém todas as indicações de "quem é usado em quem" . Assim, uma alteração em algum ponto do sistema propaga-se automaticamente, gerando todas as partes do sistema que tenham sido afetadas. Ficamos assim garantidos contra o esquecimento de algum passo necessário à geração do sistema.

- Cada arquivo permanente do sistema (por exemplo,um programa objeto) é resultado de uma "unidade", e a unidade é o elemento de descrição do sistema. Uma unidade possui um nome que a caracteriza, e esse nome é o segundo nível do rótulo do arquivo. Os arquivos permanentes terão um rótulo <nome do sistema>/<nome da unidade>, e na realidade o nome da unidade é escolhido em função do nome desejado para o arquivo (por exemplo, o programa objeto).

- Na descrição consta o destino a ser dado a cada arquivo final do sistema. Por exemplo, se temos um programa COBOL que usa uma procedure em ALGOL, temos dois programas objeto intermediários e um programa objeto final, resultado da ligação de ambos pelo BINDER. Este programa objeto final precisa ficar disponível em determinado disco com um rótulo em geral diferente, e com determinada segurança (GUARDED, etc.) , e isto é o seu destino. A cópia dos arquivos finais para seus lugares de destino, com a segurança correta, é efetuada automaticamente pelo MACYS, e constitui a instalação do sistema.

O uso do MACYS na manutenção de sistemas traz as seguintes vantagens:

- elimina as tarefas puramente mecânicas, tais como montar

jobs para recompilar partes do sistema, copiar arquivos , etc.

- elimina a preocupação de verificar se todos os passos exigidos foram executados, e todos os arquivos foram copiados para seus lugares corretos.
- reune a documentação de tudo o que foi feito no sistema , permitindo a visualização de sua estrutura e do estado em que se encontra.
- permite recuperar o sistema tal como era em alguma data anterior, bastando para isso retirar os patches posteriores a essa data.
- permite recompor discos com facilidade, em casos de destruição de seu conteúdo, através da facilidade de instalação automática.
- auxilia a alteração e testes do sistema, através do suporte ao projeto em modo interativo.

Um sistema que tenha sido projetado com o auxílio do MACYS ficará melhor estruturado e documentado, facilitando em muito sua manutenção futura. Sistemas adquiridos de terceiros, ou projetados com outras técnicas, podem ser adaptados à descrição do MACYS, aliviando o trabalho de manutenção.

A descrição do sistema se faz através da descrição de suas unidades, cada unidade correspondendo a um arquivo permanente do sistema. A descrição das unidades está contida em um único arquivo, denominado "units". O arquivo units é um arquivo de tipo SEQDATA, com 72 colunas úteis e numeração sequencial nas colunas 73 a 80. A descrição é feita através de uma linguagem de comandos em formato fixo: não se permite um número arbitrário de brancos nos comandos, e onde houver um branco ele é necessário.

Toda informação contida no arquivo units, que compareça muitas vezes e possa ser alterada, será transformada em um "nome interno" do sistema. Os nomes internos do sistema ficam descritos em um arquivo à parte, denominado "intnames".São de conhecimento exclusivo do MACYS, e serão substituídos pela

sua descrição contida no arquivo intnames. Constituem apenas mais um mecanismo de suporte à manutenção, permitindo alterar o sistema com maior facilidade.

Os rótulos de arquivos são especificados para o MACYS em um formato particular, que permite fornecer ao mesmo tempo o meio e o volume de armazenamento:

```
<título> ::= <title> <meio> .
<meio>   ::= ON <packname>
             IN <fita>
             Ø  (vazio, indicando ON DISK)
<fita>   ::= <title>(SERIALNO=<serialnumbers>)
exemplos: A/B.  (implícito ON DISK)
          A/B ON CCEUSP.
          A/B IN FONTES(SERIALNO=(100,101)).
```

A especificação IN implica sempre na execução automática pelo MACYS de uma cópia de ou para disco através de library-maintenance.

O conceito de um patch é o mesmo do SYSTEM/PATCH: um conjunto de cartões encabeçados por um cartão de controle $¶. Para o MACYS, usamos um formato fixo:

```
$¶ <nome da unidade> AAA.VV.NNN <comentarios>
```

AAA designa o autor do patch, VV sua versão e NNN um número seqüencial. O MACYS agrega os patches em um arquivo de patches , colocando-os em ordem alfabética de autor e numérica da seqüência NNN, um arquivo de patches para cada unidade. Por convenção, o autor USP fica em último na ordem alfabética.Cada patch deve possuir uma razão explicada em texto narrativo. Este texto deve ser incluído através de cartões de comentário $: imediatamente depois do cartão $¶. Um patch nunca deve ser retirado do arquivo de patches.

Como nota, mencionamos que tanto o nome do sistema como o nome de uma unidade podem conter barras (/), constituindo mais de um nível de diretório.

OS NOMES INTERNOS

Existem nomes internos pré-definidos:

@ - indica o nome do sistema.

+ - indica o nome da unidade.

[MYUSERCODE] - indica o usercode sob o qual o MACYS está sendo executado.

[MYFAMILY] - indica a família de substituição a DISK com a qual o MACYS foi iniciado. exemplo:
se FAMILY DISK = USERPACK OTHERWISE PACK
então [MYFAMILY] será substituído por USERPACK.

Os nomes internos @ e + são usados para escrever títulos de arquivos. Assim, dentro da unidade B do sistema A um label-equation da forma

```
FILE INX(TITLE=@/FONTE/+);
```

será substituído por

```
FILE INX(TITLE=A/FONTE/B);
```

Os nomes internos constantes no arquivo intnames são descritos por um nome simbólico único, que pode conter o sinal menos (-). Ao usar um nome interno, este deve ser escrito entre brackets, sem brancos ([...]), excetuando @ e +.

A descrição de um nome interno começa com um registro de identificação começando na coluna 1, na forma:

```
$NAME <nome-interno>
```

e termina por um registro contendo

```
/*
```

na primeira coluna. O conteúdo dos registros existentes entre este dois formam três categorias:

1) simples substituição de cadeia de caracteres. É permitido apenas um registro, e a cadeia a ser substituida fica entre pontos (.). Seja por exemplo:

```
$NAME PACK-DO-SISTEMA
.CCEUSP.
/*
```

Assim, um label-equation

```
FILE OUT(TITLE = @/+, PACKNAME = [PACK-DO-SISTEMA]);
```

será substituído por

```
FILE OUT(TITLE = @/+, PACKNAME = CCEUSP);
```

2) um título de um arquivo. É permitido apenas um registro contendo o título terminado por ponto (.), e opcionalmente um range de seqüência. Um título pode conter um nome interno.

Exemplos:

```
$NAME DECLARAÇOES-FORWARD
@/DECLARATIONS ON [PACK-DO-SISTEMA].
/*
$NAME PEDACO-DA-TABELA
@/TABELA. 1000 - 5000
/*
```

No primeiro caso, (TITLE=[DECLARACOES-FORWARD]) será substituído por (TITLE=@/DECLARATIONS ON CCEUSP). No segundo caso, o MACYS se encarregará de gerar um arquivo auxiliar cujo conteúdo é o range 1000 a 5000 do arquivo @/TABELA, substituindo o nome interno pelo title do arquivo auxiliar. A criação deste arquivo auxiliar pode não ser necessária, por exemplo em cartões $INCLUDE ou $FROM dos compiladores, uma vez que estes aceitam o range de seqüência. Assim,

```
$FROM [PEDACO-DA-TABELA]
```

será substituído por

```
$FROM "@/TABELA" 1000 - 5000
```

e a manipulação desta categoria de nome interno depende do contexto.

3) trechos de arquivos, notadamente trechos de programas como descrição de uma FD em COBOL, ou instruções para o BINDER muito utilizadas. Permite-se um número qualquer de registros, sendo o primeiro deles

```
DD *
```

na primeira coluna, terminando com /*. Também dependendo do contexto, o MACYS poderá criar um arquivo auxiliar com esse conteúdo. Devido a esta categoria, o arquivo intnames pode ser criado com tipo SEQDATA ou tipo COBOL, dependendo do maior número de ocorrências de $INCLUDE em ALGOL ou $FROM em COBOL .

Seja por exemplo

```
5000 ¶NAME FD-DE-ENTRADA
5100 DD *
5200 FD  ENTRADA.
5300     01 ...
 ...
5900 /*
```

Caso o arquivo intnames seja tipo COBOL, um cartão

$FROM [FD-DE-ENTRADA]

será substituído por

$FROM "@/INTNAMES" 5200 - 5800

No entanto, se o tipo for SEQDATA, o MACYS será obrigado a criar um arquivo auxiliar do tipo COBOL, substituindo o title deste arquivo auxiliar no $FROM.

Durante o projeto, em modo interativo, o arquivo de intnames é gerado manualmente com auxílio do CANDE (suporte de processamento remoto), com title @/INTNAMES/V0, e o MACYS se encarrega de ordená-lo em ordem alfabética. Após a conversão para o modo batch, qualquer alteração no arquivo intnames será feita através de patches, como para qualquer outro arquivo.

A DESCRIÇÃO DA UNIDADE

A descrição de todas as unidades constitui o arquivo units. Da mesma forma que o arquivo intnames, é gerado manualmente com auxílio do CANDE durante o projeto, com title @/UNITS/V0, e o MACYS se encarrega de ordená-lo em ordem alfabética. Após convertido para modo batch, qualquer alteração será feita através de patches.

A descrição de uma unidade começa com um registro de identificação, começando na coluna 1, na forma:

¶NAME <nome/da/unidade>

e termina com outro registro ¶NAME ou com o fim do arquivo. O conteúdo dos registros seguintes é formado por comandos que co

meçam na coluna 1, e são, nesta ordem exata:

```
.PICK
.NEEDS:
.PROPAGATE:
.SOURCE:
.PATCHFROM:
.WFL-BEFORE:
.RUN-BEFORE:
.COMPILE:
.BIND:
.WFL-AFTER:
.RUN-AFTER:
.MARK-AS-CP:
.MARK-AS-COMPILER:
.MARK-AS-SYSFILE:
.DESTINATIONS:
```

O conteúdo destes comandos é fornecido ao MACYS, terminando com um cartão /*; o MACYS gera mais os seguintes comandos para documentação

```
.RESULT:
.VERSION:
.DATES:
.USEDON:
```

Estes comandos são protegidos, e o MACYS impedirá que incidam patches sobre eles:

O comando .PICK

É usado apenas durante o projeto, em modo interativo, para o desenvolvimento de um programa. Para facilidade, o programador mantém a descrição da unidade que compilará esse programa dentro do corpo do programa fonte, fazendo as alterações necessárias em um único arquivo. O conteúdo do .PICK é um registro contendo o title do arquivo fonte, e o MACYS procurará a descrição da unidade dentro desse arquivo. Note que, como todo title, termina com ponto (.).

O comando .NEEDS:

É uma série de registros contendo cada um deles um nome de outra unidade necessária para a composição desta. O MACYS coleta a descrição .NEEDS: de todas as unidades do sistema e monta a tabela de .USEDON: , tabela que será usada para propagar uma alteração em uma unidade para outras unidades, garantindo a integração correta do sistema. Seja por exemplo uma rotina A em FORTRAN que será ligada pelo BINDER aos programas FORTRAN B, C e D, cada um deles constituindo uma unidade. Assim, na descrição de B, C e D existe o comando

```
.NEEDS:
A
/*
```

Através destes comandos, o MACYS agrega à descrição de A :

```
.USEDON:
B
C
D
/*
```

e quando efetuarmos uma alteração na rotina A, esta alteração se propaga para as unidades B, C e D gerando novos arquivos objeto atualizados.

Além de ser usado para a propagação da alteração, este comando determina a ordem de execução das unidades: se B needs A então A será executada antes de B. O resultado de A também se propaga para B: se A terminou anormalmente, fica marcada como "não executada", e isto é considerado término anormal. Assim se houver D needs B, D também não será executada.

Uma unidade pode usar alguns nomes internos e todos estes nomes internos são acrescentados automaticamente pelo MACYS ao comando .NEEDS: ; não é necessário fornecer essa informação ao MACYS. Uma alteração em um nome interno se propaga às unidades através desse "needs implícito". Por exemplo, se vários programas COBOL incluem uma mesma FD através de um nome interno de tipo 3), uma alteração nessa FD causará a recompilação daqueles programas COBOL.

O comando .SOURCE:

É uma série de registros, cada um contendo um title terminando em ponto (.). Descrevem os arquivos que serão usados nessa unidade. Aqui adotamos a seguinte convenção: apenas o primeiro arquivo pode ter patches associados, e no caso geral o primeiro arquivo é o programa fonte dessa unidade. Como tal será usado no comando .COMPILE:. O MACYS se encarrega de copiar os arquivos descritos no .SOURCE: que estejam em fita, e antes de executar a unidade testa a presença de todos os arquivos. Caso algum deles não exista, a unidade fica marcada como não executada, tendo um término anormal.

Os comandos .WFL- e .RUN-

São formados por uma série de registros que compõem um "statement" em WFL (a linguagem de comandos do computador). Um statement de WFL será passado ao MCP (sistema operacional) para ser executado sem nenhuma alteração, efetuando apenas as substituições de nomes internos @, + e [...]. Um statement de WFL termina sempre com ponto e vírgula (;) e se estende por no máximo 20 registros.

O comando .WFL- é usado para statements que não são associados a tasks, tal como CHANGE, REMOVE, e o MACYS não tem capacidade de coletar seu resultado, a menos de erros de sintaxe em WFL. Nada impede que aqui se execute um task qualquer,porém o MACYS não tentará coletar a informação de término anormal: o uso de .WFL- pressupõe término normal.

O comando .RUN- é usado para a execução de um task cujo resultado é necessário conhecer. Contém statements RUN ou COMPILE associados a um task [T], e este task é usado pelo MACYS para coletar o modo de término: um término anormal interrompe a execução da unidade e a marca não executada. Um exemplo seria:

```
RUN  @ /PROGRAMA ("PARAMETRO") [T]; FILE... etc;
```

O comando .RUN- permite a inclusão do comando WFL ?DATA, o que não se permite em .WFL- . Um arquivo "data" é identificado pelo registro contendo apenas a palavra DATA na coluna 1, e termina com o cartão /*. O arquivo data será montado pelo MACYS como um arquivo auxiliar, sendo gerado um label equation para o FILE CARD. O arquivo data poderá ser formado por arquivos descritos por nomes, internos de tipo 2) ou 3) , desde que o nome interno comece na coluna 1. No entanto, @ e + não são substituídos. Como exemplo, imaginemos que temos os nomes internos

```
¶NAME A
DD *
cartão A
/*
¶NAME B
TABELA/UM ON CCEUSP.
/*
```

e o arquivo TABELA/UM contenha um registro X. Assim, um DATA dado por

```
DATA
cartão 1
[A]
[B]
cartão 2
```

gerará um arquivo auxiliar contendo

```
cartão 1
cartão A
X
cartão 2
```

Isto é particularmente útil para conjuntos de instruções do BINDER que ocorrem em vários programas diferentes.

Notemos que um .RUN-AFTER: contendo RUN @/ + causará a execução do programa que é resultado da própria unidade.

Apenas para usuários privilegiados, o primeiro registro de um .RUN- ou .WFL- pode ser, a partir da coluna 1

```
..FAMILY <family specification>
```

e que altera o family-substitution apenas para esse RUN ou WFL.

O comando .COMPILE:

Este comando é quase sempre o núcleo de uma unidade. Associa-se ao compilador um task [T] para a coleta do resultado, e tem vários efeitos associados: O primeiro arquivo mencionado em .SOURCE: é o FILE TAPE do compilador, e é o único arquivo a ter patches. O MACYS executa a ordenação dos patches antes de passar à compilação, e introduz os label-equation apropriados para o FILE CARD e FILE TAPE do compilador; assim o comando COMPILE pode ser escrito na forma mais simples como

```
COMPILE @/+ WITH COBOL [T] LIBRARY;
```

apenas, e será completado pelo MACYS. Por convenção, o arquivo ordenado de patches que será "merged" com o primeiro .SOURCE: terá o title @/PATCHED/+. Não é permitida a inclusão de DATA neste comando, e não se permite a existência de mais de um .COMPILE: na descrição da unidade.

O comando .BIND:

É idêntico a um comando .RUN- . A única diferença é que é destinado à execução do BINDER, e o arquivo DATA terá @ e + substituídos em qualquer posição. Assim podemos escrever por exemplo

```
.BIND:
COMPILE @/+  WITH BINDER [T] LIBRARY;
DATA
HOST IS @/+;
BIND = FROM @/ROTINAS;
/*
```

O comando .PROPAGATE:

Usado em casos especiais em que desejamos executar outras unidades em conjunto com a unidade em questão, embora não haja relação de .NEEDS: . Por exemplo, quando o resultado de uma unidade A é um arquivo que não desejamos armazenar no diretório do sistema, uma unidade B que necessitar do arquivo A usará .PROPAGATE para A, causando a criação do arquivo A. Um exemplo: a unidade B é um programa objeto, e contém a execução do próprio programa, para teste, tendo como entrada o arquivo A. Então, B needs A, mas não B usedon A. Assim, uma alteração em B não se propagaria para A. Usamos então .PROPAGATE: . Uma alteração em B "propaga-se" para A, e uma vez que a ordem de execução é determinada por .NEEDS: , o MACYS executará primeiro A, criando o arquivo, e depois B, que o usa. Ao terminar o MACYS, o arquivo A por não fazer parte do diretório do sistema, não existirá na fita library-maintenance, donde a necessidade de ser gerado a cada vez. Inúmeros outros usos aparecem na prática, facilmente intuíveis.

O comando .PATCHFROM:

Destinado a casos muito particulares em que o mesmo programa fonte gera programas objeto diferentes, utilizando por exemplo recursos de "COMPILE-TIME PROCESSOR". Significa que ao ordenar os patches de uma unidade serão agregados os patches de outra unidade, e ao dizermos que é usado para gerar os compiladores ALGOL, DCALGOL e DMALGOL já caracterizamos seu uso. Os usuários em geral podem ignorar esse comando.

Os comandos .MARK-AS-

Apenas para usuários privilegiados, marca o arquivo após instalado como control program, compiler ou system file.

O comando .DESTINATIONS:

O primeiro registro descreve a classe de segurança do arquivo após instalado, e os registros seguintes são titles que descrevem o local de armazenamento permanente do resultado da unidade. A cópia e a mudança de segurança são efetuadas automaticamente pelo MACYS, sempre que todas as unidades tenham sido executadas corretamente. Como exemplo,

```
.DESTINATIONS:
<PUBLIC SECURED>
OBJECT/PROGRAMA ON CCEUSP.
/*
```

causa a execução de COPY @/+ AS OBJECT/PROGRAMA TO CCEUSP (PACK); SECURITY OBJECT/PROGRAMA ON CCEUSP PUBLIC SECURED. Lembremos que os titles terminam com ponto (.) e permitem o uso de nomes internos @,+ e [...]. Notadamente, devemos usar um nome interno de tipo 1) para manter o rótulo do pack de destino.

O comando .RESULT:

Gerado pelo MACYS, mantém várias informações internas, sendo aparente apenas a data de ativação da unidade, e o resultado de sua execução.

O comando .DATES:

Gerado pelo MACYS, mantém a data de criação do arquivo fonte e do arquivo resultado da unidade. A data de criação do arquivo fonte é mantida para recusar fontes alterados indevidamente, e a data de criação do arquivo resultado é usada para conferir o estado de instalação do sistema, ou seja,confrontar essa data com a data de cada um dos arquivos mencionados em .DESTINATIONS: , efetuando a cópia quando as datas não coincidem.

O comando .VERSION:

Gerado pelo MACYS, contém um número de versão da unidade, incrementado a cada vez que a unidade é ativada. Esse número, no futuro, será copiado para o segmento zero dos arquivos objeto ou transmitido ao compilador. A utilização ou não do número de versão é um problema a ser considerado. Por enquanto existe apenas como documentação.

O comando .USEDON:

Gerado pelo MACYS, descreve o inverso do comando .NEEDS: , e é fornecido para documentação.

OS PARÂMETROS DO MACYS

Algumas informações fixas são fornecidas ao MACYS através do parâmetro tipo string. São elas:

SYSTEM <nome do sistema>

Obrigatório, identifica o sistema e fornece a substituição de @ .

DO <nome da unidade>

Usado apenas em modo interativo, indica uma unidade a ser executada.

TAPES [S1,S3]
TAPES [(S1,S2),(S3,S4)]

Obrigatório em modo manutenção, fornece as fitas para rodízio, limitado a duas apenas. S1,S2,S3 e S4 são serialnumbers escritos conforme a sintaxe do WFL, e na segunda forma S2 e S4 indicam fitas de continuação, ou rolo 2. O rodízio dessas fitas é feito automaticamente pelo MACYS, não sendo necessá-

rio alterar o parâmetro a cada execução.

SORT

Usado apenas em modo interativo, causa a ordenação alfabética dos arquivos units e intnames. Pode ser usado em conjunto com DO <...>.

INSTAL

Usado apenas em modo manutenção, requer a verificação da instalação e cópia dos arquivos cujas datas estejam erradas.

CONVERT

Usado para converter a descrição do sistema do modo interativo para o modo batch.

O MODO BATCH

O MACYS é executado através de um job, na forma:

```
?RUN *CCE/MACYS("<parâmetros>");
?DATA
  patches
?END JOB
```

Os patches são dados pelos cartões de contrôle $¶. No caso COBOL, $¶ aparece na coluna 7, nos demais casos na coluna 1. Os patches podem existir em um arquivo independente, e existe o comando $.FILE <title>. que colocado no ?DATA faz com que o MACYS passe a ler esse arquivo, retornando depois ao ?DATA.

Os patches são ordenados por nome de unidade, e o diretório do sistema é copiado da última fita. As unidades para as quais existem patches são marcadas "a ser executada". Notemos aqui que um patch "vazio", contendo apenas o cartão $¶ não é guardado pelo MACYS no arquivo de patches, constituindo

uma forma de anular um patch já existente ou apenas de pedir a execução de uma unidade.

O MACYS percorre todas as unidades, montando as relações .USEDON: a partir de .NEEDS: . Através de .USEDON: e .PROPAGATE: marca todas as unidades a serem executadas,e em seguida as ordena numa seqüência temporal determinada pelos .NEEDS: . Copia todos os arquivos mencionados em .SOURCE: que estejam em fita. Passa então a executar essa seqüência de unidades, executando os statements de WFL de cada unidade. Términos anormais propagam-se através das relações .USEDON: (que é a inversa de .NEEDS:). Ao longo da execução são criados arquivos de "checkpoint", de modo que em caso de parada da máquina o MACYS retomará a seqüência no ponto em que foi interrompida. Ao término da execução da última unidade, copia o diretório do sistema para a próxima fita de rodízio. Caso todas as unidades tenham terminado normalmente, efetua então as cópias necessárias à instalação.

O MODO INTERATIVO

No modo interativo, ou para suporte de projeto, o MACYS assume que todos os arquivos necessários estejam presentes, não executando nenhuma cópia, e não coleta nenhum resultado de execução, uma vez que a execução não é feita através dele: o MACYS apenas cria um arquivo de comando indireto para o CANDE, contendo a substituição de todos os nomes internos e os label-equation para quaisquer arquivos auxiliares que tenha sido necessário gerar. Esse arquivo de comando indireto pode ser executado várias vezes, sem necessidade de invocar o MACYS novamente. Uma vez que a descrição da unidade fica no corpo do programa, basta editar um único arquivo.

Um exemplo típico de programação seria o de um programa COBOL que usa rotinas em ALGOL e é executado para teste com um arquivo de teste fixo. O programador cria seu programa, colocando no início a descrição da unidade com comando .COMPI-

LE: , .BIND: , .RUN-AFTER: . Ao escrever o programa, as FD necessárias devem estar no arquivo intnames, e o programador coloca $FROM [...] nos locais apropriados.Executa então o MACYS, que substitui os nomes internos, acerta os $FROM, cria arquivos auxiliares inserindo label-equations, e cria o arquivo de comando indireto. Notemos que o programa fonte não é alterado com isso. O arquivo de comando indireto, sempre que possível, gera um único comando WFL para o CANDE, de modo que a compilação, ligação e execução passarão apenas uma vez pela TASK QUEUE do CANDE. Enquanto não houver alterações no arquivo intnames ou na descrição da unidade, não será necessário invocar o MACYS novamente.

Notemos que durante essa fase, é necessário que uma pessoa coordene os esforços, principalmente quanto a modificações no arquivo de intnames.

Durante o modo interativo, as convenções do CANDE são respeitadas. Assim, os resultados das unidades serão não @/+, mas OBJECT/@/+ . O MACYS se encarrega de fazer algumas pequenas alterações para seguir essas convenções.

A CONVERSÃO DE MODOS

Após o término do projeto, o MACYS converte a descrição do modo interativo para o modo batch. Para isso são necessários os seguintes passos executados pelo MACYS:

Extrair de cada arquivo fonte a descrição da unidade, colocando-a no arquivo units em substituição ao .PICK correspondente, gerando um novo arquivo fonte "limpo".

Transformar todo OBJECT/@/+ em @/+.

Coletar as datas de criação dos fontes e objetos (ou resultados), e inicializar .DATES: .

Copiar os arquivos fonte para uma fita library-

maintenance.

Ordenar alfabeticamente os arquivos units e intnames.

Inicializar os arquivos de patches de cada unidade, contendo de início todos os cartões do fonte que contenham nomes internos. Isto para que o MACYS não tenha que percorrer todo o arquivo fonte, bastando inspecionar o arquivo de patches durante a manutenção.

Copiar o diretório do sistema para a primeira fita do rodízio.

Instalar o sistema.

CONCLUSÃO

O MACYS é uma ferramenta poderosa no suporte à manutenção, e traz um auxílio substancial ao projeto. Em uma implementação futura serão agregados novos comandos destinados a automatizar a documentação na fase de projeto, permitindo gerar automaticamente as chamadas "pasta do sistema" e "pasta do programa", bem como manuais de usuário e de operação.

EXEMPLO

Nas páginas seguintes temos como exemplo um relatório de uma execução do MACYS. Tomamos como sistema uma parte do conjunto de utilitários: dois programas, UEDIT e ESPOL6800, ambos utilizando tabelas geradas por um programa gerador de tabelas TABLEGEN. Temos três unidades apenas, e a execução do MACYS foi iniciada pelo seguinte job:

```
?JOB DEMO; CHANGE MACYS;
  CLASS 99; PRIORITY 99;
  FAMILY DISK=PACK ONLY;
BEGIN
?RUN *CCE/MACYS("SYSTEM DEMO TAPES [000099,000100]");
?DATA
$# TABLEGEN USP.9.00 FORCE COMPILE ONLY
?END JOB
```

Foi introduzido um patch vazio para a unidade TABLEGEN, que é uma maneira de pedir a execução de uma unidade. Como TABLEGEN é usado em ESPOL6800 e UEDIT, estas duas unidades também serão executadas.

Na primeira página temos a seqüência de execução de processos efetuada pelo MACYS. Na segunda e terceira, a descrição das unidades com os resultados coletados pelo MACYS, mais o histórico de cada unidade dado pelos seus patches. Na quarta página temos a descrição dos nomes internos, e finalmente um relatório interno dos statements em WFL após a substituição de nomes internos, onde se vê como o comando .COMPILE é completado.

```
                                        *MACYS* : 06/09/1981 (TUE)  00:12:42

$$ TABLEGEN USP.9.00 FORCE COMPILE ONLY

=== SORTED PATCHES ===


TABLEGEN
          $$ TABLEGEN USP.9.00 FORCE COMPILE ONLY

****************************************
DOING: TABLEGEN
****************************************
.COMPILE:

$$$ TABLEGEN: DONE OK.

****************************************
DOING: UEDIT
****************************************
.RUN-BEFORE:
.COMPILE:

$$$ UEDIT: DONE OK.

****************************************
DOING: ESPOL6800
****************************************
.RUN-BEFORE:
.COMPILE:

$$$ ESPOL6800: * SYNTAX *
```

```
                                *MACYS* : 06/09/1981 (TUE)  00:12:42

SYSTEM: DEMO

 INPUT TAPE: DEMO01     SERIALNO: 000099

OUTPUT TAPE: DEMO02     SERIALNO: 000100

================================================================================

ESPOL6800

        .RESULT:      06/09/1981 (TUE)  00:15:50 * SYNTAX *

        .VERSION:     29.006.170

        .DATES:       SOURCE =  06/08/1981 CODE    =  06/08/1981

        .NEEDS:                                                          10120000
                      TABLEGEN                                           10130000
                      [TABELA-DO-ESPOL6800]
                      [TABELA-PALAVRAS-RESERVADAS-ESPOL6800]
        .SOURCE:                                                         10150000
                      @/FONTE/* .                                        10160000
        .RUN-BEFORE:                                                     10180000
                      RUN @/TABLEGEN [T];                                10190000
                        FILE OUT=[TABELA-DO-ESPOL6800];                  10200000
                      DATA                                               10210000
                      [TABELA-PALAVRAS-RESERVADAS-ESPOL6800]             10220000
        .COMPILE:                                                        10240000
                      COMPILE @/* WITH ALGOL [T] LIBRARY;                10250000
                        STACK 3000;                                      10260000

        PATCH HISTORY:

          $# ESPOL6800 .....0 ($. CARDS) AUTOMATIC          ( 06/08/1981 (MON)  23:05:27 )
          $.SINGLE MARK COMPARE
          $# ESPOL6800 ..1..1 (INTNAMES) AUTOMATIC          ( 06/08/1981 (MON)  23:05:27 )
          $# ESPOL6800 USM.9.00 DOLLAR CARDS                ( 06/09/1981 (TUE)  00:07:30 )
          $SET MERGE LISTINCL

================================================================================

TABLEGEN

        .RESULT:      06/09/1981 (TUE)  00:14:25 DONE OK.

        .VERSION:     29.006.170

        .DATES:       SOURCE =  06/08/1981 CODE    =  06/09/1981

        .USEDON:                                                         10400000
                      VECI;                                              10410000
                      ESPOL6800                                          10420000
        .SOURCE:                                                         10440000
                      @/FONTE/* .                                        10450000
        .COMPILE:                                                        10470000
                      COMPILE @/* WITH ALGOL [T] LIBRARY;                10480000
```

```
UEDIT

        •RESULT:      06/09/1981 (TUE)  00:15:07 DONE OK•

        •VERSION:     29.006.170

        •DATES:       SOURCE =  06/08/1981 CODE   =  06/09/1981

        •NEEDS:                                                                  10620000
                      TABLEGEN                                                   10630000
                      [TABELA=DO=UEDIT]
        •SOURCE:                                                                 10650000
                      @/FONTE/* •                                                10660000
                                                                                 10680000
        •RUN=BEFORE:
                      RUN #/TABLEGEN [T];                                        10690000
                        FILE OUT=[TABELA=DO=UEDIT];                              10700000
                      DATA                                                       10710000
                      %                                                          10720000
                      VALUE ARRAY VERBS (                                        10730000
                      WHAT                                                       10740000
                      REM                                                        10750000
                      SAVE                                                       10760000
                      END                                                        10770000
                      HELP                                                       10780000
                      SHOW                                                       10790000
                      REPEAT                                                     10800000
                      %       VERBS ABOVE WONT BE PUSHED ON LAST ENTRIES•        10810000
                      %       FIND MUST BE FIRST VERB TO BE PUSHED •••           10820000
                      FIND                                                       10830000
                      GET                                                        10840000
                      GO                                                         10850000
                      );                                                         10860000
                      %                                                          10870000
                      $COPY                                                      10880000
                                                                                 10890000
                        DEFINE LASTVERB = (VERBSSIZE - 2) #;                     10900000
                                                                                 10910000
                      $END                                                       10920000
                      $OFF                                                       10930000
        •COMPILE:                                                                10950000
                      COMPILE @/* WITH ALGOL [T] LIBRARY;                        10960000
                        PL PRIORITY 99; PRIORITY 99;                             10970000

      PATCH HISTORY:

        $# UEDIT .....0 ($• CARDS) AUTOMATIC                    ( 06/08/1981 (MON)  22:59:07 )
        $.SINGLE MARK COMPARE
        $# UEDIT ..1..1 (INTNAMES) AUTOMATIC                    ( 06/08/1981 (MON)  22:59:07 )
        $# UEDIT USP.9.00 DOLLAR CARDS                          ( 06/09/1981 (TUE)  00:07:41 )
        $SET MERGE LISTINCL
```

```
                                *MACYS* : 06/09/1981 (TUE)  00:12:42

SYSTEM: DEMO   -   INTERNAL NAMES:

TABELA-DC-ESPOL6800

        (TABELA-DO-UEDIT)                                          00100100

TABELA-DC-UEDIT

        @/AUX/+.                                                   00100400

TABELA-PALAVRAS-RESERVADAS-ESPOL6800

        DD *                                                       00100700
        %                                                          00100800
        VALUE ARRAY RESWORDS(                                      00100900
        WHILE                                                      00101000
        DO                                                         00101100
        BEGIN                                                      00101200
        END                                                        00101300
        $OFF                                                       00101400
```

```
**********************************************
DOING: TABLEGEN
**********************************************
.COMPILE:
----------------------------------------------
WFLER
----------------------------------------------
COMPILE DEMO/TABLEGEN WITH ALGOL [T] LIBRARY;PL FILE TAPE=DE
MO/FONTE/TABLEGEN;PL FILE CARD(KIND=DISK,FILETYPE=7,TITLE=DE
MO/FONTE/TABLEGEN);                                    ;X:=
T(HISTORY);RUN*CCE/GETHISTORY ON PACK(X,"DEMO/AUX/HISTORY.")
;END JOB

### TABLEGEN: DONE OK.

**********************************************
DOING: UEDIT
**********************************************
.RUN-BEFORE:
----------------------------------------------
WFLER
----------------------------------------------
RUN DEMO/TABLEGEN [T];FILE CARD(KIND=DISK,FILETYPE=7,TITLE=D
EMO/AUX/UEDIT/DATA001 ON PACK);
                          FILE OUT=DEMO/AUX/UEDIT;
                                ;X:=T(HISTORY);RUN*CCE/GETH
ISTORY ON PACK(X,"DEMO/AUX/HISTORY.");END JOB
.COMPILE:
----------------------------------------------
WFLER
----------------------------------------------
COMPILE DEMO/UEDIT WITH ALGOL [T] LIBRARY;PL FILE TAPE=DEMO/
FONTE/UEDIT;PL FILE CARD(KIND=DISK,FILETYPE=7,TITLE=DEMO/PAT
CHED/UEDIT);                                       PL PRIORI
TY 99; PRIORITY 99;
 ;X:=T(HISTORY);RUN*CCE/GETHISTORY ON PACK(X,"DEMO/AUX/HISTO
RY.");END JOB
```

No exemplo dado, os programas fonte não foram copiados de fita; na prática o comando .SOURCE: seria
@/FONTE/+ IN [SYMBOLTAPE].
ficando a especificação da fita em nome interno.

A unidade TABLEGEN foi descrita apenas com dois comandos, .SOURCE: e .COMPILE: , de forma extremamente simples; o comando .USEDON: é gerado pelo MACYS.

Tanto no ESPOL6800 como no UEDIT temos o comando .NEEDS: TABLEGEN, ao qual foi acrescentado pelo MACYS a relação dos nomes internos mencionados na unidade. Ambos possuem um comando .RUN- para executar o TABLEGEN, e este usa um arquivo de entrada, colocado em DATA. No primeiro caso usamos um nome interno tipo DD *, e no segundo caso consta da própria descrição da unidade (veja o comando .RUN-).

A execução do TABLEGEN gera um arquivo OUT, e em ambos os casos seu title foi modificado através de um nome interno. Como vemos na descrição dos nomes internos, TABELA-DO-ESPOL6800 aponta para TABELA-DO-UEDIT. Isto foi feito apenas para mostrar que é possível, e ressaltar o fato de que
@/AUX/+
tem substituições diferentes em unidades diferentes. Por convenção, o diretório @/AUX é removido ao fim da execução de cada unidade, de modo que esse é um arquivo temporário.

O arquivo OUT do TABLEGEN é incluído no compilador através de um cartão $INCLUDE [...]. Estes cartões do programa fonte que mencionam nomes internos são colocados pelo MACYS nos patches ..1..1 , de forma que a busca de nomes internos não necessita ser feita dentro do arquivo fonte.

Finalmente, a compilação do ESPOL6800 não terminou normalmente. Isto inibe o processo de instalação do sistema, e o programa UEDIT não seria copiado para seu destino. Como este é um caso em que o sistema é composto de programas independentes, seria interessante que a instalação de cada um também se-

ja independente. O comando .DESTINATIONS: será modificado para indicar quando se deseja uma instalação independente ou não.