Este exemplar corresponde à redação final da tese defendida por Plínio de Sá Leitão Júnior e aprovada pela Comissão Julgadora em 27/08/1992.

Mario Jino
Orientador

SUPORTE AO TESTE DE PROGRAMAS COBOL NO AMBIENTE POKE-TOOL

PLÍNIO DE SÁ LEITÃO JÚNIOR

ORIENTADOR: PROF. DR. MARIO JINO

CO-ORIENTADOR: PROF. DR. JOSÉ CARLOS MALDONADO

ESTE EXEMPLAR CORRESPONDE À REDAÇÃO FINAL DA DISSERTAÇÃO APRESENTADA À FACULDADE DE ENGENHARIA ELÉTRICA DA UNICAMP, PARA OBTENÇÃO DO TÍTULO DE MESTRE EM ENGENHARIA ELÉTRICA, E APROVADA PELA COMISSÃO JULGADORA.

CAMPINAS, AGOSTO DE 1992

Aos meus Pais

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Dr. Mario Jino, pela oportunidade de trabalho, pelo fornecimento de subsídios técnicos e por despertar em mim a importância da atividade de Teste de Software.

Ao Prof. Dr. José Carlos Maldonado cuja participação foi fundamental no desenvolvimento deste trabalho; ressalto suas opiniões, sua amizade e seu exemplo ético.

Ao Rubens Pontes Fonseca que, através de seus pequenos e grandes gestos, transmitiu confiança e companheirismo constantes.

Ao Marcos Lordello Chaim pelo suporte técnico fornecido durante a configuração da ferramenta POKE-TOOL.

À Sílvia Regina Vergílio pela paciência e incentivo demonstrados principalmente nos momentos finais deste trabalho.

Aos companheiros de moradia: Brito, Rudolf, Menoti e Paulo sempre presentes em todos os momentos.

Ao Prof. Dr. Paulo César Bezerra pela amizade e pela oprtunidade de ingresso no curso de mestrado.

Ao Renato Helônio Pinheiro Braga pelo exemplo profissional muito importante para meu enriquecimento pessoal.

À Cyndia pela sua presença e carinho.

Aos meus irmãos: Ângela, Ricardo, Elisabeth, Jaqueline e Patrícia que sempre torceram pelo meu sucesso.

Aos meus pais: Plínio e Eliezita o meu sincero obrigado pelo exemplo, educação e apoio, presentes valiosos e de difícil retribuição. Espero, como forma maior de agradecimento, honrar e aperfeiçoar tudo aquilo por vocês ensinado.

Ao CNPQ pelo suporte financeiro.

RESUMO

Neste trabalho são discutidos os aspectos de aplicação de teste estrutural baseado em análise de fluxo de dados em programas COBOL. Com esse propósito, uma ferramenta denominada POKE-TOOL [MAL89,CHA91b], que apóia a utilização dos critérios Potenciais Usos [MAL88,MAL91], foi configurada para a linguagem COBOL. São abordados: a caracterização de unidade em programas COBOL; a abstração das estruturas de controle que compõem essa linguagem; o fluxo de dados presente em programas COBOL; e aspectos referentes à instrumentação de programas COBOL. A discussão ressalta a não trivialidade dessas tarefas: por exemplo, a inexistência de estrutura de bloco na linguagem COBOL dificulta a atividade de instrumentação de código fonte. Como produto principal desta dissertação,a versão operacional da POKE-TOOL para a linguagem COBOL constitui uma contribuição importante no sentido de apoiar o teste estrutural de unidades escritas em linguagem COBOL.

ABSTRACT

This work explores the issues involved in applying data flow based structural testing in COBOL programs; a tool named POKE-TOOL [MAL89,CHA91b] that supports Potential Uses Criteria application [MAL88,MAL91] was configured for the COBOL programming language. The following topics are discussed: characterization of units in COBOL; abstraction of control flow structures; data flow in COBOL; and program instrumentation. The presentation points out that these tasks are non-trivial: for example, the lack of block structuring in COBOL makes difficult the instrumentation of source code. As the main product of this thesis, the operational version of the POKE-TOOL for COBOL language constitute an important contribution to structural testing of units implemented in COBOL.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Neste capítulo são introduzidas as motivações para o desenvolvimento desta Tese e os objetivos principais a serem atingidos. A organização da Tese é apresentada na última seção deste capítulo.

1.1 MOTIVAÇÃO

Soluções implementadas por software têm crescentemente sido aplicadas em diversas atividades presentes em nossa sociedade; tal tendência revela a existência de vários domínios de atuação de sistemas de software. Como conseqüência, esforços são direcionados para a redução de custo e para a melhoria de qualidade dessas soluções.

No contexto de sistemas de software voltados para aplicações comerciais, que são caracterizadas por computações relativamente simples e grande volume de informação de entrada e saída, a linguagem COBOL (*COmmon Business Oriented Language*) ocupa um lugar de destaque devido a sua grande difusão. O propósito do projeto da linguagem COBOL foi fornecer uma linguagem comum orientada para negócios, comum no sentido que um programa fonte fosse portável entre computadores de diversos fabricantes. Com o apoio do governo norte-americano, diversos fabricantes investiram no desenvolvimento de compiladores para essa linguagem; em virtude disto, a linguagem COBOL tornou-se uma linguagem amplamente utilizada no dia-a-dia de empreendimentos de negócios. Apesar do surgimento de novas ferramentas e linguagens orientadas a aplicações comerciais, uma considerável parcela do desenvolvimento de novos sistemas ainda utiliza a linguagem COBOL; além disto, a linguagem COBOL é apontada como uma das linguagens que serão mais utilizadas na década de 90 [NAN92]. Adicionalmente, cabe lembrar que grande parte dos sistemas já desenvolvidos em linguagem COBOL continua em operação recebendo manutenção.

Segundo Deutsch, o desenvolvimento de sistemas de software envolve

uma série de atividades de produção, onde as oportunidades de introdução de falhas humanas são enormes; por este motivo, o desenvolvimento de software deve ser acompanhado por atividades de garantia de qualidade [DEU82]. Teste de software é uma das atividades de garantia de qualidade de software [PRE87], tendo sido amplamente utilizado para aumentar a confiança no produto desenvolvido [CLA82]. O objetivo principal da atividade de teste de software é revelar a presença de erros [MYE79].

Teste de software consiste de gerar um conjunto de casos de teste de acordo com alguma estratégia de teste e, então, avaliar os resultados obtidos em relação aos resultados esperados.

Duas técnicas são geralmente usadas na seleção de dados para a aplicação da atividade de teste: funcional e estrutural. O teste funcional utiliza-se principalmente da especificação e dos aspectos funcionais do software para derivar os requisitos de teste; ou seja, trata o programa como uma caixa preta, onde os detalhes de implementação são ignorados. O teste estrutural baseia-se na estrutura interna da implementação e o objetivo é caracterizar um conjunto de componentes de programa que devem ser exercitados.

No contexto desta dissertação, são explorados os aspectos referentes ao teste estrutural de unidades de programa. O teste estrutural é predominantemente aplicado em pequenos módulos de software, freqüentemente mencionados como unidades de programa. Os componentes elementares de programa, derivados dessa técnica, que devem ser exercitados, dependem da abordagem adotada; diferentes abordagens correspondem a diferentes tipos de componentes como, por exemplo, comandos, seqüência de comandos, laços, etc. Os componentes requeridos são caracterizados por um critério de teste e, após a aplicação de casos de teste, os componentes efetivamente executados são avaliados quanto à adequação ao critério utilizado.

Informação de fluxo de dados pode ser utilizada para derivar requisitos de teste [HEC77]; vários critérios de teste são baseados em análise de fluxo de dados [HER76,RAP82,RAP85,FRA88,MAL88]. Os critérios *Potenciais Usos* [MAL88,MAL91] estão incluídos nesta classe de critérios e requerem basicamente que as interações que envolvem definições de variáveis -- mudanças de estado do programa -- e subseqüentes potenciais referências a essas definições, sejam exercitadas.

Com o crescimento do tamanho e da complexidade dos sistemas de software, o esforço requerido para testar esses sistemas tem crescido muito além das expectativas; até 40% do custo total de desenvolvimento de software é gasto no teste de programas [PRE87]. Essa situação ensejou o

desenvolvimento de ferramentas automáticas para auxiliar na produção de testes efetivos e na análise dos resultados de testes [BER91].

O suporte automatizado ao uso de critérios de teste estrutural requer facilidades, tais como: instrumentação de programas; monitoração dos componentes efetivamente executados pelos casos de teste; e, meios para decidir se o critério foi satisfeito ou não. Adicionalmente, no contexto de teste estrutural baseado em análise de fluxo de dados, aspectos de fluxo de dados são requeridos para a caracterização e avaliação de conjuntos de casos de teste.

Um fato importante a ser considerado é que não existem ferramentas comerciais de teste estrutural de suporte à aplicação de critérios baseados em análise de fluxo de dados. As ferramentas disponíveis constituem protótipos desenvolvidos em universidades [HER76,FRA87,PRI90,MAL89,CHA91b], ou são ferramentas de domínio restrito [HOR92]. Ainda, não se tem conhecimento de ferramentas dessa natureza que apoiem o teste de programas escritos em linguagem COBOL.

A ferramenta POKE-TOOL (*POtential Uses CRiteria TOOL for program testing*) [MAL89,CHA91b], a qual automatiza a aplicação dos critérios Potenciais Usos, realiza a análise de adequação de um conjunto de casos de teste e auxilia o projeto de casos de teste. Uma característica peculiar dessa ferramenta é que ela pode ser configurada para diversas linguagens de programação. Em sua versão inicial, a ferramenta suporta o teste de programas escritos na linguagem C; Maldonado, em [MAL91], apresenta o resultado da aplicação de um "benchmark", onde é avaliado o custo de utilização dos critérios Potenciais Usos em programas escritos em linguagem C.

1.2 OBJETIVOS DA TESE

O objetivo principal desta dissertação concentra-se no apoio à aplicação de técnicas de teste baseadas em análise de fluxo de dados em programas escritos em linguagem COBOL; mais especificamente, são enfocados os critérios Potenciais Usos. Os aspectos de configuração da ferramenta POKE-TOOL são analisados para a linguagem COBOL.

A caracterização das estruturas de controle, a instrumentação do código fonte e a análise de fluxo de dados são atividades pertinentes e fundamentais à aplicação de critérios de teste baseados em análise de fluxo de dados e, consequentemente, à implementação de ferramentas; a partir dessa

necessidade, são derivados modelos que constituem um mecanismo básico de suporte ao teste de programas. Esses aspectos são aplicados à linguagem COBOL, viabilizando, portanto, a utilização de critérios em programas escritos em linguagem COBOL.

Um outro objetivo consiste em investigar a aplicabilidade de teste de unidades em programas COBOL. Para isto, é realizado um estudo da estrutura de chamadas da linguagem COBOL; tal estudo visa fornecer meios para a caracterização de unidades.

1.3 ORGANIZAÇÃO DA TESE

No Capítulo 2 são introduzidos os conceitos básicos referentes às atividades de teste estrutural, os critérios Potenciais Usos e os aspectos associados à configuração da ferramenta POKE-TOOL. Os modelos de implementação dos critérios Potenciais Usos vinculados à configuração da POKE-TOOL para a linguagem COBOL são explorados no Capítulo 3; os aspectos de implementação desses modelos são discutidos no Capítulo 4. As conclusões e trabalhos futuros são apresentados no Capítulo 5. O Apêndice A contém a descrição da Linguagem Intermediária (LI), de onde se abstrai o fluxo de controle de um Programa [CAR91]. Os Apêndices B e C apresentam tabelas descritoras da linguagem COBOL, contendo, respectivamente, uma descrição léxica e uma descrição gramatical. Um exemplo completo de aplicação da POKE-TOOL no teste de um programa escrito em linguagem COBOL é apresentado no Apêndice D.

CAPÍTULO 2

TESTE ESTRUTURAL DE PROGRAMAS BASEADO EM ANÁLISE DE FLUXO DE DADOS

Neste capítulo são apresentados os aspectos de aplicação de informação de fluxo de dados na atividade de teste estrutural de programas.

Dentre os critérios baseados em análise de fluxo de dados, são explorados os critérios Potenciais Usos que requerem, essencialmente, a execução de caminhos livres de definição a partir de cada nó *i* contendo definições de variáveis, mesmo se não existir uso dessas variáveis através desses caminhos. Se pode ocorrer um uso em um caminho -- um *potencial uso* -- esse caminho é requerido durante as atividades de teste.

Para a automação de teste estrutural de programas, é necessário, em geral, caracterizar o fluxo de execução do programa e instrumentar o código fonte para a determinação dos caminhos efetivamente executados durante a aplicação dos casos de teste. Adicionalmente, no contexto de teste baseado em análise de fluxo de dados, informações de fluxo de dados são agregadas aos aspectos de fluxo de controle do programa para a caracterização dos componentes requeridos pelo critério de teste utilizado.

O uso de critérios de teste estrutural, particularmente de critérios baseados em análise de fluxo de dados, sem um suporte de ferramenta automatizado, é limitado a programas muito simples [KOR85]. Uma ferramenta chamada POKE-TOOL foi desenvolvida para habilitar o uso dos critérios Potenciais Usos. A POKE-TOOL constitui uma ferramenta multi-linguagem e, por isto, nesta dissertação são caracterizadas as funções da ferramenta que são alvo da atividade de configuração para uma nova linguagem; essas funções são baseadas em modelos aplicados na implementação dos critérios Potenciais Usos. Esses modelos serão instanciados para a linguagem COBOL no Capítulo 3. Os passos necessários à configuração da POKE-TOOL são apresentados no final deste capítulo.

2.1 CONCEITOS BÁSICOS

Um programa pode ser decomposto em um conjunto de blocos disjuntos de comandos tal que, a execução do primeiro comando de um bloco acarreta na execução de todos os outros comandos, na ordem dada, desse bloco. A representação de um programa P como um grafo de fluxo de controle (grafo de programa) -- um grafo dirigido, com um único nó de entrada e um único nó de saída -- consiste em estabelecer uma correspondência entre nós e blocos, e em indicar possíveis fluxos de controle entre nós através dos arcos.

Seja um grafo de fluxo de controle $G = (N,E,s)$ onde N representa o conjunto de nós, E representa o conjunto de arcos, e s o nó de entrada. Um *caminho* é uma seqüência de nós $(n1,n2,...,nk)$, $k \geq 2$, tal que existe um arco do nó ni para $ni+1$ para $i = 1,2,...,k-1$. Um caminho é um *caminho simples* se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos; se todos os nós são distintos diz-se que esse é um *caminho livre de laço*. Um *caminho completo* é um caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída.

A abstração do fluxo de execução de um programa é freqüentemente utilizada na derivação de requisitos de teste estrutural; vários critérios de teste baseiam-se unicamente no fluxo de controle de programas, onde os mais conhecidos são: *todos os nós*, *todos os ramos* e *todos os caminhos*. O critério todos os nós requer que todos os comandos sejam executados pelo menos uma vez. O critério todos os ramos requer que toda transferência de controle seja exercitada pelo menos uma vez. O critério todos os caminhos requer que todos os caminhos possíveis do programa sejam executados [HOW75].

Uma outra classe de critérios utiliza informação de fluxo de dados para derivar requisitos de teste; tais critérios são conhecidos como critérios de teste baseados em análise de fluxo de dados [HER76,RAP82,RAP85, FRA88,MAL88]. Os tipos de ocorrências de variáveis em um programa são definidos por um *modelo de fluxo de dados* que é usado nessa classe de critérios; uma ocorrência de variável caracteriza uma *definição*, um *uso* ou uma *indefinição*. Em um programa, em geral, uma ocorrência de variável é uma definição se seu valor é alterado em um comando de atribuição, de entrada ou de chamada de procedimento. A ocorrência de uma variável é um uso quando a referência a essa variável não a estiver definindo; um uso computacional (*c-uso*) afeta diretamente a computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado; um uso predicativo (*p-uso*) afeta diretamente o fluxo de controle de um programa. Uma variável

está indefinida quando, ou não se tem acesso ao seu valor, ou sua localização deixa de estar amarrada na memória.

Os tipos de ocorrência de variáveis são utilizados na caracterização de elementos do fluxo de execução de um programa. Um caminho $(i,n_1,...,n_m,j)$, $m \geq 0$, que não contenha definição de variável nos nós $n_1,...,n_m$ é chamado *caminho livre de definição* com respeito a (*c.r.a*) x do nó i ao nó j e do nó i ao arco (n_m,j). Diz-se que um nó i possui uma *definição global* de uma variável x se ocorre uma definição de x no nó i e existe um caminho livre de definição de i para algum nó ou para algum arco que contém um uso da variável x. Um caminho $(n_1,n_2,...,n_j,n_k)$ é um *du-caminho* c.r.a variável x se n_1 tiver uma definição global de x e: (1) ou n_k tem um c-uso de x e $(n_1,n_2,...,n_j,n_k)$ é um caminho simples livre de definição c.r.a x; ou (2) (n_j,n_k) tem um p-uso de x e $(n_1,n_2,...,n_j,n_k)$ é um caminho livre de definição c.r.a x e $n_1,n_2,...,n_j$ é um caminho livre de laço.

Os critérios baseados em análise de fluxo de dados requerem que as interações que envolvem definições de variáveis em um programa e subsequentes referências a essas definições sejam testadas. Esses critérios baseiam-se, portanto, para a derivação de casos de teste, nas associações entre a definição de uma variável e os seus possíveis usos subsequentes; com exceção dos critérios Potenciais Usos, que serão discutidos na próxima seção, eles requerem a ocorrência explícita de um uso de uma definição de variável para caracterizar e requerer essa interação. Rapps e Weyuker [RAP82,RAP85] introduziram uma família de critérios de Fluxo de Dados, onde os três critérios centrais são: *todas as definições*, *todos os usos* e *todos-du-caminhos*. O critério todas as definições requer que cada definição de variável seja exercitada pelo menos por um p-uso ou um c-uso. O critério todos os usos requer que todas as associações entre uma definição de uma variável e subsequentes c-usos e p-usos dessa variável sejam exercitadas pelos casos de teste. O critério todos-du-caminhos requer que toda associação entre uma definição de variável e subsequentes c-usos e p-usos dessa variável seja exercitada por todos os possíveis du-caminhos c.r.a essa definição de variável.

2.2 CRITÉRIOS POTENCIAIS USOS

Maldonado, Chaim e Jino introduziram a família de critérios Potenciais Usos básicos -- *todos-potenciais-usos*, *todos-potenciais-usos/du* e *todos-potenciais-du-caminhos* -- que requerem basicamente que caminhos livres de definição, em relação a qualquer nó i que possua definição de variável e qualquer variável x definida em i, sejam executados independente de ocorrer uso dessa variável nesses caminhos [MAL88]. Neste sentido, pode-se verificar, por exemplo, que o valor de x não foi alterado nesses caminhos (possivelmente devido a efeitos colaterais) ganhando-se, desta forma, maior confiança que a computação correta é realizada; isto está de acordo com uma filosofia discutida por Myers [MYE79]: um erro está claramente presente se um programa não faz o que supõe-se que ele faça, mas erros também estão presentes se um programa faz o que supõe-se que não faça. Ainda, a aplicação dos critérios Potenciais Usos torna mais fácil a detecção de erros causados por dependências de fluxo de dados ausentes [POD90] originadas, por exemplo, por uso ausente de variáveis.

O conceito *potencial uso* introduz algumas particularidades em relação à maioria dos critérios baseados em análise de fluxo de dados. Os elementos a serem requeridos são caracterizados, como mencionado acima, independentemente da ocorrência explícita de uma referência a uma determinada definição; se um uso dessa definição pode existir -- um *potencial uso* -- a potencial associação caracterizada entre a definição e o potencial uso é requerida. Neste sentido, a idéia de du-caminho é estendida para *potencial-du-caminho*.

. **Critério Todos-potenciais-usos**:

Um conjunto de caminhos Π satisfaz o critério todos-potenciais-usos se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui pelo menos um caminho livre de definição c.r.a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i.

. **Critério Todos-potenciais-usos/du:**

Um conjunto de caminhos Π satisfaz o critério todos-potenciais-usos/du se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui pelo menos um potencial-du-caminho c.r.a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i.

. **Critério Todos-potenciais-du-caminhos**:

Um conjunto de caminhos Π satisfaz o critério todos-potenciais-du-caminhos se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui todos potenciais-du-caminhos c.r.a x em relação ao nó i.

Os componentes requeridos pelos critérios Potenciais Usos são caracterizados como *associações* entre definições de variáveis e potenciais usos dessas variáveis, onde uma associação é caracterizada pela existência de pelo menos um caminho livre de definição c.r.a essas variáveis. O que diferencia cada critério consiste no conjunto de caminhos requeridos, onde o critério mais fraco requer pelo menos um caminho livre de definição para cobrir uma associação, e o critério mais forte requer todos potenciais-du-caminhos para cobrir essa associação.

2.3 AUTOMAÇÃO DE TESTE ESTRUTURAL DE PROGRAMAS

As ferramentas de teste estrutural de programas fazem, em quase sua totalidade, análise de cobertura segundo algum critério de teste selecionado, de um conjunto de casos de teste. Apesar de não auxiliarem diretamente na determinação das entradas de um programa, necessárias para a execução de caminhos específicos, a maioria das ferramentas de teste apresenta, ao usuário, quais os requisitos de teste exigidos para que os critérios sejam satisfeitos. Assim, de certa forma, orientam e auxiliam os usuários na elaboração dos casos de teste.

Chaim [CHA91b] descreve várias ferramentas de apoio ao teste estrutural de programas. A seguir, citamos algumas ferramentas de teste estrutural de software.

RXVP80 [DEU82] é uma ferramenta comercial, distribuída pela General Research Corporation, Santa Barbara, California, EUA; basicamente, realiza a análise de cobertura através do critério todos os ramos em programas escritos em linguagem FORTRAN. Além do suporte ao teste dinâmico, essa ferramenta fornece ainda: análise estática do código fonte com a geração do grafo de chamada dos módulos (unidades) constituintes do sistemas em teste; geração do grafo de fluxo de controle dos módulos; referências cruzadas entre código fonte, variáveis e módulos; entre outras.

A ferramenta TCAT (*Test-Coverage Analysis Tool*) [REI90] realiza o teste de unidades segundo o critério todos os ramos; produz estatísticas para o último caso de teste ou para todos os casos de teste. Existem versões disponíveis para Ada, C, COBOL, FORTRAN e Pascal. É uma ferramenta comercial fornecida por Software Research Corporation, San Francisco, Califórnia, EUA.

A ferramenta de Herman [HER76] suporta a aplicação do primeiro critério baseado em análise fluxo de dados; tal critério é semelhante ao critério *todos-c-usos* [RAP82,RAP85] proposto posteriormente. A ferramenta realiza a análise cobertura desse critério para programas escritos em linguagem FORTRAN.

ASSET (*A System to Select and Evaluate Tests*) [FRA87] suporta basicamente um conjunto de critérios baseados em análise fluxo de dados, desenvolvido por Rapps e Weyuker [RAP82,RAP85], em programas escritos em linguagem Pascal; foi desenvolvida na New York University.

PROTESTE [PRI90] tem como objetivo um ambiente completo para suporte ao teste estrutural de programas, incluindo tanto critérios baseados unicamente no fluxo de controle (por exemplo, critérios todos os nós e todos os ramos) como critérios baseados em análise de fluxo de dados (por exemplo, critérios de Rapps e Weyuker e Potenciais Usos); suporta o teste de programas escritos em linguagem Pascal. PROTESTE é um protótipo desenvolvido na Universidade Federal do Rio Grande do Sul.

ATAC (*Automatic Test Analysis for C*) é uma ferramenta desenvolvida no Bell Communications Research, e suporta o teste de unidades escritas em linguagem C; realiza de análise cobertura através dos critérios todos os nós, todos os ramos e todos os usos. Ela informa os elementos requeridos pelo critério mas não executados na aplicação dos casos de teste. É uma ferramenta de domínio público, mas de uso restrito a universidades [HOR92].

Um fato importante a ser considerado é que, até o momento, não existem ferramentas comerciais de teste estrutural de apoio à aplicação dos critérios baseados em análise de fluxo de dados. Ainda, os protótipos existentes não suportam o teste de programas escritos em linguagem COBOL. A ferramenta POKE-TOOL, discutida na seção seguinte, está incluída nessa classe de ferramentas, e auxilia o uso prático dos critérios Potenciais Usos.

2.4 POKE-TOOL - UMA FERRAMENTA DE APOIO À APLICAÇÃO DOS CRITÉRIOS POTENCIAIS USOS

A ferramenta POKE-TOOL (*Potential Uses CRiteria TOOL for program testing*) [MAL89,CHA91b], para uma dada unidade de programa, seleciona os elementos que satisfazem os critérios Potenciais Usos e avalia os caminhos percorridos pelos casos de teste, informando a cobertura desses caminhos. Adicionalmente, é fornecido o conjunto de elementos requeridos não executados, auxiliando, dessa forma, o projeto de casos de teste.

A POKE-TOOL, cuja arquitetura está ilustrada na Figura 2.1, é constituída de nove funções descritas abaixo:

GRAFO DE FLUXO DE CONTROLE:

Esta função produz o grafo de fluxo de controle do programa fonte a ser testado. Para a produção do grafo de fluxo de controle para várias linguagens utiliza-se uma tabela descritora da sintaxe da linguagem de implementação que é utilizada na análise sintática do programa fonte; é gerada uma versão em linguagem intermediária (LI) [CAR91] do programa fonte, de onde se extrai o fluxo de controle. Note-se que a incorporação de novas linguagens far-se-á unicamente com inserção das tabelas descritoras que permitem à ferramenta produzir a linguagem intermediária a partir do código fonte. A correspondência entre os comandos da linguagem fonte e os comandos da linguagem intermediária faz parte do processo de configuração e é de responsabilidade do usuário configurador.

CÁLCULO DOS ARCOS PRIMITIVOS:

Calcula os arcos primitivos [CHU87] do grafo de fluxo de controle. O conjunto de arcos primitivos servirá de base para a instrumentação do programa fonte e para a construção dos autômatos utilizados na avaliação de adequação de conjuntos de casos de teste.

EXTENSÃO DO GRAFO DE FLUXO DE CONTROLE:

Associa a cada nó *i* do grafo de fluxo de controle o conjunto de variáveis definidas no nó *i*, produzindo um grafo denominado *grafo def*, de acordo com o modelo de fluxo de dados pré-estabelecido; este modelo será apresentado na Seção 2.5.3.

INTERFACE GRÁFICA:

Apresenta os grafos de fluxo de controle para o usuário.

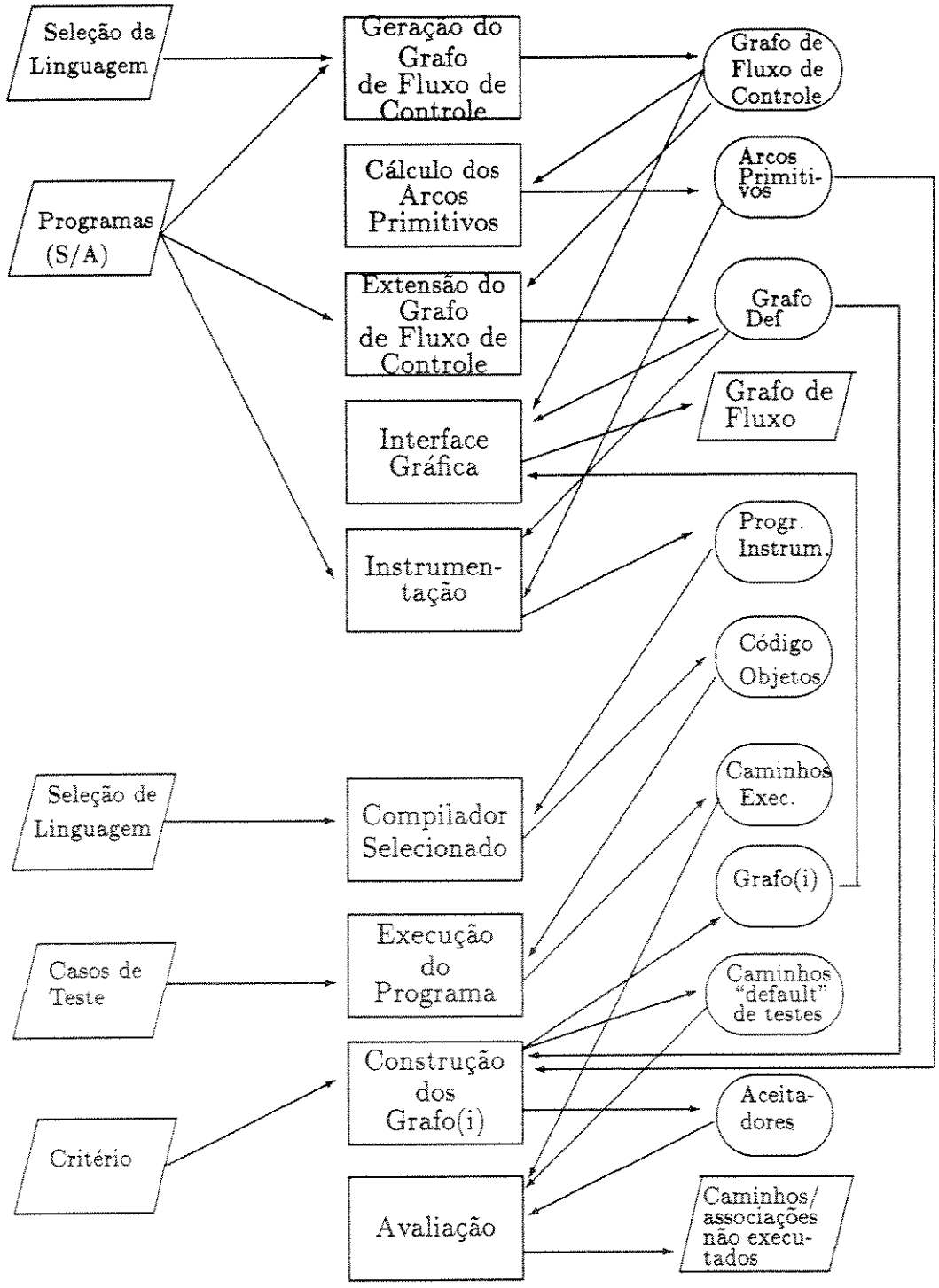


Figura 2.1: Arquitetura da Ferramenta POKE-TOOL

INSTRUMENTAÇÃO:

Insere comandos de escrita (pontas de prova) no programa fonte para cada um dos nós da unidade em teste, gerando uma nova versão do programa -- versão instrumentada -- que produz um "trace" da execução dos casos de teste.

COMPILADOR SELECIONADO:

Consiste de um compilador da linguagem fonte na qual o programa em teste foi implementado.

EXECUÇÃO DO PROGRAMA:

Controla a execução do programa em teste -- unidade em teste -- produzindo um conjunto dos caminhos executados pelos casos de teste fornecidos. Adicionalmente, produz um registro de pares <entrada,saída> associados aos respectivos caminhos executados. Os caminhos executados são descritos através de seqüências de números dos nós do grafo de fluxo de controle do programa fonte.

CONSTRUÇÃO DOS GRAFO(i):

Com base no grafo def e no conjunto de arcos primitivos, esta função constrói os grafo(i) [MAL88]. Adicionalmente, fornece um conjunto de caminhos e associações requeridos para satisfazer os critérios Potenciais Usos. Ainda são fornecidos os descritores de caminhos e associações, em termos dos arcos primitivos, a serem utilizados na avaliação de um conjunto de casos de teste qualquer.

AVALIAÇÃO:

Esta função verifica se o conjunto de caminhos ouassociações executados satisfaz o critério selecionado. Em caso negativo, produz uma relação de caminhos (associações) requeridos pelo critério e não executados e uma medida percentual da cobertura provida pelo conjunto de casos de teste, ou seja, uma relação entre os caminhos (associações) executados e o número de caminhos (associações) requeridos.

De forma resumida, pode-se dizer que a partir do programa fonte, a POKE-TOOL determina o grafo de fluxo de controle. A seguir, este grafo é estendido incorporando-se informações de fluxo de dados, obtendo-se o grafo def; o conjunto de arcos primitivos é calculado, obtendo-se o grafo com redução de herdeiros para fluxo de dados [MAL91]. Estes arcos primitivos são utilizados para construir descritores (expressões regulares) dos caminhos (associações) requeridos. A instrumentação auxilia na determinação dos caminhos efetivamente executados pelo conjunto de casos de teste fornecido. Os descritores são utilizados para verificar se o critério selecionado foi

satisfeito; esta verificação dá-se pela implementação de aceitadores de expressões regulares. A partir do grafo def e do conjunto de arcos primitivos constroem-se os grafo(i), caracterizando-se os arcos primitivos em cada um desses grafo(i). Em seguida, os descritores dos caminhos (associações) requeridos são elaborados. Na hipótese da POKE-TOOL ser utilizada na avaliação da adequação de um conjunto de casos de teste, são determinados os caminhos (associações) efetivamente executados e verifica-se se os aceitadores correspondentes aos descritores dos caminhos (associações) requeridos estão no estado final (o critério foi satisfeito); caso contrário, é fornecida ao usuário uma lista de caminhos requeridos pelo critério e não executados pelo conjunto de casos de teste. No caso da POKE-TOOL ser utilizada para auxiliar na geração de casos de teste, o conjunto "default" de caminhos requeridos pelo critério (obtido durante a construção dos grafo(i)) é fornecido.

Dentre as funções implementadas pela ferramenta POKE-TOOL, distinguem-se as funções dependentes do código fonte, permitindo que as tarefas correspondentes sejam isoladas das tarefas que realizam funções independentes da linguagem; tais funções são: Grafo de Fluxo de Controle, Extensão do Grafo de Fluxo de Controle e Instrumentação.

2.5 MODELOS DE IMPLEMENTAÇÃO DOS CRITÉRIOS POTENCIAIS USOS

Na implementação dos critérios potenciais usos foram utilizados modelos que descrevem as funções que compõem a ferramenta POKE-TOOL. Nesta seção apresentam-se os modelos referentes às funções dependentes da linguagem em que está escrita a unidade em teste, a saber: Modelo de Fluxo de Controle, Modelo de Instrumentação e Modelo de Fluxo de Dados. Os conceitos apresentados representam uma síntese dos modelos apresentados em [CHA91b,MAL91].

2.5.1 MODELO DE FLUXO DE CONTROLE

No modelo de fluxo de Controle adotado, um programa P é representado por um grafo dirigido $G(N,A,s)$, onde os blocos disjuntos de comandos correspondentes da LI são associados aos nós $n \in N$ e os possíveis fluxos de controle entre os

blocos associados aos arcos $e \in A$; o nó s representa o nó de entrada. Esse grafo é usualmente conhecido como grafo de fluxo de controle ou grafo de programa.

Os comandos que compõem a linguagem intermediária (LI) implementam construções básicas de fluxo de controle como, por exemplo, seqüência, seleção, iteração e desvio; a descrição precisa da LI é mostrada no Apêndice A. As Figuras 2.2, 2.3, 2.4 e 2.5 representam as construções básicas de fluxo de controle adotadas para os comandos da linguagem LI. O grafo de fluxo de controle de uma unidade é obtido pela simples concatenação dessas construções básicas. Observe-se que o grafo de fluxo de controle da unidade em teste é independente da linguagem de implementação da unidade; no entanto, o usuário configurador da POKE-TOOL deverá levar esse modelo em consideração pois, de certa forma, ele reflete os aspectos semânticos da LI e terá forte influência na instrumentação da unidade em teste.

A representação dos comandos de desvios incondicionais introduz modificações profundas no fluxo de controle e, conseqüentemente, no modelo e na implementação. Por exemplo, o comando de desvio incondicional irrestrito *goto* encerra a caracterização de um bloco, sendo que existirá um arco do bloco que contém o comando *goto* para o bloco que contém o rótulo associado a ele. Maiores considerações acerca da presença de comandos de desvio em estruturas de seleção e iteração podem ser encontradas em [CHA91b,MAL91], pois não são utilizadas na aplicação do Modelo de Fluxo de Controle para a linguagem COBOL.

2.5.2 MODELO DE INSTRUMENTAÇÃO

A instrumentação fornece facilidades para a análise posterior à execução dos casos de teste, como por exemplo, a análise de adequação de um dado conjunto de casos de teste. Para tanto, este módulo modifica, com inserção de código fonte, a própria unidade em teste, gerando uma nova versão do programa, usualmente denominada versão instrumentada; no caso da POKE-TOOL, para a linguagem COBOL, essa versão é denominada TESTEPROG.CBL. O arquivo TESTEPROG.CBL, do Apêndice D, ilustra a unidade instrumentada para o exemplo apresentado.

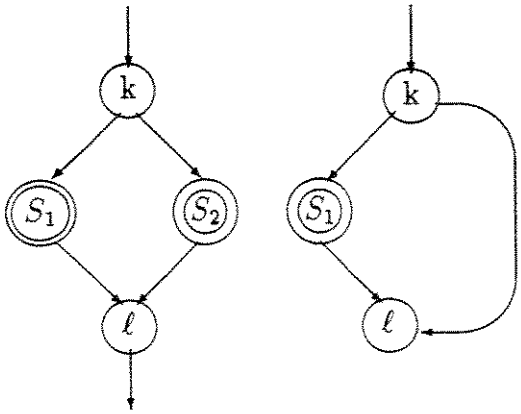
A instrumentação consiste essencialmente em inserir pontas de prova nos blocos de comandos correspondentes a cada nó do grafo de programa da unidade em teste, possibilitando a identificação do caminho executado pelo caso de teste fornecido. Uma ponta de prova consiste basicamente em um

comando de escrita do número do nó em um arquivo (arquivo PATH.TES, na versão atual da POKE-TOOL), produzindo um "trace" do caso de teste fornecido; esta informação é imprescindível para a função de avaliação da POKE-TOOL.

Obviamente, a instrumentação deve ser tal que reflita a semântica dos comandos da LI e ao mesmo tempo viabilize a correta avaliação dos comandos efetivamente executados; observa-se também que a instrumentação está fortemente restrita ao modelo de fluxo de controle adotado. As figuras 2.2, 2.3, 2.4 e 2.5 ilustram a instrumentação associada aos comandos da LI; considerações acerca da presença de comandos de desvio em estruturas de seleção e iteração podem ser encontradas em [CHA91b,MAL91], pois não são utilizadas na aplicação do Modelo de Instrumentação para a linguagem COBOL. Um fato importante a ser observado é que nenhuma modificação é introduzida nos comandos originais do programa fonte, sendo que somente pontas de prova são inseridas. Ainda, são também inseridos comandos para a abertura e fechamento de um arquivo que conterá o caminho executado (arquivo PATH.TES), assim como, comandos de declaração e de iniciação das variáveis necessárias para a correta implementação desta função.

Comandos de seleção

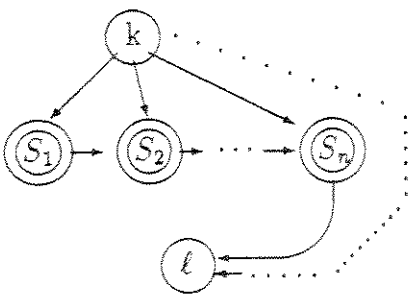
< if > ::= < if-atm >< cond-atm>< statement$_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statment$_2$ >



S_1 – representa o subgrafo correspondente ao < statement$_1$ >
S_2 – representa o subgrafo correspondente ao statement$_2$.
k – associado a <cond-atm>.

instrumentação: Sejam i e j os nós de entrada dos subgrafos S_1 e S_2, respectivamente. No início dos blocos k, i, j e l são inseridas pontas de prova que caracterizam a execução destes blocos, ou seja, pontas de prova k, i, j e l.

< CASE >::= < case-atm >< case-cond-atm > { { (< rotc-atm > | < rotd-atm >) { < statement > } } }



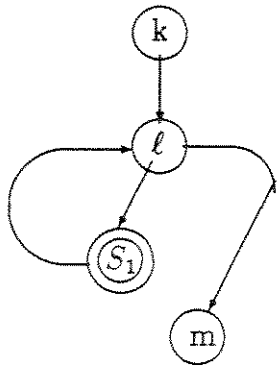
S_i i=1, ..., n representa os subgrafos correspondentes ao < statement$_i$ > relativos a cada uma das alternativas da seleção múltipla
k – associado a <case-cond-atm>.

instrumentação:
Sejam i_1, i_2,..., i_n os nós de entrada dos subgrafos S_1, S_2,..., S_n, respectivamente. No início dos blocos k, i_1, i_2,..., i_n, l são inseridas pontas de prova que caracterizam a execução destes blocos, ou seja, pontas de prova k, i_1, i_2,..., i_n e l.

Figura 2.2: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos de Seleção.

Comandos de Iteração

< while > ::= < while-atm >< cond-while-atm>< statement>

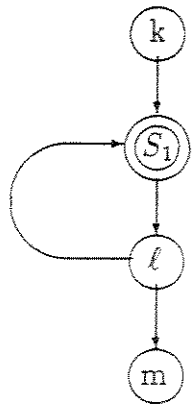


S_1 – representa o subgrafo correspondente a < *statement* >, ou seja, ao corpo do laço

ℓ – associado a < cond-while-atm>

instrumentação: seja i o nó de entrada do subgrafo S_1. No nó k é inserida a ponta de prova k; no nó i as pontas de prova l e i e no nó m, as pontas de prova l e m.

<repeat-until > ::= < repeat-atm >< statement >< until-atm>< cond-until-atm>



S_1 – representa o subgrafo correspondente a < statement >

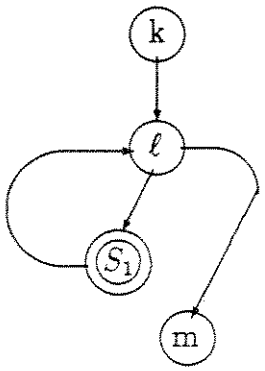
ℓ – associado a < cond-until-atm>

instrumentação: seja i o nó de entrada do subgrafo S_1 e j o nó saída de S_1. No início dos nós k, i e m são inseridas as pontas de provas k, i e m. Adicionalmente, no início do nó j, é inserida a ponta de prova l.

Figura 2.3: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos de Iteração -- "while" e "repeat".

Comandos de Iteração

< for > ::= < for-atm >< S_1 >< cond-for-atm>< S_2 >< statement>



(S_1) – representa o subgrafo correspondente a < *statement* >, ou seja, ao corpo do laço

(k) – associado aos comandos de iniciação da variável de controle do "for", ou seja, aos comandos representados por < S_1 >.

(j) – associado aos comandos que alteram a variável de controle, ou seja, aos comandos representados por < S_2 >, onde o nó j é o nó saída do subgrafo S_1.

(ℓ) – associado a < cond-for-atm>

instrumentação:

sejam i e j os nós de entrada e saída, respectivamente, do subgrafo < S_1 >.
No início do nó k, é inserida a ponta de prova k; no início do nó i as pontas de provas l e i e no início do nó m, as pontas de provas l e m.
No nó j é também inserida a ponta de prova j.

Figura 2.4: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comando de Iteração "for".

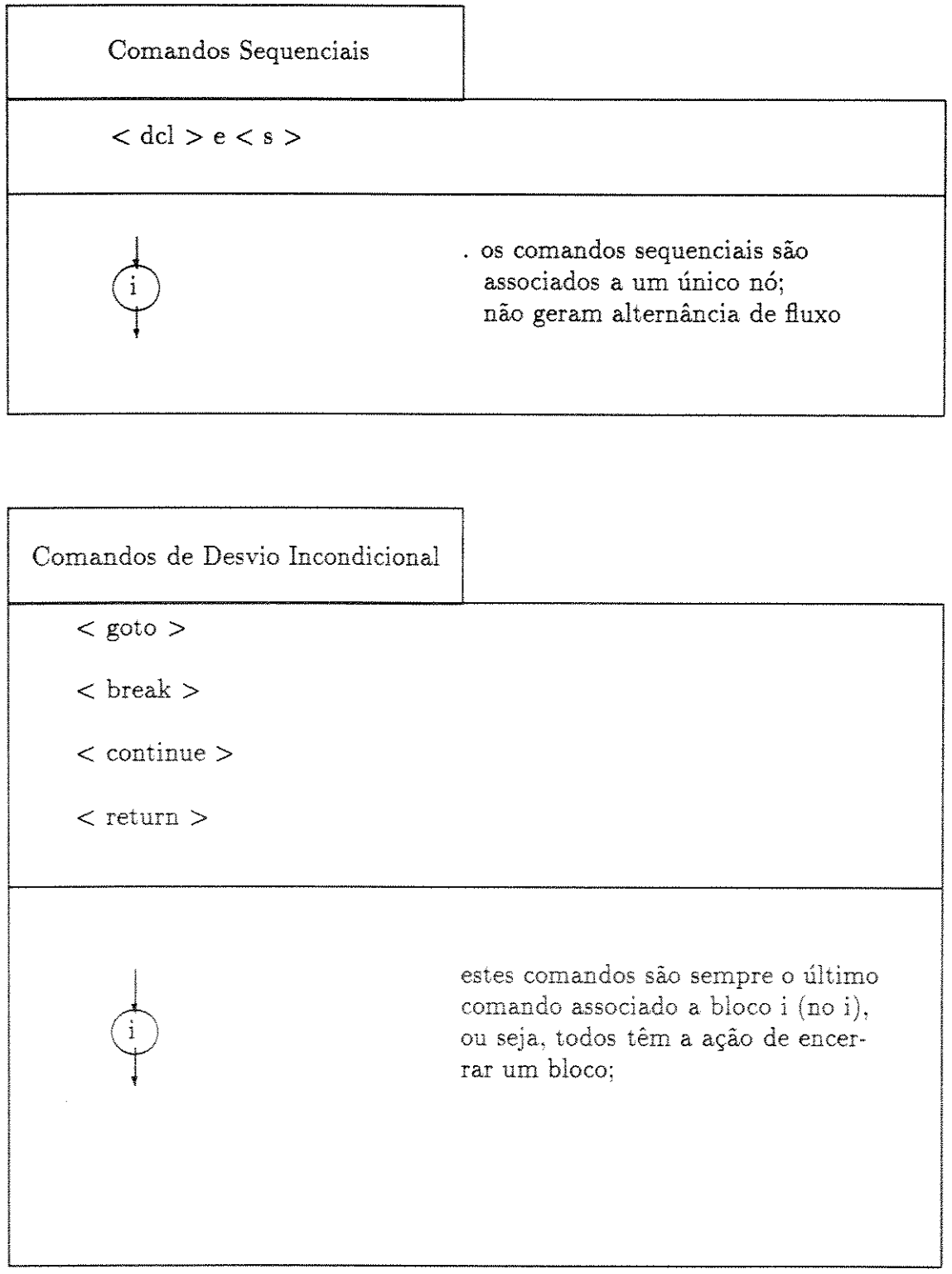


Figura 2.5: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos Seqüenciais e de Desvio Incondicional.

2.5.3 MODELO DE FLUXO DE DADOS

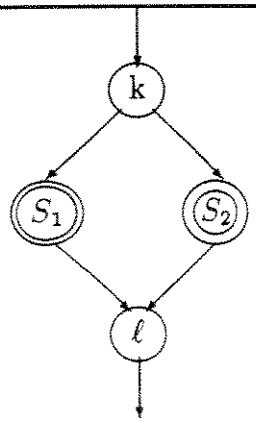
No contexto de teste de software, a análise de fluxo de dados usualmente é utilizada para estender o grafo de programa pela associação de tipos de ocorrências de variáveis aos elementos deste grafo que, posteriormente, é utilizado para determinação dos caminhos e associações a serem requeridos. No caso dos critérios Potenciais Usos é suficiente associar a cada nó *i* do grafo de programa o conjunto de variáveis definidas no bloco de comandos correspondente; esta extensão denomina-se *grafo def*.

Para a correta geração do grafo def é necessário precisar o que considera-se uma definição de variável; tal consideração foi discutida na Seção 2.1. Adicionalmente, a passagem de valores entre procedimentos através da passagem de parâmetros pode ser feita por: *valor*, *referência* ou *nome* [GHE87]. Se a variável é passada por referência ou por nome considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são distinguidas das demais e são ditas *definidas por referência*; esta distinção é utilizada na geração dos grafo(i) [CHA91b], ou seja, na determinação dos caminhos livres de definição para as variáveis definidas em *i*.

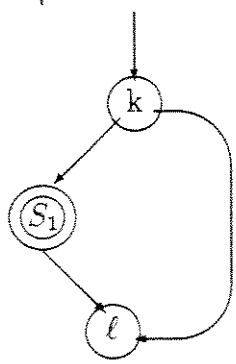
No modelo de fluxo de dados adotado, considera-se que no nó de entrada ocorre uma definição dos parâmetros e das variáveis globais que ocorrem na unidade em teste. As figuras 2.6, 2.7 e 2.8 sintetizam os conceitos e hipóteses básicos para a determinação do grafo def a partir do modelo de fluxo de controle dos principais comandos da LI. Observe-se que para os comandos seqüenciais a extensão é obvia, uma vez que esses comandos estão sempre associados a um único nó.

Comandos de Seleção

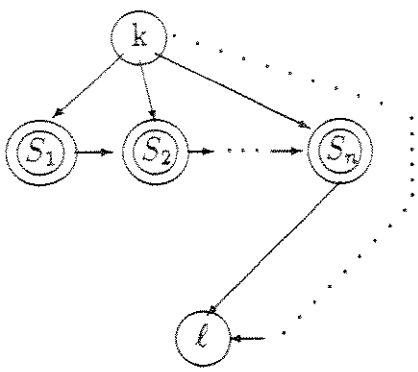
< if > ::= < if-atm >< cond-atm>< statement$_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statement$_2$ >



Ao nó k são atribuídos o conjunto de variáveis definidas no bloco de comandos associado a este nó e o conjunto de variáveis definidas na condição associada a <cond-atm>. Ao nó l é atribuído o conjunto de variáveis definidas no bloco de comandos associado a este nó. A extensão dos subgrafos S_1 e S_2 depende dos comandos associados a < statement$_1$ > e < statement$_2$ >, respectivamente.



<case>::=<case-atm>< case-cond-atm>{{ [<rotc-atm>| <rotd-atm>] {<statement>} }}

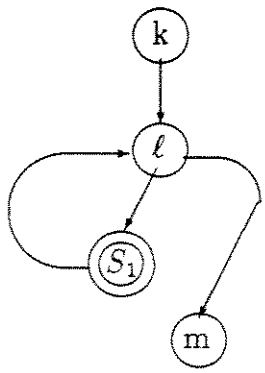


Ao nó k são atribuídos o conjunto de variáveis definidas no bloco de comandos associados a este nó e o conjunto de variáveis definidas na condição do case associada a < case-cond-atm >. Ao nó l é atribuído o conjunto de variáveis definidas no bloco de comandos associado a a este nó. A extensão dos subgrafos S_n depende dos comandos associados a cada uma das alternativas da seleção múltipla.

Figura 2.6: Diretrizes para a Expansão do Grafo de Programa para a Obtenção do Grafo Def: Comandos de Seleção.

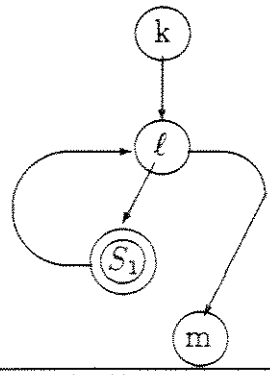
Comandos de iteração

< while > ::≡ < while-atm >< cond-while-atm>< statement>



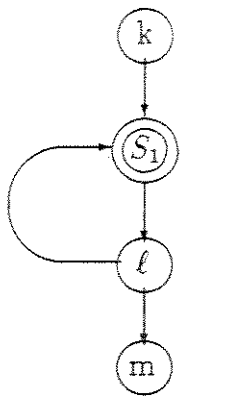
Ao nó k é associada nenhuma definição de variável devido ao comando < while >, somente devido aos demais comandos eventualmente associados ao nó k. Ao nó l é associado conjunto de variáveis definidas na condição associada a < cond-while-atm >. Ao nó m somente é atribuído o conjunto de variáveis definidas devido a outros comandos associados a este nó. A extensão do subgrafo S_1 depende dos comandos do corpo do laço, ou seja, associados a < statement >.

< for > ::≡ < for-atm >< S_1 ><cond-for-atm>< S_2 >< statement>



Seja x o nó de saída de S_1. Ao nó k é associado o conjunto de variáveis definidas pelos comandos representados por < S_1 >. Aos nós l e m idem ao comando < while >. Ao nó x é atribuído o conjunto de variáveis definidas pelos comandos representados por < S_2 > e, se for o caso, o conjunto de variáveis definidas por outros comandos associados ao nó x. A extensão do subgrafo S_1 depende dos comandos do corpo do laço associados a <statement>.

<repeat-until> ::≡ <repeat-until><statement>< until-atm><cond-until-atm>



Ao nó k não é associada nenhuma definição de variável devido ao comando <repeat-until >, somente devido aos demais comandos eventualmente associados ao nó k. Ao nó l é associado o conjunto de variáveis definidas na condição associada a <cond-until-atm>. Ao nó m, somente é atribuído o conjunto de variáveis definidas devido a outros comandos eventualmente associados a este nó. A extensão do subgrafo S_1 depende dos comandos do corpo do laço, ou seja, associados a < statement >.

Figura 2.7: Diretrizes para a Expansão do Grafo de Programa para a Obtenção do Grafo Def: Comandos de Iteração.

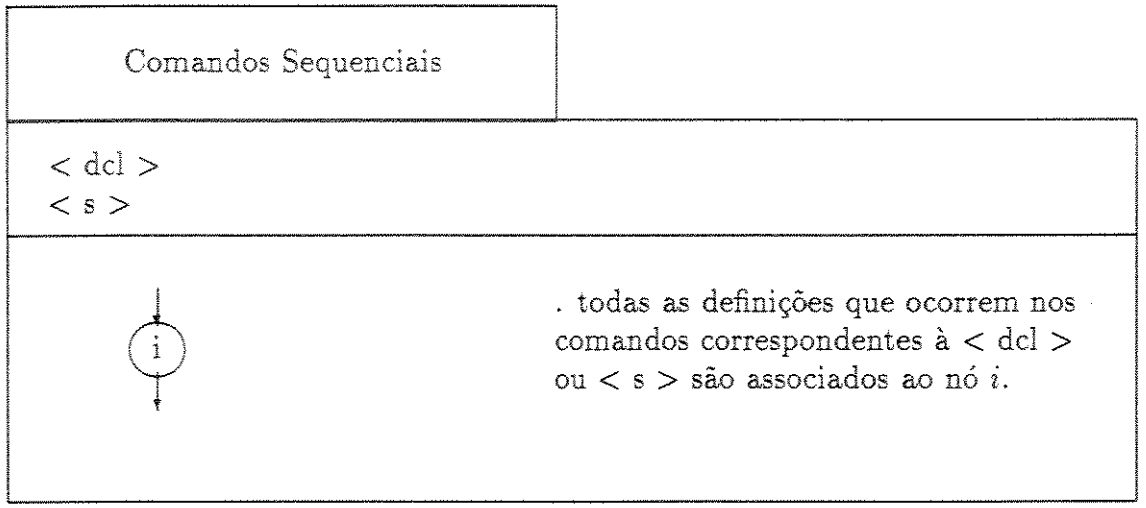


Figura 2.8: Diretrizes para a Expansão do Grafo de Programa para a Obtenção do Grafo Def: Comandos Seqüenciais.

2.6 ASPECTOS DE CONFIGURAÇÃO DA FERRAMENTA POKE-TOOL

A arquitetura da POKE-TOOL estabeleceu uma distinção entre as tarefas dependentes do código fonte das demais, permitindo que tais tarefas sejam isoladas em módulos distintos dos módulos que realizam tarefas independentes da linguagem [CHA91b].

As funções *Extensão do Grafo de Fluxo de Controle* e *Instrumentação* são implementadas pelo módulo **pokernel** da POKE-TOOL. Assim, esse módulo determina quais variáveis são definidas em um dado nó e, insere os comandos para a definição de pontas de prova, declaração de variáveis auxiliares e comandos para a manipulação de arquivos.

A função *Grafo de Fluxo de Controle* é subdividida em dois módulos: o primeiro -- o módulo **li** -- realiza o mapeamento da unidade em teste para a unidade em teste em LI; o segundo -- o módulo **chanomat** -- realiza a geração do grafo de fluxo de controle a partir da unidade em LI. Aparentemente essa função seria extremamente dependente da linguagem de programação da unidade em teste; entretanto, essa dependência se restringe ao mapeamento para a LI, pois a geração do grafo de fluxo de controle é genérica.

O módulo **li** da POKE-TOOL é dependente da linguagem fonte, pois é ele quem faz a tradução da unidade em teste para a unidade em teste em LI. Na configuração da POKE-TOOL para a linguagem C, foi utilizado o módulo gerador da versão em LI da unidade em teste da ferramenta *GFC* [CAR91]. No entanto, esse módulo gerador não realiza a análise dos "include files", prejudicando o mapeamento entre a linguagem C e a linguagem LI como, por exemplo, a caracterização dos tipos de dados definidos pelo usuário que são freqüentemente declarados nos "include files". Leitão [LEI91] reconfigurou o módulo **li** da POKE-TOOL para a linguagem C, visando corrigir tal deficiência; nessa configuração foi utilizado o mesmo enfoque adotado na configuração do módulo **pokernel**: autômatos finitos na análise léxica e gramáticas livres de contexto na análise sintática, segundo [SET81] e [WIR76], respectivamente.

A seguir, apresentamos uma relação das tarefas a serem realizadas por um usuário configurador da POKE-TOOL para obter uma configuração da ferramenta para uma nova linguagem [CHA91a]. Os passos 1, 2 e 4 são aplicados tanto para a configuração do módulo **li**, quanto para a configuração do módulo **pokernel**; o passo 3 está associado somente ao módulo **pokernel**.

1. Configurar o analisador léxico da POKE-TOOL. Essa tarefa demanda que o usuário conheça bem os aspectos léxicos da linguagem alvo e está dividida

em algumas subtarefas:

1.1. Desenvolver um autômato finito [SET81] que realize a análise léxica da linguagem, e especificar a tabela de palavras reservadas da linguagem.

1.2. Identificar as ações semânticas associadas às transições do autômato finito.

1.3. Transcrever esse autômato para a notação aceita pelo analisador genérico da POKE-TOOL.

1.4. Complementar o analisador léxico através de rotinas ("ações semânticas") que realizem a separação dos átomos da linguagem fonte, colocando-os nas estruturas de dados da POKE-TOOL utilizadas para armazená-los.

1.5. Depurar o analisador léxico conjuntamente com as ações semânticas desenvolvidas.

2. Configurar o analisador sintático da POKE-TOOL. Essa tarefa demanda que o usuário conheça bem a sintaxe e a semântica da linguagem alvo e está dividida em algumas subtarefas:

2.1. Descrever a gramática da linguagem fonte na forma de *grafos sintáticos* [WIR76].

2.2. Identificar os pontos onde devem ser ativadas as rotinas semânticas nos grafos sintáticos.

2.3. Transcrever os grafos sintáticos na notação aceita pela POKE-TOOL.

2.4. Complementar o analisador sintático com rotinas semânticas; no caso do módulo **li**, essas rotinas identificam as eqüivalências entre a linguagem alvo e a LI e geram os átomos da LI que irão compôr a versão em linguagem intermediária da unidade em teste; no caso do módulo **pokernel**, essas rotinas realizam a identificação das variáveis definida, gerando, portanto, o grafo def.

2.5. Depurar o analisador sintático conjuntamente com as rotinas desenvolvidas.

3. Ajustar os procedimentos que inserem código fonte na nova versão da unidade em teste com as instruções da nova linguagem de programação aceita pela POKE-TOOL.

4. Compilar e ligar os arquivos que constituem os analisadores léxico e sintático genéricos e as ações e rotinas semânticas com os demais arquivos que constituem a POKE-TOOL; no caso do módulo **pokernel**, também são agregados os procedimentos de inserção de código fonte.

2.7 CONSIDERAÇÕES FINAIS

Foram caracterizados os aspectos de aplicação de informação de fluxo de dados no teste estrutural de programas. O suporte automatizado de critérios baseados em análise de fluxo de dados limita-se, até o momento, a protótipos desenvolvidos em universidades, ou a ferramentas de uso restrito. Ainda, não existem ferramentas disponíveis para a aplicação desses critérios em programas escritos em linguagem COBOL. Com o intuito de automatizar a utilização dos critérios Potenciais Usos para o teste de programas escritos em linguagem COBOL, foram apresentados os pontos pertinentes à configuração da ferramenta POKE-TOOL.

A ferramenta POKE-TOOL implementa funções dependentes e funções não dependentes da linguagem em que está escrito o programa em teste. As funções dependentes da linguagem, que são alvo da atividade de configuração, são *Grafo de Fluxo de Controle*, *Extensão do Grafo de Fluxo de Controle* e *Instrumentacão*; elas estão relacionadas com a análise estática do código fonte, onde os processos de análise léxica e de análise sintática devem ser instanciados para a linguagem em questão.

A POKE-TOOL leva em consideração os modelos de implementação dos critérios Potenciais Usos. Dentre tais modelos, foram discutidos aqueles pertinentes à configuração da ferramenta, ou seja, os que fornecem diretamente subsídios para que o usuário prepare a POKE-TOOL para aceitar programas em novas linguagens: *Modelo de Fluxo de Controle*, *Modelo de Instrumentação e Modelo de Fluxo de Dados*. O Modelo de Fluxo de Controle caracteriza os caminhos de fluxo de execução em programas. O Modelo de Fluxo de Dados estabelece diretrizes gerais para a geração do grafo def. O Modelo de Instrumentação estabelece a localização das pontas de prova que devem ser inseridas no código fonte. Esses modelos serão instanciados para a linguagem COBOL no Capítulo 3.

CAPÍTULO 3

A LINGUAGEM COBOL NO CONTEXTO DE APLICAÇÃO DOS CRITÉRIOS POTENCIAIS USOS

Neste capítulo são discutidos os aspectos de aplicação dos critérios Potenciais Usos ao teste estrutural de Programas escritos em linguagem COBOL.

Inicialmente, são abordados os aspectos históricos e técnicos que nortearam o projeto da linguagem COBOL; são evidenciados os motivos que resultaram na pouca divulgação do COBOL na comunidade acadêmica, e são apresentadas as suas contribuições e as suas falhas técnicas.

A linguagem COBOL é apresentada no contexto de teste de unidades baseado em análise de fluxo de dados, onde são caracterizadas unidades em programas COBOL e identificadas as ocorrências de definição de variáveis. Os modelos de implementação dos critérios Potenciais Usos, apresentados na Seção 2.5, são instanciados para a linguagem COBOL. Apresentam-se diretrizes para a aplicação desses modelos.

Considerando que a abstração do grafo de fluxo de controle requer o entendimento da semântica dos comandos que alteram o fluxo de execução em um programa, estabeleceu-se o relacionamento entre os comandos dessa natureza da linguagem COBOL e da linguagem LI. Os comandos da linguagem COBOL que alteram o fluxo de execução de um programa são explorados sob o ponto de vista dos Modelos de Fluxo de Controle, de Instrumentação e de Fluxo de Dados.

3.1 A LINGUAGEM COBOL

A linguagem COBOL (*Common Business Oriented Language*) é voltada para aplicações comerciais, as quais têm como objetivo principal registrar, atualizar e recuperar informação utilizada no dia-a-dia de empreendimentos de negócios [NIC75]. Aplicações comerciais são caracterizadas por computações relativamente simples e grande volume de informação de entrada e saída. A estrutura de dados básica, utilizada nesse tipo de aplicação, é o registro. Um registro é composto por um conjunto de elementos consistentes com um propósito; por exemplo, os campos que formam uma ocorrência em um cadastro ou relatório compõem um registro. Um programa típico consiste de uma laço principal, onde arquivos de entrada são lidos, enquanto várias operações são conduzidas. Arquivos auxiliares (temporários) podem ser utilizados para ordenar/intercalar dados úteis para a estruturação das saídas; as informações de saída assumem, geralmente, a forma de arquivos de dados, relatórios e telas exibidas em transações "on-line".

O propósito do projeto da linguagem COBOL foi fornecer uma linguagem comum orientada para negócios; comum no sentido de que um programa fonte fosse portável entre computadores de diversos fabricantes. O projeto iniciou-se em 1959 por iniciativa do governo norte-americano; os desenvolvedores da linguagem COBOL eram da comunidade comercial: fabricantes e usuários de grandes sistemas de processamento de dados na indústria e governo. Da mesma forma que outras linguagens amplamente utilizadas, a linguagem COBOL sofreu uma série de revisões de projeto; sua versão inicial foi construída em 1960 e em 1985 foi liberada a última versão revisada. Sebesta [SEB89] acredita que o COBOL sobreviveu devido ao apoio do governo norte-americano; qualquer companhia que desejasse vender ou alugar computadores para o governo deveria ter um compilador COBOL, ao menos que fosse claramente comprovado que este não era necessário para a classe de problemas envolvidos.

O desenvolvimento do COBOL foi um esforço pioneiro para avançar o estado da arte de processamento de dados e de projeto de linguagens de programação [TOM83]. Apesar da grande importância e do amplo uso na comunidade de processamento de dados comerciais, o COBOL ocupa uma posição desprivilegiada no meio acadêmico. Tompkins [TOM83] e Shneiderman [SHN85] ilustram o relacionamento entre o COBOL e a comunidade de ciência da computação; aspectos históricos e técnicos evidenciados são ilustrados a seguir.

Quatro aspectos do desenvolvimento histórico do COBOL contribuem

para a sua pouca divulgação na comunidade de ciência da computação:

- os cientistas acadêmicos da ciência da computação não participaram da equipe de projeto;
- os desenvolvedores da linguagem COBOL tinham, aparentemente, pouco interesse em aspectos acadêmicos ou científicos; as publicações quase não continham referências prévias e trabalhos relatados. Em maio de 1962, a revista *Communications of the ACM* continha 13 publicações descrevendo a linguagem COBOL, onde cada artigo foi escrito por pessoal da indústria ou do governo; somente quatro desses artigos tinham qualquer referência bibliográfica;
- a notação Backus-Naur Form [PRA84,SEB89] não foi usada para especificar a gramática da linguagem. O estilo COBOL de meta-linguagem -- CBL (COBOL-*like*) -- tem-se tornado amplamente usado e pode ser considerado uma contribuição importante [PRA84];
- os cientistas de computação em 1960 não estavam interessados no domínio de aplicação de programas COBOL; estavam mais envolvidos em problemas de engenharia, análise numérica, física e sistemas de programação.

Em síntese, as pessoas e os problemas no mundo COBOL eram diferentes das pessoas e dos problemas que estavam formando a base da ciência da computação.

Em termos de aspectos técnicos, o COBOL contribuiu, principalmente, com a estrutura de registro, definição explícita de estrutura de arquivo e separação de definição de dados e aspectos procedimentais. Algumas contribuições técnicas vão abaixo discriminadas:

- a estrutura de registro e de arquivo influenciou o projeto de PL/1 e de Pascal; a agregação de itens dissimilares -- itens de tipos diferentes -- que compõem um registro constituiu, na época, avanço sobre a linguagem FORTRAN;
- os registros variantes do Pascal podem ter sido influenciados pela cláusula REDEFINES do COBOL (essa cláusula é referenciada na Seção 3.3);
- as definições explícitas de estrutura de arquivos, hierarquia de nomes para campos e declaração de dados em uma porção separada do programa foram os predecessores do conceito de sistema de gerenciamento de banco de dados;
- algumas estruturas de controle eram sofisticadas para a época; a estrutura *if-then-else* reduziu a necessidade de *goto's* e permitiu a criação de código mais compreensível; a variedade de instruções de invocação de sub-rotinas (comando PERFORM e suas variantes) favoreceu a construção de *loop's* e de projeto modular;
- a inclusão de grupos de instruções presentes em uma biblioteca encorajou a cooperação com padrões organizacionais e reuso de código.

Segundo Shneiderman, algumas falhas técnicas poderiam ter sido evitadas, através do envolvimento de cientistas da computação, mas outros problemas poderiam não ter sido reconhecidos até o início dos anos 70 [SHN85]. Algumas dessas falhas técnicas são citadas abaixo:

- a omissão de definição explícita de função ou de procedimento com parâmetros e variáveis de escopo local;
- os projetistas acreditavam que instruções "English-like" tornariam os programas legíveis por gerentes e outros não programadores, mas esqueceram de que a semântica de programação é tão importante quanto a sua sintaxe;
- o escopo de estruturas do tipo *if-then-else* é limitado pelo caractere ponto, o que facilita a ocorrência de erros de omissão ou localização indevida desse caractere;
- as instruções que compõem o corpo de uma estrutura de laço não estão agregadas à condição do laço; o corpo do laço é composto pela invocação de uma sub-rotina, a qual contém as instruções que formam o corpo do laço.

3.2 CARACTERIZAÇÃO DE UNIDADE EM PROGRAMAS COBOL

A ferramenta POKE-TOOL apóia o teste estrutural de unidades através da automação dos critérios Potenciais Usos. Em linguagens do estilo ALGOL, funções ou procedimentos são encarados como unidades de programa; na linguagem COBOL, não se observa essa correspondência direta, já que não existe uma definição explícita de função ou de procedimento. Esta seção caracteriza unidades presentes em programas COBOL.

Um programa COBOL consiste de um número de divisões -- Identification Division, Environment Division, Data Division e Procedure Division -- construídas de acordo com uma sintaxe bem definida e um propósito especializado; essas divisões são ilustradas na Figura 3.1 [NIC75].

As três primeiras divisões contêm informação declarativa. A Identification Division é constituída de informação descritiva do programa (autor, data em que foi escrito, etc). A Environment Division especifica o ambiente para execução do programa, incluindo equipamento, tal como periféricos, e especificações do sistema, por exemplo, área de memória; fornece um local onde fatores críticos de dependência de máquina podem ser observados. A Data Division contém especificações de dados e arquivos; todos os dados e arquivos devem ser declarados antes de serem referenciados.

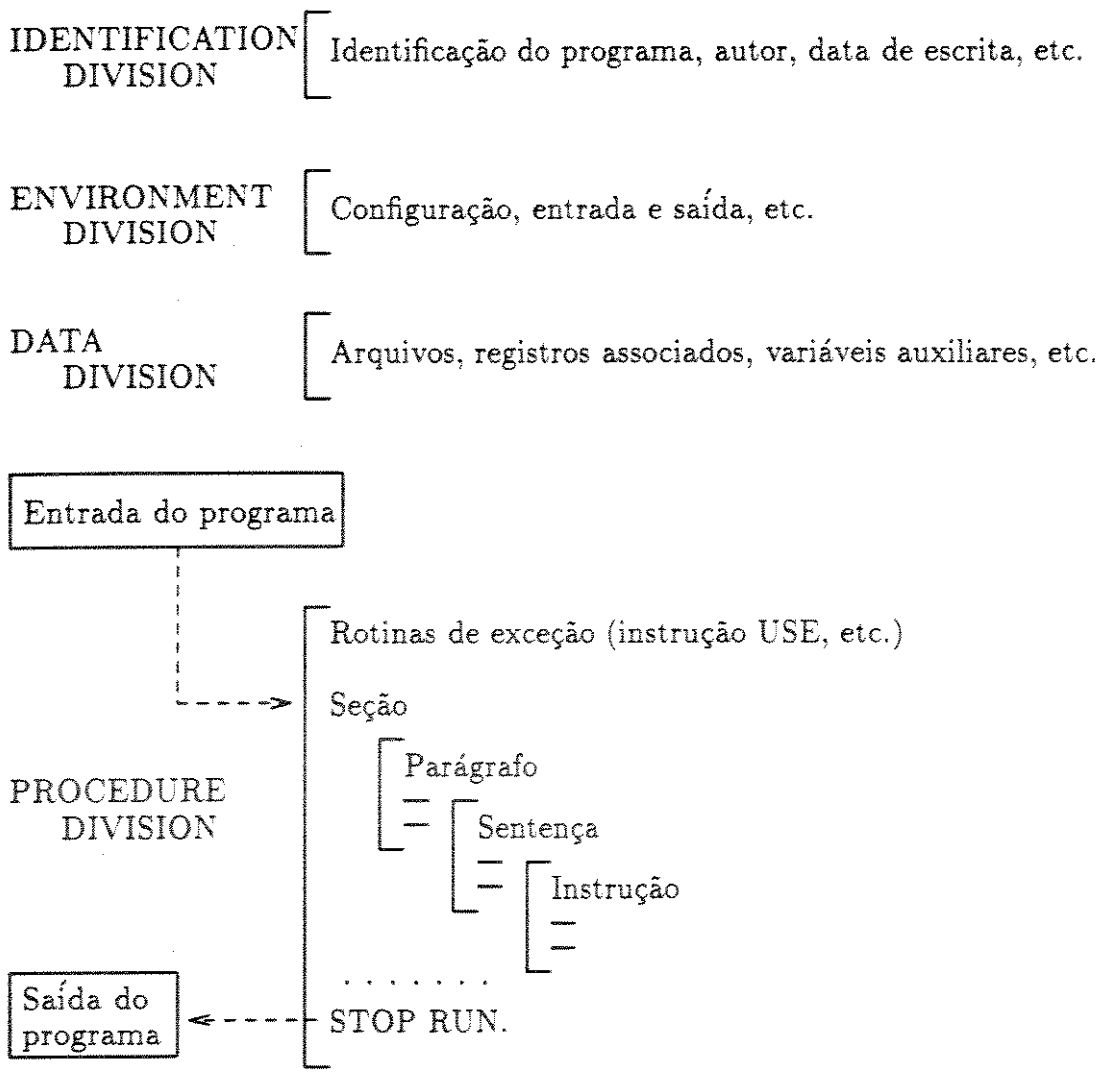


Figura 3.1: Estrutura de Programa COBOL.

Na outra divisão -- Procedure Division -- observa-se uma estrutura hierárquica: seção, parágrafo, sentença e instrução. A seção é um bloco de código nomeado que pode ser referenciado pelas instruções de desvio de controle da linguagem. Sintaticamente, o nome de uma seção é seguido do item léxico SECTION. Uma seção é terminada pelo início de outra seção; a ausência de delimitadores de seção significa que seção dentro de seção não é permitido. O parágrafo também constitui um bloco nomeado de código; um parágrafo é simplesmente um grupo de sentenças precedido por um nome de parágrafo [MaC86]. O nome de um parágrafo tenta reproduzir, da maneira mais direta possível, a idéia expressa no comando abstrato correspondente [CARD81]. Uma seção pode conter zero ou mais parágrafos. Da mesma forma que em uma seção, a ausência de delimitadores de parágrafo significa que parágrafo dentro de parágrafo não é permitido. No exemplo abaixo, INITIAL e READIN são parágrafos.

```
000100 PROCEDURE DIVISION.
000101 PART-1 SECTION.
000102 INITIAL.
       ...
000120 READIN.
       ...
000140 ADJUSTMENT SECTION.
```

Uma sentença é construída por uma série de instruções terminadas por um ponto. As instruções são ações computacionais.

Em um programa COBOL, podem ser especificadas rotinas para tratamento de exceção, que são automaticamente invocadas quando certas condições de exceção ocorrem como, por exemplo, um erro na leitura de um arquivo. Essas rotinas são organizadas em seções, e estão situadas no início da Procedure Division (Figura 3.1); associada a cada seção está uma instrução USE, a qual declara a natureza da exceção a ser tratada.

A caracterização de unidade dentro de um programa COBOL é extremamente dependente do comando PERFORM e torna-se necessário conhecer a semântica desse comando. Em sua forma mais simples, esse comando desvia o controle para uma posição do programa identificada por um rótulo, o qual caracteriza uma seção ou parágrafo. A instrução

```
PERFORM EDIT-ACCT-NUM
```

causa o desvio do controle para a posição do programa onde localiza-se o rótulo EDIT-ACCT-NUM; as instruções que seguem esse rótulo são executadas e o controle retorna, ao encontrar um parágrafo, caso EDIT-ACCT-NUM seja um parágrafo, ou ao encontrar uma seção, caso EDIT-ACCT-NUM seja uma seção, para

a próxima instrução executável após o comando PERFORM. Na instrução

PERFORM EDIT-ACCT-NUM THRU EDIT-ZIP-CODE

o controle passa para o rótulo EDIT-ACCT-NUM e a execução segue até o rótulo EDIT-ZIP-CODE ser encontrado. As instruções que seguem este rótulo são também executadas, e o controle retorna, ao encontrar um outro parágrafo, caso os rótulos envolvidos sejam parágrafos, ou ao encontrar uma seção, caso os rótulos envolvidos sejam seções, para a próxima instrução após o comando PERFORM. A Figura 3.2 [LIS78] ilustra a semântica desta ocorrência do comando PERFORM, onde os rótulos envolvidos são nomes de parágrafos.

Pratt afirma que o comando PERFORM fornece uma estrutura "call-return"; uma simples chamada de procedimento sem parâmetros. Inexistem fronteiras explícitas de procedimentos dentro da Procedure Division; no entanto, parágrafos e seções podem ser usados como simples procedimentos [PRA84].

Por sua vez, Nicholls denomina de "procedure-names" os identificadores associados às seções e parágrafos. Por implicação, seções e parágrafos são, portanto, procedimentos, embora sejam consideravelmente diferentes de procedimentos de ALGOL e PL/1; um "procedure-name" em COBOL é similar a um "label" em outras linguagens [NIC75].

Lister e Yourdon mostram como construir programas estruturados em COBOL; definem um módulo de programa como um grupo contínuo de instruções, tendo um único nome através do qual ele pode ser invocado como uma unidade. Em COBOL, isto significa que um módulo pode ser um parágrafo ou uma seção [LIS78].

A unidade que é caracterizada pelo "driver" de controle (similar à função "main" da linguagem C [KER88]) é a seqüência inicial de instruções da Procedure Division terminando no comando STOP RUN; o "driver" de controle constitui uma unidade que é alvo de interesse da atividade de teste.

Cada porção de código para onde existe uma possível transferência de controle, devido ao comando PERFORM, também caracteriza uma unidade do programa. Essas unidades são hierarquizadas em níveis de ocorrência de chamada (grafo de chamada [PRE87]). Assim, as primeiras ocorrências do comando PERFORM invocam unidades de nível 1, unidades invocadas por unidades nível 1 estão no nível 2, e assim por diante. Parágrafos e seções podem ser usados como unidades simples, mas o controle também pode fluir através de parágrafos ou seções na sequência normal de execução; neste caso, a unidade é composta por vários parágrafos ou seções.

Quando as rotinas para tratamento de exceção são especificadas em um programa COBOL, a invocação de tais rotinas é efetuada automaticamente

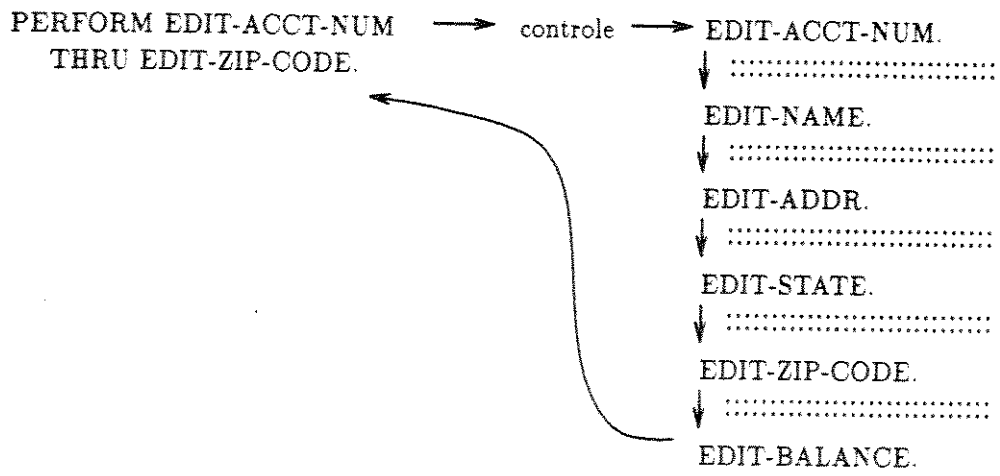


Figura 3.2: Transferência de Controle: Comando PERFORM.

quando as condições correspondentes a essas rotinas são alcançadas. Nessa situação, ocorre uma transferência implícita de controle, similar àquela realizada pelo comando PERFORM. Assim, cada rotina para tratamento de exceção também constitui uma unidade do programa.

Resumindo, uma unidade em um programa COBOL é caracterizada:

1 - pelas instruções que constituem o "driver" de controle do programa, de onde se iniciam as chamadas das demais unidades;

2 - por porções de código invocadas por um comando PERFORM, em qualquer nível de chamada;

3 - por cada porção de código que constitui uma rotina para tratamento de exceção.

Como exemplo, considere a Procedure Division abaixo [LIS78]:

```
000100 PROCEDURE DIVISION.
000110 A1-CONVERT-INPUT-FORM.
000120      OPEN RECD-IN INPUT
000130           RECD-OUT OUTPUT.
000140      MOVE "NO" TO RECD-EOF.
000150      PERFORM B1-READ-A-RECORD.
000160      PERFORM PROCESS-LOOP
000170        UNTIL RECD-EOF EQUAL "YES".
000180      CLOSE RECD-IN
000190            RECD-OUT.
000200      STOP RUN.
000210 PROCESS-LOOP.
000220      MOVE INPUT-RECORD TO OUTPUT-RECORD.
000230      PERFORM B2-WRITE-A-RECORD.
000240      PERFORM B1-READ-A-RECORD.
000250 B1-READ-A-RECORD.
000260      READ RECD-IN
000270           AT END
000280              MOVE "YES" TO RECD-EOF.
000290 B2-WRITE-A-RECORD.
000300      WRITE OUTPUT-RECORD.
```

No trecho de programa acima, a unidade que constitui o "driver" de controle é composta pelo intervalo de seqüência 110-200; as demais unidades são compostas pelos intervalos 210-240, 250-280 e 290-300, nomeadas PROCESS-LOOP, B1-READ-A-RECORD e B2-WRITE-A-RECORD, respectivamente.

O conceito de unidade em linguagem COBOL, introduzido nesta seção, difere em relação a linguagens de estilo ALGOL, pois inexiste o conceito de escopo local de variáveis, e podem ocorrer desvios incondicionais para fora dos limites de uma unidade, como também sobreposição de unidades. No entanto, a abordagem adotada é a que mais se aproxima da noção de unidade de programa.

Normalmente, o teste em linguagem COBOL é realizado considerando-se

um programa inteiro como um módulo em teste; o conceito de unidade em linguagem COBOL suporta essa prática, através do teste simultâneo de unidades. Essa facilidade foi incorporada à POKE-TOOL e será discutida na Seção 4.1.

3.3 FLUXO DE DADOS EM PROGRAMAS COBOL

Na linguagem COBOL, todas as declarações de dados ocorrem em uma parte específica do programa denominada Data Division; essa constitui o único ambiente de referência global que é usado através da Procedure Division. Portanto, todas as variáveis de um programa COBOL são globais às unidades; os únicos identificadores que não aparecem na Data Division são os rótulos de parágrafos e de seções na Procedure Division, não sendo, no entanto, locais, mas implicitamente globais [NIC75].

A linguagem COBOL é caracterizada pela inexistência de aninhamento de blocos; as regras de escopo de variáveis aplicadas às linguagens do estilo ALGOL (visibilidade, tempo de vida de variáveis) não são necessárias, pois, em COBOL, a alocação de memória é estática e todas as variáveis são globais. Assim, em linguagem COBOL, todas as variáveis são acessíveis durante a execução de um programa.

Os tipos de dados básicos são números e cadeia de caracteres com precisão especificada pelo programador [GHE87]. O princípio central de representação dos dados é que todas as variáveis são armazenadas na forma de cadeia de caracteres, com exceção dos itens de dados mencionados explicitamente para a representação binária de máquina [PRA84]; isto pode ser realizado através da cláusula COMPUTATIONAL e suas variantes. Por exemplo, na declaração

```
01 VAR PIC X(08)
```

o valor da variável VAR é armazenado em uma cadeia de oito caracteres; a declaração

```
01 AMT PIC 9(04)V99 COMPUTATIONAL
```

indica um valor de dado composto de seis dígitos, com ponto decimal entre o quarto e o quinto dígitos, armazenado na forma binária.

Os tipos de dados estruturados são registros e vetores. Registros são declarados usando um formato pré-definido de níveis aninhados, cada um com um número de nível. Registros podem ter como elementos componentes outros

registros; a estrutura de registro forma uma hierarquia similar a uma estrutura em árvore. Como exemplo, no trecho abaixo, a variável FUNCIONÁRIO é um registro com três componentes -- NOME, DAT-NAS e ENDERECO, onde DAT-NAS, também constitui um registro.

```
000100 01 FUNCIONARIO.
000101    02 NOME        PIC X(40).
000102    02 DAT-NAS.
000103       03 DIA      PIC 9(02).
000104       03 MES      PIC 9(02).
000105       03 ANO      PIC 9(02).
000106    02 ENDERECO    PIC X(40).
```

Quando uma ação é aplicada em um registro, essa ação é considerada como tendo sido aplicada em todos os seus itens componentes [KAO84]. Definições associadas a variáveis caracterizadas como registros resulta na definição das variáveis componentes desses registros; caso quaisquer desses componentes sejam também registros, os componentes destes registros são considerados definidos. A referência explícita a uma variável folha da árvore de declaração de um registro somente afeta a posição de memória atribuída a essa variável. No exemplo acima, uma ocorrência de definição da variável FUNCIONARIO resulta na definição das variáveis NOME, DIA, MES, ANO e ENDERECO; uma ocorrência de definição da variável MES não afeta as demais variáveis.

Vetores e matrizes são freqüentemente mencionados como tabelas; são declarados através da cláusula OCCURS. Tabelas podem ser compostas de registros e vice-versa. No exemplo a seguir, o registro POPULATION-DATA é composto pelas variáveis TOTAL-US-POP e STATE; a variável STATE representa um vetor de 50 elementos.

```
000100 01 POPULATION-DATA.
000101    02 TOTAL-US-POP    PIC 9(10).
000102    02 STATE           PIC X(30) OCCURS 50 TIMES.
```

No modelo de fluxo de dados estabelecido para a extensão do grafo de fluxo de controle [CHA91b,MAL91], a definição de um elemento de uma tabela implica na definição da tabela; ou seja, as variáveis que têm ocorrência repetitiva (necessitam de índices para serem referenciadas) são tratadas como variáveis de ocorrência única. No exemplo acima, uma ocorrência de definição do elemento STATE(I), onde I representa o índice que especifica um elemento da variável STATE, resulta na definição da variável STATE. Ignorar índices e

tratar todos os elementos de um "array" como uma única variável não constitui uma abordagem ideal para a detecção de anomalias de fluxo de dados [HUA79], mas tal abordagem foi adotada por ser impossível se determinar estaticamente que elemento de uma tabela estaria sendo referenciado.

No exemplo abaixo, extraído de [PRA84], o registro POPULATION-DATA é composto da variável simples TOTAL-US-POP e da tabela STATE; cada elemento desta tabela representa um registro composto de quatro variáveis simples (STATE-NAME, STATE-POP, CAPITAL-NAME e CAPITAL-POP).

```
000100 01 POPULATION-DATA.
000102     02 TOTAL-US-POP          PIC 9(10).
000103     02 STATE OCCURS 50 TIMES.
000104        03 STATE-NAME         PIC X(30).
000105        03 STATE-POP          PIC 9(9).
000106        03 CAPITAL-NAME       PIC X(30).
000107        03 CAPITAL-POP        PIC 9(8).
```

Aplicando o enfoque mencionado acima, ignora-se a cláusula OCCURS e a variável STATE passa a ser um registro com quatro componentes. Assim, uma ocorrência de definição do elemento STATE(I) resulta na definição dos itens componentes do registro STATE (STATE-NAME, STATE-POP, CAPITAL-NAME e CAPITAL-POP); uma ocorrência de definição de STATE-NAME(I) resulta somente na definição do item STATE-NAME, pertencente ao registro STATE.

A linguagem COBOL permite o conceito de redefinição de variáveis (sinonímia). Uma declaração de registro serve para reservar um bloco de armazenamento na memória e associar nomes e tipos para os vários elementos do registro. A cláusula REDEFINES habilita mais que uma forma para nomear uma mesma área de armazenamento, ou parte dessa área de armazenamento. Sinônimos de variáveis são analisados como novas variáveis disponíveis para uso/definição dentro de uma unidade; esse enfoque é conservador na aplicação dos critérios Potenciais Usos, pois nenhum caminho ou associação deixa de ser requerido.

A Data Division é constituída de duas seções principais: WORKING-STORAGE SECTION e FILE SECTION. Na primeira, declaram-se as variáveis simples e estruturas de dados utilizadas nas computações do programa; na segunda, declaram-se os registros transmitidos entre a memória e arquivos externos, ou seja, especificam-se posições de memória que são transferidas de/para os "buffers" dos arquivos, através dos comandos de saída/entrada. Do ponto de vista da aplicação dos critérios Potenciais Usos, ambas as classes de variáveis são tratadas indistintamente.

O comando PERFORM, na sua forma mais simples, é tratado como uma

chamada de unidade. Nenhuma variável é explicitamente passada como parâmetro, mas todas as variáveis do programa estão disponíveis para uso ou definição dentro de qualquer unidade. Para cada ocorrência do comando PERFORM, convencionou-se a definição por referência de todas as variáveis do programa, ocasionando a seleção mais rigorosa de caminhos e associações para a aplicação dos critérios Potenciais Usos. Já o comando CALL, que invoca a execução de subprogramas, permite a passagem explícita de variáveis por referência; tais variáveis são também ditas definidas por referência.

Na análise individual de unidades, como qualquer variável já poderia ter sido definida anteriormente, assumiu-se uma abordagem de pior caso: todas as variáveis são consideradas definidas no nó inicial da unidade. Uma exceção acontece na unidade que constitui o "driver" de controle do programa; somente as variáveis declaradas e iniciadas através da cláusula VALUE na Data Division são consideradas definidas na análise dessa unidade; as demais assumem o estado de indefinidas.

A seguir, é apresentado um resumo do modelo de fluxo de dados, com vistas à linguagem COBOL, para a obtenção do Grafo Def:

1 - ocorre definição de variável no comando de entrada READ;

2 - uma variável que recebe valor em um comando de atribuição é assumido como uma definição. Os comandos de atribuição são caracterizados por verbos que denotam a ação correspondente (ACCEPT, ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE, MOVE, STRING, UNSTRING, INSPECT);

3 - a definição de um elemento de uma variável estruturada do tipo vetor ou matriz implica na definição da variável;

4 - as definições associadas às variáveis caracterizadas como registros resultam na definição dos itens componentes desses registros; a definição de qualquer item componente de um registro resulta somente na definição desse item, estando sujeito às regras 1,2,3 e 4;

5 - a definição por referência ocorre para todas as variáveis do programa quando o comando PERFORM é utilizado. No comando CALL, o qual invoca a execução de subprogramas, as variáveis (parâmetros) explicitamente passadas como referência também são ditas definidas por referência;

6 - no início de uma unidade, todas as variáveis passíveis de definição são

consideradas definidas. Na unidade caracterizada pelo "driver" de controle do programa, somente são definidas as variáveis iniciadas na Data Division.

3.4 DIRETRIZES PARA A APLICAÇÃO DOS MODELOS DE IMPLEMENTAÇÃO DOS CRITÉRIOS POTENCIAIS USOS PARA A LINGUAGEM COBOL

Nesta seção são apresentadas as diretrizes para a aplicação dos modelos de fluxo de controle, de instrumentação e de fluxo de dados para a linguagem COBOL; são analisados os aspectos semânticos dos comandos através de suas influências no fluxo de controle.

O modelo de fluxo de controle sumarizado na Seção 2.5.1 reflete a semântica da LI do ponto de vista de fluxo de controle. O relacionamento semântico entre os comandos da linguagem LI e da linguagem COBOL é obtido através do mapeamento de suas construções sintáticas.

A instrumentação deve ser tal que reflita a semântica dos comandos da LI e ao mesmo tempo viabilize a correta avaliação dos caminhos efetivamente executados; observa-se, também, que a instrumentação está fortemente restrita ao modelo de fluxo de controle adotado. Além da inserção de pontas de prova, a versão instrumentada da unidade em teste também contém, através de comentários inseridos, a localização, no programa fonte, dos comandos associados aos nós; isto facilita as atividades de teste, assim como outras correlatas, como depuração e manutenção. Não se objetiva realizar qualquer modificação nos comandos originais do programa fonte, como ocorre no modelo instanciado para unidades escritas na linguagem C [CHA91b]; infelizmente, isto não foi alcançado na instrumentação de alguns comandos da linguagem COBOL, como pode ser observado no decorrer desta seção, principalmente decorrente do fato dessa linguagem não possuir estrutura de bloco.

Para cada comando considerado a seguir, são apresentados:

- a semântica, do ponto de vista de fluxo de controle, através de exemplos;
- a sintaxe na notação BNF [PRA84,SEB89];
- o mapeamento para a LI, através de um tabela de equivalência sintática, e o grafo de fluxo de controle associado ao exemplo apresentado. A tabela de equivalência representa o relacionamento sintático entre o comando da linguagem COBOL e seu equivalente na LI. A definição sintática dos comandos

da LI é mostrada no Apêndice A;
- a versão em LI do exemplo;
- a versão instrumentada do exemplo;
- as diretrizes para a construção do Grafo Def; conduz a determinação dos nós onde ocorre definição de dados. São associadas as ocorrências de definição de variáveis aos nós do grafo de fluxo de controle.

Para auxiliar a instrumentação de cada exemplo, é utilizada a pseudo-função *ponta_de_prova*, a qual registra os nós percorridos durante a aplicação dos casos de teste; *ponta_de_prova(n)* grava o parâmetro *n* no arquivo de caminhos executados.

3.4.1 COMANDOS SEQÜENCIAIS

Os comandos seqüenciais representam a maioria dos comandos presentes na linguagem COBOL. Os comandos seqüenciais consistem basicamente de comandos aritméticos (por exemplo, ADD e SUBTRACT), de comandos de manipulação de variáveis (por exemplo, MOVE e STRING), de comandos de ordenação de arquivos (por exemplo, SORT e MERGE), de comandos de chamada de subprogramas (por exemplo, CALL), e de comandos de invocação de unidades (por exemplo, PERFORM).

. Exemplos de Comandos Seqüenciais

A seguir, são apresentados cinco exemplos de comandos seqüenciais.

```
000100      ADD 1 TO CONTADOR
000101      MOVE 0 TO INDICE
000102      READ CADASTRO
000103      SORT ARQ-SORT
000104          ON ASCENDING KEY KEY-SORT
000105          USING  ARQ-ENTRADA
000106          GIVING ARQ-SAIDA
000107      CALL "FORMATA-RELATORIO"
```

. Sintaxe dos Comandos Seqüenciais

<sequenciais> ::= <add> | <subtract> | ... | <move> | <string> |
<sort> | <merge> | ... | <call> | <perform>

Na sintaxe apresentada acima, os elementos à direita do símbolo '::=' representam elementos não terminais, os quais são definidos em outras regras gramaticais do COBOL, não sendo de nosso interesse mostrar os detalhes sintáticos de cada comando seqüencial.

. Mapeamento dos Comandos Seqüenciais para a LI

Os comandos seqüenciais da linguagem COBOL são mapeados diretamente para os comandos seqüenciais da linguagem intermediária (LI). A sintaxe dos comandos seqüenciais da LI, apresentada no Apêndice A, tem o seguinte formato:

<seqüência> ::= <s> ¦ <dcl>

A equivalência sintática entre o COBOL e a LI, para os comandos seqüenciais, é mostrada na Tabela 3.1; os elementos à esquerda pertencem à definição sintática da linguagem COBOL e os elementos à direita pertencem à definição sintática da linguagem LI. O grafo de fluxo de controle desse tipo de comando é apresentado na Figura 3.3.

. Versão em LI dos Exemplos

```
$S    500 17 15
$S    529 16 16
$S    557 13 17
$S    582 117 18
$S    711 24 22
```

. Versão Instrumentada dos Exemplos

Para instrumentar os comandos seqüenciais, deve-se inserir no código fonte uma chamada à função *ponta-de-prova*; esta função registrará o nó associado ao bloco de comandos que contém o comando seqüencial.

```
000100** n **
000101      ponta_de_prova (n)
000102      ADD 1 TO CONTADOR
000103      MOVE 0 TO INDICE
000104      READ CADASTRO
000105      SORT ARQ-SORT
000106         ON ASCENDING KEY KEY-SORT
000107         USING ARQ-ENTRADA
000108         GIVING ARQ-SAIDA
000109      CALL "FORMATA-RELATORIO"
```

. Diretrizes para a Construção do Grafo Def para os Comandos Seqüenciais

Ao nó *n* é atribuído o conjunto de variáveis definidas no bloco de comandos associado a esse nó; no exemplo, ao nó *n* são associadas as variáveis CONTADOR, INDICE e as variáveis componentes do registro que descreve o arquivo CADASTRO.

Tabela 3.1: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comandos Seqüenciais.

COBOL	LI
<add>	<s>
<subtract>	<s>
.	.
.	.
.	.
<move>	<s>
<string>	<s>
.	.
.	.
.	.
<sort>	<s>
<merge>	<s>
.	.
.	.
.	.
<call>	<s>
.	.
.	.
.	.
<perform>	<s>

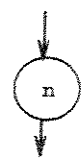


Figura 3.3: Grafo de Fluxo de Controle: Comandos Seqüenciais.

3.4.2 COMANDOS DE SELEÇÃO

As estruturas de seleção na linguagem COBOL estão presentes no comando IF e nas cláusulas para tratamento de exceção, apresentados a seguir.

3.4.2.1 Comando IF

Este comando tem a mesma semântica do comando "if" das linguagens do estilo ALGOL.

. Exemplo do Comando IF

```
000100      IF NOVA-PAGINA
000101         ADD 1 TO CONT-PAGINA
000102         MOVE O TO CONT-LINHA
000103      ELSE
000104         ADD 1 TO CONT-LINHA.
```

. Sintaxe do Comando IF

Na sintaxe apresentada abaixo, o elemento <statement> representa todos os possíveis comandos da linguagem COBOL.

<if> ::= **IF** <boolean expression> <statement> . |
IF <boolean expression> <statement> **ELSE** <statement> .

. Mapeamento do Comando IF para a LI

O comando IF equivale ao comando 'if' da LI; a sintaxe do comando 'if' da LI, apresentada do Apêndice A, tem o seguinte formato:

<if> ::= <if_atm> <cond_atm> <statement> |
<if_atm> <cond_atm> <statement> <else_atm> <statement>

A equivalência sintática entre o COBOL e a LI, para o comando IF, é mostrada na Tabela 3.2; os elementos à esquerda pertencem à definição sintática da linguagem COBOL e os elementos à direita pertencem à definição sintática da linguagem LI. O grafo de fluxo de controle correspondente ao exemplo é apresentado na Figura 3.4.

. Versão em LI do Exemplo

O exemplo apresentado seria traduzido para a LI como:

```
$IF          500 2 15
$C01(01)     503 11 15
{            0 0 0
$S01         529 20 16
$S02         564 20 17
}            0 0 0
$ELSE        596 4 18
{            0 0 0
$S03         615 19 19
}            634 1 19
```

. Versão Instrumentada do Exemplo

Para instrumentar o comando IF, devem-se inserir, no código fonte, chamadas à função *ponta_de_prova*, que registrará os nós exercitados durante a execução dos casos de teste. Assim, cada nó executado resultará em uma chamada a esta função. A versão instrumentada do exemplo é apresentada abaixo.

```
000100** n1 **
000101       ponta_de_prova (n1)
             .
             .
000102       IF NOVA-PAGINA
000103** n2 **
000104       ponta_de_prova (n2)
000105          ADD 1 TO CONT-PAGINA
000106          MOVE 0 TO CONT-LINHA
000107       ELSE
000108** n3 **
000109          ponta_de_prova (n3)
000110          ADD 1 TO CONT-LINHA.
000111** n4 **
000112       ponta_de_prova (n4)
```

. Diretrizes para a Construção do Grafo Def para o Comando IF

Ao nó *n*1 é atribuído somente o conjunto de variáveis definidas no bloco de comandos associado a esse nó; não é associada qualquer variável relativa à condição. As atribuições presentes nos nós *n*2 e *n*3 ou nos subgrafos correspondentes dependem dos comandos envolvidos. Ao nó *n*4 é associado o conjunto de variáveis definidas nos comandos associados a esse nó.

Tabela 3.2: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando IF.

COBOL	LI
IF ---------->	<if_atm>
<boolean expression> ---->	<cond_atm> <{>
<statement> ---------->	<statement>
ELSE ---------->	<}> <else_atm> <{>
. ---------->	<}>

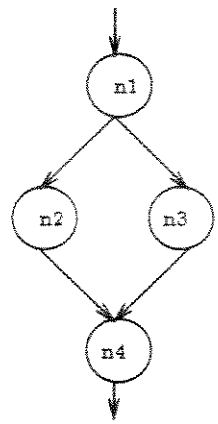


Figura 3.4: Grafo de Fluxo de Controle: Comando IF.

3.4.2.2 Cláusulas para Tratamento de Exceção

As cláusulas para tratamento de exceção habilitam o uso de instruções apropriadas, quando ocorrem situações de exceção nos comandos a que estão agregadas; essas cláusulas podem ser utilizadas nos comandos de entrada/saida (por exemplo, READ e WRITE) e nos comandos aritméticos (por exemplo, ADD e COMPUTE).

. Exemplos das Cláusulas para Tratamento de Exceção

Como exemplos, considere as cláusulas AT, ON e INVALID nos comandos abaixo:

```
000100      READ CADAST
000101      AT END
000102         DISPLAY "FIM DE CADASTRO"
000103         MOVE 1 TO FIM-CADAST.
000104      COMPUTE SDO-ATUAL = SDO-ANTERIOR - DEB + CRE
000105      ON SIZE ERROR
000106         DISPLAY "ERRO NO CALCULO DO SALDO"
000107         MOVE 1 TO ERRO-CALCULO.
000108      WRITE REG-CADAST
000109      INVALID KEY
000110         DISPLAY "ERRO NA GRAVACAO DO CADASTRO"
000111         MOVE 1 TO ERRO-GRAVACAO.
```

Na linha 101, é tratada a condição de final de arquivo na leitura do arquivo CADAST; na linha 105, é prevista a situação onde o resultado da operação aritmética ultrapassa o valor máximo que a variável SDO-ATUAL pode assumir; na linha 109, são tratados possíveis erros durante a gravação do registro REG-CADAST.

. Sintaxe das Cláusulas para Tratamento de Exceção

Nas regras sintáticas apresentadas abaixo, o elemento <statement>$_1$ representa todos os possíveis comandos da linguagem COBOL.

```
<at>       ::= AT <aux1> <statement>1 .
<aux1>     ::= END | EOP | END-OF-PAGE

<on>       ::= ON <aux2> <statement>1 .
<aux2>     ::= SIZE ERROR | OVERFLOW

<invalid>  ::= INVALID KEY <statement>1 .
```

. Mapeamento das Cláusulas para Tratamento de Exceção para a LI

As cláusulas para tratamento de exceção são equivalentes ao comando 'if' da LI; a sintaxe do comando 'if' da LI, apresentada no Apêndice A, tem o seguinte formato:

<if> ::= <if_atm> <cond_atm> <statement> ¦
<if_atm> <cond_atm> <statement> <else_atm> <statement>

A equivalência sintática entre as cláusulas para tratamento de exceção da linguagem COBOL e o comando 'if' da LI é mostrada na Tabela 3.3. O grafo de fluxo de controle correspondente ao exemplo é apresentado na Figura 3.5.

. Versão em LI dos Exemplos

```
$S01        500 11 15
$IF         523 2 16
$C(01)01    526 3 16
{           0 0 0
$S02        544 25 17
$S03        584 20 18
}           604 1 18
$S04        617 44 19
$IF         673 2 20
$C(01)02    676 10 20
{           0 0 0
$S05        701 34 21
$S06        750 22 22
}           772 1 22
$S07        785 16 23
$IF         813 7 24
$C(01)03    821 3 24
{           0 0 0
$S08        838 38 25
$S09        891 23 26
}           914 1 26
```

. Versão Instrumentada dos Exemplos

Para instrumentar as cláusulas para tratamento de exceção, procede-se de maneira similar ao comando IF da linguagem COBOL, como foi abordado na Seção 3.4.2.1.

Tabela 3.3: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Cláusulas para Tratamento de Exceção.

COBOL	LI
AT ---------->	<if_atm>
ON ---------->	<if_atm>
INVALID ---------->	<if_atm>
<aux1> ---------->	<cond_atm> <{>
<aux2> ---------->	<cond_atm> <{>
KEY ---------->	<cond_atm> <{>
<statement>$_1$ ---------->	<statement>
. ---------->	<}>

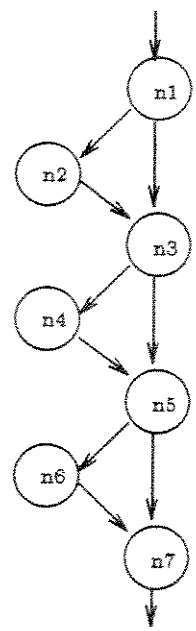


Figura 3.5: Grafo de Fluxo de Controle: Cláusulas para Tratamento de Exceção.

```
000100** n1 **
000101      ponta_de_prova (n1)
            :
000102      READ CADAST
000103      AT END
000104** n2 **
000105          ponta_de_prova (n2)
000106          DISPLAY "FIM DE CADASTRO"
000107          MOVE 1 TO FIM-CADAST.
000108** n3 **
000109      ponta_de_prova (n3)
000110      COMPUTE SDO-ATUAL = SDO-ANTERIOR - DEB + CRE
000111      ON SIZE ERROR
000112** n4 **
000113          ponta_de_prova (n4)
000114          DISPLAY "ERRO NO CALCULO DO SALDO"
000115          MOVE 1 TO ERRO-CALCULO.
000116** n5 **
000117      ponta_de_prova (n5)
000118      WRITE REG-CADAST
000119      INVALID KEY
000120** n6 **
000121          ponta_de_prova (n6)
000122          DISPLAY "ERRO NA GRAVACAO DO CADASTRO"
000123          MOVE 1 TO ERRO-GRAVACAO.
```

. **Diretrizes para a Construção do Grafo Def para as Cláusulas para Tratamento de Exceção**

Aos nós *n*1, *n*3 e *n*5 é atribuído somente o conjunto de variáveis definidas no bloco de comandos associado a esses nós; as atribuições presentes nos nós *n*2, *n*4 e *n*6 ou nos subgrafos correspondentes dependem dos comandos envolvidos em cada nó.

3.4.3 COMANDOS DE ITERAÇÃO

Os comandos PERFORM-UNTIL, PERFORM-TIMES, PERFORM-VARYING, SEARCH (busca serial) e SEARCH (busca binária) implementam as estruturas de iteração presentes na linguagem COBOL.

3.4.3.1 Comando PERFORM-UNTIL

O comando PERFORM associado à cláusula UNTIL itera a execução da unidade até que uma expressão condicional seja verdadeira. Se, inicialmente, a condição já for verdadeira, a unidade não é ativada e nenhuma transferência de controle é realizada.

. Exemplo do Comando PERFORM-UNTIL

```
000100      PERFORM PROCESSA THRU PROCESSA-EXIT
000101        UNTIL CHAVE-MOVIM GREATER CHAVE-CADAST
```

. Sintaxe do Comando PERFORM-UNTIL

<perf_until> ::= <perform> **UNTIL** <boolean expression>
<perform> :: = **PERFORM** <identifier> ¦
PERFORM <identifier> **THRU** <identifier>

. Mapeamento do Comando PERFORM-UNTIL para a LI

Este formato do comando PERFORM equivale ao comando 'while' da LI, com a diferença de que a iteração é realizada enquanto a condição for falsa. A sintaxe do comando 'while' da LI, apresentada no Apêndice A, tem o seguinte formato:

<while> ::= <while_atm> <cond_while_atm> <statement>

A equivalência sintática entre o COBOL e a LI, para o comando PERFORM-UNTIL, é mostrado na Tabela 3.4; o símbolo ø indica que o átomo da LI não possui correspondência direta no arquivo fonte. O grafo de fluxo de controle correspondente ao exemplo é exibido na Figura 3.6.

. Versão em LI do Exemplo

O exemplo apresentado será traduzido para a LI como:

```
$WHILE    0 0 0
$NC       549 38 16
$S        500 35 15
```

. Versão Instrumentada do Exemplo

Para instrumentar o comando PERFORM-UNTIL, criou-se uma sub-rotina específica para cada ocorrência do comando. Caso a expressão condicional (representada por <boolean expression> na sintaxe do comando) seja avaliada como falsa, essa sub-rotina registrará a execução de um percurso completo pelo corpo do laço. A sub-rotina é definida pelos comandos que seguem o rótulo *<ROTULO>-PUNTIL-ii*, onde *<ROTULO>* constitui os sete primeiros caracteres do rótulo para onde o fluxo de controle foi desviado e *ii* representa a ordem de aparição física do comando no programa fonte.

Tabela 3.4: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando PERFORM-UNTIL.

COBOL	LI
ø ---------->	<while_atm>
UNTIL <boolean expression> ---->	<cond_while_atm>
<perform> ---------->	<s>

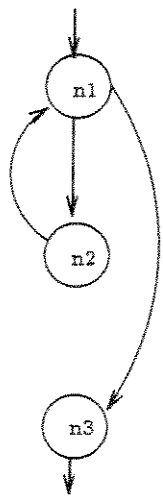


Figura 3.6: Grafo de Fluxo de Controle: Comando PERFORM-UNTIL.

```
000100** n1 **
000101** n2 **
000102      PERFORM PROCESS-PUNTIL-01
000103        UNTIL CHAVE-MOVIM GREATER CHAVE-CADAST
000104      ponta_de_prova (n1)
000105      ponta_de_prova (n3)
              .
              .
              .
000200 PROCESS-PUNTIL-01.
000201      ponta_de_prova (n1)
000202      ponta_de_prova (n2)
000203      PERFORM PROCESSA THRU PROCESSA-EXIT.
```

. Diretrizes para a Construção do Grafo Def para o Comando PERFORM-UNTIL

Observando o grafo da Figura 3.6, o nó *n*1 representa a condição de execução do laço, o nó *n*2 o corpo do laço e o nó *n*3 refere-se à primeira sentença executável após o comando PERFORM-UNTIL. Ao nó *n*1 não é associada qualquer definição de variável; ao nó *n*2 é atribuída a definição por referência de todas as variáveis do programa, pois o corpo do laço é composto por uma invocação de unidade; ao nó *n*3 somente é atribuído o conjunto de variáveis definidas no bloco de comandos associado a esse nó.

3.4.3.2 Comando PERFORM-TIMES

O comando PERFORM associado à cláusula TIMES itera a execução da unidade em um número finito de vezes. O número de iterações pode ser fixo, sendo especificado através de uma constante inteira, ou corresponder a um valor armazenado em uma variável inteira do programa, como ocorre no exemplo a seguir.

. Exemplo do Comando PERFORM-TIMES

```
000100      PERFORM PROCESSA THRU PROCESSA-EXIT
000101        NUM-REG TIMES
```

. Sintaxe do Comando PERFORM-TIMES

```
<perf_times> ::= <perform> <aux> TIMES
<aux>        ::= <constant> | <identifier>
```

. Mapeamento do Comando PERFORM-TIMES para a LI

A semântica deste comando assemelha-se ao comando 'for' da LI, ou seja, a execução contada do laço. A sintaxe do comando 'for' da LI, apresentada no Apêndice A, tem o seguinte formato:

<for> ::= <for_atm> <s1> <cond_for_atm> <s2> <statement>

A equivalência sintática entre o COBOL e a LI, para o comando PERFORM-TIMES, é apresentada na Tabela 3.5; o símbolo ø indica que o átomo da LI não possui correspondência no arquivo fonte. O grafo de fluxo de controle correspondente ao exemplo é exibido na Figura 3.7.

. Versão em LI do Exemplo

O exemplo apresentado seria traduzido para a LI como:

```
$FOR        0 0 0
$S01        0 0 0
$C(01)01    549 13 16
$S02        0 0 0
$S03        500 35 15
```

. Versão Instrumentada do Exemplo

Para instrumentar o comando PERFORM-TIMES, criou-se uma sub-rotina específica para cada ocorrência do comando, a qual registra a execução de um percurso completo pelo corpo do laço. Essa sub-rotina é definida pelos comandos que seguem o rótulo *<ROTULO>-PTIMES-ii*, onde *<ROTULO>* constitui os sete primeiros caracteres do rótulo para onde o fluxo de controle foi desviado e *ii* representa a ordem de aparição física do comando no programa fonte.

```
000100** n1 **
000101      ponta_de_prova (n1)
            .
000102** n2 **
000103** n3 **
000104      PERFORM PROCESS-PTIMES-01
000105        NUM-REG TIMES
000106      ponta_de_prova (n2)
000107      ponta_de_prova (n4)
            .
            .
            .
000200 PROCESS-PTIMES-01.
000201      ponta_de_prova (n2)
000202      ponta_de_prova (n3)
000203      PERFORM PROCESSA THRU PROCESSA-EXIT.
```

Tabela 3.5: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando PERFORM-TIMES.

COBOL	LI
ø ---------->	<for_atm> <s1>
<aux> TIMES -------->	<cond_for_atm> <s2>
<perform> ---------->	<s>

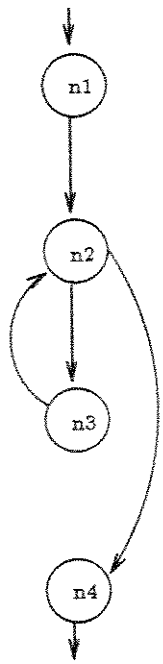


Figura 3.7: Grafo de Fluxo de Controle: Comando PERFORM-TIMES.

. **Diretrizes para a Construção do Grafo Def para o Comando PERFORM-TIMES**

Para a construção do Grafo Def, a variável que serve como contador durante a iteração não é considerada na análise de fluxo de dados, nem compõe o elenco de variáveis disponíveis para o programador. A execução contada do laço apenas reflete a semântica do comando PERFORM-TIMES, não ocasionando, portanto, a definição de variáveis associadas ao controle do laço. Deste modo, apenas são consideradas as definições relativas à invocação de uma unidade do programa; ou seja, ao nó *n*3 é atribuída a definição por referência de todas as variáveis do programa. Ao nó *n*4, que é o nó de saída do laço, é atribuído o conjunto de variáveis definidas no bloco de comandos associado a este nó.

3.4.3.3 Comando PERFORM-VARYING

O comando PERFORM associado à cláusula VARYING itera a execução da unidade através do aninhamento de um a três laços. Para cada nível de aninhamento é associado uma variável, a qual desempenha o papel de variável contadora do laço, e uma condição de saída do laço.

. **Exemplo do Comando PERFORM-VARYING**

No exemplo a seguir, as variáveis I, J e K desempenham o papel de variável contadora nos laços mais externo, intermediário e mais interno, respectivamente; nesta mesma ordem, as condições associadas aos laços são I > MAX-I, J > MAX-J e K > MAX-K.

```
000100      PERFORM PROCESSA THRU PROCESSA-EXIT
000101        VARYING I FROM 1 BY 1 UNTIL I > MAX-I
000102        AFTER  J FROM 1 BY 1 UNTIL J > MAX-J
000103        AFTER  K FROM 1 BY 1 UNTIL K > MAX-K
```

. **Sintaxe do Comando PERFORM-VARYING**

```
<perf_var> ::= <perform> VARYING <identifier> FROM <aux> BY <aux>
                 UNTIL <boolean expression> |
               <perform> VARYING <identifier> FROM <aux> BY <aux>
                 UNTIL <boolean expression> <loop>
<loop>     ::= AFTER <identifier> FROM <aux> BY <aux>
                 UNTIL <boolean expression> |
               <loop> AFTER <identifier> FROM <aux> BY <aux>
```

UNTIL <boolean expression>

<aux> ::= <identifier> ¦ <constant>

. Mapeamento do Comando PERFORM-VARYING para a LI

O comando PERFORM-VARYING é considerado como um conjunto de comandos 'for' da LI aninhados. A sintaxe do comando 'for' da LI, apresentada no Apêndice A, tem o seguinte formato:

<for> ::= <for_atm> <s1> <cond_for_atm> <s2> <statement>

A equivalência sintática entre o COBOL e a LI, para o comando PERFORM-VARYING, é mostrada na Tabela 3.6. O grafo de fluxo de controle do exemplo é apresentado na Figura 3.8.

. Versão em LI do Exemplo

O exemplo apresentado seria traduzido para a LI como:

```
$FOR        549 7 16
$S01        555 8 16
$NC(01)01   569 15 16
$S02        564 4 16
$FOR        598 5 17
$S03        606 8 17
$NC(01)02   620 15 17
$S04        615 4 17
$FOR        649 5 18
$S05        657 8 18
$NC(01)03   671 15 18
$S06        666 4 18
$S07        500 35 15
```

. Versão Instrumentada do Exemplo

Para instrumentar o comando PERFORM-VARYING, criou-se um conjunto de sub-rotinas numeradas para cada ocorrência desse comando. Cada conjunto de subrotinas é definido pelos comandos que seguem os rótulos *<ROTULO>-PVA-ii-jj*, onde *<ROTULO>* representa os sete primeiros caracteres do rótulo para onde o fluxo de controle foi desviado, *ii* é um número que identifica a ordem de aparição física do comando PERFORM-VARYING no arquivo fonte e *jj* numera os rótulos necessários à instrumentação deste comando; *jj* varia proporcionalmente com a quantidade de laços envolvidos. No exemplo apresentado, o comando PERFORM-VARYING consiste de três laços aninhados, o que resulta em três sub-rotinas necessárias para sua instrumentação.

Tabela 3.6: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando PERFORM-VARYING.

COBOL	LI
VARYING ---------->	<for_atm>
AFTER ---------->	<for_atm>
<identifier> **FROM** <aux> ------>	<s1>
UNTIL <boolean expression> --->	<cond_for_atm>
BY <aux> ---------->	<s2>
<perform> ---------->	<s>

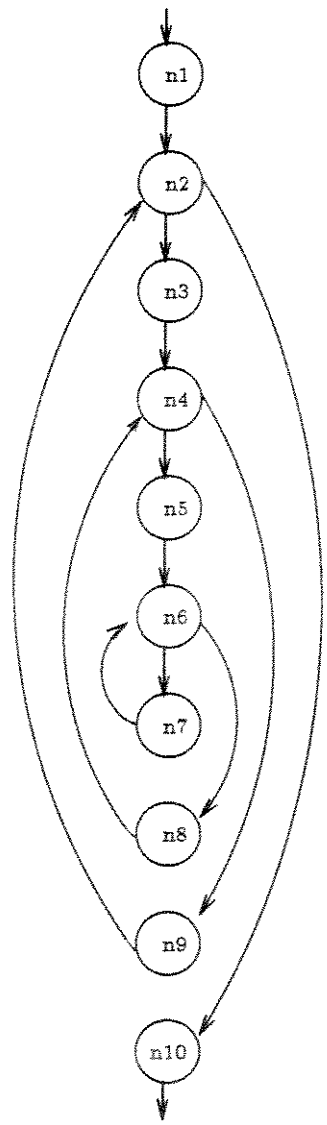


Figura 3.8: Grafo de Fluxo de Controle: Comando PERFORM-VARYING.

```
000100** n1 **
000101       ponta_de_prova (n1)
             :
000102** n2 **
000103** n9 **
000104       PERFORM PROCESS-PVA-01-01
000105         VARYING I FROM 1 BY 1 UNTIL I > MAX-I
000106       ponta_de_prova (n2)
000107       ponta_de_prova (n10)
             :
             :
             :
             :
000200 PROCESS-PVA-01-01.
000201       ponta_de_prova (n2)
000202       ponta_de_prova (n3)
000203** n3 **
000204** n4 **
000205** n8 **
000206       PERFORM PROCESS-PVA-01-02
000207         VARYING J FROM 1 BY 1 UNTIL J > MAX-J
000208       ponta_de_prova (n4)
000209       ponta_de_prova (n9).
000210 PROCESS-PVA-01-02.
000211       ponta_de_prova (n4)
000212       ponta_de_prova (n5)
000213** n5 **
000214** n6 **
000215** n7 **
000216       PERFORM PROCESS-PVA-01-03
000217         VARYING J FROM 1 BY 1 UNTIL J > MAX-J
000218       ponta_de_prova (n6)
000219       ponta_de_prova (n8).
000220 PROCESS-PVA-01-03.
000221       ponta_de_prova (n6)
000222       ponta_de_prova (n7)
000223       PERFORM PROCESSA THRU PROCESSA-EXIT.
```

. **Diretrizes para a Construção do Grafo Def para o Comando PERFORM-VARYING**

Observando o grafo da Figura 3.8, os nós $n1$,$n3$ e $n5$ representam a iniciação das variáveis contadoras; os nós $n7$, $n8$ e $n9$ contém os comandos de incremento da variável contadora; os nós $n2$, $n4$ e $n6$ representam as condições de saída dos laços aninhados; e o nó $n7$ também contém o corpo do laço mais interno. Ao nó $n1$ é atribuído a variável iniciada pelo laço mais externo; ao nó $n3$ é atribuído a variável iniciada pelo laço intermediário; ao nó $n5$ é atribuído a variável iniciada pelo laço mais interno; aos nós $n2$, $n4$ e $n6$ não é associada qualquer definição de variável; no nó $n7$, que contém uma invocação de unidade, todas as variáveis são consideradas definidas por referência; adicionalmente, ao nó $n7$ também é atribuída a variável contadora do laço mais interno; ao nó $n8$ é associada a variável contadora do laço intermediário; ao nó $n9$ é atribuída a variável contadora do laço mais externo; ao nó $n10$ é

atribuído o conjunto de variáveis definidas no bloco de comandos associado a esse nó.

3.4.3.4 Comando SEARCH (busca linear)

O comando SEARCH é utilizado para pesquisar numa tabela por um elemento que satisfaça uma condição específica. Associado à tabela objeto do comando, tem-se uma variável usada como índice da tabela durante a pesquisa. Caso a pesquisa termine com sucesso -- algum elemento da tabela satisfez uma condição -- o índice estará ajustado para indicar a posição do elemento encontrado.

O índice usado na pesquisa constitui uma variável que é declarada junto com a tabela (através da cláusula INDEXED BY), como a variável IND associada à tabela TAB-HOMONIMOS no exemplo a seguir.

. Exemplo do Comando SEARCH (busca serial)

```
000050 01 TAB-HOMONIMOS OCCURS 50 TIMES INDEXED BY IND.
000051    02 NOME         PIC X(40).
000052    02 DAT-NAS      PIC 9(06).
000053    02 RG           PIC 9(10).
000054    02 CIC          PIC 9(11).
             .
             .
             .
000100       SEARCH TAB-HOMONIMOS
000101         AT END  MOVE 0 TO ACHA-HOMONIMO
000102         WHEN    RG (IND)  EQUAL M-RG
000103                 PERFORM FORMATA-CLIENTE
000104                 PERFORM EXIBE-CLIENTE.
```

A pesquisa inicia-se a partir da posição da tabela referenciada pelo valor corrente do índice. A expressão condicional associada a cada cláusula WHEN (no exemplo, 'RG(IND) EQUAL M-RG') é avaliada e, caso todas as expressões sejam falsas, o índice é incrementado e cada expressão é novamente avaliada; são permitidas várias expressões condicionais para serem avaliadas, mas, no exemplo, apenas uma cláusula WHEN foi especificada. Caso alguma dessas expressões seja avaliada como verdadeira, as instruções atribuídas a esta expressão (no exemplo, 'PERFORM FORMATA-CLIENTE' e 'PERFORM EXIBE-CLIENTE') são executadas e o controle é transferido para a próxima sentença executável após o comando SEARCH. O processo se repete até que alguma das condições seja verdadeira, ou o valor do índice ultrapasse o limite superior de indexação da tabela (no exemplo, 50); nessa situação, as instruções atribuídas à cláusula AT END (no exemplo, 'MOVE 0 TO ACHA-HOMONIMO'), se esta

cláusula tiver sido especificada, serão executadas. Caso a cláusula VARYING seja utilizada, o que não ocorre no exemplo apresentado, o identificador que sintaticamente segue essa cláusula assumirá o mesmo valor do índice usado na pesquisa; ou seja, durante a busca, esse identificador acompanhará a variação do índice.

. Sintaxe do comando SEARCH (busca linear)

Na sintaxe apresentada abaixo, o elemento <statement> representa todos os possíveis comandos da linguagem COBOL.

<search> ::= **SEARCH** <identifier>$_1$<body> . |
SEARCH <identifier>$_1$ **AT END** <statement>$_2$<body> .|
SEARCH <identifier>$_1$ **VARYING** <identifier>$_2$<boby> . |
SEARCH <identifier>$_1$ **VARYING** <identifier>$_2$
AT END <statement>$_2$ <body> .
<body> ::= **WHEN** <boolean expression> <statement>$_1$ |
<body> **WHEN** <boolean expression> <statement>$_1$

. Mapeamento do comando SEARCH (busca linear) para a LI

O comando SEARCH é mapeado para o comando 'for' da LI, onde a variável de controle do laço é aquela que faz o papel de índice na pesquisa; dentro do corpo do laço encontra-se um ou mais 'if's relativo a cada cláusula 'WHEN'. A sintaxe do comando 'for' da LI, apresentada no Apêndice A, tem o seguinte formato:

<for> ::= <for_atm> <s1> <cond_for_atm> <s2> <statement>

No grafo do exemplo, apresentado na Figura 3.9, o nó *n*6 não constitui o próximo comando executável após o comando SEARCH; as instruções associadas à cláusula AT END correspondem ao bloco de comandos que forma esse nó. Assim, caso o valor do índice ultrapasse o limite de indexação da tabela -- nenhum elemento da tabela, a partir do valor inicial do índice, satisfez quaisquer das condições envolvidas -- ocorrerá um desvio para o nó *n*6.

No corpo do laço, quando uma condição é avaliada como verdadeira, as instruções associadas ao corpo desse 'if' (no grafo, nó *n*4) são executadas e o controle é desviado para a próxima instrução após o comando SEARCH; no grafo abaixo, ocorre uma saída forçada do laço para o nó *n*7. O comando de saída forçada do laço (linha tracejada no grafo) não caracteriza o comando 'break' da LI, pois este desvia para o próximo nó após o corpo da iteração, o

qual é composto pelas instruções associadas à cláusula AT END. Assim, esse desvio é mapeado para o comando 'goto' da LI com o rótulo *SEARCHnn*, onde *nn* representa a ordem de aparição física do comando SEARCH no programa; esse rótulo localiza-se após os comandos relativos à cláusula AT END.

A Tabela 3.7 apresenta a equivalência sintática entre o COBOL e a LI para o comando SEARCH (busca linear).

. Versão em LI do exemplo

```
$FOR        500 6 15
$S01        0 0 0
$C(01)01    507 13 15
$S02        0 0 0
{           0 0 0
$IF         580 4 17
$C(01)02    589 21 17
{           0 0 0
$S03        633 23 18
$S04        679 21 19
$GOTO       0 0 0
SEARCH01    0 0 0
}           0 0 0
}           700 1 19
$S05        543 23 16
SEARCH01    0 0 0
.
.
.
```

. Versão instrumentada do Exemplo

Para auxiliar a instrumentação do comando SEARCH, criou-se uma sub-rotina específica para cada ocorrência deste comando. Cada sub-rotina é definida pelas instruções que seguem o rótulo *<TABELA>-SEARCH-ll*, onde *<TABELA>* representa os sete primeiros caracteres da tabela alvo do comando e *ll* é um número que identifica a ordem de aparição física.

A partir do índice que indexa a tabela, pode-se concluir o número de execuções do corpo do "for", ou seja, quantas iterações ocorreram durante a pesquisa. Inicialmente, torna-se necessário "guardar" o valor do índice. A diferença entre o valor inicial e o valor final do índice ao final da pesquisa fornece o número de vezes de execução do laço. Uma variável auxiliar (IND-AUX) é utilizada para "guardar" o valor do índice antes da pesquisa ser realizada; ao final da pesquisa, o número de iterações efetuadas no corpo do laço pode ser calculado através da expressão 'IND - IND-AUX'.

Tabela 3.7: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando SEARCH (busca serial).

COBOL	LI
SEARCH ---------->	<for_atm>
<identifier>$_1$ ---------->	<s1> <cond_for_atm> <s2>
<identifier>$_1$ **VARYING** <identifier>$_2$ -->	<s1> <cond_for_atm> <s2>
WHEN ---------->	<if_atm>
<boolean expression> ---------->	<cond_atm> <{>
<statement>$_1$ ---------->	<statement> <goto_atm> SEARCHnn <}>
<statement>$_2$ ---------->	SEARCHnn <statement>
. ---------->	<}>

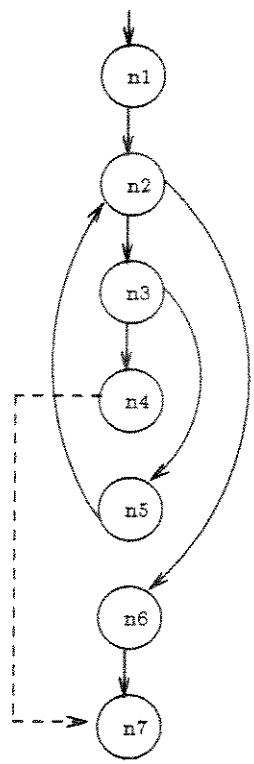


Figura 3.9: Grafo de Fluxo de Controle: Comando SEARCH (busca serial).

```
000100** n1 **
000101      ponta_de_prova (n1)
            .
            .
000102** n2 **
000103** n5 **
000104      SET IND-AUX TO IND
000105      SEARCH TAB-HOMONIMOS
000106        AT END   PERFORM TAB-HOM-SEARCH-01
000107                   (IND - IND-AUX) TIMES
000108                 ponta_de_prova (n2)
000109** n6 **
000110                 ponta_de_prova (n6)
000111                 MOVE 0 TO ACHA-HOMONIMO
000112        WHEN     RG (IND) EQUAL M-RG
000113                 PERFORM TAB-HOM-SEARCH-01
000114                   (IND - IND-AUX) TIMES
000115                 ponta_de_prova (n2)
000116                 ponta_de_prova (n3)
000117** n4 **
000118                 ponta_de_prova (n4)
000119                 PERFORM FORMATA-CLIENTE
000120                   PERFORM EXIBE-CLIENTE.
            .
            .
            .
            .
000200 TAB-HOM-SEARCH-01.
000201      ponta_de_prova (n2)
000202      ponta_de_prova (n3)
000203      ponta_de_prova (n5).
```

. **Diretrizes para a Construção do Grafo Def para o comando SEARCH (busca serial)**

Observando o grafo da Figura 3.9, o nó *n*1 representa a iniciação da variável contadora do "for" e o nó *n*5 representa o comando seqüencial que altera essa variável a cada iteração do "for". Ao nó *n*5 é associada a variável utilizada como índice durante a pesquisa; ao nó *n*1 não é associada qualquer variável, visto que a pesquisa é iniciada a partir do valor corrente do índice. Os nós *n*2 e *n*3 representam expressões condicionais e a esses nós não é associada qualquer definição de variável. Aos nós *n*4 e *n*6 somente é atribuído o conjunto de variáveis definidas devido aos comandos relativos a esses nós ou subgrafos correspondentes. Ao nó *n*7 é atribuído o conjunto de variáveis definidas devido ao bloco de comandos associado a esse nó.

3.4.3.5 Comando SEARCH (busca binária)

Neste formato, o comando SEARCH é usado para pesquisar numa tabela , através de busca binária, por um elemento que satisfaça uma condição específica. Antes de iniciar a pesquisa, a tabela deve estar ordenada por uma chave explicitamente indicada. As condições de parada da pesquisa são as mesmas aplicadas para o comando em busca serial (Seção 3.4.3.4).

. Exemplo do Comando SEARCH (busca binária)

No exemplo abaixo, a tabela TAB-HOMONIMOS é definida para que seus elementos estejam ordenados, de modo crescente, pela variável RG.

```
000050 01 TAB-HOMONIMOS OCCURS 50 TIMES
000051                    ASCENDING KEY IS RG
000052                    INDEXED BY IND.
000053    02 NOME       PIC X(40).
000054    02 DAT-NAS    PIC 9(06).
000055    02 RG         PIC 9(10).
000056    02 CIC        PIC 9(11).
           .
           .
           .
000100      SEARCH ALL TAB-HOMONIMOS
000101        AT END  MOVE 0 TO ACHA-HOMONIMO
000102        WHEN    RG (IND)  EQUAL M-RG
000103                PERFORM FORMATA-CLIENTE
000104                PERFORM EXIBE-CLIENTE.
```

. Sintaxe do comando SEARCH (busca binária)

Na sintaxe apresentada abaixo, o elemento <statement> representa todos os possíveis comandos da linguagem COBOL. Em relação à sintaxe apresentada para o formato anterior -- comando SEARCH (busca serial) -- acrescenta-se a cláusula ALL, a qual indica que a busca abrange toda a tabela; assim, o valor inicial da variável utilizada como índice não será usado para indicar a posição da tabela onde será iniciada a pesquisa. Uma outra diferença fundamental consiste na existência de apenas uma cláusula WHEN para esse formato do comando SEARCH; somente uma expressão condicional é avaliada na pesquisa.

```
<search> ::= SEARCH ALL <identifier>1 <body> . !
             SEARCH ALL <identifier>1 AT END <statement>2 <body> . !
             SEARCH ALL <identifier>1 VARYING <identifier>2 <boby> . !
             SEARCH ALL <identifier>1 VARYING <identifier>2
               AT END <statement>2 <body> .
<body>   ::= WHEN <boolean expression> <statement>1
```

. Mapeamento do Comando SEARCH (busca binária) para a LI

As considerações de mapeamento para a LI realizadas para a pesquisa serial (Seção 3.4.3.4) são também aplicadas para a pesquisa binária. A Tabela 3.8 contém a equivalência sintática entre o COBOL e a LI para o comando SEARCH (busca binária). O grafo de fluxo de controle correspondente ao exemplo é apresentado na Figura 3.10.

. Versão em LI do Exemplo

```
$FOR        500 10 15
$S01        0 0 0
$C(01)01    511 13 15
$S02        0 0 0
{           0 0 0
$IF         584 4 17
$C(01)02    593 21 17
{           0 0 0
$S03        637 23 18
$S04        683 21 19
$GOTO       0 0 0
SEARCH01    0 0 0
}           0 0 0
}           704 1 19
$S05        543 23 16
SEARCH01    0 0 0
.
.
.
```

. Versão Instrumentada do Exemplo

A pesquisa binária, implementada pela linguagem COBOL para o comando SEARCH-ALL, inicialmente verifica se o elemento central da tabela satisfaz a condição associada à cláusula WHEN. Em caso negativo, se o elemnto procurado é menor que o elemento central, a pesquisa continua considerando somente a metade inferior da tabela ou, por outro lado, a metade superior da tabela. O processo se repete até que, após repetidas divisões da tabela, o elemento central da porção considerada da tabela corresponda ao elemento procurado, ou a porção considerada da tabela se torne vazia.

Para instrumentar o comando SEARCH-ALL, são utilizadas as variáveis auxiliares LIM-INF, LIM-SUP e MEIO, que armazenam, respectivamente, o limite inferior da tabela, o limite superior da tabela, e o índice que aponta para o elemento central da porção da tabela definida pelas variáveis LIM-INF e LIM-SUP. Os valores inicialmente associados a essas variáveis são atribuídos imediatamente antes do comando SEARCH-ALL. A sub-rotina *<TABELA>-SEARCH-II*, citada na instrumentação do comando SEARCH (busca serial), que registra a execução completa do corpo do laço, também realiza a divisão da tabela para determinar a porção pertinente da tabela para onde o elemento procurado

Tabela 3.8: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando SEARCH (busca binária).

COBOL	LI
SEARCH ALL ---------->	<for_atm>
<identifier>$_1$ ---------->	<s1> <cond_for_atm> <s2>
<identifier>$_1$ **VARYING** <identifier>$_2$ -->	<s1> <cond_for_atm> <s2>
WHEN ---------->	<if_atm>
<boolean expression> ---------->	<cond_atm> <{>
<statement>$_1$ ---------->	<statement> <goto_atm> SEARCHnn <}>
<statement>$_2$ ---------->	SEARCHnn <statement>
. ---------->	<}>

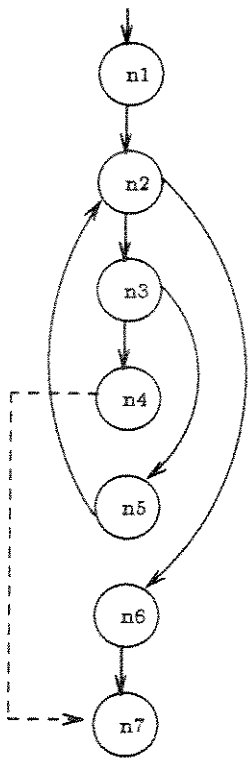


Figura 3.10: Grafo de Fluxo de Controle: Comando SEARCH (busca binária).

converge, como pode ser observado na sub-rotina TAB-HOM-SEARCH-01 da versão instrumentada do exemplo, apresentada a seguir:

```
000100** n1 **
000101      ponta_de_prova (n1)
            :
000102** n2 **
000103** n5 **
000104      MOVE 1 TO LIM-INF
000105      MOVE 50 TO LIM-SUP
000106      COMPUTE MEIO = (LIM-INF + LIM-SUP) / 2
000107      SEARCH ALL TAB-HOMONIMOS
000108        AT END  PERFORM TAB-HOM-SEARCH-01
000109                  UNTIL LIM-SUP LESS LIM-INF
000110                ponta_de_prova (n2)
000111** n6 **
000112                ponta_de_prova (n6)
000113                MOVE 0 TO ACHA-HOMONIMO
000114        WHEN    RG (IND)  EQUAL M-RG
000115                PERFORM TAB-HOM-SEARCH-01
000116                  UNTIL MEIO EQUAL IND
000117                ponta_de_prova (n2)
000118                ponta_de_prova (n3)
000119** n4 **
000120                ponta_de_prova (n4)
000121                PERFORM FORMATA-CLIENTE
000122                PERFORM EXIBE-CLIENTE.
            :
            :
000200 TAB-HOM-SEARCH-01.
000201      ponta_de_prova (n2)
000202      ponta_de_prova (n3)
000203      ponta_de_prova (n5)
000204      COMPUTE MEIO = (LIM-INF + LIM-SUP) / 2
000205      IF MEIO LESS IND
000206         COMPUTE LIM-INF = MEIO + 1
000207      ELSE
000208         COMPUTE LIM-SUP = MEIO - 1.
```

Note que existem duas chamadas à sub-rotina TAB-HOM-SEARCH-01, situadas nas linhas de seqüência 108 e 115, respectivamente. A primeira refere-se à situação onde o elemento procurado não se encontra na tabela, ou seja, essa sub-rotina é executada até que a expressão condicional 'LIM-SUP LESS LIM-INF' seja verdadeira. A segunda ocorre quando o elemento procurado pertence à tabela e, nesse caso, a sub-rotina é executada até que a expressão condicional 'MEIO EQUAL IND' seja satisfeita.

. Diretrizes para a Construção do Grafo Def para o comando SEARCH (busca binária)

Observando o grafo da Figura 3.10, o nó *n*1 representa a iniciação da variável contadora do "for" e o nó *n*5 representa o comando seqüencial que altera essa variável a cada iteração do "for". Aos nós *n*1 e *n*5 é associada a variável utilizada como índice durante a pesquisa. Os nós *n*2 e *n*3 representam expressões condicionais e a esses nós não é associada qualquer definição de variável. Aos nós *n*4 e *n*6 somente é atribuído o conjunto de variáveis definidas devido aos comandos relativos a esses nós ou subgrafos correspondentes. Ao nó *n*7 é atribuído o conjunto de variáveis definidas devido ao bloco de comandos associado a esse nó.

3.4.4 COMANDOS DE DESVIO

Os comandos GOTO (simples) e GOTO-DEPENDING, apresentados a seguir, formam os comandos de desvio da linguagem COBOL.

3.4.4.1 Comando GOTO (simples)

Em sua forma mais simples, o comando GOTO causa a transferência incondicional de controle para um rótulo do programa (parágrafo ou seção). A transferência de controle realizada por esse comando deve restringir-se aos limites da unidade a que pertencem. No entanto, existem programas, principalmente os mais antigos, que não obedecem tal restrição; na análise individual de unidades, qualquer desvio dessa natureza não ocasiona prejuízo na adequação de um conjunto de casos de teste aos critérios Potenciais Usos, pois os caminhos e as associações requeridos simplesmente deixariam de ser satisfeitas.

. Exemplo do Comando GOTO (simples)

```
                :
000100          GOTO PROCESSA-EXIT
                :
000200 PROCESSA-EXIT.
```

. Sintaxe do Comando GOTO (simples)

<goto> ::= **GOTO** <identifier>

. Mapeamento do Comando GOTO (simples) para a LI

A semântica do comando GOTO (simples) não difere do tradicional "goto" das linguagens do estilo ALGOL, sendo mapeado para o comando 'goto' da LI; a sintaxe do comando 'goto' da LI, apresentada no apêndice A, tem o seguinte formato:

<goto> ::= <goto_atm> <label_atm>

A equivalência sintática entre o COBOL e a LI, para o comando GOTO (simples), é mostrada na Tabela 3.9. O grafo de fluxo de controle correspondente ao exemplo é apresentado na Figura 3.11.

. Versão em LI do Exemplo

```
.
.
$GOTO            500 4 15
PROCESSA-EXIT  505 13 15
.
.
.
PROCESSA-EXIT  700 13 30
```

. Versão Instrumentada do Exemplo

```
                .
                .
000100** n1 **
000101          ponta_de_prova (n1)
                .
000102          GOTO PROCESSA-EXIT
                .
                .
                .
                .
                .
000200 PROCESSA-EXIT.
000201** n2 **
000202          ponta_de_prova (n2)
                .
                .
```

. Diretrizes para a Construção do Grafo Def para o Comando GOTO (simples)

O comando GOTO do exemplo pertence ao nó *n*1 do grafo da Figura 3.11; a esse nó é atribuído o conjunto de variáveis definidas devido a outros comandos associados a esse nó.

Tabela 3.9: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando GOTO (simples).

COBOL		LI
GOTO	---------->	<goto_atm>
<identifier>	------->	<label_atm>

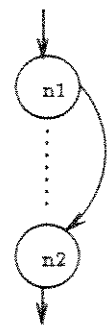


Figura 3.11: Grafo de Fluxo de Controle: Comando GOTO (simples).

3.4.4.2 Comando GOTO-DEPENDING

O comando GOTO associado à cláusula DEPENDING constitui uma estrutura de transferência de controle para um entre vários rótulos do programa. Uma variável é utilizada na escolha do rótulo para onde haverá o desvio do fluxo de execução; para cada rótulo presente no comando está associado um valor dessa variável. Caso ela possua um valor que não referencie um dos rótulos do comando, o controle é transferido para o próximo comando após o comando GOTO-DEPENDING.

. Exemplo do Comando GOTO-DEPENDING

```
              :
              :
000100        GOTO INCLUI-REGISTRO
000101             REMOVE-REGISTRO
000102          DEPENDING ON COD-REG
              :
```

No exemplo, se COD-REG = 1, o controle é desviado para o rótulo INCLUI-REGISTRO; se COD-REG = 2, ocorre desvio para o rótulo REMOVE-REGISTRO; caso contrário, a execução prossegue para a próxima sentença executável após o comando GOTO-DEPENDING.

. Sintaxe do Comando GOTO-DEPENDING

<goto> ::= **GOTO** <identfier_list> **DEPENDING ON** <identifier>$_2$

<identifier_list> ::= <identifier>$_1$ ¦
<identifier_list> <identifier>$_1$

. Mapeamento do Comando GOTO-DEPENDING para a LI

O comando GOTO-DEPENDING é considerado como um comando de seleção múltipla; portanto, equivale ao comando 'case' da LI; a sintaxe do comando 'case' da LI, apresentada no Apêndice A, é a seguinte:

<case> ::= <case_atm> <case_cond_atm> <aux1>

<aux1> ::= <aux2> <statement> ¦
<aux1> <aux2> <statement>

<aux2> ::= <rotc_atm> ¦ <rotd_atm>

A equivalência sintática entre o COBOL e a LI, para o comando GOTO-DEPENDING, é mostrada na Tabela 3.10; o símbolo ø indica que o átomo da LI não possui

correspondência direta no código fonte. O grafo de fluxo de controle correspondente ao exemplo é apresentado na Figura 3.12.

. Versão em LI do Exemplo

```
    .
    .
    .
    $CASE          500 4 15
    $CC            579 7 17
    {              0 0 0
    $ROTC          0 0 0
    $GOTO          0 0 0
    INCLUI-REGISTRO 505 15 15
    $ROTC          0 0 0
    $GOTO          0 0 0
    DELETA-REGISTRO537 15 16
    }              0 0 0
    .
    .
    .
```

. Versão Instrumentada do Exemplo

Para instrumentar o comando GOTO-DEPENDING, criou-se uma sub-rotina específica para cada ocorrência do comando. Essa sub-rotina é definida pelos comandos que seguem o rótulo *<IDENT>-GOTODE-ii*, onde *<IDENT>* constitui os sete primeiros caracteres da variável que controla o desvio e *ii* é um número que identifica a ordem de aparição física

```
000100** n1 **
000101       ponta_de_prova (n1)
             .
             .
000102       PERFORM COD-REG-GOTODE-01
000103** n2 **
000104** n3 **
000105       GOTO INCLUI-REGISTRO
000106            REMOVE-REGISTRO
000107         DEPENDING ON COD-REG.
000108       ponta_de_prova (n4)
             .
             .
000200 COD-REG-GOTODE-01.
000201       IF COD-REG EQUAL 1
000202          ponta_de_prova (n2).
000203       IF COD-REG EQUAL 2
000204          ponta_de_prova (n3).
```

. Diretrizes para a Construção do Grafo Def para o Comando GOTO-DEPENDING

Ao nó *n1* é atribuído o conjunto de variáveis definidas no bloco de comandos associado a esse nó; aos nós *n2* e *n3* não é atribuída qualquer definição de variável; ao nó *n4* é associado o conjunto de variáveis definidas neste nó.

Tabela 3.10: Equivalência Sintática entre a Linguagem COBOL e a Linguagem LI: Comando GOTO-DEPENDING.

COBOL	LI
GOTO ---------->	<case_atm>
<identifier>$_1$ -------->	<rotc_atm> <goto_atm> <label_atm>
<identifier>$_2$ -------->	<case_cond_atm> <{>
ø ---------->	<}>

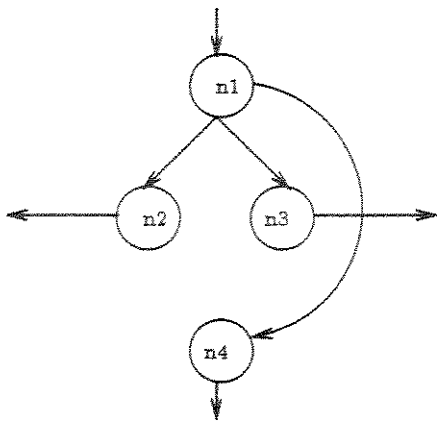


Figura 3.12: Grafo de Fluxo de Controle: Comando GOTO-DEPENDING.

3.5 CONSIDERAÇÕES FINAIS

Os aspectos de implementação dos critérios Potenciais Usos para uma linguagem particular são obtidos através da aplicação dos modelos de implementação desses critérios e constituem uma atividade fundamental na configuração da POKE-TOOL. Foram instanciados, para a linguagem COBOL, os modelos de implementação dos critérios Potenciais Usos.

A caracterização de unidade em linguagem COBOL mostrou-se dependente da estrutura de chamada presente em um programa; tal abordagem foi adotada, pois a linguagem COBOL não possui declarações explícitas de procedimentos e funções. O conceito de unidade, proposto neste capítulo, difere da noção de unidade presente em linguagens de estilo ALGOL, pois: inexistem variáveis de escopo local, e podem ocorrer transferências incondicionais de controle para fora do escopo de uma unidade e sobreposição de unidades.

No contexto de configuração da POKE-TOOL, a caracterização das estruturas de controle que compõem uma linguagem é estabelecida através de um mapeamento entre os aspectos de controle da linguagem alvo da configuração e da linguagem LI. O relacionamento semântico entre as estruturas de controle da linguagem COBOL e da linguagem LI não resulta em uma correspondência direta, diferentemente do observado na configuração da POKE-TOOL para a linguagem C [CHA91b]; alguns comandos da linguagem COBOL implementam estruturas de controle complexas, como, por exemplo, o comando PERFORM-VARYING que implementa uma estrutura de laços aninhados.

A instrumentação de unidades COBOL não constitui uma tarefa trivial; a inexistência de estruturas de bloco dificulta a monitoração dos nós que compõem as estruturas de controle que implementam laços (comandos PERFORM-UNTIL, PERFORM-TIMES, PERFORM-VARYING e SEARCH) e desvios para um dentre vários rótulos (comando GOTO-DEPENDING); foram criadas sub-rotinas dedicadas à monitoração desses nós.

Na instanciação do modelo de fluxo de dados não foi necessário considerar a visibilidade de variáveis, pois todas as variáveis são globais às unidades de um programa. Cada vez que uma unidade é invocada, as variáveis compartilhadas entre a unidade chamadora e a unidade chamada são ditas definidas por referência na análise da primeira unidade; assim, ao nó que contém uma invocação de unidade é atribuída a definição por referência de todas as variáveis do programa. Na análise individual de unidades, todas as variáveis são consideradas definidas no nó de entrada de cada unidade; uma exceção ocorre na unidade de controle principal de um programa, onde somente

são consideradas definidas no nó inicial da unidade as variáveis explicitamente definidas quando declaradas.

Ainda com respeito ao modelo de fluxo de dados, no tratamento de variáveis estruturadas, estabeleceu-se um enfoque onde as variáveis componentes de um registro são consideradas como variáveis simples e, por isso, passíveis de definição; uma ocorrência de definição em uma variável do tipo registro resulta na definição de suas variáveis componentes. No caso de vetores e matrizes, assumiu-se um enfoque simplificado, onde uma ocorrência de definição de um elemento de um vetor resulta na definição do vetor.

CAPÍTULO 4

ASPECTOS DE IMPLEMENTAÇÃO DA CONFIGURAÇÃO DA POKE-TOOL PARA A LINGUAGEM COBOL

Basicamente, para configurar a POKE-TOOL, é necessário que o usuário configurador especifique o analisador léxico, o analisador sintático e as rotinas necessárias à instrumentação da linguagem alvo. Em [CHA91a] é descrito como realizar tais tarefas; uma síntese desse processo foi fornecida na Seção 2.6. Este capítulo apresenta os aspectos essenciais da implementação da configuração da POKE-TOOL para a linguagem COBOL.

A utilização de uma ferramenta que apóia o teste estrutural de unidades pode permitir ao usuário fazer a análise de cobertura de várias unidades simultaneamente; essa sofisticação foi incorporada à POKE-TOOL durante a sua configuração para a linguagem COBOL. Em decorrência desse fato, algumas modificações foram realizadas no projeto original da POKE-TOOL; tais modificações são apresentadas na Seção 4.1.

A Seção 4.2 aborda a preparação do analisador léxico e é complementada pelo Apêndice B, que contém a tabela de transições léxicas da linguagem COBOL. A configuração do analisador sintático da POKE-TOOL é explorada na Seção 4.3; a descrição gramatical da linguagem COBOL, mostrada no Apêndice C, levou em consideração a existência de vários dialetos para a linguagem COBOL [BRA89]. Os procedimentos associados à instrumentação do código fonte são abordados na Seção 4.4.

4.1 MODIFICAÇÃO NO PROJETO DA POKE-TOOL

Na Seção 3.1 foi caracterizado o conceito de unidade de programa em linguagem COBOL; um programa COBOL pode ser decomposto em várias unidades, que são definidas a partir da estrutura de chamada de procedimentos. A atividade de teste apoiada pela ferramenta POKE-TOOL envolve a definição dos elementos requeridos para a unidade em análise e posterior avaliação dos casos de teste aplicados. Na configuração da POKE-TOOL para a linguagem COBOL, foram caracterizados os elementos requeridos para cada unidade de programa, bem como as unidades presentes em um programa foram todas instrumentadas, permitindo ao usuário fazer a análise de cobertura de várias unidades simultaneamente. Essa abordagem está de acordo com a filosofia normalmente utilizada no teste em linguagem COBOL, que consiste em considerar-se um módulo em teste como um programa inteiro; ainda, constitui uma característica desejável em uma ferramenta de teste [HOR92].

Esta seção apresenta as modificações realizadas no projeto da POKE-TOOL, com o intuito de testar várias unidades concomitantemente; tal característica requer a aplicação das funções associadas à definição de elementos requeridos e à avaliação dos casos de teste nas unidades escritas em linguagem COBOL.

A POKE-TOOL é composta de vários módulos que se comunicam através de arquivos. Esses módulos implementam funções ou parte de funções, as quais foram descritas na Seção 2.4. Na Figura 4.1 é apresentado um diagrama contendo os módulos e os diversos produtos gerados. Nessa figura, os *retângulos* representam os módulos, os *losangos* representam as entradas fornecidas pelos usuários à ferramenta e os *círculos* os produtos gerados; as linhas tracejadas representam o *fluxo de controle* e as linhas cheias o *fluxo de informação*.

O módulo **poketool** é o responsável pela comunicação entre a ferramenta e o usuário e pela seqüenciação das atividades de teste através da ativação dos demais módulos. O módulo **li** realiza a tradução da unidade para uma nova versão escrita na linguagem intermadiária (LI). O módulo **chanomat** gera o fluxo de controle da unidade e uma nova versão da unidade em LI, versão numerada, onde cada comando da LI está associado ao nó correspondente. Os módulos **li** e **chanomat** implementam a função *Grafo de Fluxo de Controle* da POKE-TOOL. O módulo **pokernel** é o responsável pelo restante da análise estática da unidade em teste, onde são implementadas as seguintes funções da POKE-TOOL: *Cálculo dos Arcos Primitivos*, *Extensão do Grafo de Fluxo de*

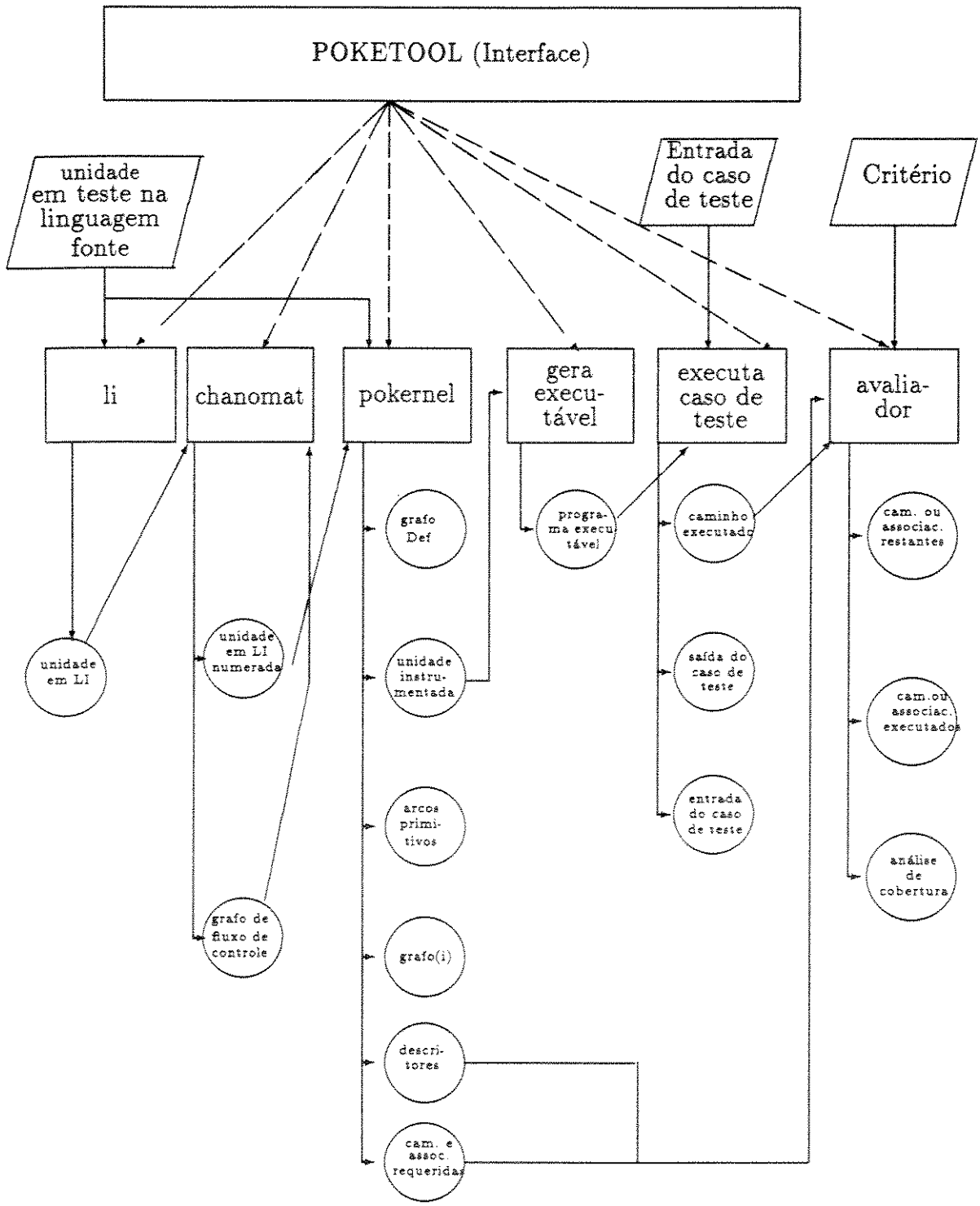


Figura 4.1: Projeto da POKE-TOOL

Controle, *Instrumentação*, *Construção do Grafo (i) e Geração dos Descritores*. O módulo **gera executável** fornece as condições para a geração do programa executável da versão instrumentada e engloba a função *Compilador Selecionado*. O módulo **executa caso de teste**, como o próprio nome diz, controla a execução dos casos de teste salvando as entradas, a saída e o caminho executado para cada caso de teste; implementa a função *Execução do Programa* da POKE-TOOL. Finalmente, o módulo **avaliador** verifica quais os caminhos ou associações executados pelo caso de teste e fornece uma análise de cobertura do conjunto de casos de teste fornecido; implementa a função *Avaliação* da POKE-TOOL.

A Figura 4.2 introduz as modificações realizadas no projeto da POKE-TOOL; nessa figura, os círculos concatenados representam um conjunto de arquivos, onde cada arquivo refere-se a uma unidade do programa. As alterações efetuadas são apresentadas a seguir.

O módulo **li** não sofreu alteração na geração da versão em linguagem intermediária; no entanto, o código em LI gerado reflete o programa inteiro, onde todas as unidades estão presentes. Adicionalmente, é gerada uma tabela de invocações de unidades, que, posteriormente, será utilizada na determinação das unidades presentes no programa.

O módulo **chanomat** é conservado em sua versão original mas, a exemplo do módulo li, os produtos gerados dizem respeito ao programa inteiro em teste; o programa em LI é numerado e o grafo de fluxo de controle equivalente é produzido. Uma vez que a versão numerada é gerada, as unidades presentes no programa podem ser identificadas a partir de intervalos de nós correspondentes.

O módulo **pokernel** foi subdividido em dois módulos -- **kernel e descr** -- e o módulo **separa** foi criado. O módulo **kernel** é responsável pela geração do grafo def do programa inteiro, englobando as funções *Extensão do Grafo de Fluxo de Controle* e *Instrumentação* da POKE-TOOL. O módulo **separa** realiza a decomposição do programa em unidades, onde são criados subdiretórios ao diretório corrente, que conterão informação relativa a cada unidade do programa; adicionalmente, são produzidos uma tabela de unidades e, para cada subdiretório criado, o grafo def e o grafo de fluxo de controle correspondente a uma unidade do programa. O módulo **descr** implementa as funções *Cálculo dos Arcos Primitivos* e *Construção dos Grafo (i)*. Esse módulo é chamado repetidamente, onde cada invocação produz informação associada a uma unidade no subdiretório correspondente.

Os módulos **gera executável** e **executa caso de teste** são conservados em sua forma original; o arquivo de caminhos executados produzido pelo segundo módulo diz respeito aos nós do grafo de fluxo de controle do programa

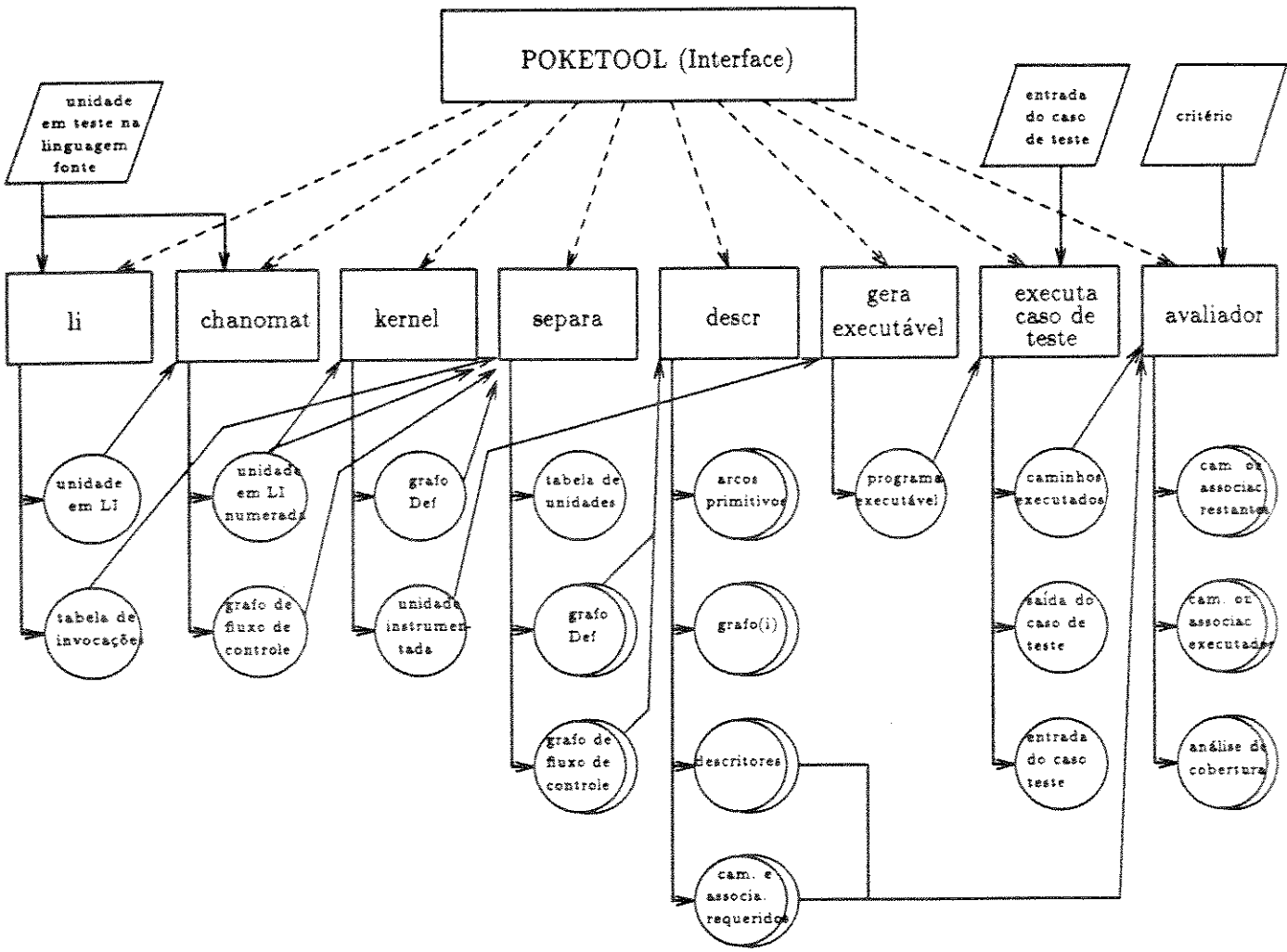


Figura 4.2: Projeto Modificado da POKE-TOOL

inteiro, abrangendo, portanto, todas as unidades.

O módulo **avaliador**, que implementa a função *Avaliação* da POKE-TOOL, é invocado para cada unidade do programa. Os caminhos percorridos pelos casos de teste são selecionados a partir do arquivo de caminhos executados do programa e, então, o subconjunto resultante é utilizado na avaliação da unidade em análise. Os resultados produzidos são também organizados nos subdiretórios referentes às unidades do programa.

4.2 ANÁLISE LÉXICA

O passo inicial na análise de código fonte é o processo de análise léxica. A análise léxica separa os itens constituintes de uma linguagem -- itens léxicos -- e determina suas classes.

A POKE-TOOL implementa um algoritmo genérico [SET81] que utiliza autômatas finitos na análise léxica. Esse algoritmo interpreta um autômata finito cujas transições foram organizadas numa estrutura de dados criada através de uma entrada do usuário. A interpretação do autômata é dirigida pela estrutura de dados que representa as transições e por caracteres lidos do programa fonte. Associadas às transições do autômato estão as *ações semânticas*, isto é, rotinas que são executadas quando uma determinada transição ocorre. Por exemplo, uma rotina que verifica se um identificador é uma palavra reservada ou não, é muito útil, pois evita que sejam criadas transições e estados para identificar cada palavra reservada. Assim, o usuário configurador da POKE-TOOL deve especificar a Tabela de Transições Léxicas, a Tabela de Palavras Reservadas e as Ações Semânticas; as duas tabelas foram definidas para a linguagem COBOL segundo as referências [IBM87,UNI87,MIC88].

O Apêndice B contém a tabela de transições léxicas definida para a linguagem COBOL. Cada linha da tabela (transição do autômato) é composta de cinco campos: argumento de comparação com o caractere lido, estado corrente, próximo estado, número da ação semântica que será executada e índice relativo ao próximo estado. Como exemplo, considere a quarta linha da tabela: caso o caractere lido seja uma letra, ocorrerá a transição do estado 0 para o estado 10, a ação semântica 1 será executada e as transições do estado 10 localizam-se após a vigésima terceira linha da tabela. Na tabela pode-se observar que as transições que têm o mesmo estado corrente estão agrupadas

consecutivamente e que existe apenas um estado inicial (estado 0) e um estado final (estado 1000).

As ações semânticas são rotinas que não recebem argumentos e nem retornam valor; portanto, elas executam suas tarefas manipulando variáveis globais. Na configuração da POKE-TOOL para a linguagem COBOL foram especificadas 14 ações semânticas. Como exemplo, será apresentada, a seguir, a ação semântica que decide se um item léxico pode ser caracterizado como um identificador.

VARIÁVEIS GLOBAIS:

saida : estrutura de dados composta de 5 campos. O primeiro campo se chama *label* e deve armazenar o item léxico propriamente dito. O segundo campo, *classe*, vai armazenar a classe do item léxico. Os demais campos são *inicio, comprimento e linha*, e como os próprios nomes revelam, dizem respeito à posição no arquivo fonte onde se inicia o item léxico, ao comprimento do item léxico e à linha onde se inicia o item léxico, respectivamente.

FUNÇÕES OU PROCEDIMENTOS CHAMADOS:

pesq_tab(cadeia) : pesquisa a variável **cadeia** na tabela de palavras reservadas; retorna TRUE se o elemento encontra-se na tabela; FALSE, caso contrário.

dev_ch : "devolve" o último caractere lido para o arquivo fonte, e esse caractere fica disponível para compôr outro item léxico; a leitura de um caractere além dos limites do item léxico é neceesária para determinar o final do item léxico.

calcula_posicao : preenche os campos início, comprimento e linha da variável global **saida**.

```
begin
  if pesq_tab(saida.label) then
    saida.classe := saida.label
  else
    saida.classe := "IDENT";
  dev_ch;
  calcula_posicao;
end
```

4.3 ANÁLISE SINTÁTICA

O processo de reconhecimento de sentenças em uma linguagem qualquer é conhecido como análise sintática. A POKE-TOOL realiza a análise sintática do programa fonte a ser testado, reconhecendo as sentenças da linguagem e extraindo informações essenciais, através da associação de rotinas, ao reconhecimento dessas sentenças.

O analisador sintático implementado pela POKE-TOOL segue a abordagem proposta por Wirth [WIR76]; o processo é dirigido por um grafo sintático da linguagem alvo da configuração. Esse grafo é colocado na memória através da leitura de um arquivo que contém a descrição da gramática a ser reconhecida. O que ocorre é que a POKE-TOOL além de reconhecer sentenças de uma linguagem, obtém também informações a partir do reconhecimento das sentenças. O algoritmo utilizado permite que rotinas sejam executadas quando é reconhecido um elemento da gramática da linguagem; essas rotinas são denominadas *rotinas semânticas*. Assim, a configuração do analisador sintático da POKE-TOOL consiste em definir a gramática da linguagem e especificar rotinas semânticas que serão invocadas ao se reconhecer sentenças da linguagem.

A estrutura gramatical utilizada na configuração do analisador sintático segue o padrão ANSI 74 da linguagem COBOL. O que se verifica, na prática, é que existem vários dialetos da linguagem COBOL [BRA89] e que, para uma definição compatível entre diversos compiladores, é fundamental considerar as diferenças existentes, bem como traduzi-las em construções sintáticas. Várias foram as referências utilizadas no entendimento dos comandos que compõem a linguagem COBOL e na construção de suas regras sintáticas [IBM87,UNI87,MIC88,PAQ89].

A forma utilizada pela POKE-TOOL para descrever uma gramática é baseada na notação estendida de Backus-Naur [SEB89]. A diferença básica entre a notação de Backus-Naur e a notação entendida pela POKE-TOOL consiste nos meta-símbolos: "=" indica a atribuição de uma regra gramatical; "," representa a ocorrência de sentenças alternativas; "[" e "]" limitam uma estrutura que pode ocorrer zero ou mais vezes; e "*" designa que o reconhecimento da regra gramatical é opcional. Os elementos entre plicas ("'") indicam os símbolos terminais, ou seja, as classes de itens léxicos caracterizados na análise léxica; os demais elementos auxiliam a definição gramatical da linguagem. Ainda, a notação aceita pela POKE-TOOL requer que cada regra sintática seja terminada por um ponto (".". O Apêndice C

apresenta a especificação gramatical da linguagem COBOL na notação acima mencionada.

Na Seção 2.6 foram apresentadas as tarefas necessárias à configuração da POKE-TOOL, onde os módulos **li** e **pokernel** da ferramenta requerem a preparação do analisador sintático. Portanto, devem ser especificadas rotinas semânticas para a geração da versão em LI do programa e para a geração do grafo def. Note que, como visto na Seção 4.1, o módulo **pokernel** foi subdividido, resultando nos módulos **kernel** e **descr**. O módulo **kernel** implementa as funções Extensão do Grafo de Fluxo de Controle e Instrumentação, sendo, portanto, alvo da atividade de configuração. Desse modo, o analisador sintático deve ser especificado para os módulos **li** e **kernel** da POKE-TOOL.

Na configuração do módulo **li** foram especificadas 40 rotinas semânticas. Como exemplo, considere o reconhecimento do item léxico relativo à condição do comando IF da linguagem COBOL. Observando o quadro de equivalência sintática entre o COBOL e a LI para o comando IF (Seção 3.4), o reconhecimento da condição associada a esse comando conduz à geração dos átomos <cond_atm> e <{> da LI. A rotina semântica que realiza essa tarefa é apresentada abaixo; as variáveis, os procedimentos e as funções utilizados são introduzidos inicialmente.

VARIÁVEIS GLOBAIS:

pred : armazena o número de predicados da condição.

open_blk : indica o número de blocos iniciados na versão em LI do código fonte.

FUNÇÕES OU PROCEDIMENTOS CHAMADOS:

grava_li(qtde,atomo) : grava, no arquivo que contém o fonte modificado, o elemento da LI descrito pela variável **atomo**. O parâmetro **qtde** informa a quantidade de elementos que compõem o símbolo do átomo gravado. No caso de uma condição, além do indicativo de condição ($C), são também considerados o número de predicados da condição e a ordem de aparição física dessa condição no arquivo fonte (ver Apêndice A).

zera_apont : zera os apontadores de início, comprimento e linha, que descrevem o código fonte mapeado para um átomo da LI; esse procedimento se aplica aos casos onde não existe uma correspondência direta da linguagem fonte para algum átomo do comando da LI correspondente.

calcula_posicao : inicia os apontadores de início, comprimento e linha, que descrevem o código fonte mapeado para algum átomo da LI; esse

procedimento é utilizado na iniciação de um novo ponto de mapeamento.

```
{ escreve <cond_atm> e <{> no fonte modificado }

begin
  grava_li(3,"$C");
  pred := 1;
  zera_apont;
  grava_li(1,"{");
  open_blk := open_blk + 1;
  calcula_posicao;
end
```

As rotinas semânticas especificadas para o módulo **kernel** são as responsáveis pela identificação das variáveis que são definidas em um dado comando. Dentre as 27 rotinas definidas, apresentamos, a seguir, a rotina semântica que trata uma ocorrência de definição de uma variável do tipo registro. Essa rotina utiliza uma estrutura de dados que descreve a hierarquia de variáveis, ou seja, uma estrutura do tipo árvore onde os componentes de um registro são determinados pelos elementos subordinados ao elemento que descreve esse registro.

VARIÁVEIS GLOBAIS:

reg : cadeia de caracteres que descreve a variável do tipo registro.

tree_var : apontador para um elemento da árvore de variáveis declaradas. Cada elemento da árvore é composto pelo campos *id*, *prox* e *comp*: id armazena um número inteiro de identifica unicamente a variável; prox aponta para o próximo elemento da árvore que descreve uma variável de mesmo nível e, também está subordinada ao mesmo registro; e comp aponta para o primeiro elemento componente, caso o elemento descrito caracterize um registro.

num_no : número do nó onde ocorreu a definição de variável.

VARIÁVEIS LOCAIS:

tree_var_aux : armazena o apontador para o elemento da árvore de variáveis declaradas, o qual descreve o registro definido.

FUNÇÕES OU PROCEDIMENTOS CHAMADOS:

retr_tree_var(descr) : retorna um apontador para o elemento da árvore de variáveis declaradas, o qual é descrito por **descr**.

def_record(tree_var,num_no) : inclui as variáveis componentes do registro descrito por **tree_var** no conjunto de variáveis definidas no nó **num_no**.

```
{ trata a definição de uma variável do tipo registro }

begin
  tree_var_aux := retr_tree_var(reg);
  def_record(tree_var_aux^.comp,num_no);
end
```

O procedimento **def_record** realiza efetivamente a determinação das variáveis componentes de um registro. Ao ser identificada uma definição de uma variável do tipo registro, as variáveis componentes desse registro são incluídas no conjunto de variáveis definidas do nó associado ao comando que define esse registro; caso qualquer variável componente seja também caracterizada como um registro, ocorre uma chamada recursiva ao referido procedimento, o qual é mostrado a seguir; as variáveis e funções e procedimentos utilizados, e ainda não descritos nesta seção, são também introduzidos.

PROCEDIMENTO: **def_record**

PARÂMETROS DE ENTRADA:

tree_var : apontador para um elemento da árvore de variáveis declaradas, que descreve o primeiro elemento componente do registro definido.

num_no : número do nó onde ocorreu a definição do registro.

VARIÁVEIS GLOBAIS:

info_no : é um apontador para uma estrutura de dados que armazena informações a respeito de cada nó da unidade em teste. Dentre os campos que compõem essa estrutura de dados, destacamos *defg_i* que é um vetor de "bits", onde cada elemento de defg_i refere-se a uma variável do programa; se o "bit" é 1 a variável foi definida, se o "bit" é 0 não foi definida.

FUNÇÕES OU PROCEDIMENTOS CHAMADOS:

set_bit(id,vetor) : torna 1 o "bit" de número **id** do vetor de bits **vetor**.

```
{ inclui as variáveis componentes de um registro no conjunto de variáveis
  definidas de um determinado nó }

begin
  if tree_var^.comp = nil then
    set_bit(tree_var^.id,info_no[num_no].defg_i);
```

```
  else
    def_record(tree_var^.comp,num_no);
  if tree_var^.next <> nil then
    def_record(tree_var^.next,num_no);
end
```

4.4 PROCEDIMENTOS DE INSTRUMENTAÇÃO

A instrumentação, como já foi dito, é a função que gera uma versão da unidade em teste preparada para a atividade de teste. Esse processo depende essencialmente da identificação dos comandos que alteram o fluxo de execução do programa, pois o processo constitui-se na inserção de pontas de prova (instruções de escrita) e alguns comandos no programa fonte. A POKE-TOOL realiza essa operação através de uma análise sintática da versão em LI do programa em teste, onde são identificados os comandos que alteram o fluxo de controle.

O analisador sintático da LI vai reconhecendo os átomos e comandos da LI e executando as ações necessárias para a realização da tarefa de instrumentação. Por exemplo, quando o analisador da LI identifica um comando "while", ele insere uma ponta de prova dentro do corpo do "while" para indicar a passagem pelo nó da condição, e uma ponta de prova, correspondente a esse mesmo nó, imediatamente depois do corpo do "while" para indicar a passagem pelo nó da condição quando da saída do laço. Portanto, para se instrumentar o programa fonte para uma determinada linguagem de programação, deve-se, à medida que se reconhecem os elementos que compõem a versão em LI, inserir as pontas de prova necessárias à instrumentação de um comando particular.

O analisador sintático da linguagem LI é implementado pela POKE-TOOL por um "parser" onde as regras sintáticas da LI são definidas no código fonte desse analisador. As ações necessárias à instrumentação -- procedimentos de instrumentação -- são invocadas pelo analisador sintático nos pontos que identificam a necessidade da geração de comandos no arquivo que conterá a versão instrumentada do programa em teste; tais pontos caracterizam a inserção dos comandos originais do programa fonte e dos comandos necessários à instrumentação do programa.

Como exemplo, será apresentado o código utilizado na instrumentação do comando PERFORM-UNTIL. Na Seção 3.4 estabeleceu-se que esse comando

equivale a uma estrutura de controle do tipo "while". A análise sintática do comando "while" da LI é mostrada a seguir; ela reflete a estrutura sintática da LI apresentada no Apêndice A. Antes de iniciar o algoritmo, vamos introduzir algumas estruturas de dados, procedimentos e funções utilizados nesse algoritmo.

VARIÁVEIS LOCAIS:

num_while : armazena o número do nó associado à condição do laço.

VARIÁVEIS GLOBAIS:

arqli : arquivo que contém o programa em LI.

mod_arq : arquivo que armazena a versão instrumentada do programa.

aux_arq : arquivo que armazena as sub-rotinas auxiliares à instrumentação.

seq_mod : variável inteira que seqüencia os comandos da versão instrumentada do programa.

seq_aux : variável inteira que seqüencia os comandos das sub-rotinas auxiliares à instrumentação.

pnewsymbol : estrutura de dados que descreve o átomo da LI lido do arquivo **mod_arq**; é composta pela descrição do átomo (simbolo), posição de início do código fonte mapeado para o átomo em LI (inicio), comprimento do código fonte equivalente ao átomo em LI (comprimento) e linha onde se inicia o código fonte correspondente (linha).

num_no : armazena o número do nó associado ao átomo da LI lido do arquivo **mod_arq**.

tab_counter : determina o número de caracteres associado à tabulação da versão instrumentada do programa.

FUNÇÕES OU PROCEDIMENTOS CHAMADOS:

in_while_monitor(num_no) : inicia a monitoração dos comandos do tipo "while"; esta função será melhor descrita no decorrer desta seção.

fim_while_monitor(num_no) : finaliza a monitoração dos comandos do tipo "while"; esta função será melhor descrita no decorrer desta seção.

getsymli(pnewsymbol,arqli) : obtém o próximo átomo da LI, a partir do arquivo que contém o programa em LI, e armazena sua descrição na variável **atomo_li**.

ajusta_formato(arq,tab) : ajusta a tabulação do arquivo descrito por **arq** pela quantidade de caracteres especificado pela variável **tab**.

copy_command(pnewsymbol,arq,seq) : copia o código fonte descrito na estrutura **pnewsymbol** no arquivo da variável **arq**; o número de seqüência associado

ao código copiado é informado pela variável **seq**.

error(string) : encerra a execução do analisador sintático da LI informando ao usuário da POKE-TOOL o motivo especificado em **string**.

```
{ análise sintática do comando "while" da LI }
      .
      .
   if pnewsymbol.simbolo = "$WHILE" then
     begin
     var num_while:integer;
     in_while_monitor(num_no);
     num_while := num_no;
     getsymli(pnewsymbol, arqli);
     if pnewsymbol.simbolo = "$C"  or
        pnewsymbol.simbolo = "$NC" then
       begin
       ajusta_formato(mod_arq,tab_counter+2);
       copy_command(pnewsymbol,mod_arq,seq_mod);
       getsymli(pnewsymbol,arqli);
       if pnewsymbol.simbolo = "{" then
         begin
         getsymli(pnewsymbol,arqli);
         fim_while_monitor(pnewsymbol,num_while);
         end
       else
         error("Erro Fatal: esperado "{" em while");
       end
     else
       error("Erro Fatal: condicao esperada em while");
     getsymli(pnewsymbol,arqli);
     end
      .
      .
```

Uma característica da linguagem COBOL consiste na presença de números de seqüência em cada linha de um programa fonte; para suportar tal necessidade, o procedimento **copy_command** foi alterado com o intuito de se inserir a ocorrência desse número no programa instrumentado.

Os procedimentos **in_while_monitor** e **fim_while_monitor** foram instanciados para a linguagem COBOL; as particularidades relativas à instrumentação do comando PERFORM-UNTIL são tratadas para refletir o modelo de instrumentação especificado na Seção 3.4; esses procedimentos são apresentados a seguir. As variáveis e os procedimentos usados e que ainda não foram citados nesta seção são mostrados.

{ inicia a monitoração dos comandos do tipo "while" }

PROCEDIMENTO: **in_while_monitor**

PARÂMETROS DE ENTRADA:

num_no : número do nó associado à condição do laço.

VARIÁVEIS GLOBAIS:

position_mod : armazena uma posição associada ao arquivo da versão instrumentada do programa em teste.

FUNÇÕES OU PROCEDIMENTOS CHAMADOS:

grava_no(arq,seq,num_no) : grava no arquivo da variável **arq**, através de comentários e com seqüência **seq**, o número do nó contido na variável **num_no**.

nova_linha(arq,seq) : cria nova linha com seqüência **seq** no arquivo da variável **arq**.

write_file(arq,string) : grava **string** no arquivo da variável **arq**.

get_position(arq) : retorna posição corrente do arquivo da variável **arq**.

```
begin
  { grava no fonte modificado, através de comentários, a ocor-
    rência dos nós referentes à condição e ao corpo do laço }
  grava_no(mod_arq,seq_mod,num_no);
  grava_no(mod_arq,seq_mod,num_no+1);

  { inclui chamada no fonte modificado para a sub-rotina que au-
    xilia a monitoração do comando PERFORM-TIMES }
  nova_linha(mod_arq,seq_mod);
  ajusta_formato(mod_arq,tab_counter);
  write_file(mod_arq,"PERFORM ");

  { salva posição onde se inicia a chamada para a sub-rotina au-
    xiliar à monitoração do comando PERFORM-TIMES }
  position_mod := get_position(mod_arq);

  { reserva no fonte modificado o local da chamada para a
    sub-rotina auxiliar a monitoração do comando PERFORM-TIMES }
  ajusta_formato(mod_arq,20);
end
```

{ finaliza a monitoração dos comandos do tipo "while" }

PROCEDIMENTO: **fim_while_monitor**

PARÂMETROS DE ENTRADA:

pnewsymbol : estrutura que armazena o átomo LI que descreve o corpo do laço.

num_no : número do nó associado à condição do laço.

VARIÁVEIS LOCAIS:

posicao : variável auxiliar utilizada para armazenar posição do arquivo da versão instrumentada.

ide_rotulo : cadeia de caracteres que armazena o nome da sub-rotina auxiliar à instrumentação.

FUNÇÕES OU PROCEDIMENTOS CHAMADOS:

recup_rotulo(pnewsymbol,ide_rotulo) : recupera o rótulo da sub-rotina auxiliar à instrumentação a partir da estrutura **pnewsymbol**, a qual armazena o corpo do laço.

grava_ponta_de_prova(arq,seq,tab,num_no) : grava ponta de prova do nó **num_no** no arquivo da variável **arq**, com número de seqüência **seq**, e obedecendo a tabulação **tab**.

change_position(arq,pos) : altera a posição corrente do arquivo da variável **arq** para a posição **pos**.

def_ref_all(num_no) : inclui todas as variáveis do programa no conjunto de variáveis definidas por referência do nó **num_no**.

```
begin
  recup_rotulo(pnewsymbol,ide_rotulo);
  grava_ponta_de_prova(mod_arq,seq_mod,tab_counter,num_no);
  { inclui no fonte modificado a chamada para a sub-rotina que
    auxilia a instrumentação do comando PERFORM-TIMES }
  posicao := get_position(mod_arq);
  change_position(mod_arq,position_mod);
  write_file(mod_arq,ide_rotulo);
  change_position(mod_arq,posicao);
  { inclui no arquivo auxiliar o rótulo da sub-rotina que auxilia
    monitoração do comando PERFORM-UNTIL }
  nova_linha(aux_arq,seq_aux);
  write_file(aux_arq,ide_rotulo);
  write_file(aux_arq,".");
  { inclui no arquivo auxiliar pontas de prova referentes à
    condição e ao corpo do laço }
  grava_ponta_de_prova(aux_arq,seq_aux,5,num_no);
```

```
  grava_ponta_de_prova(aux_arq,seq_aux,5,num_no+1);
  { inclui todas as variáveis no conjunto de variáveis definidas
    por referência devido à chamada de unidade do corpo do laço }
  def_ref_all(num_no+1);
  { inclui no arquivo auxiliar a invocação da unidade }
  nova_linha(aux_arq,seq_aux);
  ajusta_formato(aux_arq,5);
  copy_command(pnewsymbol,aux_arq,seq_aux);
  write_file(aux_arq,".");
end
```

4.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram ressaltadas as atividades realizadas na configuração da POKE-TOOL para a linguagem COBOL, e as modificações realizadas no projeto da POKE-TOOL, para permitir o teste simultâneo das unidades presentes em um programa COBOL.

Os aspectos discutidos refletem o ponto de vista que o usuário configurador da POKE-TOOL tem da ferramenta. Foram apresentados exemplos de rotinas desenvolvidas na preparação dos analisadores léxico e sintático, e na monitoração do programa fonte; essas rotinas implementam as diretrizes de fluxo de controle, de fluxo de dados e de instrumentação, enfocadas no Capítulo 3, para a linguagem COBOL.

A estrutura gramatical definida para a linguagem COBOL, mostrada no Apêndice C, procurou considerar as diferenças existentes entre diversos compiladores, tornando-se, portanto, o mais abrangente possível.

As modificações realizadas no projeto da POKE-TOOL, devido à facilidade de se testar várias unidades simultaneamente, resultou na caracterização de elementos requeridos e na avaliação de adequação de um conjunto de casos de teste para cada unidade de programa. Nesse sentido, o grafo de fluxo de controle do programa e a extensão de fluxo de dados correspondente foram organizados por unidades; isso viabilizou a execução repetitiva dos módulos de descrição de elementos requeridos e de avaliação resultando, portanto, na facilidade de se testar concomitantemente as unidades presentes em um programa COBOL.

Um esforço inicial no sentido de validar a configuração consistiu em submeter a ferramenta aos exemplos especificados na Seção 3.4; a aplicação de um "benchmark", como um trabalho futuro, teria, entre outros, o objetivo

de validação da ferramenta.

A POKE-TOOL versão COBOL tem seu código fonte escrito em linguagem C, estando disponível para ambientes do tipo PC em plataforma DOS. A extensão da ferramenta para outros ambientes consiste na utilização das funções disponíveis na biblioteca do compilador C desses ambientes.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo são apresentadas as conclusões obtidas deste trabalho e os trabalhos futuros que podem ser conduzidos no sentido de enriquecer a versão da POKE-TOOL para a linguagem COBOL.

5.1 CONCLUSÕES

Nesta tese foram explorados os aspectos de utilização de teste estrutural baseado em análise de fluxo de dados para programas escritos em linguagem COBOL. Com esse propósito, uma ferramenta que suporta a aplicação dos critérios Potenciais Usos -- POKE-TOOL -- foi configurada para a linguagem COBOL.

A POKE-TOOL suporta o teste de unidade de programa; uma das dificuldades encontradas foi definir o que é uma unidade em linguagem COBOL; em COBOL não existem declarações explícitas de procedimentos ou funções, o que leva a crer que uma unidade em teste seria caracterizada como um programa inteiro. Este seria o raciocínio imediato; contudo, concluiu-se que cada porção de código para onde há uma transferência de controle similar a um mecanismo "call-return" é o que mais se aproxima da noção de unidade de programa; por exemplo, uma transferência explícita de controle devido ao comando PERFORM ou uma invocação implícita de rotinas para tratamento de exceção definem uma unidade de programa. Entretanto, essa abordagem difere em relação a linguagens do estilo ALGOL, pois inexiste o conceito de escopo local de variáveis, podendo ainda ocorrerem desvios incondicionais para fora dos limites de uma unidade e sobreposição de unidades.

O observado na prática é que a atividade de teste em linguagem COBOL é conduzida considerando-se um programa inteiro como um módulo em teste. Essa característica é suportada pelo conceito de unidade em linguagem

COBOL, através do teste simultâneo das unidades que compõem um programa. A POKE-TOOL versão COBOL suporta tal abordagem, fornecendo subsídios para a avaliação concomitante das unidades de um programa, o que constitui uma característica desejável de uma ferramenta de teste estrutural [HOR92].

A aplicação de critérios baseados em análise de fluxo de dados envolve atividades que viabilizem a determinação dos elementos requeridos e a obtenção dos elementos efetivamente executados, tais como: abstração do fluxo de controle, caracterização dos tipos de ocorrência de variáveis e instrumentação de código fonte. A abordagem utilizada para a automação dos critérios Potenciais Usos pela POKE-TOOL incorpora essas necessidades, que são sistematizadas em 3 modelos fundamentais à configuração da ferramenta: Modelo de Fluxo de Controle, Modelo de Fluxo de Dados e Modelo de Instrumentação. Esses modelos foram instanciados para a linguagem COBOL, como requisito básico para a preparação da POKE-TOOL.

A abstração do fluxo de execução de um programa é realizada pela POKE-TOOL através da correspondência entre os aspectos de fluxo de controle da linguagem alvo da configuração e da linguagem LI. Na instanciação do modelo de fluxo de controle, observou-se que o relacionamento entre as estruturas de controle da linguagem COBOL e da linguagem LI não resultou em uma correspondência direta, diferentemente do que ocorre para a linguagem C [CHA91b]; alguns comandos da linguagem COBOL implementam estruturas de controle complexas, como, por exemplo, o comando PERFORM-VARYING que implementa uma estrutura de laços aninhados.

A instrumentação de unidades COBOL não constitui uma tarefa trivial; a inexistência de estrutura de bloco na linguagem COBOL dificulta a atividade de instrumentação de código fonte; esse fato foi observado na monitoração dos comandos que implementam laços (comandos PERFORM-UNTIL, PERFORM-TIMES, PERFORM-VARYING e SEARCH) e na monitoração dos comandos que implementam desvios para um dentre vários rótulos (comando GOTO-DEPENDING); foram criadas sub-rotinas dedicadas à monitoração de tais comandos.

Na instanciação do modelo de fluxo de dados para a linguagem COBOL, foram estabelecidas diretrizes que fornecem subsídios à aplicação dos critérios Potenciais Usos. Variáveis do tipo registro são tratadas de maneira diferente ao enfoque adotado para a linguagem C [CHA91b], onde a definição de um elemento de um registro resulta na definição desse registro; em vez disso, as variáveis componentes de um registro são tratadas como variáveis simples, onde uma ocorrência de definição de uma variável dessa natureza não afeta as demais e a definição de um registro resulta na definição de suas variáveis componentes. Essa abordagem é a que melhor se adequa à aplicação dos

critérios baseados em análise de fluxo de dados, pois é mais precisa na determinação de associações requeridas, além de ser ideal para a detecção de anomalias de fluxo de dados [HUA79]. No caso de variáveis do tipo vetor ou matriz, a definição de um elemento componente de uma variável resulta na definição da variável; esse enfoque foi adotado por ser impossível determinar-se estaticamente que elemento estaria sendo referenciado.

Ainda com respeito ao modelo de fluxo de dados, salientou-se que a ocorrência de comandos de invocação de unidades resulta na definição por referência das variáveis presentes em um programa COBOL. No que tange à determinação das variáveis definidas no nó de entrada de uma unidade, convencionou-se a definição de todas as variáveis do programa. Esses enfoques são conservadores na aplicação dos critérios Potenciais Usos, pois nenhum caminho ou associação deixa de ser requerido [MAL91].

A caracterização de unidade e a aplicação dos modelos de implementação dos critérios Potenciais Usos à linguagem COBOL constituem pontos pertinentes à atividade de teste estrutural de programas. A versão operacional da POKE-TOOL para a linguagem COBOL, como principal produto desta tese, estabelece uma contribuição importante no sentido de suportar o teste de unidades escritas em linguagem COBOL.

Cabe ressaltar que, até o momento, não se tem conhecimento de ferramentas que apoiem o teste estrutural baseado em análise de fluxo de dados em linguagem COBOL. Portanto, a POKE-TOOL versão COBOL constitui a primeira ferramenta que suporta a aplicação de critérios baseados em análise de fluxo de dados para a linguagem COBOL.

Um esforço inicial no sentido de validar a configuração consistiu em submeter a ferramenta aos exemplos especificados na Seção 3.4. A POKE-TOOL versão COBOL tem seu código fonte escrito em linguagem C, estando disponível para ambientes do tipo PC em plataforma DOS. A extensão da ferramenta para outros ambientes consiste na utilização das funções disponíveis do compilador C desses ambientes.

5.2 TRABALHOS FUTUROS

No sentido de dar continuidade ao trabalho realizado nesta tese, são apresentadas, a seguir, algumas atividades que enriquecerão o suporte ao teste estrutural de programas COBOL.

. Aplicar um "benchmark" para a POKE-TOOL versão COBOL no sentido de validar a ferramenta e avaliar o custo de aplicação dos critérios Potenciais Usos em unidades COBOL.

. Incorporar heurísticas à POKE-TOOL versão COBOL para a caracterização de caminhos não executáveis em programas COBOL; a existência desses caminhos dificulta a avaliação de adequação a um critério de teste. Já foram implementadas heurísticas dessa natureza para a versão C da POKE-TOOL [VER92].

. Estender a POKE-TOOL versão COBOL através da introdução de outros critérios de teste estrutural. Os modelos de implementação dos critérios Potenciais Usos incluem aspectos comuns à automação de critérios de teste estrutural, o que favorece a inclusão de outros critérios de teste na ferramenta POKE-TOOL.

APÊNDICE A

A LINGUAGEM INTERMEDIÁRIA (LI)

Neste apêndice é apresentada uma síntese da descrição da linguagem LI [CAR91,CHA91b].

A linguagem intermediária (LI) tem como objetivo identificar o fluxo de execução em um programa. Basicamente, a LI tem dois tipos de comandos: comandos *seqüenciais* e comandos de *controle de fluxo.* Os comandos seqüenciais da LI indicam os comandos das linguagens procedurais que representam uma declaração de variável ou uma computação (comandos de atribuição ou chamadas de procedimentos) e que, portanto, não alteram o fluxo de execução. Os comandos de controle de fluxo da LI são equivalentes aos comandos das linguagens procedurais que causam *seleção, seleção múltipla, iteração* e *transferência incondicional.*

Uma característica própria da LI é que todos os *átomos* da linguagem são seguidos por números que identificam, respectivamente, o início do átomo no arquivo fonte da unidade em teste (a quantos bytes do começo do arquivo se inicia o átomo), o comprimento do átomo (quantos bytes tem o átomo) e a linha onde se inicia o átomo. Dessa maneira, utilizando a notação de Backus-Naur, um átomo da LI teria a seguinte estrutura:

<atm_li> ::= *<atomo><inicio><comprimento><linha>*.

Onde

<atomo> ::= *$DCL* | *$S* | *$IF* | *$CASE* | *$ROTC* | *$ROTD*
| *$WHILE* | *$FOR* | { | } | *$C* | *$NC* | *$REPEAT* | *$UNTIL*
| *$GOTO* | *LABEL* | *$BREAK* | *$CONTINUE* | *$RETURN*
| *$CC* | *$ELSE*

<inicio> ::= *NUM*,
<comprimento> ::= *NUM* e
<linha> ::= *NUM*.

Os terminais acima representam a seqüência de caracteres indicada pelos próprios terminais; com exceção de $\$S$ que indica a seqüência de caracteres $\$Sd^n$ onde d pertence a {0,1,...,9} e n>0, $\$C$ que indica a seqüência $\$C(d^n)d^n$, $\$NC$ que indica a seqüência $\$NC(d^n)d^n$, *NUM* que indica a seqüência d^n e *LABEL* que indica uma seqüência de letras e caracteres sempre começando por um caractere.

A utilidade desses ponteiros dos átomos é possibilitar o acesso ao código fonte associado ao átomo LI, o que vai ser necessário, por exemplo, para a extensão do grafo de fluxo de controle para a geração do *grafo def* e para a instrumentação da unidade em teste.

Na convenção de Backus-Naur [SEB89] adotada aqui, os terminais serão representados em itálico, os não-terminais entre "<" e ">" e os meta-símbolos sublinhados.

Os comandos seqüenciais são os seguintes:

<*dcl*> ::= $\$DCL$<*inicio*><*comprimento*><*linha*> e
<*s*> ::= $\$S$<*inicio*><*comprimento*><*linha*>.

Onde <*dcl*> denota uma *declaração de variável* e <*s*> uma *computação*; sendo que uma computação pode ser uma atribuição de valor a uma variável (através de uma expressão ou chamada de função), ou somente uma chamada de procedimento.

Os comandos de controle de fluxo são os seguintes:

(1) **Seleção**

<*if*> ::= <*if_atm*><*cond_atm*><*statement*1> |
<*if_atm*><*cond_atm*><*statement*1><*else_atm*><*statement*2>

onde

<*if_atm*> ::= $\$IF$<*inicio*><*comprimento*><*linha*>,
<*cond_atm*> ::= $\$C$<*inicio*><*comprimento*><*linha*>,
<*else_atm*> ::= $\$ELSE$<*inicio*><*comprimento*><*linha*> e

<*statement*i> é o não-terminal que denota todos os possíveis comandos da LI, agrupados ou não; <*statement*i> será definido formalmente mais adiante. Este comando significa o "if" tradicional das linguagens do estilo ALGOL, ou seja, os "comandos" em <*statement*1> serão executados se <*cond_atm*> for

verdadeiro e os "comandos" em *<statement2>* serão executados caso contrário.

A solução para a ambigüidade que poderia ser gerada pelo encadeamento de *$IF* e *$ELSE* sem delimitadores de bloco (e) foi tomada inspirada na maioria das linguagens do estilo ALGOL, onde o *$ELSE* é associado com o mais recente *$IF* se *$ELSE*, salvo o uso explícito de chaves que podem forçar a associação apropriada. Observe-se que *$S1* acima representa o primeiro comando sequencial do programa em LI, ou seja, o número que segue os caracteres "*$S*" indica a ordem em que aparece o comando sequencial no programa. Os números que aparecem em *$C(1)1* indicam, respectivamente, o número de predicados que possui a condição e a ordem de aparição.

(2) **Seleção Múltipla:**

<case> ::=
<case_atm><case_cond_atm>{{(*<rotc_atm>*¦*<rotd_atm>*){*<statement>*}}+}
onde
<case_atm> ::= *$CASE<inicio><comprimento><linha>*,
<case_cond_atm> ::= *$CC<inicio><comprimento><linha>*,
<rotc_atm> ::= *$ROTC<inicio><comprimento><linha>* e
<rotd_atm> ::= *$ROTD<inicio><comprimento><linha>*.

O não-terminal *<case_atm>* representa o átomo que inicia o comando de seleção múltipla, *<case_cond_atm>* representa a condição do comando e *<rotc_atm>* os possíveis rótulos para as seqüências de comandos indicadas por *<statement>*. O não-terminal *<rotd_atm>* representa o rótulo para a seqüência de comandos a ser executada quando a condição não combina com nenhum rótulo *<rotc_atm>*.

A semântica do comando acima é equivalente ao comando "switch" da linguagem C, isto é, a execução começa no rótulo que ocorreu a combinação e executa os comandos desse rótulo mais os comandos dos rótulos que o seguem, a menos que seja encontrado um comando do tipo "break", que causa a imediata saída do comando de seleção múltipla, ou um comando de desvio incondicional. Esses comandos serão discutidos mais adiante.

(3) **Iteração**

A LI fornece comandos para iteração tanto para um número fixo de repetições quanto para um número de repetições que depende de uma condição.

No caso de um número fixo de repetições, tem-se um comando

semelhante ao "for" das linguagens do estilo ALGOL. O "for" da LI é definido como

<for> ::= *<for_atm><s1><cond_for-atm><s2><statement>*
onde
<for_atm> ::= *$FOR<inicio><comprimento><linha>* e
<cond_for_atm> ::= *($C|$NC)<inicio><comprimento><linha>.*

O não-terminal *<for-atm>* indica o comando "for" da LI, o não terminal *<s1>* representa a iniciação das variáveis de controle do "for" através de um comando sequencial, *<cond_for_atm>* representa a condição, *<s2>* representa o comando sequencial que altera as variáveis de controle a cada iteração do "for" e *<statement>* representa o corpo do comando. O comando "for" da LI é inspirado no comando equivalente da linguagem C, possuindo a mesma semântica. Note que a condição é composta de $C e $NC; no primeiro caso, o corpo do laço é executado enquanto a condição for verdadeira ou, no segundo caso, o corpo do laço é executado enquanto a condição for falsa.

Os comandos de iteração cujo número de repetições é dirigido por uma condição são também equivalentes aos tradicionais "while" e "repeat-until" das linguagens do estilo ALGOL. O "while" da LI é definido como

<while> ::= *<while-atm><cond_while_atm><statement>*
onde
<while_atm> ::= *$WHILE<inicio><comprimento><linha>* e
<cond_while_atm> ::= *($C|$NC)<inicio><comprimento><linha>.*

<while-atm> indica o comando "while", *<cond_while_atm>* a condição e *<statement>* o corpo do "while". Note que a condição é composta de $C e $NC; no primeiro caso, o corpo do laço é executado enquanto a condição for verdadeira ou, no segundo caso, o corpo do laço é executado enquanto a condição for falsa.

O "repeat-until" da LI é definido como

<repeat_until> ::=
<repeat_atm><statement><until_atm><cond_until_atm>,
onde
<repeat_atm> ::= *$REPEAT<inicio><comprimento><linha>*,
<until_atm> ::= *$UNTIL<inicio><comprimento><linha>* e
<cond_until_atm> ::= *($C|$NC)<inicio><comprimento><linha>.*

<repeat_atm> indica o início do comando "repeat-until", *<statement>* o corpo do comando, *<until_atm>* indica o fim do comando e *<cond_until_atm>* representa a condição de término. A semântica desse comando da LI é igual ao comando equivalente das linguagens "ALGOL-like", ou seja, o corpo da iteração é executado pelo menos uma vez e o teste da condição é realizado depois da execução do corpo. Note-se que a condição de término é composta por **$C** ou **$NC**, isto ocorre porque, na maioria das linguagens "ALGOL-like", o corpo do "repeat-until" é executado até que uma dada condição se verifique; porém, em algumas linguagens como C, o comando "repeat-until" equivalente funciona de maneira que o corpo é executado enquanto a condição permanece verdadeira. Devido a esse fato, a condição desse comando pode ser **$C**, se for o primeiro caso, e aí estaria indicando um "repeat-until" tradicional, ou **$NC**, se for o segundo caso, e nesta situação teríamos que o laço se repete até a negação da condição.

Os comandos de desvio incondicional provocam a mudança do fluxo de execução em um programa. A LI possui um comando de tranferência incondicional irrestrito do tipo "goto" e comandos de transferência incondicional controlada; estes últimos têm sua utilização limitada a algumas situações e seu efeito bem previsível.

O comando "goto" da LI é definido como

<goto> ::= *<goto_atm><label_atm>*
onde
<goto_atm> ::= *$GOTO<inicio><comprimento><linha>* e
<label_atm> ::= *LABEL<inicio><comprimento><linha>*.

<goto_atm> representa o comando "goto" da LI e *<label_atm>* representa o rótulo para onde deve ser dirigido o fluxo de execução quando é encontrado o comando "goto".

Existem mais três comandos de desvio incondicional na LI, são eles

<break> ::= *$BREAK<inicio><comprimento><linha>*,
<continue> ::= *$CONTINUE<inicio><comprimento><linha>* e
<return> ::= *$RETURN<inicio><comprimento><linha>*.

Esses comandos de transferência incondicional foram inspirados nos seus homônimos da linguagem C e, por isso, possuem efeitos idênticos. O "break" causa o fim do comando de iteração ("for', "while" ou "repeat-until")

mais próximo que o engloba. Ainda, dentro de um comando "case" da LI, o "break" causa o desvio para o primeiro comando fora do "case". O comando "continue" provoca o desvio para a *próxima iteração* do laço que o engloba. No caso dos comandos "repeat-until" e "while", ao encontrar-se o "continue", o fluxo de execução é desviado para o teste da condição da iteração; no caso do comando "for", o fluxo de execução é desviado para o comando que altera as variáveis de controle. O comando "return" causa o fim do procedimento que está sendo executado e o retorno para a unidade que o chamou.

Até aqui foram descritos os comandos individuais da LI, entretanto, para concluir a definição da LI, ainda falta definir como se agrupam comandos na LI e como esses comandos são organizados em um programa. O não-terminal *<statement>*, muito utilizado acima, representa um único comando da LI ou um agrupamento deles e é definido como

<statement> ::= ({*<statement>*}) ¦
<dcl> ¦
<s> ¦
<if> ¦
<case> ¦
<for> ¦
<while> ¦
<repeat_until> ¦
LABEL*<statement>* ¦
<goto> ¦
<break> ¦
<continue> ¦
<return>.

Os programas em LI são definidos da seguinte maneira:

<program> ::= {{*<dcl>*¦*<s>*}}{*<statement>*}.

Um exemplo de tradução de um programa fonte para a LI é fornecido no Apêndice D, que consiste na tradução do programa DEMO.CBL para a linguagem LI.

A associação dos comandos do código fonte para a LI não é sempre direta como poderia ser concluído do exemplo citado acima; considere o trecho de programa em Pascal [GHE87]:

```
type
 day = (sunday,monday,tuesday,wednesday,thursday,friday,saturday);
var
 week_day: day;
 ...
    for week_day := monday to friday do
        ...;
```

seria traduzido para

$DCL	51	71	2
$DCL	125	19	4
	...		
$FOR	303	3	7
$S1	307	18	7
$C(1)1	0	0	0
$S2	326	9	7

Observe-se que na condição do comando "for" os ponteiros são iguais a 0, indicando que o átomo da LI não possui correspondência no arquivo fonte. Obviamente, a decisão de como mapear a linguagem da unidade para a LI é do usuário configurados da POKE-TOOL.

APÊNDICE B

TABELA DE TRANSIÇÕES LÉXICAS DA LINGUAGEM COBOL

Este Apêndice apresenta a tabela de transições léxicas utilizada na configuração do analisador léxico da POKE-TOOL para a linguagem COBOL. As informações contidas nesta tabela foram descritas na Seção 4.2.

```
"i \r\t\f" 0 0 0 0
"c\"\r\t\nld-*@&" 0 1000 6 1000
"i\n" 0 1 0 10
"il" 0 10 1 23
"id" 0 11 1 25
"i-" 0 15 1 30
"i*" 0 20 1 32
"i\"" 0 25 1 34
"i@" 0 50 10 39
"i&" 0 55 10 43
"id " 1 2 0 11
"id " 2 3 0 12
"id " 3 4 0 13
"id " 4 5 0 14
"id " 5 6 0 15
"id " 6 7 0 16
"c*/$" 7 0 0 0
"i*/" 7 8 0 19
"i$" 7 9 1 21
"c\r" 8 8 0 19
"i\r" 8 0 0 0
"c\r" 9 9 2 21
"i\r" 9 1000 14 1000
"ild-" 10 10 2 23
"cld-" 10 1000 3 1000
"id" 11 11 2 25
"i." 11 12 2 28
"cd." 11 1000 4 1000
"id" 12 11 2 25
"cd" 12 1000 9 1000
"id" 15 11 2 25
"cd" 15 1000 7 1000
"i*" 20 1000 8 1000
"c*" 20 1000 7 1000
"c\"\r" 25 25 2 34
"i\r" 25 26 2 37
"i\"" 25 1000 5 1000
"c\"" 26 26 2 37
"i\"" 26 25 2 34
```

```
"c." 50 50 2 39
"i." 50 51 2 41
"i\"ld" 51 50 2 39
"c\"ld" 51 1000 11 1000
"i \t\r\n" 55 55 0 43
"c \t\r\n" 55 56 2 45
"c .\t\r" 56 56 2 45
"i." 56 57 2 48
"i \t\r" 56 1000 12 1000
"c \t\r" 57 56 2 45
"i \t\r" 57 1000 13 1000
```

APÊNDICE C

DESCRIÇÃO SINTÁTICA DA LINGUAGEM COBOL

Neste apêndice é apresentada a especificação sintática utilizada na configuração do analisador sintático da POKE-TOOL para a linguagem COBOL. A notação utilizada aproxima-se da notação estendida de Backus-Naur [SEB89], sintetizada na Seção 4.3.

```
unitra   = [ 'DIRCOMP' ] DIiden DIenvi DIdata DIproc .

DIiden   = 'IDENTIFICATION' 'DIVISION' '.' [ DIidenA1 ] .
DIidenA1 = 'PROGRAM-ID'     '.'     'IDENT'   '.' ,
           'AUTHOR'         '.' 22 'SEQCOD' '.' ,
           'INSTALLATION'   '.' 22 'SEQCOD' '.' ,
           'DATE-WRITTEN'   '.' 22 'SEQCOD' '.' ,
           'DATE-COMPILED'  '.' 22 'SEQCOD' '.' ,
           'SECURITY'       '.' 22 'SEQCOD' '.' .

DIenvi   = 'ENVIRONMENT' 'DIVISION' '.' [ DIenviA1 ] .
DIenviA1 = SEconf , SEinou .

SEconf   = 'CONFIGURATION' 'SECTION' '.' [ SEconfA1 ] .
SEconfA1 = 'SOURCE-COMPUTER' '.' 'IDENT' '.' ,
           'OBJECT-COMPUTER' '.' 'IDENT' virgul [ SEconfA2 virgul ] '.' ,
           'SPECIAL-NAMES'   '.' [ SEconfA2 virgul ] '.' .
SEconfA2 = 'MEMORY' , 'SIZE' , 'WORDS' , 'CHARACTERS' , 'MODULES' ,
           'PROGRAM' , 'COLLATING' , 'SEQUENCE' , 'SEGMENT-LIMIT' ,
           'ON' , 'OFF' , 'ARE' , 'STATUS' , 'ALPHABET' , 'NATIVE' ,
           'EBCDIC' , 'CLASS' , 'CURRENCY'  'SIGN' , 'DECIMAL-POINT' ,
           'COMMA' , 'IS' , 'IDENT' , 'CONST'  'SWITCH' , 'STANDARD-1' ,
           'STANDARD-2' , 'SYMBOLIC' , 'NUMERIC' , 'TRAILING' ,
           'SEPARATE' , 'ASCII' .

SEinou   = 'INPUT-OUTPUT' 'SECTION' '.' [ SEinouA1 ] .
SEinouA1 = 'FILE-CONTROL' '.' 1 [ SEinouA2 '.' ] ,
           'I-O-CONTROL'  '.' 1 [ SEinouA4 '.' ] .
SEinouA2 = select [ SEinouA3 virgul ] .
SEinouA3 = reserv , access , organi , status , key .
SEinouA4 = [ SEinouA5 virgul ] .
SEinouA5 = rerun , same .
```

```
select   = 'SELECT' selectA1 'IDENT' 'ASSIGN' selectA2 selectA3
           selectA4 .
selectA1 = 'OPTIONAL' , * .
selectA2 = 'TO' , * .
selectA3 = 'EXTERNAL' , 'DYNAMIC' , * .
selectA4 = 'DISK' , 'KEYBOARD' , 'DISPLAY' , 'PRINTER' ,
           'MULTIPLE' selectA5 , 'CONST' , 'IDENT' .
selectA5 = 'REEL' , 'UNIT' .

reserv   = 'RESERVE' reservA1 reservA2 reservA3 .
reservA1 = 'CONST' , 'NO' .
reservA2 = 'ALTERNATE' , * .
reservA3 = 'AREA' , 'AREAS' .

access   = 'ACCESS' accessA1 aux01 accessA2 .
accessA1 = 'MODE' , * .
accessA2 = 'SEQUENTIAL' , 'RANDOM' , 'DYNAMIC' .

organi   = 'ORGANIZATION' aux01 organiA1 .
organiA1 = 'SEQUENTIAL' , 'RELATIVE' , 'INDEXED' , 'LINE' 'SEQUENTIAL' .

status   = 'FILE' 'STATUS' aux01 identi , 'STATUS' aux01 identi .

key      = 'RELATIVE' 'KEY' aux01 identi ,
           'ACTUAL' 'KEY' aux01 identi ,
           'RECORD' 'KEY' aux01 identi ,
           'ALTERNATE' 'RECORD' 'KEY' aux01 identi keyA1 .
keyA1    = 'WITH' 'DUPLICATES' , 'DUPLICATES' , * .

rerun    = 'RERUN' 'ON' rerunA1 rerunA2 rerunA3 .
rerunA1  = 'IDENT' , 'ERROR' , 'ERRORS' .
rerunA2  = 'EVERY' , * .
rerunA3  = 'END' 'OF' rerunA4 rerunA5 'IDENT' ,
           'CONST' 'RECORDS' rerunA5 'IDENT' .
rerunA4  = 'REEL' , 'UNIT' .
rerunA5  = 'OF' , * .

same     = 'SAME' sameA1 sameA2 sameA3 [ 'IDENT' virgul ] .
sameA1   = 'RECORD' , 'SORT' , 'SORT-MERGE' .
sameA2   = 'AREA' , * .
sameA3   = 'FOR' , * .


DIdata   = 'DATA' 'DIVISION' '.' [ DIdataA1 ] .
DIdataA1 = SEfile , SEwork , SElink .

SEfile   = 'FILE' 'SECTION' '.' [ SEfileA1 ] .
SEfileA1 = SEfileA2 'IDENT' [ ATfile virgul ] '.' [ Ddados '.' ] .
SEfileA2 = 'FD' , 'SD' .

ATfile   = 'RECORD' , 'BLOCK' , 'CONTAINS' , 'CONST' , 'CHARACTERS' ,
           'IN' , 'RECORDING' , 'MODE' , 'IS' , 'IDENT' , 'LABEL' ,
           'RECORDS', 'VALUE' , 'OF' , 'TO' , 'DATA' , 'OMITTED' ,
           'STANDARD' , 'SIZE' , 'EXTERNAL' , 'GLOBAL' , 'VARYING' ,
           'FROM' , 'DEPENDING' , 'ARE' .

SElink   = 'LINKAGE' 'SECTION' '.' [ Ddados '.' ] .
```

```
SEwork   = 'WORKING-STORAGE' 'SECTION' '.' [ Ddados '.' ] .

Ddados   = 'CONST' DdadosA1 [ DdadosA2 virgul ] .
DdadosA1 = 'IDENT' , 'FILLER' .
DdadosA2 = pictur , redefi , rename , blank , justif , value ,
           occurs , synchr .

pictur   = picturA1 'FORMATO' usage .
picturA1 = 'PICTURE' , 'PIC' .

usage    = usageA1 usageA2 .
usageA1  = 'USAGE' aux01 , * .
usageA2  = 'DISPLAY' , 'COMPUTATIONAL' , 'COMP' , 'COMPUTATIONAL-1' ,
           'COMP-1' , 'COMPUTATIONAL-2' , 'COMP-2' , 'COMPUTATIONAL-3' ,
           'COMP-3' , 'INDEX' , 'DISPLAY-ST' , * .

redefi   = 'REDEFINES' 'IDENT' .

rename   = 'RENAMES' 'IDENT' renameA1 .
renameA1 = 'THRU' 'IDENT' , 'THROUGH' 'IDENT' , * .

blank    = 'BLANK' blankA1 'ZERO' .
blankA1  = 'WHEN' , * .

justif   = 'JUSTIFIED' 'RIGHT' , 'JUST' 'RIGHT' .

value    = valueA1 aux01 [ consta valueA2 virgul ] .
valueA1  = 'VALUE' , 'VALUES' .
valueA2  = 'THRU' consta , * .

occurs   = 'OCCURS' 'CONST' occursA1 occursA2 occursA3 occursA4
           occursA5 .
occursA1 = 'TO' 'CONST' , * .
occursA2 = 'TIMES' , * .
occursA3 = 'DEPENDING' 'ON' 'IDENT' , * .
occursA4 = 'ASCENDING'  'KEY' aux01 [ 'IDENT' virgul ] ,
           'DESCENDING' 'KEY' aux01 [ 'IDENT' virgul ] , * .
occursA5 = 'INDEXED' 'BY' [ 'IDENT' virgul ] .

synchr   = 'SYNC' synchrA1 , 'SYNCHRONIZED' synchrA1 .
synchrA1 = 'LEFT' , 'RIGHT' , * .


DIproc   = 'PROCEDURE' 'DIVISION' DIprocA1 '.' declar [ DIprocA2 ] .
DIprocA1 = 'USING' [ identi virgul ] , * .
DIprocA2 = label , Cseque 2 DIprocA3 , Cdesvi DIprocA3 , Ccondi '.' .
DIprocA3 = '.' , * .

declar   = 'DECLARATIVES' '.' [ DIprocA2 ] 'END' 'DECLARATIVES' '.' , * .

label    = 'IDENT' labelA1 '.' labelA3 .
labelA1  = 'SECTION' labelA2 , * .
labelA2  = 'CONST' , * .
labelA3  = use , * .

LC       = [ LCA1 ] .
LCA1     = Cseque , Cdesvi , Ccondi .
```

```
aux01    = 'IS' , 'ARE' , * .
virgul   = ',' , * .
id_cte   = identi , consta .

Cseque   = accept , add , alter , call , cancel , close , comput ,
           delete , displa , divide , exit , inspec , merge , move ,
           multip , next , open , read , releas , return , set ,
           sort , start , string , subtra , unstri , write .

Cdesvi   = goto , perfor , stop .

Ccondi   = search , if , at , on , invali .

accept   = 'ACCEPT' identi acceptA1 .
acceptA1 = 'FROM' acceptA2 , * .
acceptA2 = 'DATE' , 'DAY' , 'TIME' , 'IDENT' .

add      = 'ADD' addA1 [ id_cte virgul ] addA2 addA3 .
addA1    = 'CORR' , 'CORRESPONDING' , * .
addA2    = 'TO'      [ addA4  addA5 virgul ] , * .
addA3    = 'GIVING' [ identi addA5 virgul ] , * .
addA4    = identi , consta .
addA5    = 'ROUNDED' , * .

alter    = 'ALTER' [ 'IDENT' 'TO' alterA1 'IDENT' virgul ] .
alterA1  = 'PROCEED' 'TO' , * .

at       = 'AT' atA1 LC .
atA1     = 'END' , 'EOP' , 'END-OF-PAGE' .

call     = 'CALL' id_cte 'USING' callA1 [ callA2 ] .
callA1   = 'BY' , * .
callA2   = 'REFERENCE' [ identi virgul ] callA1 ,
           'CONTENT'   [ identi virgul ] callA1 .

cancel   = 'CANCEL' [ id_cte virgul ] .

close    = 'CLOSE' [ 'IDENT' closeA1 closeA2 closeA3 virgul ] .
closeA1  = 'REEL' , 'UNIT' , * .
closeA2  = 'WITH' , * .
closeA3  = 'NO' 'REWIND' , 'LOCK' , 'POSITIONING' , * .

comput   = 'COMPUTE' [ identi computA1 virgul ] '=' EX .
computA1 = 'ROUNDED' , * .

delete   = 'DELETE' identi deleteA1 .
deleteA1 = 'RECORD' , * .

displa   = 'DISPLAY' [ id_cte virgul ] displaA1 .
displaA1 = 'UPON' displaA2 , * .
displaA2 = 'CONSOLE' , 'SYSPUNCH' , 'SYSOUT' , 'IDENT' .

divide   = 'DIVIDE' id_cte divideA1 divideA2 divideA5 .
divideA1 = 'INTO'   [ divideA3 divideA4 virgul ] ,
           'BY'       divideA3 .
divideA2 = 'GIVING' [ identi divideA4 virgul ] .
divideA3 = identi , consta .
divideA4 = 'ROUNDED' , * .
```

```
divideA5 = 'REMAINDER' identi , * .

exit     = 'EXIT' .

goto     = 'GO' 'TO' [ 'IDENT' ] gotoA1 .
gotoA1   = 'DEPENDING' 'ON' identi  , * .

if       = 'IF' EX LC ifA1 .
ifA1     = 'ELSE' LC , * .

inspec   = 'INSPECT' identi [ inspecA1 ] .
inspecA1 = 'TALLYING'  identi 'FOR' inspecA2
           [ ',' identi 'FOR' inspecA2 ] ,
           'REPLACING' inspecA4 [ ',' inspecA4 ] .
inspecA2 = [ inspecA3 virgul ] .
inspecA3 = 'CHARACTERS' inspecA6 ,
           inspecA5 [ inspecA8 inspecA6 virgul ] .
inspecA4 = 'CHARACTERS' 'BY' inspecA8 inspecA6 ,
           inspecA5 [ inspecA8 'BY' inspecA8 inspecA6 virgul ] .
inspecA5 = 'ALL' , 'LEADING' , 'FIRST' .
inspecA6 = [ inspecA7 virgul ] .
inspecA7 = 'AFTER'  'INITIAL' inspecA8 ,
           'BEFORE' 'INITIAL' inspecA8 .
inspecA8 = identi , consta .

invali   = 'INVALID' invaliA1 LC .
invaliA1 = 'KEY' .

merge    = 'MERGE' 'IDENT' [ mergeA1 virgul ] mergeA2 mergeA3 .
mergeA1  = 'ASCENDING' , 'DESCENDING' , 'KEY' , 'ON' , 'COLLATING' ,
           'SEQUENCE' , 'IS' , identi .
mergeA2  = 'USING' [ 'IDENT' virgul ] ,
           'INPUT' 'PROCEDURE' aux01 'IDENT' mergeA4 .
mergeA3  = 'GIVING' [ 'IDENT' virgul ] ,
           'OUTPUT' 'PROCEDURE' aux01 'IDENT' mergeA4 .
mergeA4  = 'THROUGH' 'IDENT' , 'THRU' 'IDENT' , * .

move     = 'MOVE' moveA1 id_cte 'TO' [ identi virgul ] .
moveA1   = 'CORR' , 'CORRESPONDING' , * .

multip   = 'MULTIPLY' id_cte multipA1 multipA2 .
multipA1 = 'BY'      [ multipA3 multipA4 virgul ] .
multipA2 = 'GIVING' [ identi   multipA4 virgul ] , * .
multipA3 = identi , consta .
multipA4 = 'ROUNDED' , * .

next     = 'NEXT' 'SENTENCE' .

on       = 'ON' onA1 LC .
onA1     = 'OVERFLOW' , 'SIZE' 'ERROR' .

open     = 'OPEN' [ openA1 'IDENT' openA2 virgul ] .
openA1   = 'INPUT' , 'OUTPUT' , 'I-O' , 'EXTEND' .
openA2   = 'REVERSED' , 'WITH' 'NO' 'REWIND' , 'WITH' 'LOCK' ,
           'NO' 'REWIND' , 'LOCK' , * .
```

```
perfor   = 'PERFORM' 'IDENT' perforA1 perforA2 .
perforA1 = 'THRU' 'IDENT' , * .
perforA2 = perforA3 , perforA4 , perforA5 , * .
perforA3 = 'UNTIL' EX .
perforA4 = perforA6 'TIMES' .
perforA5 = 'VARYING' identi 'FROM' perforA6 'BY' perforA6 'UNTIL' EX
           [ 'AFTER' identi 'FROM' perforA6 'BY' perforA6 'UNTIL' EX ] .
perforA6 = identi , consta .

read     = 'READ' identi [ readA1 ] readA2 .
readA1   = 'RECORD' , 'NEXT' .
readA2   = 'INTO' identi , * .

releas   = 'RELEASE' identi releasA1 .
releasA1 = 'FROM' identi , * .

return   = 'RETURN' identi returnA1 returnA2 .
returnA1 = 'RECORD' , * .
returnA2 = 'INTO' identi , * .

search   = 'SEARCH' searchA1 identi searchA2 searchA3 searchA4 .
searchA1 = 'ALL' , * .
searchA2 = 'VARYING' identi , * .
searchA3 = 'AT' 'END' LC , * .
searchA4 = [ 'WHEN' EX LC ] .

set      = 'SET' [ identi virgul ] setA1 id_cte .
setA1    = 'TO' , 'UP' 'BY' , 'DOWN' 'BY' .

sort     = 'SORT' 'IDENT' [ sortA1 virgul ] sortA2 sortA3 .
sortA1   = 'ASCENDING' , 'DESCENDING' , 'KEY' , 'ON' , 'COLLATING' ,
           'SEQUENCE' , 'IS' , identi .
sortA2   = 'USING' [ 'IDENT' virgul ] ,
           'INPUT' 'PROCEDURE' aux01 'IDENT' sortA4 .
sortA3   = 'GIVING' [ 'IDENT' virgul ] ,
           'OUTPUT' 'PROCEDURE' aux01 'IDENT' sortA4 .
sortA4   = 'THROUGH' 'IDENT' , 'THRU' 'IDENT' , * .

start    = 'START' 'IDENT' startA1 .
startA1  = 'KEY' aux01 EXrelaA1 identi , * .

stop     = 'STOP' stopA1 .
stopA1   = stopA2 , stopA3 .
stopA2   = 'RUN' .
stopA3   = 'CONST' .

string   = 'STRING' stringA1 [ ',' stringA1 ] 'INTO' identi stringA2 .
stringA1 = [ stringA3 virgul ] 'DELIMITED' 'BY' stringA3 .
stringA2 = 'WITH' 'POINTER' identi , 'POINTER' identi , * .
stringA3 = identi , consta , 'SIZE' .

subtra   = 'SUBTRACT' subtraA1 [ id_cte virgul ] subtraA2 subtraA3 .
subtraA1 = 'CORR' , 'CORRESPONDING' , * .
subtraA2 = 'FROM'    [ subtraA4 subtraA5 virgul ] .
subtraA3 = 'GIVING' [ identi   subtraA5 virgul ] , * .
subtraA4 = identi , consta .
subtraA5 = 'ROUNDED' , * .
```

```
unstri    = 'UNSTRING' identi 'DELIMITED' [ unstriA1 virgul ]
            'INTO' [ identi unstriA5 unstriA6 virgul ] unstriA7
            unstriA8 .
unstriA1  = unstriA2 unstriA3 unstriA4 .
unstriA2  = 'BY' , 'OR' .
unstriA3  = 'ALL' , * .
unstriA4  = identi , consta .
unstriA5  = 'DELIMITER' 'IN' identi , * .
unstriA6  = 'COUNT' 'IN' identi , * .
unstriA7  = 'WITH' 'POINTER' identi , 'POINTER' identi , * .
unstriA8  = 'TALLYING' 'IN' identi , * .

use       = 'USE' 'AFTER' [ useA1 ] 'PROCEDURE' useA2 useA3 '.' .
useA1     = 'STANDARD' , 'EXCEPTION' , 'ERROR' .
useA2     = 'ON' , * .
useA3     = 'INPUT' , 'OUTPUT' , 'I-O' , 'EXTEND' , [ 'IDENT' virgul ] .

write     = writeA1 identi writeA2 [ writeA3 ] .
writeA1   = 'WRITE' , 'REWRITE' .
writeA2   = 'FROM' 'IDENT' , * .
writeA3   = 'AFTER' , 'BEFORE' , 'ADVANCING' , 'POSITIONING' , 'IDENT' ,
            'CONST' , 'LINES' , 'LINE' , 'PAGE' .


EX        = EXelog [ 'AND' EXelog ] .

EXelog    = EXrela [ 'OR' EXrela ] .

EXrela    = EXmult [ EXrelaA1 EXmult ] .
EXrelaA1  = '>' EXrelaA2 , '<' EXrelaA2 , '=' EXrelaA3 ,
            'GREATER' EXrelaA2 , 'LESS' EXrelaA2 , 'EQUAL' EXrelaA3 .
EXrelaA2  = 'THAN' , * .
EXrelaA3  = 'TO' , * .

EXmult    = EXadit [ EXmultA1 EXadit ] .
EXmultA1  = '*' , '/' , '**' .

EXadit    = EXunar [ EXaditA1 EXunar ] .
EXaditA1  = '+' , '-' .

EXunar    = EXaux [ EXunarA1 ] .
EXunarA1  = 'NUMERIC' , 'ALPHABETIC' , 'POSITIVE' , 'NEGATIVE' ,
            'ZERO' , 'ZEROES' .

EXaux     = EXprim EXauxA1 .
EXauxA1   = 'NOT' , 'IS' EXauxA2 , * .
EXauxA2   = 'NOT' , * .

EXprim    = identi , consta , '(' EX ')' , 'NOT' EXprim .

consta    = 'CONST' , 'ZERO' , 'ZEROS' , 'ZEROES' , 'SPACE' , 'SPACES' ,
            'HIGH-VALUE' , 'HIGH-VALUES' , 'QUOTE' , 'QUOTES' ,
            'LOW-VALUE' , 'LOW-VALUES' , 'ALL' consta .

identi    = 'IDENT' identiA1 identiA2 .
identiA1  = 'OF' 'IDENT' , 'IN' 'IDENT' , * .
identiA2  = '(' [ id_cte identiA3 virgul ] ')' , * .
identiA3  = '+' consta , '-' consta , * .
```

APÊNDICE D

UM EXEMPLO COMPLETO

Neste apêndice é apresentado um exemplo completo utilizado para ilustrar a aplicação da POKE-TOOL no teste de programas escritos em linguagem COBOL; são mostradas as informações geradas pela ferramenta em uma sessão de teste.

O programa usado aqui, chamado DEMO.CBL, tem como função atualizar um cadastro de itens de estoque, denominado CADAST.DAT. O arquivo de movimentos, nomeado MOVIM.DAT, contém informação de inclusão e exclusão de itens para esse cadastro. O cadastro atualizado é gerado no arquivo CADAST.OUT. Adicionalmente, é produzido um relatório de ocorrências de movimentação de estoque, denominado OCORR.PRN.

Para distinguir entre informações geradas pela POKE-TOOL e comentários, foi adotada a seguinte convenção: o que for relativo à ferramenta será descrito em "font" de maquina de escrever e os comentários são escritos em itálico.

O programa DEMO.CBL é apresentado a seguir:

```
000000 IDENTIFICATION DIVISION.
000000 PROGRAM-ID. TESTE01.
000000 AUTHOR. PLINIO DE SA LEITAO JUNIOR.
000000 ENVIRONMENT DIVISION.
000000 CONFIGURATION SECTION.
000000 SOURCE-COMPUTER. NEXUS3600.
000000 OBJECT-COMPUTER. NEXUS3600.
000000 SPECIAL-NAMES.
000000     DECIMAL-POINT IS COMMA.
000000 INPUT-OUTPUT SECTION.
000000 FILE-CONTROL.
000000     SELECT CAD-TAB ASSIGN TO "TABELA.DAT"
000000       ORGANIZATION IS LINE SEQUENTIAL.
000000     SELECT CAD-IN  ASSIGN TO "CADAST.DAT"
000000       ORGANIZATION IS LINE SEQUENTIAL.
000000     SELECT CAD-OUT ASSIGN TO "CADAST.OUT"
000000       ORGANIZATION IS LINE SEQUENTIAL.
000000     SELECT MOVIM   ASSIGN TO "MOVIM.DAT"
000000       ORGANIZATION IS LINE SEQUENTIAL.
000000     SELECT RELAT   ASSIGN TO "OCORR.PRN"
000000       ORGANIZATION IS LINE SEQUENTIAL.
```

```
000000 DATA DIVISION.
000000 FILE SECTION.
000000 FD  CAD-TAB.
000000 01  REG-CAD-TAB                    PIC 9(04).
000000 FD  CAD-IN
000000     RECORD CONTAINS 50 CHARACTERS.
000000 01  REG-CAD-IN.
000000     02  CI-NUM-ITEM                 PIC 9(04).
000000     02  FILLER                      PIC X(46).
000000 FD  CAD-OUT
000000     RECORD CONTAINS 50 CHARACTERS.
000000 01  REG-CAD-OUT                    PIC X(50).
000000 FD  MOVIM
000000     RECORD CONTAINS 60 CHARACTERS
000000     DATA RECORDS ARE REG-MOVIM-1
000000                      REG-MOVIM-2.
000000 01  REG-MOVIM-1.
000000     02  M-DATA-N                    PIC 9(06).
000000     02  FILLER                      PIC X(54).
000000 01  REG-MOVIM-2.
000000     02  M-COD-REG                   PIC X(03).
000000     02  M-INFO.
000000         03  M-NUM-ITEM              PIC 9(04).
000000         03  M-DESCR-ITEM            PIC X(30).
000000         03  M-QTDE-ITEM             PIC 9(03).
000000         03  M-PRECO-ITEM            PIC 9(09)V99.
000000     02  FILLER                      PIC X(09).
000000 FD  RELAT
000000     RECORD CONTAINS 132 CHARACTERS.
000000 01  REG-RELAT                      PIC X(132).
000000*
000000 WORKING-STORAGE SECTION.
000000 01  TABELA-ITENS  OCCURS 50 TIMES
000000                   ASCENDING KEY ITEM-TAB
000000                   INDEXED BY IND-TAB.
000000     02  ITEM-TAB                    PIC 9(04).
000000 01  CONTADORES.
000000     02  CONT-LIN                    PIC 9(02).
000000     02  CONT-PAG                    PIC 9(02).
000000 01  FLAGS.
000000     02  FLAG-FIM-CAD-TAB            PIC 9(01).
000000         88  FIM-CAD-TAB VALUE 1.
000000     02  FLAG-ITEM-VALIDO            PIC 9(01).
000000         88  ITEM-NAO-VALIDO VALUE 0.
000000         88  ITEM-VALIDO     VALUE 1.
000000*
000000*    DEFINICAO DE LINHAS DE IMPRESSAO
000000*
000000 01  W-CABEC-1.
000000     02  FILLER                      PIC X(44) VALUE
000000        "** COMPANHIA DE PRODUTOS INDUSTRIALIZADOS - ".
000000     02  FILLER                      PIC X(32) VALUE
000000        "DIVISAO DE ESTOQUE **      PAG. ".
000000     02  W-PAG                       PIC 9(03).
000000 01  W-CABEC-2.
000000     02  FILLER                      PIC X(42) VALUE
000000        "OCORRENCIAS DE MOVIMENTACAO DE ESTOQUE EM ".
```

```
000000      02  W-DATA                  PIC 99/99/99.
000000 01   W-DETALHE.
000000      02  W-COD-REG               PIC X(03).
000000      02  FILLER                  PIC X(02) VALUE SPACES.
000000      02  W-NUM-ITEM              PIC 9(04).
000000      02  FILLER                  PIC X(02) VALUE SPACES.
000000      02  W-DESCR-ITEM            PIC X(30).
000000      02  FILLER                  PIC X(02) VALUE SPACES.
000000      02  W-QTDE-ITEM             PIC 9(03).
000000      02  FILLER                  PIC X(02) VALUE SPACES.
000000      02  W-PRECO-ITEM            PIC ZZZ.ZZZ.ZZ9,99.
000000      02  FILLER                  PIC X(02) VALUE SPACES.
000000      02  W-OBS                   PIC X(20).
000000/*****************************
000000* DIVISAO DE PROCEDIMENTOS
000000*****************************
000000 PROCEDURE DIVISION.
000000 PROCESSAMENTO SECTION.
000000 PROGRAMA.
000000*
000000* ABRE ARQUIVOS
000000*
000000      OPEN INPUT  MOVIM
000000           INPUT  CAD-IN
000000           OUTPUT CAD-OUT
000000           OUTPUT RELAT.
000000*
000000* INICIA VARIAVEIS
000000*
000000      PERFORM INICIA-PROC
000000*
000000* PROCESSA TRANSACOES DE MOVIMENTACAO DE REGISTROS ATE
000000*   FINAL DOS ARQUIVOS DE CADASTRO DE ENTRADA E MOVIMENTO
000000*
000000      PERFORM PROCESSA-TRANSACAO
000000         UNTIL M-NUM-ITEM  EQUAL 9999  AND
000000               CI-NUM-ITEM EQUAL 9999
000000*
000000* FECHA ARQUIVOS
000000*
000000      CLOSE MOVIM
000000            CAD-IN
000000            CAD-OUT
000000            RELAT.
000000*
000000* ENCERRA PROCESSAMENTO
000000*
000000      STOP RUN.
000000/*****************************
000000*    INICIACAO DE VARIAVEIS
000000*****************************
000000 INICIA-PROC.
000000*
000000* INICIA TABELA DE ITENS VALIDOS
000000*
000000      OPEN INPUT CAD-TAB
000000      MOVE 0  TO FLAG-FIM-CAD-TAB
```

```
000000      PERFORM LE-CAD-TAB
000000      PERFORM INICIA-TAB-ITENS
000000        VARYING IND-TAB FROM 1 BY 1
000000        UNTIL   IND-TAB > 50
000000      CLOSE CAD-TAB
000000*
000000* INICIA CONTADORES
000000*
000000      MOVE 99 TO CONT-LIN
000000      MOVE 0  TO CONT-PAG
000000*
000000* INICIA CABECALHO PARA LINHA DE IMPRESSAO
000000*
000000      PERFORM LE-MOV
000000      MOVE M-DATA-N TO W-DATA
000000*
000000* LE CADASTRO DE ENTRADA E LE MOVIMENTO
000000*
000000      PERFORM LE-CAD
000000      PERFORM LE-MOV.
000000/**************************************
000000*    CARREGA TABELA DE ITENS VALIDOS
000000**************************************
000000 INICIA-TAB-ITENS.
000000      IF FIM-CAD-TAB
000000         MOVE 9999 TO ITEM-TAB (IND-TAB)
000000      ELSE
000000         MOVE REG-CAD-TAB TO ITEM-TAB (IND-TAB)
000000         PERFORM LE-CAD-TAB.
000000/**********************************
000000* PROCESSA TRANSACAO DE MOVIMENTO
000000**********************************
000000 PROCESSA-TRANSACAO.
000000*
000000* PESQUISA ITEM DE MOVIMENTO NA TABELA DE ITENS VALIDOS
000000*
000000      SEARCH ALL TABELA-ITENS
000000        AT END
000000              MOVE 0 TO FLAG-ITEM-VALIDO
000000        WHEN  ITEM-TAB (IND-TAB) EQUAL M-NUM-ITEM
000000              MOVE 1 TO FLAG-ITEM-VALIDO.
000000*
000000* TESTA SE O ITEM DO MOVIMENTO EXISTE
000000* TRATA CODIGO DO MOVIMENTO
000000*
000000      IF ITEM-NAO-VALIDO
000000         MOVE "ITEM NAO VALIDO" TO W-OBS
000000      ELSE
000000        IF M-COD-REG = "INC"
000000           PERFORM PROCESSA-INCLUSAO
000000        ELSE
000000           IF M-COD-REG = "EXC"
000000              PERFORM PROCESSA-EXCLUSAO
000000           ELSE
000000              MOVE M-COD-REG TO W-COD-REG
000000              MOVE "COD. REG. INVALIDO" TO W-OBS.
```

```
000000*
000000* IMPRIME REGISTRO DE MOVIMENTO NO RELATORIO DE OCORRENCIAS
000000*
000000      PERFORM IMPR-RELAT
000000*
000000* LE PROXIMO REGISTRO DE MOVIMENTO
000000*
000000      PERFORM LE-MOV
000000*
000000* CASO SEJA FINAL DO ARQUIVO DE MOVIMENTOS, COPIAR OS
000000*   REGISTROS RESTANTES DO CADASTRO DE ENTRADA PARA O
000000*   CADASTRO DE SAIDA
000000*
000000      IF M-NUM-ITEM EQUAL 9999
000000         PERFORM COPIA-CAD
000000           UNTIL CI-NUM-ITEM EQUAL 9999.
000000/************************************************
000000*    PROCESSA MOVIMENTO DE INCLUSAO DE REGISTRO
000000************************************************
000000 PROCESSA-INCLUSAO.
000000*
000000* POSICIONA O CADASTRO DE ENTRADA, COPIANDO REGISTROS DESSE
000000*   CADASTRO PARA O CADASTRO DE SAIDA
000000*
000000      PERFORM COPIA-CAD
000000        UNTIL CI-NUM-ITEM NOT LESS M-NUM-ITEM
000000*
000000* TESTA SE O REGISTRO DO MOVIMENTO JAH EXISTE NO CADASTRO DE
000000*   ENTRADA; EM CASO NEGATIVO, INCLUI O REGISTRO DO
000000*   MOVIMENTO NO CADASTRO DE SAIDA
000000*
000000      IF M-NUM-ITEM EQUAL CI-NUM-ITEM
000000         MOVE "REG. JA EXISTENTE" TO W-OBS
000000      ELSE
000000         MOVE "REG. INCLUIDO" TO W-OBS
000000         WRITE REG-CAD-OUT FROM M-INFO.
000000/************************************************
000000*    PROCESSA MOVIMENTO DE EXCLUSAO DE REGISTRO
000000************************************************
000000 PROCESSA-EXCLUSAO.
000000*
000000* POSICIONA O CADASTRO DE ENTRADA, COPIANDO REGISTROS DESSE
000000*   CADASTRO PARA O CADASTRO DE SAIDA
000000*
000000      PERFORM COPIA-CAD
000000        UNTIL CI-NUM-ITEM NOT LESS M-NUM-ITEM
000000*
000000* TESTA SE O REGISTRO DO MOVIMENTO NAO EXISTE NO CADASTRO DE
000000*   ENTRADA; EM CASO NEGATIVO, NAO INCLUI O REGISTRO DO
000000*   CADASTRO DE ENTRADA NO CADASTRO DE SAIDA
000000*
000000      IF M-NUM-ITEM GREATER CI-NUM-ITEM
000000         MOVE "REG. INEXISTENTE" TO W-OBS
000000      ELSE
000000         MOVE "REG. EXCLUIDO" TO W-OBS
000000         PERFORM LE-CAD.
```

```
000000/*****************************************************************
000000* COPIA REGISTRO DE ENTRADA PARA O CADASTRO DE SAIDA
000000*****************************************************************
000000 COPIA-CAD.
000000*
000000* GERA REGISTRO NO CADASTRO DE SAIDA, A PARTIR DE UM
000000*   REGISTRO DO CADASTRO DE ENTRADA
000000*
000000     WRITE REG-CAD-OUT FROM REG-CAD-IN
000000*
000000* LE PROXIMO REGISTRO DO CADASTRO DE ENTRADA
000000*
000000     PERFORM LE-CAD.
000000/*****************************************************************
000000*    IMPRIME MOVIMENTOS NO RELATORIO DE OCORRENCIAS
000000*****************************************************************
000000 IMPR-RELAT.
000000     IF CONT-LIN > 60
000000        ADD 1 TO CONT-PAG
000000        MOVE CONT-PAG TO W-PAG
000000        WRITE REG-RELAT FROM W-CABEC-1 BEFORE 1 LINE
000000        WRITE REG-RELAT FROM W-CABEC-2 BEFORE 2 LINE
000000        MOVE 4 TO CONT-LIN.
000000     MOVE M-COD-REG      TO W-COD-REG
000000     MOVE M-NUM-ITEM     TO W-NUM-ITEM
000000     MOVE M-DESCR-ITEM   TO W-DESCR-ITEM
000000     MOVE M-QTDE-ITEM    TO W-QTDE-ITEM
000000     MOVE M-PRECO-ITEM   TO W-PRECO-ITEM
000000     WRITE REG-RELAT FROM W-DETALHE
000000     ADD 1 TO CONT-LIN.
000000/*************************************************
000000*    LE O PROXIMO REGISTRO DO CADASTRO DE ENTRADA
000000*************************************************
000000 LE-CAD.
000000* LE REGISTRO NO CADASTRO DE MOVIMENTO; SE FINAL DE ARQUIVO,
000000*   MOVE 9999 PARA NUMERO DE ITEM DO REGISTRO DESSE CADASTRO
000000*
000000     READ CAD-IN
000000       AT END
000000          MOVE 9999 TO CI-NUM-ITEM.
000000/*************************************************
000000*    LE O PROXIMO REGISTRO DO CADASTRO DE SAIDA
000000*************************************************
000000 LE-MOV.
000000*
000000* LE REGISTRO NO CADASTRO DE SAIDA; SE FINAL DE ARQUIVO,
000000*   MOVE 9999 PARA NUMERO DE ITEM DO REGISTRO DESSE CADASTRO
000000*
000000     READ MOVIM
000000       AT END
000000          MOVE 9999 TO M-NUM-ITEM.
000000/*********************************
000000*    LE CADASTRO DE ITENS VALIDOS
000000*********************************
000000 LE-CAD-TAB.
000000*
000000* LE CADASTRO DE ITENS VALIDOS; SE FINAL DE ARQUIVO,
```

```
000000*   SETA FLAG DE FINAL DESSE ARQUIVO
000000*
000000      READ CAD-TAB
000000        AT END
000000           MOVE 1 TO FLAG-FIM-CAD-TAB.
```

O arquivo DEMO.LI, que contém a versão em linguagem intermediária (LI) do programa DEMO.CBL, é apresentado abaixo:

```
$DCL        8      24       1
$DCL        41      20       2
$DCL        70      35       3
$DCL        114      21       4
$DCL        144      22       5
$DCL        175      27       6
$DCL        211      27       7
$DCL        247      50       8
$DCL        306      21      10
$DCL        336      13      11
$DCL        362      84      12
$DCL        459      84      14
$DCL        556      84      16
$DCL        653      83      18
$DCL        749      83      20
$DCL        841      14      22
$DCL        864      13      23
$DCL        886      12      24
$DCL        907      42      25
$DCL        958      53      26
$DCL        1020      15      28
$DCL        1048      38      29
$DCL        1099      38      30
$DCL        1146      54      31
$DCL        1209      42      33
$DCL        1260     134      34
$DCL        1403      16      38
$DCL        1432      38      39
$DCL        1483      38      40
$DCL        1530      16      41
$DCL        1559      38      42
$DCL        1610      11      43
$DCL        1638      34      44
$DCL        1689      34      45
$DCL        1740      34      46
$DCL        1791      37      47
$DCL        1841      38      48
$DCL        1888      53      49
$DCL        1950      43      51
$DCL        2011      24      53
$DCL        2044     128      54
```

```
$DCL          2185      38      57
$DCL          2232      15      58
$DCL          2260      38      59
$DCL          2311      38      60
$DCL          2358      10      61
$DCL          2381      38      62
$DCL          2436      24      63
$DCL          2473      38      64
$DCL          2528      28      65
$DCL          2573      28      66
$DCL          2673      14      70
$DCL          2700     106      71
$DCL          2819      94      73
$DCL          2926      38      75
$DCL          2973      14      76
$DCL          3000     104      77
$DCL          3117      41      79
$DCL          3167      14      80
$DCL          3194     110      81
$DCL          3317      93      83
$DCL          3419      14      85
$DCL          3446      38      86
$DCL          3497      51      87
$DCL          3561      38      88
$DCL          3612      51      89
$DCL          3676      38      90
$DCL          3727      51      91
$DCL          3791      38      92
$DCL          3842      51      93
$DCL          3906      47      94
$DCL          3966      51      95
$DCL          4030      38      96
$DCL          4181      19     100
{             0       0       0
PROCESSAMENTO 4209      22     101
PROGRAMA      4240       9     102
{             0       0       0
$S01          4303     111     106
$S02          4471      19     113
$WHILE        0       0       0
$NC(02)01     4699      77     119
$S03          4657      26     118
$S04          4831      87     124
$RETURN       4980       9     131
}             0       0       0
INICIA-PROC   5105      12     135
{             0       0       0
$S05          5188      18     139
$S06          5219      27     140
$S07          5259      18     141
$FOR          5329       7     143
$S08          5337      14     143
$NC(01)02     5371      20     144
$S09          5352       4     143
$S10          5290      24     142
$S11          5404      13     145
$S12          5475      19     149
```

```
$S13          5507      19     150
$S14          5607      14     154
$S15          5634      23     155
$S16          5735      14     159
$S17          5762      15     160
}             0       0       0
INICIA-TAB-ITENS 5920     17     164
{             0       0       0
$IF           5950       2     165
$C(01)03      5953      11     165
{             0       0       0
$S18          5980      31     166
}             0       0       0
$ELSE         6024       4     167
{             0       0       0
$S19          6044      38     168
$S20          6098      18     169
}             6116       1     169
}             0       0       0
PROCESSA-TRANSACAO 6251      19     173
{             0       0       0
$FOR          6364      10     177
$S21          0       0       0
$C(01)04      6375      12     177
$S22          0       0       0
{             0       0       0
$IF           6470       4     180
$C(01)05      6476      35     180
{             0       0       0
$S24          6532      26     181
$GOTO         0       0       0
SEARCH01      0       0       0
}             0       0       0
}             6558       1     181
$S23          6429      26     179
SEARCH01      0       0       0
$IF           6670       2     186
$C(01)06      6673      15     186
{             0       0       0
$S25          6704      31     187
}             0       0       0
$ELSE         6748       4     188
{             0       0       0
$IF           6767       2     189
$C(01)07      6770      17     189
{             0       0       0
$S26          6805      25     190
}             0       0       0
$ELSE         6845       4     191
{             0       0       0
$IF           6867       2     192
$C(01)08      6870      17     192
{             0       0       0
$S27          6908      25     193
}             0       0       0
$ELSE         6951       4     194
{             0       0       0
```

```
$S28          6976      27     195
$S29          7024      34     196
}             0         0      0
}             0         0      0
}             7058      1      196
$S30          7157      18     200
$S31          7248      14     204
$IF           7433      2      209
$C(01)09      7436      21     209
{             0         0      0
$WHILE        0         0      0
$NC(01)10     7508      28     211
$S32          7473      17     210
}             7536      1      211
}             0         0      0
PROCESSA-INCLUSAO 7713  18     215
{             0         0      0
$WHILE        0         0      0
$NC(01)11     7906      37     221
$S33          7874      17     220
$IF           8143      2      227
$C(01)12      8146      28     227
{             0         0      0
$S34          8190      33     228
}             0         0      0
$ELSE         8236      4      229
{             0         0      0
$S35          8256      29     230
$S36          8301      29     231
}             8330      1      231
}             0         0      0
PROCESSA-EXCLUSAO 8507  18     235
{             0         0      0
$WHILE        0         0      0
$NC(01)13     8700      37     241
$S37          8668      17     240
$IF           8951      2      247
$C(01)14      8954      30     247
{             0         0      0
$S38          9000      32     248
}             0         0      0
$ELSE         9045      4      249
{             0         0      0
$S39          9065      29     250
$S40          9110      14     251
}             9124      1      251
}             0         0      0
COPIA-CAD     9346      10     255
{             0         0      0
$S41          9493      33     260
$S42          9609      15     264
}             0         0      0
IMPR-RELAT    9845      11     268
{             0         0      0
$IF           9869      2      269
$C(01)15      9872      13     269
{             0         0      0
```

```
$S43          9901      17     270
$S44          9934      22     271
$S45          9972      44     272
$S46          10032      44     273
$S47          10092      44     274
$S48          10152      18     275
}             10170       1     275
$S49          10184      31     276
$S50          10228      32     277
$S51          10273      34     278
$S52          10320      33     279
$S53          10366      34     280
$S54          10413      30     281
$S55          10456      18     282
}             0       0       0
LE-CAD        10656       7     286
{             0       0       0
$S56          10821      11     290
$IF           10847       2     291
$C(01)16      10850       3     291
{             0       0       0
$S57          10871      24     292
}             10895       1     292
}             0       0       0
LE-MOV        11072       7     296
{             0       0       0
$S58          11242      10     301
$IF           11267       2     302
$C(01)17      11270       3     302
{             0       0       0
$S59          11291      23     303
}             11314       1     303
}             0       0       0
LE-CAD-TAB    11449      11     307
{             0       0       0
$S60          11595      12     312
$IF           11622       2     313
$C(01)18      11625       3     313
{             0       0       0
$S61          11646      26     314
}             11672       1     314
```

O programa em LI é numerado através da inserção de números de nós; esta nova versão, denominada DEMO.NLI, é apresentada abaixo:

```
$DCL    1     8    24     1
$DCL    1    41    20     2
$DCL    1    70    35     3
$DCL    1   114    21     4
$DCL    1   144    22     5
$DCL    1   175    27     6
$DCL    1   211    27     7
$DCL    1   247    50     8
$DCL    1   306    21    10
$DCL    1   336    13    11
$DCL    1   362    84    12
$DCL    1   459    84    14
$DCL    1   556    84    16
$DCL    1   653    83    18
$DCL    1   749    83    20
$DCL    1   841    14    22
$DCL    1   864    13    23
$DCL    1   886    12    24
$DCL    1   907    42    25
$DCL    1   958    53    26
$DCL    1  1020    15    28
$DCL    1  1048    38    29
$DCL    1  1099    38    30
$DCL    1  1146    54    31
$DCL    1  1209    42    33
$DCL    1  1260   134    34
$DCL    1  1403    16    38
$DCL    1  1432    38    39
$DCL    1  1483    38    40
$DCL    1  1530    16    41
$DCL    1  1559    38    42
$DCL    1  1610    11    43
$DCL    1  1638    34    44
$DCL    1  1689    34    45
$DCL    1  1740    34    46
$DCL    1  1791    37    47
$DCL    1  1841    38    48
$DCL    1  1888    53    49
$DCL    1  1950    43    51
$DCL    1  2011    24    53
$DCL    1  2044   128    54
$DCL    1  2185    38    57
$DCL    1  2232    15    58
$DCL    1  2260    38    59
$DCL    1  2311    38    60
$DCL    1  2358    10    61
$DCL    1  2381    38    62
$DCL    1  2436    24    63
$DCL    1  2473    38    64
$DCL    1  2528    28    65
$DCL    1  2573    28    66
$DCL    1  2673    14    70
```

$DCL	1	2700	106	71
$DCL	1	2819	94	73
$DCL	1	2926	38	75
$DCL	1	2973	14	76
$DCL	1	3000	104	77
$DCL	1	3117	41	79
$DCL	1	3167	14	80
$DCL	1	3194	110	81
$DCL	1	3317	93	83
$DCL	1	3419	14	85
$DCL	1	3446	38	86
$DCL	1	3497	51	87
$DCL	1	3561	38	88
$DCL	1	3612	51	89
$DCL	1	3676	38	90
$DCL	1	3727	51	91
$DCL	1	3791	38	92
$DCL	1	3842	51	93
$DCL	1	3906	47	94
$DCL	1	3966	51	95
$DCL	1	4030	38	96
$DCL	1	4181	19	100
{	1	0	0	0
PROCESSAMENTO	2	4209	22	101
PROGRAMA	2	4240	9	102
{	2	0	0	0
$S01	2	4303	111	106
$S02	2	4471	19	113
$WHILE	3	0	0	0
$NC(02)01	3	4699	77	119
{	4	0	0	0
$S03	4	4657	26	118
}	4	0	0	0
$S04	5	4831	87	124
$RETURN	5	4980	9	131
}	5	0	0	0
INICIA-PROC	6	5105	12	135
{	6	0	0	0
$S05	6	5188	18	139
$S06	6	5219	27	140
$S07	6	5259	18	141
$FOR	6	5329	7	143
$S08	6	5337	14	143
$NC(01)02	7	5371	20	144
$S09	8	5352	4	143
{	8	0	0	0
$S10	8	5290	24	142
}	8	0	0	0
$S11	9	5404	13	145
$S12	9	5475	19	149
$S13	9	5507	19	150
$S14	9	5607	14	154
$S15	9	5634	23	155
$S16	9	5735	14	159
$S17	9	5762	15	160
}	9	0	0	0
INICIA-TAB-ITENS	10	5920	17	164

{	10	0	0	0
$IF	10	5950	2	165
$C(01)03	10	5953	11	165
{	11	0	0	0
$S18	11	5980	31	166
}	11	0	0	0
$ELSE	12	6024	4	167
{	12	0	0	0
$S19	12	6044	38	168
$S20	12	6098	18	169
}	12	6116	1	169
}	13	0	0	0
PROCESSA-TRANSACAO	14	6251	19	173
{	14	0	0	0
$FOR	14	6364	10	177
$S21	14	0	0	0
$C(01)04	15	6375	12	177
$S22	18	0	0	0
{	16	0	0	0
$IF	16	6470	4	180
$C(01)05	16	6476	35	180
{	17	0	0	0
$S24	17	6532	26	181
$GOTO	17	0	0	0
SEARCH01	17	0	0	0
}	17	0	0	0
}	18	6558	1	181
$S23	19	6429	26	179
SEARCH01	20	0	0	0
$IF	20	6670	2	186
$C(01)06	20	6673	15	186
{	21	0	0	0
$S25	21	6704	31	187
}	21	0	0	0
$ELSE	22	6748	4	188
{	22	0	0	0
$IF	22	6767	2	189
$C(01)07	22	6770	17	189
{	23	0	0	0
$S26	23	6805	25	190
}	23	0	0	0
$ELSE	24	6845	4	191
{	24	0	0	0
$IF	24	6867	2	192
$C(01)08	24	6870	17	192
{	25	0	0	0
$S27	25	6908	25	193
}	25	0	0	0
$ELSE	26	6951	4	194
{	26	0	0	0
$S28	26	6976	27	195
$S29	26	7024	34	196
}	26	0	0	0
}	27	0	0	0
}	28	7058	1	196
$S30	29	7157	18	200
$S31	29	7248	14	204

```
$IF                     29   7433     2   209
$C(01)09                29   7436    21   209
{                       30      0     0     0
$WHILE                  30      0     0     0
$NC(01)10               30   7508    28   211
{                       31      0     0     0
$S32                    31   7473    17   210
}                       31      0     0     0
}                       32   7536     1   211
}                       33      0     0     0
PROCESSA-INCLUSAO       34   7713    18   215
{                       34      0     0     0
$WHILE                  35      0     0     0
$NC(01)11               35   7906    37   221
{                       36      0     0     0
$S33                    36   7874    17   220
}                       36      0     0     0
$IF                     37   8143     2   227
$C(01)12                37   8146    28   227
{                       38      0     0     0
$S34                    38   8190    33   228
}                       38      0     0     0
$ELSE                   39   8236     4   229
{                       39      0     0     0
$S35                    39   8256    29   230
$S36                    39   8301    29   231
}                       39   8330     1   231
}                       40      0     0     0
PROCESSA-EXCLUSAO       41   8507    18   235
{                       41      0     0     0
$WHILE                  42      0     0     0
$NC(01)13               42   8700    37   241
{                       43      0     0     0
$S37                    43   8668    17   240
}                       43      0     0     0
$IF                     44   8951     2   247
$C(01)14                44   8954    30   247
{                       45      0     0     0
$S38                    45   9000    32   248
}                       45      0     0     0
$ELSE                   46   9045     4   249
{                       46      0     0     0
$S39                    46   9065    29   250
$S40                    46   9110    14   251
}                       46   9124     1   251
}                       47      0     0     0
COPIA-CAD               48   9346    10   255
{                       48      0     0     0
$S41                    48   9493    33   260
$S42                    48   9609    15   264
}                       48      0     0     0
IMPR-RELAT              49   9845    11   268
{                       49      0     0     0
$IF                     49   9869     2   269
$C(01)15                49   9872    13   269
{                       50      0     0     0
$S43                    50   9901    17   270
```

$S44	50	9934	22	271
$S45	50	9972	44	272
$S46	50	10032	44	273
$S47	50	10092	44	274
$S48	50	10152	18	275
}	50	10170	1	275
$S49	51	10184	31	276
$S50	51	10228	32	277
$S51	51	10273	34	278
$S52	51	10320	33	279
$S53	51	10366	34	280
$S54	51	10413	30	281
$S55	51	10456	18	282
}	51	0	0	0
LE-CAD	52	10656	7	286
{	52	0	0	0
$S56	52	10821	11	290
$IF	52	10847	2	291
$C(01)16	52	10850	3	291
{	53	0	0	0
$S57	53	10871	24	292
}	53	10895	1	292
}	54	0	0	0
LE-MOV	55	11072	7	296
{	55	0	0	0
$S58	55	11242	10	301
$IF	55	11267	2	302
$C(01)17	55	11270	3	302
{	56	0	0	0
$S59	56	11291	23	303
}	56	11314	1	303
}	57	0	0	0
LE-CAD-TAB	58	11449	11	307
{	58	0	0	0
$S60	58	11595	12	312
$IF	58	11622	2	313
$C(01)18	58	11625	3	313
{	59	0	0	0
$S61	59	11646	26	314
}	59	11672	0	314
}	60	11672	0	314
}	60	11672	0	314

O arquivo TESTEPROG.CBL é apresentado abaixo; note que, a pseudo-função **ponta_de_prova**, *utilizada na Seção 3.4, para registrar os nós percorridos pelos casos de teste, foi implementada através dos comandos 'MOVE N TO R-NO (R-IND)' e 'PERFORM PONTA-DE-PROVA', onde N denota o número do nó que está sendo monitorado.*

```
000001*    ** VERSAO INSTRUMENTADA **
000002 IDENTIFICATION DIVISION.
000003 PROGRAM-ID. TESTE01.
000004 AUTHOR. PLINIO DE SA LEITAO JUNIOR.
000005 ENVIRONMENT DIVISION.
000006 CONFIGURATION SECTION.
000007 SOURCE-COMPUTER. NEXUS3600.
000008 OBJECT-COMPUTER. NEXUS3600.
000009 SPECIAL-NAMES.
000010     DECIMAL-POINT IS COMMA.
000011 INPUT-OUTPUT SECTION.
000012 FILE-CONTROL.
000013     SELECT PATH ASSIGN TO "PATH.TES"
000014       ORGANIZATION IS LINE SEQUENTIAL.
000015     SELECT CAD-TAB ASSIGN TO "TABELA.DAT"
000016       ORGANIZATION IS LINE SEQUENTIAL.
000017     SELECT CAD-IN  ASSIGN TO "CADAST.DAT"
000018       ORGANIZATION IS LINE SEQUENTIAL.
000019     SELECT CAD-OUT ASSIGN TO "CADAST.OUT"
000020       ORGANIZATION IS LINE SEQUENTIAL.
000021     SELECT MOVIM   ASSIGN TO "MOVIM.DAT"
000022       ORGANIZATION IS LINE SEQUENTIAL.
000023     SELECT RELAT   ASSIGN TO "OCORR.PRN"
000024       ORGANIZATION IS LINE SEQUENTIAL.
000025 DATA DIVISION.
000026 FILE SECTION.
000027 FD PATH.
000028 01 R-PATH.
000029    02 R-NO   PIC 99B  OCCURS 10 TIMES
000030             INDEXED BY R-IND.
000031 FD  CAD-TAB.
000032 01  REG-CAD-TAB                  PIC 9(04).
000033 FD  CAD-IN
000034     RECORD CONTAINS 50 CHARACTERS.
000035 01  REG-CAD-IN.
000036   02  CI-NUM-ITEM              PIC 9(04).
000037   02  FILLER                   PIC X(46).
000038 FD  CAD-OUT
000039     RECORD CONTAINS 50 CHARACTERS.
000040 01  REG-CAD-OUT                  PIC X(50).
000041 FD  MOVIM
000042     RECORD CONTAINS 60 CHARACTERS
000043     DATA RECORDS ARE REG-MOVIM-1
000044                      REG-MOVIM-2.
000045 01  REG-MOVIM-1.
000046   02  M-DATA-N                 PIC 9(06).
```

```
000047    02  FILLER                      PIC X(54).
000048 01  REG-MOVIM-2.
000049    02  M-COD-REG                   PIC X(03).
000050    02  M-INFO.
000051      03  M-NUM-ITEM              PIC 9(04).
000052      03  M-DESCR-ITEM            PIC X(30).
000053      03  M-QTDE-ITEM             PIC 9(03).
000054      03  M-PRECO-ITEM            PIC 9(09)V99.
000055      02  FILLER                      PIC X(09).
000056 FD  RELAT
000057     RECORD CONTAINS 132 CHARACTERS.
000058 01  REG-RELAT                        PIC X(132).
000059 WORKING-STORAGE SECTION.
000060*
000061* Variaveis auxiliares a monitoracao do comando SEARCH
000062*
000063 01 BEFORE-SEARCH PIC 9(06).
000064 01 LOOP-SEARCH   PIC 9(06).
000065 01 INS-LIM-INF   PIC 9(06).
000066 01 INS-LIM-SUP   PIC 9(06).
000067 01 INS-MEIO      PIC 9(06).
000068*
000069 01  TABELA-ITENS  OCCURS 50 TIMES
000070                   ASCENDING KEY ITEM-TAB
000071                   INDEXED BY IND-TAB.
000072    02  ITEM-TAB                   PIC 9(04).
000073 01  CONTADORES.
000074    02  CONT-LIN                   PIC 9(02).
000075    02  CONT-PAG                   PIC 9(02).
000076 01  FLAGS.
000077    02  FLAG-FIM-CAD-TAB           PIC 9(01).
000078    88  FIM-CAD-TAB VALUE 1.
000079    02  FLAG-ITEM-VALIDO           PIC 9(01).
000080    88  ITEM-NAO-VALIDO VALUE 0.
000081    88  ITEM-VALIDO     VALUE 1.
000082 01  W-CABEC-1.
000083 02  FILLER                      PIC X(44) VALUE
000084        "** COMPANHIA DE PRODUTOS INDUSTRIALIZADOS - ".
000085 02  FILLER                      PIC X(32) VALUE
000086        "DIVISAO DE ESTOQUE **      PAG. ".
000087    02  W-PAG                      PIC 9(03).
000088 01  W-CABEC-2.
000089 02  FILLER                      PIC X(42) VALUE
000090        "OCORRENCIAS DE MOVIMENTACAO DE ESTOQUE EM ".
000091    02  W-DATA                     PIC 99/99/99.
000092 01  W-DETALHE.
000093    02  W-COD-REG                  PIC X(03).
000094    02  FILLER                     PIC X(02) VALUE SPACES.
000095    02  W-NUM-ITEM                 PIC 9(04).
000096    02  FILLER                     PIC X(02) VALUE SPACES.
000097    02  W-DESCR-ITEM               PIC X(30).
000098    02  FILLER                     PIC X(02) VALUE SPACES.
000099    02  W-QTDE-ITEM                PIC 9(03).
000100    02  FILLER                     PIC X(02) VALUE SPACES.
000101    02  W-PRECO-ITEM               PIC ZZZ.ZZZ.ZZ9,99.
000102    02  FILLER                     PIC X(02) VALUE SPACES.
000103    02  W-OBS                      PIC X(20).
```

```
000104 PROCEDURE DIVISION.
000105 PROCESSAMENTO SECTION.
000106 PROGRAMA.
000107     OPEN OUTPUT PATH
000108     SET R-IND TO 1
000109** 02 **
000110     MOVE 02 TO R-NO (R-IND)
000111     PERFORM PONTA-DE-PROVA
000112     OPEN INPUT  MOVIM
000113          INPUT  CAD-IN
000114          OUTPUT CAD-OUT
000115          OUTPUT RELAT.
000116     PERFORM INICIA-PROC
000117** 03 **
000118** 04 **
000119     PERFORM PROCESS-PUNTIL-01
000120       UNTIL M-NUM-ITEM  EQUAL 9999  AND
000121             CI-NUM-ITEM EQUAL 9999
000122     MOVE 03 TO R-NO (R-IND)
000123     PERFORM PONTA-DE-PROVA
000124** 05 **
000125     MOVE 05 TO R-NO (R-IND)
000126     PERFORM PONTA-DE-PROVA
000127     CLOSE MOVIM
000128           CAD-IN
000129           CAD-OUT
000130           RELAT.
000131** 60 **
000132     MOVE 60 TO R-NO (R-IND)
000133     PERFORM PONTA-DE-PROVA
000134     SET R-IND TO 99
000135     PERFORM PONTA-DE-PROVA
000136     CLOSE PATH
000137     STOP RUN.
000138 INICIA-PROC.
000139** 06 **
000140     MOVE 06 TO R-NO (R-IND)
000141     PERFORM PONTA-DE-PROVA
000142     OPEN INPUT CAD-TAB
000143     MOVE 0  TO FLAG-FIM-CAD-TAB
000144     PERFORM LE-CAD-TAB
000145** 07 **
000146** 08 **
000147     PERFORM INICIA--PVA-01-01
000148       VARYING IND-TAB FROM 1
000149       BY 1
000150       UNTIL   IND-TAB > 50
000151     MOVE 07 TO R-NO (R-IND)
000152     PERFORM PONTA-DE-PROVA
000153** 09 **
000154     MOVE 09 TO R-NO (R-IND)
000155     PERFORM PONTA-DE-PROVA
000156     CLOSE CAD-TAB
000157     MOVE 99 TO CONT-LIN
000158     MOVE 0  TO CONT-PAG
000159     PERFORM LE-MOV
000160     MOVE M-DATA-N TO W-DATA
```

```
000161      PERFORM LE-CAD
000162      PERFORM LE-MOV.
000163 INICIA-TAB-ITENS.
000164** 10 **
000165      MOVE 10 TO R-NO (R-IND)
000166      PERFORM PONTA-DE-PROVA
000167      IF FIM-CAD-TAB
000168** 11 **
000169        MOVE 11 TO R-NO (R-IND)
000170        PERFORM PONTA-DE-PROVA
000171        MOVE 9999 TO ITEM-TAB (IND-TAB)
000172      ELSE
000173** 12 **
000174        MOVE 12 TO R-NO (R-IND)
000175        PERFORM PONTA-DE-PROVA
000176        MOVE REG-CAD-TAB TO ITEM-TAB (IND-TAB)
000177        PERFORM LE-CAD-TAB.
000178** 13 **
000179      MOVE 13 TO R-NO (R-IND)
000180      PERFORM PONTA-DE-PROVA.
000181 PROCESSA-TRANSACAO.
000182** 14 **
000183      MOVE 14 TO R-NO (R-IND)
000184      PERFORM PONTA-DE-PROVA
000185** 15 **
000186** 18 **
000187      MOVE 1 TO INS-LIM-INF
000188      MOVE 50 TO INS-LIM-SUP
000189      COMPUTE INS-MEIO = (1 + 50) / 2
000190      SEARCH ALL TABELA-ITENS
000191        AT END
000192          PERFORM TABELA--SEARCH-01
000193            UNTIL (INS-MEIO EQUAL IND-TAB  OR
000194                   INS-LIM-SUP LESS INS-LIM-INF)
000195          MOVE 15 TO R-NO (R-IND)
000196          PERFORM PONTA-DE-PROVA
000197** 19 **
000198          MOVE 19 TO R-NO (R-IND)
000199          PERFORM PONTA-DE-PROVA
000200          MOVE 0 TO FLAG-ITEM-VALIDO
000201** 16 **
000202        WHEN ITEM-TAB (IND-TAB) EQUAL M-NUM-ITEM
000203          PERFORM TABELA--SEARCH-01
000204            UNTIL (INS-MEIO EQUAL IND-TAB  OR
000205                   INS-LIM-SUP LESS INS-LIM-INF)
000206          MOVE 15 TO R-NO (R-IND)
000207          PERFORM PONTA-DE-PROVA
000208          MOVE 16 TO R-NO (R-IND)
000209          PERFORM PONTA-DE-PROVA
000210** 17 **
000211          MOVE 17 TO R-NO (R-IND)
000212          PERFORM PONTA-DE-PROVA
000213          MOVE 1 TO FLAG-ITEM-VALIDO.
000214** 20 **
000215      MOVE 20 TO R-NO (R-IND)
000216      PERFORM PONTA-DE-PROVA
000217      IF ITEM-NAO-VALIDO
```

```
000218** 21 **
000219         MOVE 21 TO R-NO (R-IND)
000220         PERFORM PONTA-DE-PROVA
000221         MOVE "ITEM NAO VALIDO" TO W-OBS
000222       ELSE
000223** 22 **
000224         MOVE 22 TO R-NO (R-IND)
000225         PERFORM PONTA-DE-PROVA
000226         IF M-COD-REG = "INC"
000227** 23 **
000228           MOVE 23 TO R-NO (R-IND)
000229           PERFORM PONTA-DE-PROVA
000230           PERFORM PROCESSA-INCLUSAO
000231           MOVE 28 TO R-NO (R-IND)
000232           PERFORM PONTA-DE-PROVA
000233         ELSE
000234** 24 **
000235           MOVE 24 TO R-NO (R-IND)
000236           PERFORM PONTA-DE-PROVA
000237           IF M-COD-REG = "EXC"
000238** 25 **
000239             MOVE 25 TO R-NO (R-IND)
000240             PERFORM PONTA-DE-PROVA
000241             PERFORM PROCESSA-EXCLUSAO
000242             MOVE 27 TO R-NO (R-IND)
000243             PERFORM PONTA-DE-PROVA
000244             MOVE 28 TO R-NO (R-IND)
000245             PERFORM PONTA-DE-PROVA
000246           ELSE
000247** 26 **
000248             MOVE 26 TO R-NO (R-IND)
000249             PERFORM PONTA-DE-PROVA
000250             MOVE M-COD-REG TO W-COD-REG
000251             MOVE "COD. REG. INVALIDO" TO W-OBS
000252             MOVE 27 TO R-NO (R-IND)
000253             PERFORM PONTA-DE-PROVA
000254             MOVE 28 TO R-NO (R-IND)
000255             PERFORM PONTA-DE-PROVA.
000256** 29 **
000257       MOVE 29 TO R-NO (R-IND)
000258       PERFORM PONTA-DE-PROVA
000259       PERFORM IMPR-RELAT
000260       PERFORM LE-MOV
000261       IF M-NUM-ITEM EQUAL 9999
000262** 30 **
000263** 31 **
000264         PERFORM COPIA-C-PUNTIL-02
000265           UNTIL CI-NUM-ITEM EQUAL 9999
000266         MOVE 30 TO R-NO (R-IND)
000267         PERFORM PONTA-DE-PROVA.
000268** 33 **
000269       MOVE 33 TO R-NO (R-IND)
000270       PERFORM PONTA-DE-PROVA.
000271 PROCESSA-INCLUSAO.
000272** 34 **
000273       MOVE 34 TO R-NO (R-IND)
000274       PERFORM PONTA-DE-PROVA
```

```
000275** 35 **
000276** 36 **
000277      PERFORM COPIA-C-PUNTIL-03
000278        UNTIL CI-NUM-ITEM NOT LESS M-NUM-ITEM
000279      MOVE 35 TO R-NO (R-IND)
000280      PERFORM PONTA-DE-PROVA
000281** 37 **
000282      MOVE 37 TO R-NO (R-IND)
000283      PERFORM PONTA-DE-PROVA
000284      IF M-NUM-ITEM EQUAL CI-NUM-ITEM
000285** 38 **
000286        MOVE 38 TO R-NO (R-IND)
000287        PERFORM PONTA-DE-PROVA
000288        MOVE "REG. JA EXISTENTE" TO W-OBS
000289      ELSE
000290** 39 **
000291        MOVE 39 TO R-NO (R-IND)
000292        PERFORM PONTA-DE-PROVA
000293        MOVE "REG. INCLUIDO" TO W-OBS
000294        WRITE REG-CAD-OUT FROM M-INFO.
000295** 40 **
000296      MOVE 40 TO R-NO (R-IND)
000297      PERFORM PONTA-DE-PROVA.
000298 PROCESSA-EXCLUSAO.
000299** 41 **
000300      MOVE 41 TO R-NO (R-IND)
000301      PERFORM PONTA-DE-PROVA
000302** 42 **
000303** 43 **
000304      PERFORM COPIA-C-PUNTIL-04
000305        UNTIL CI-NUM-ITEM NOT LESS M-NUM-ITEM
000306      MOVE 42 TO R-NO (R-IND)
000307      PERFORM PONTA-DE-PROVA
000308** 44 **
000309      MOVE 44 TO R-NO (R-IND)
000310      PERFORM PONTA-DE-PROVA
000311      IF M-NUM-ITEM GREATER CI-NUM-ITEM
000312** 45 **
000313        MOVE 45 TO R-NO (R-IND)
000314        PERFORM PONTA-DE-PROVA
000315        MOVE "REG. INEXISTENTE" TO W-OBS
000316      ELSE
000317** 46 **
000318        MOVE 46 TO R-NO (R-IND)
000319        PERFORM PONTA-DE-PROVA
000320        MOVE "REG. EXCLUIDO" TO W-OBS
000321        PERFORM LE-CAD.
000322** 47 **
000323      MOVE 47 TO R-NO (R-IND)
000324      PERFORM PONTA-DE-PROVA.
000325 COPIA-CAD.
000326** 48 **
000327      MOVE 48 TO R-NO (R-IND)
000328      PERFORM PONTA-DE-PROVA
000329      WRITE REG-CAD-OUT FROM REG-CAD-IN
000330      PERFORM LE-CAD.
```

```
000331 IMPR-RELAT.
000332** 49 **
000333      MOVE 49 TO R-NO (R-IND)
000334      PERFORM PONTA-DE-PROVA
000335      IF CONT-LIN > 60
000336** 50 **
000337        MOVE 50 TO R-NO (R-IND)
000338        PERFORM PONTA-DE-PROVA
000339        ADD 1 TO CONT-PAG
000340        MOVE CONT-PAG TO W-PAG
000341        WRITE REG-RELAT FROM W-CABEC-1 BEFORE 1 LINE
000342        WRITE REG-RELAT FROM W-CABEC-2 BEFORE 2 LINE
000343        MOVE 4 TO CONT-LIN.
000344** 51 **
000345      MOVE 51 TO R-NO (R-IND)
000346      PERFORM PONTA-DE-PROVA
000347      MOVE M-COD-REG      TO W-COD-REG
000348      MOVE M-NUM-ITEM     TO W-NUM-ITEM
000349      MOVE M-DESCR-ITEM   TO W-DESCR-ITEM
000350      MOVE M-QTDE-ITEM    TO W-QTDE-ITEM
000351      MOVE M-PRECO-ITEM   TO W-PRECO-ITEM
000352      WRITE REG-RELAT FROM W-DETALHE
000353      ADD 1 TO CONT-LIN.
000354 LE-CAD.
000355** 52 **
000356      MOVE 52 TO R-NO (R-IND)
000357      PERFORM PONTA-DE-PROVA
000358      READ CAD-IN
000359      AT END
000360** 53 **
000361        MOVE 53 TO R-NO (R-IND)
000362        PERFORM PONTA-DE-PROVA
000363        MOVE 9999 TO CI-NUM-ITEM.
000364** 54 **
000365      MOVE 54 TO R-NO (R-IND)
000366      PERFORM PONTA-DE-PROVA.
000367 LE-MOV.
000368** 55 **
000369      MOVE 55 TO R-NO (R-IND)
000370      PERFORM PONTA-DE-PROVA
000371      READ MOVIM
000372      AT END
000373** 56 **
000374        MOVE 56 TO R-NO (R-IND)
000375        PERFORM PONTA-DE-PROVA
000376        MOVE 9999 TO M-NUM-ITEM.
000377** 57 **
000378      MOVE 57 TO R-NO (R-IND)
000379      PERFORM PONTA-DE-PROVA.
000380 LE-CAD-TAB.
000381** 58 **
000382      MOVE 58 TO R-NO (R-IND)
000383      PERFORM PONTA-DE-PROVA
000384      READ CAD-TAB
000385      AT END
000386** 59 **
000387        MOVE 59 TO R-NO (R-IND)
```

```
000388        PERFORM PONTA-DE-PROVA
000389        MOVE 1 TO FLAG-FIM-CAD-TAB
000390** 60 **
000391      MOVE 60 TO R-NO (R-IND)
000392      PERFORM PONTA-DE-PROVA.
010000/*****************************************
010001* ROTINAS AUXILIARES PARA A INSTRUMENTACAO
010002*****************************************
010003 ROTINAS-INSTRUMENTACAO SECTION.
010004 PONTA-DE-PROVA.
010005      IF R-IND < 10
010006         SET R-IND UP BY 1
010007      ELSE
010008         WRITE R-PATH
010009         MOVE SPACES TO R-PATH
010010         SET R-IND TO 1.
010011 PROCESS-PUNTIL-01.
010012      MOVE 03 TO R-NO (R-IND)
010013      PERFORM PONTA-DE-PROVA
010014      MOVE 04 TO R-NO (R-IND)
010015      PERFORM PONTA-DE-PROVA
010016      PERFORM PROCESSA-TRANSACAO.
010017 INICIA--PVA-01-01.
010018      MOVE 07 TO R-NO (R-IND)
010019      PERFORM PONTA-DE-PROVA
010020      MOVE 08 TO R-NO (R-IND)
010021      PERFORM PONTA-DE-PROVA
010022      PERFORM INICIA-TAB-ITENS.
010023 TABELA--SEARCH-01.
010024      MOVE 15 TO R-NO (R-IND)
010025      PERFORM PONTA-DE-PROVA
010026      MOVE 16 TO R-NO (R-IND)
010027      PERFORM PONTA-DE-PROVA
010028      MOVE 18 TO R-NO (R-IND)
010029      PERFORM PONTA-DE-PROVA
010030      COMPUTE INS-MEIO = (INS-LIM-INF + INS-LIM-SUP) / 2
010031      IF INS-MEIO LESS IND-TAB
010032         COMPUTE INS-LIM-INF = INS-MEIO + 1
010033      ELSE
010034         COMPUTE INS-LIM-SUP = INS-MEIO - 1.
010035 COPIA-C-PUNTIL-02.
010036      MOVE 30 TO R-NO (R-IND)
010037      PERFORM PONTA-DE-PROVA
010038      MOVE 31 TO R-NO (R-IND)
010039      PERFORM PONTA-DE-PROVA
010040      PERFORM COPIA-CAD.
010041 COPIA-C-PUNTIL-03.
010042      MOVE 35 TO R-NO (R-IND)
010043      PERFORM PONTA-DE-PROVA
010044      MOVE 36 TO R-NO (R-IND)
010045      PERFORM PONTA-DE-PROVA
010046      PERFORM COPIA-CAD.
010047 COPIA-C-PUNTIL-04.
010048      MOVE 42 TO R-NO (R-IND)
010049      PERFORM PONTA-DE-PROVA
010050      MOVE 43 TO R-NO (R-IND)
010051      PERFORM PONTA-DE-PROVA
```

```
010052       PERFORM COPIA-CAD.
```

O arquivo UNIDADES.TES, apresentado abaixo, constitui uma tabela que descreve as unidades presentes no programa DEMO.CBL.

```
UNIDADES PRESENTES NO MODULO demo.cbl

Quantidade de unidades = 11

No = Numero da unidade
Rotulo Inicial = Rotulo onde se inicia a unidade
Rotulo Final = Ultimo rotulo presente na unidade
Ini = No' de entrada da unidade
Fim = No' de saida da unidade
```

No	Rotulo Inicial	Rotulo Final	Ini	Fim
00	main	main	002	005
01	INICIA-PROC	INICIA-PROC	006	009
02	PROCESSA-TRANSACAO	PROCESSA-TRANSACAO	014	033
03	LE-CAD-TAB	LE-CAD-TAB	058	060
04	INICIA-TAB-ITENS	INICIA-TAB-ITENS	010	013
05	LE-MOV	LE-MOV	055	057
06	LE-CAD	LE-CAD	052	054
07	PROCESSA-INCLUSAO	PROCESSA-INCLUSAO	034	040
08	PROCESSA-EXCLUSAO	PROCESSA-EXCLUSAO	041	047
09	IMPR-RELAT	IMPR-RELAT	049	051
10	COPIA-CAD	COPIA-CAD	048	048

Nesse ponto, as informações foram agrupadas em subdiretórios, estando, portanto, organizadas por unidade. No restante deste apêndice, iremos nos referenciar a unidade 7, nomeada pelo rótulo PROCESSA-INCLUSÃO, para ilustrar as informações geradas pela POKE-TOOL. O grafo de fluxo de controle dessa unidade, contido no arquivo UNIT07\DEMO.GFC, é mostrado a seguir:

```
07

34
  35   0
35
  36   37   0
36
  35   0
37
  38   39   0
38
  40   0
39
  40   0
40
  0
```

A extensão do grafo de controle da unidade 7 do programa DEMO.CBL é apresentada abaixo; essa informação está contida no arquivo UNIT07\GRAFODEF.TES.

```
VARIAVEIS DEFINIDAS nos Nos da UNIDADE 07
(rotulos PROCESSA-INCLUSAO 'a PROCESSA-INCLUSAO do modulo demo.cbl)

Variaveis Definidas = Vars Defs
Variaveis possivelmente definidas por Referencia = Vars Refs

NO' 34
Vars defs:  REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N M-COD-REG
 M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM REG-RELAT
 IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
 FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
 W-QTDE-ITEM W-PRECO-ITEM W-OBS
Vars refs:
NO' 35
Vars defs:
Vars refs:
NO' 36
Vars defs:
Vars refs:  REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N M-COD-REG
 M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM REG-RELAT
 IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
 FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG  W-NUM-ITEM  W-DESCR-ITEM
 W-QTDE-ITEM W-PRECO-ITEM W-OBS
NO' 37
Vars defs:
Vars refs:
NO' 38
Vars defs:  W-OBS
Vars refs:
```

```
NO' 39
Vars defs:  REG-CAD-OUT W-OBS
Vars refs:
NO' 40
Vars defs:
Vars refs:
```

O arquivo UNIT07\ARCPRIM.TES, que contém os arcos primitivos da unidade 7 do programa DEMO.CBL, é mostrado abaixo:

```
ARCOS PRIMITIVOS DO MODULO demo.cbl

arco (35,36) e' primitivo
arco (37,38) e' primitivo
arco (37,39) e' primitivo
```

Iremos aplicar o critério Todos-potenciais-usos ao teste da unidade 7 do programa DEMO.CBL; com esse propósito, apresentamos o arquivo UNIT07\PUASSOC.TES, que contém as associações requeridas pelos critérios Todos-potenciais-usos e Todos-potenciais-usos/du para a unidade 7.

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-
USOS/DU


Associacoes requeridas pelo Grafo(34)

1) <34,(37,39),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
   M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
   REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
   FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
   W-QTDE-ITEM W-PRECO-ITEM W-OBS }>
2) <34,(37,38),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
   M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
   REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
   FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
   W-QTDE-ITEM W-PRECO-ITEM W-OBS }>
3) <34,(36,35),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
   M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
   REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
   FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
   W-QTDE-ITEM W-PRECO-ITEM W-OBS }>
```

```
4) <34,(35,36),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
   M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
   REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
   FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
   W-QTDE-ITEM W-PRECO-ITEM W-OBS }>

Associacoes requeridas pelo Grafo(36)

1) <36,(37,39),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
   M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
   REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
   FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
   W-QTDE-ITEM W-PRECO-ITEM W-OBS }>
2) <36,(37,38),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
   M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
   REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
   FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
   W-QTDE-ITEM W-PRECO-ITEM W-OBS }>
3) <36,(35,36),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
   M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
   REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
   FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
   W-QTDE-ITEM W-PRECO-ITEM W-OBS }>


Associacoes requeridas pelo Grafo(38)

8) <38,( , ),{ W-OBS }>


Associacoes requeridas pelo Grafo(39)

9) <39,( , ),{ W-OBS, REG-CAD-OUT }>
```

Os descritores do critério Todos-potenciais-usos são exibidos abaixo; essa informação está contida no arquivo UNIT07\DES_PU.TES.

```
DESCRITORES PARA O CRITERIO TODOS POT-USOS

N = 34 35 36 37 38 39 40


Descritores para o Grafo(34)

Ni = 34 35 36 37 38 39 40
Nt = 35 40

1) N* 34 Nnv* 37 [ Nnv* 37 ]* 39
Nnv = 35 36 37 40
```

```
2) N* 34 Nnv* 37 [ Nnv* 37 ]* 38
Nnv = 35 36 37 40

3) N* 34 Nnv* 36 [ Nnv* 36 ]* 35
Nnv = 35 36 37 40

4) N* 34 Nnv* 35 [ Nnv* 35 ]* 36
Nnv = 35 36 37 40


Descritores para o Grafo(36)

Ni = 35 36 37 38 39 40
Nt = 36 40

5) N* 36 Nnv* 37 [ Nnv* 37 ]* 39
Nnv = 35 37 40

6) N* 36 Nnv* 37 [ Nnv* 37 ]* 38
Nnv = 35 37 40

7) N* 36 Nnv* 35 [ Nnv* 35 ]* 36
Nnv = 35 37 40


Descritores para o Grafo(38)

Ni = 38 40
Nt = 40

8) N* 38
Nnv = 38 40


Descritores para o Grafo(39)

Ni = 39 40
Nt = 40

9) N* 39
Nnv = 39 40


Numero Total de Descritores = 9
```

Após a aplicação de um caso de teste, o seguinte arquivo, nomeado PATH.TES, informa os nós executados por esse caso de teste:

```
02 06 58 07 08 10 12 58 13 07
08 10 12 58 13 07 08 10 12 58
13 07 08 10 12 58 13 07 08 10
12 58 13 07 08 10 12 58 59 60
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 08 10 11 13 07 08 10 11
13 07 09 55 57 52 54 55 57 03
04 14 15 16 18 15 16 18 15 16
18 15 16 18 15 16 18 15 16 18
15 16 17 20 22 23 34 35 36 48
52 54 35 37 39 40 28 29 49 50
51 55 57 33 03 04 14 15 16 18
15 16 18 15 16 18 15 16 18 15
16 17 20 22 23 34 35 37 38 40
28 29 49 51 55 57 33 03 04 14
15 16 18 15 16 18 15 16 18 15
16 17 20 22 23 34 35 36 48 52
53 54 35 37 39 40 28 29 49 51
55 57 33 03 04 14 15 16 18 15
16 18 15 16 18 15 16 17 20 22
23 34 35 37 39 40 28 29 49 51
55 56 57 30 33 03 05 60
```

Uma avaliação de cobertura em relação ao critério Todos-potenciais-usos para a unidade 7 é apresentada abaixo; essa informação está armazenada no arquivo UNIT07\PUOUTPUT.TES. O caso de teste utilizado nessa avaliação refere-se ao arquivo PATH.TES apresentado acima.

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS não executadas:

<36,(35,36),{ REG-CAD-TAB CI-NUM-ITEM REG-CAD-OUT M-DATA-N
  M-COD-REG M-NUM-ITEM M-DESCR-ITEM M-QTDE-ITEM M-PRECO-ITEM
  REG-RELAT IND-TAB ITEM-TAB CONT-LIN CONT-PAG FLAG-FIM-CAD-TAB
  FLAG-ITEM-VALIDO W-PAG W-DATA W-COD-REG W-NUM-ITEM W-DESCR-ITEM
  W-QTDE-ITEM W-PRECO-ITEM W-OBS }>

Cobertura Total = 88.888889

Media da Cobertura dos Grafo(i) = 91.666663
```

REFERÊNCIAS

[BER91] Bertolino, A., "An Overview of Automated Software Testing", *J. Systems Software*, 1991, No. 15, 133-138.

[BRA89] Brasil, J.P., "Compatibilidade de Programação COBOL - Programação Compatível em COBOL ANS IBM e COBOL Burroughs", in *Proc. XII Congresso Nacional de Processamento de Dados*, São Paulo, SP, 1979, pp. 223-227.

[CARD81] Cardoso, A.P., *Programação Estruturada em Cobol*, Livros Técnicos e Científicos Editora, Rio de Janeiro, 1981.

[CAR91] Carnassale, M., "GFC - Uma Ferramenta Multilinguagem para a Geração do Grafo de Programa", *Tese de Mestrado*, DCA/FEE/UNICAMP, Campinas, SP, Brasil, Fev. 1991.

[CHA91a] Chaim, M.L., Maldonado, J.C. e Jino, M., *Manual de Configuração da POKE-TOOL*, *Relatório Técnico DCA/RT*/008/91 - DCA/FEE/UNICAMP - 1991.

[CHA91b] Chaim, M.L., "POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise Fluxo de Dados", *Tese de Mestrado*, DCA/FEE/UNICAMP, Campinas, SP, Brasil, Abril 1991.

[CHU87] Chusho, T., "Test Data Selection and Quality Estimation Based on the Concept of Essencial Branches for Path Testing", *IEEE Trans. Software Eng.*, Vol. SE-13, No. 5, Maio 1987, pp. 509-517.

[CLA82] Clarke, L.A., Hassell, J. and Richardson, D.J., " A Close Look at Domain Testing", *IEEE Trans. Software Eng.*, Vol. SE-8, No. 4, July 1982.

[DEU82] Deutsch, M.S., *Software Verification and Validation*, Englewood Cliffs, Prentice-Hall, 1982.

[FRA88] Frankl, F.G. and Weyuker, E. J., "An Applicable Family of Data Flow Testing Criteria", *IEEE Trans. on Software Eng.*, Vol. 14, No. 10, Oct. 1988, pp. 1483-1498.

[GHE87] Ghezzi, C., Jazayeri, M., *Programming Languages Concepts*, John Wiley and Sons, segunda edição, New York, 1987.

[HEC77] Hecht, M.S., *Flow Analysis of Computer Programs*, North Holland, New York, 1977.

[HER76] Herman, P.M., " A Data Flow Analysis Approach to Program Testing", *The Australian Computer Journal*, Vol. 8, No. 3, Nov. 1976, pp 92-96.

[HOR92] Horgan, J.R. and Mathur, A.P., "Assessing Testing Tools in Research and Education", *IEEE Software*, May. 1992, pp. 61-69.

[HOW75] Howden, W.E., "Methodology for the Generation of Program Test Data", *IEEE Trans. on Computer*, C-24(5), 1975, pp. 554-559.

[HOW87] Howden, W.E., *Functional Program Testing and Analysis*, McGraw-Hill, USA, 1987.

[HUA79] Huang, J.C., "Detection of Data Flow Anomaly Through Program Instrumentation", *IEEE Trans. Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 226-236.

[IBM87] IBM Corporation, *VS COBOL II Application Programming Language Reference*, 1987.

[KAO84] Kao, H. and Chen, T.Y., "Data Flow Analysis for COBOL", *SIGPLAN Notices*, Vol. 19, No. 7, July 1984.

[KER88] Kernighan, B.W., Ritchie, D.R., *C a Linguagem de Programação: padrão ANSI*, Rio de Janeiro, Editora Campus, 1988.

[KOR85] Korel, B. and Laski, J., "A Tool for Data Flow Oriented Program Testing", in *Proc. Softfair II*, San Francisco, CA, Dez. 1985, pp. 34-38.

[LEI91] Leitão, P.S., *Configuração do Módulo LI da POKE-TOOL para a Linguagem C, Relatório Técnico*, DCA/FEE/UNICAMP - 1991.

[LIS78] Lister, T.R., Yourdon, E., *Learning to Program in Structured COBOL - Part 2*, Yourdon, Inc., New York, 1978.

[MaC86] McCracken, D.D., *Manual de COBOL Estruturado*, Editora Campus Ltda., Rio de Janeiro, 1986.

[MAL88] Maldonado, J.C., Chaim, M.L., Jino, M., "Seleção de Casos de Teste baseada nos Critérios Potenciais Usos", in Proc. *II Simpósio Brasileiro de Engenharia de Software*, Canela, RS, Brasil, Out. 1988, pp. 24-35.

[MAL89] Maldonado, J.C., Chaim, M.L., Jino, M., "Arquitetura de uma Ferramenta de Teste de Apoio aos Critérios Potenciais Usos", in *Proc. XXII Congresso Nacional de Informática* São Paulo, SP, Brasil, Set. 1989.

[MAL91] Maldonado, J.C., "Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software", *Tese de Doutorado*, DCA/FEE/UNICAMP, Campinas, SP, Brasil, Julho 1991.

[MIC88] Microsoft Corporation, *Microsoft COBOL Optimizing Compiler Version 3.0 - Language Reference Manual*, 1988.

[MYE79] Myers, G.J., *The Art of Software Testing*, Wiley, New York, 1979.

[NAN92] Nance,B., "The Future of Software Technology", *BYTE Special Edition OUTLOOK* 92, pp. 69-76.

[NIC75] Nicholls, J.E., *The Structure and Design of Programming Languages*, Addison Wesley Publishing Company, Inc., Philippines, 1975.

[PAQ89] Paquette, G.A., *Structured COBOL: revision edition*, Wm. C. Brown Publishers, Dubuque, Indiana, 1989.

[POD90] Podgurski, A. and Clarke, L.A., "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance", *IEEE Trans. Software Eng.*, Vol. SE-16, No. 9, Set. 1990, pp. 965-979.

[PRA84] Pratt, T.W., *Programming Languages: Design and Implementation*, Prentice-Hall, Inc., Second Edition, New Jersey, 1984.

[PRE87] Pressman, R.B., *Software Engineering: a Practitioner's Approach*, McGraw-Hill, New York, Second Edition, 1987.

[PRI90] Price, A.M., Garcia, F. e Purper, C.B., "Visualizando o Fluxo de Controle de Programas", in *Proc. IV Simp. Bras. Eng. de Software*, Águas de São Pedro, S.P., Out, 1990.

[RAP82] Rapps, S. and Weyuker, E.J., "Data Flow Analysis for Test Data Selection", in *Proc. Int. Conf. Software Eng.*, Tokio, Japão, pp. 272-278, Sep. 1982.

[RAP85] Rapps, S. and Weyuker, E.J., "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, Vol. SE-11, Abr. 1985, pp. 367-375.

[REI90] Reisman, S., "Management and Integrated Tools", *IEEE Software*, Vol. 7, No. 3, Maio 1990.

[SEB89] Sebesta, R.W., *Concepts of Programming Languages*, Benjamin Cummings Publishing Company, Inc., Redwood City, 1989.

[SET81] Setzer, V.W. and Melo, I.S.H., *A Construção de um Compilador*, terceira edição, Editora Campus, R.J., Brasil, 1981.

[SHN85] Shneiderman, B., "The Relationship between COBOL and Computer Science", *Annals of the History of Computing*, AFIPS, Reston, Vol. 7, No. 4, Oct. 1985.

[TAI80] Tai, K., "Program Testing Complexity And Testing Criteria", *IEEE Trans. Software Eng.*, Vol. SE-6, No. 6, Nov. 1980.

[TOM83] Tompkins, H.E., "In Defense of Teaching Structured COBOL as Computer Science", *SIGPLAN Notices*, Vol. 18, No. 4, April 1983.

[UNI87] UNISYS Corporation, *COBOL ANSI-74 Programming Reference Manual*, july, 1987.

[VER92] Vergílio, S.R., "Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas", *Tese de Mestrado*, DCA/FEE/UNICAMP, Campinas, SP, Brasil, Janeiro 1992.

[WHI85] White, L.J. and Sahay, P.N., "A Computer System for Generating Test Data Using the Domain Strategy", in *Proc. Softfair II*, San Francisco, CA, Dec. 1985, pp. 38-45.

[WIR76] Wirth, N., *Algorithms + Data Structures = Programs*, Englewood Cliffs, NJ, Prentice-Hall, 1976.