**UNIVERSIDADE ESTADUAL DE CAMPINAS**
**FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO**
**DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL**

# TESTE ESTRUTURAL DE PROGRAMAS DE APLICAÇÃO DE BANCO DE DADOS RELACIONAL

**Edmundo Sérgio Spoto**

Orientador: **Prof. Dr. Mario Jino**
Co-orientador: **Prof. Dr. José Carlos Maldonado**

Tese submetida à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Doutor em Engenharia Elétrica.
**Área de concentração**: Engenharia de Computação.

**Campinas – SP – Brasil**
**2000**

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP**

Sp68t

Spoto, Edmundo Sérgio
Teste estrutural de programas de aplicação de banco de dados relacional / Edmundo Sérgio Spoto.--Campinas, SP: [s.n.], 2000.

Orientadores: Mario Jino, José Carlos Maldonado.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Banco de dados relacionais. 2. SQL (Linguagem de programação de computador). 3. Software – Testes. 4. Engenharia de software. 5. Software – Desenvolvimento. I. Jino, Mario. II. Maldonado, José Carlos. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

**UNIVERSIDADE ESTADUAL DE CAMPINAS**
**FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO**
**DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL**

# TESTE ESTRUTURAL DE PROGRAMAS DE APLICAÇÃO DE BANCO DE DADOS RELACIONAL

**Edmundo Sérgio Spoto**

Tese de Doutorado defendida e aprovada, em 20 de dezembro de 2000, pela Banca Examinadora constituída pelos Professores:

**Prof. Dr. Mario Jino, Presidente**
**DCA/FEEC/UNICAMP**

**Prof. Dr. Ivan Luis Marques Ricarte**
**DCA/FEEC/UNICAMP**

**Prof. Dr. Léo Pini Magalhães**
**DCA/FEEC/UNICAMP**

**Prof. Dr. Ricardo Ribeiro Gudwin**
**FEEC/UNICAMP**

**Prof. Dr. Antonio Francisco Prado**
**UFSCAR**

**Profa. Dra. Silvia Regina Virgilio**
**UFPR**

**Dr. Plínio Souza Vilela**
**TELCORDIA/EUA**

**Campinas – SP – Brasil**
**2000**

# Resumo

Uma abordagem é proposta para o teste estrutural de programas de Aplicação de Banco de Dados Relacional (ABDR) com SQL embutida. Uma aplicação é vista como um conjunto de programas e cada programa como um conjunto de unidades. Devido à natureza da ABDR, dois modelos de fluxo de dados são propostos: *intra-modular* e *inter-modular*. O modelo de fluxo de dados *intra-modular* aplica-se ao teste de unidade e ao teste de integração das unidades de um programa. O *inter-modular* aplica-se à integração dos programas que compõem uma aplicação. O teste de unidade pode ser apoiado por qualquer critério de teste estrutural de unidade de programas procedimentais. Duas abordagens são propostas para o teste de integração *intra-modular*: teste baseado no grafo de chamadas e teste baseado em dependência de dados. Critérios baseados nos modelos *intra-modular* e *inter-modular* são apresentados e discutidos com exemplos de sua aplicação; são também apresentados e discutidos resultados de experimentos. A análise da complexidade dos critérios mostra que ela é de ordem exponencial, para o pior caso. Contudo, os resultados de sua aplicação, em programas reais, indicam que eles demandam um número pequeno de casos de teste.



# Abstract

An approach is proposed for structural testing of programs of Relational Database Applications (RDA) with embedded **SQL**. An application is regarded as a set of programs and each program as a set of units. Due to the nature of RDA two data-flow models are proposed: *intra-modular* and *inter-modular*. The *intra-modular* data flow model addresses the unit testing and the integration testing of each program. The *inter-modular* data flow model addresses the integration testing of RDA's programs. Unit testing can be supported by any structural unit testing criterion of procedural programs. Two approaches are proposed for the *intra-modular* integration testing: testing based on the call-graph and testing based on data dependence. Criteria based on the *intra-modular* and *inter-modular* data flow models are presented and discussed with examples of their application; results from experiments are also presented and discussed. Analyses of the complexity of those

criteria show them to be of exponential order, in the worst case. However, results from their application indicate that they demand a small number of test cases.

# Agradecimentos

Agradeço aos meus orientadores Prof. Dr. Mario Jino e Prof. Dr. José Carlos Maldonado, pela orientação segura e a confiança que me passaram ao longo da elaboração desta tese.

Aos amigos e companheiros do grupo de teste do LCA: Ana, Adalberto, Chaim, Claudia, Cristina, Daniela, Inês, Ivan, Letícia, Nelson, Paulo, Plínio e Silvia, pelas sugestões e companheirismo ao longo desses anos, a todos vocês meu sincero agradecimento. Agradeço também a Mitie e Priscila pela amizade.

Aos Professores do DCA: Eleri, Ivan Ricarte, Leo Pini, Mario Jino, Maurício e Ting por todo aprendizado ao longo dos anos e pela amizade. Aos membros da banca: Ivan Ricarte, Leo Pini, Plínio Vilela, Prado, Ricardo Gudwin e Silvia Vergílio agradeço as contribuições.

Aos demais amigos do LCA: Adela, Galo, Gonzaga, José, Luis, Lizet, Márcio Leandro, Miriam, Rodrigo, Viviane e a todos os demais que passaram por lá e deixaram lembranças pela amizade; valeu a força de todos.

A Família Tambascia que foi minha segunda família: Sr. Cláudio, Dna. Vanda, Claudia, Luciana, Rafael, Sr. Bilu e Dna. Guiomar pelo carinho que sempre me acolheram.

A todos os demais amigos e colegas de Campinas que de maneira direta ou indireta contribuíram com a realização deste trabalho.

A minha esposa Rosana pelo amor, carinho e dedicação que tem me dado.

Aos meus pais Yeda e Angelo e todos irmãos. A todos tios, tias, primos e primas meu muito obrigado pela força e ajuda que vocês sempre deram.

A todos os professores do DIN, de modo especial aos Professores: Delamaro e Sandra pela contribuição; e aos demais Professores Airton, Carniel, Elisa, Flávio, Itana, Jose Roberto, Maurício, Olguin, Osvaldo, Sica, Tarso, Xavier pela força no dia a dia. Aos funcionários e alunos pela amizade e compreensão.

Ao Professor Doherty Andrade do DMA pela ajuda.

Aos Órgãos de apoio a pesquisa: Capes e CNPq, bem como as Universidades UEM e UNICAMP.

# SUMÁRIO


RESUMO. . . . . *v*

ABSTRACT. . . . . *v*

AGRADECIMENTOS . . . . . *vii*

SUMÁRIO. . . . . *ix*

LISTA DE FIGURAS . . . . . *xiii*

LISTA DE TABELAS. . . . . *xvii*

1 Introdução. . . . . 001

1.1 Teste de Software . . . . . 001

1.2 Motivação . . . . . 003

1.3 Objetivos da Tese . . . . . 004

1.4 Organização da Tese . . . . . 005

2 Conceitos Básicos e Terminologia . . . . . 007

2.1 Teste Estrutural Baseado em Fluxo de Dados . . . . . 007

2.1.1 Teste de Unidade. . . . . 011

2.1.2 Teste de Integração . . . . . 012

2.1.3 Trabalhos Relacionados ao Teste Estrutural . . . . . 013

2.2 Banco de Dados Relacional . . . . . 021

2.2.1 Modelo de Dados Relacional . . . . . 021

2.2.2 Aplicação de Banco de Dados Relacional (ABDR) . . . . . 024

2.2.2.1 Linguagem de Banco de Dados Relacional – SQL . . . . . 026

2.2.2.2 Comandos da Linguagem SQL embutida . . . . . 028

2.3 Trabalhos Relacionados. . . . . 032

2.4 Considerações Finais. . . . . 034

3 Teste Estrutural de Aplicação de Banco de Dados Relacional:Conceitos e Terminologia . . . . . 037

3.1 Teste Estrutural de ABDR: Conceitos Básicos e Terminologia . . . . . 037

3.1.1 Teste de Unidade . . . . . 041

3.1.2 Teste de Integração. . . . . 050

3.1.2.1 Teste de integração baseado no grafo de chamadas. . . . . 051
3.1.2.2 Teste de integração baseado na dependência de dados . . . . 053
3.2 Considerações Finais . . . . . 057
4 Critérios de Testes: Definição e Análise de Propriedades. . . . . 061
4.1 Considerações Iniciais . . . . . 061
4.2 Tipos de Fluxos de Dados. . . . . 061
4.2.1 Fluxo Intra-Modular . . . . . 062
4.2.2 Fluxo Inter-Modular . . . . . 064
4.3 Os Critérios Intra-Modulares. . . . . 065
4.3.1 Critérios aplicados ao teste de unidade . . . . . 066
4.3.2 Critérios aplicados ao teste de integração intra-modular. . . . . 068
4.3.2.1 Critérios de Teste de Integração baseados no grafo de chamadas . . . . . 069
4.3.2.2 Critérios de teste de integração baseados na dependência de dados. . . . . 072
4.4 Critérios de integração inter-modular. . . . . 075
4.5 Análise de Propriedades. . . . . 076
4.5.1 Análise de Inclusão entre os critérios. . . . . 076
4.5.2 Análise de Complexidade. . . . . 084
4.6 Considerações Finais . . . . . 088
5 Os Modelos de Implementação dos Critérios de Teste. . . . . 091
5.1 Considerações Iniciais. . . . . 091
5.2 O Modelo de Fluxo de Controle. . . . . 092
5.3 Os Modelos de Instrumentação . . . . . 099
5.3.1 Instrumentando o programa original . . . . . 101
5.3.2 Identificando as Tuplas . . . . . 103
5.4 Modelo de Fluxo de Dados . . . . . 107
5.4.1 Modelo de Fluxo de Dados Intra-Modular. . . . . 108
5.4.2 Modelo de Fluxo de Dados Inter-Modular. . . . . 112
5.5 Modelo de Descrição dos Elementos Requeridos . . . . . 113
5.6 Considerações Finais . . . . . 115

6 Exemplos de Utilização dos Critérios de Teste. . . . . 117
6.1 Considerações Iniciais. . . . . 117
6.2 Descrição das Tabelas, Dependências e Fluxos de Dados . . . . . 118
6.3 Resultados dos Exemplos de Utilização . . . . . 121
6.3.1 Teste Intra-Modular . . . . . 121
6.3.1.1 Teste de Unidades. . . . . 121
6.3.1.2 Teste de Integração baseado na dependência de dados. . . . . 126
6.3.2 Teste de Integração Inter-Modular. . . . . 132
6.4 Comparação da utilização dos critérios. . . . . 136
6.5 Considerações Finais. . . . . 139
7 Conclusões e Trabalhos Futuros. . . . . 141
7.1 Conclusões. . . . . 141
7.2 Contribuições da Tese. . . . . 142
7.3 Trabalhos Futuros . . . . . 143
Referências Bibliográficas. . . . . 145
Apêndices. . . . . 153
A: A Ferramenta POKE-TOOL. . . . . 153
B: Comandos de SQL . . . . . 159
C: Um Exemplo Completo das etapas de integração: *intra-modular* e *inter-modular* . . . . . 175
Índice Remissivo. . . . . 199

# Lista de Figuras


2.1 Ambiente para o teste de unidade . . . . . 011

2.2 Representação gráfica do fluxo de controle dos comandos executáveis da SQL gerados pelas ações do comando WHENEVER <condição> <ação> . . . . . 030

3.1 Uma Aplicação de Banco de Dados Relacional. . . . . 038

3.2 Grafo de Programa da Unidade de Programa Selemp do Módulo de Programa $Mod_3$. . . . . 040

3.3 Exemplo de um grafo de programa com SQL embutida. . . . . 043

3.4 Função com os principais comandos de manipulação da SQL para uma *tabela t* . . . . . 046

3.5 Exemplo de um grafo de chamada entre duas UPs (C e R) . . . . . 051

3.6a Grafo de Dependência interna de Dados c.r.a. DEPARTMENT $GD_{PA}(Insdep_{MOD3} \times Seldep_{MOD3})_{DEPARTMENT}$ . . . . . 056

3.6b Grafo de Dependência externa de Dados c.r.a. CUSTOMER entre as UPs Inscus($Mod_2$) e Relcli($Mod_4$). $GD_{PA}(Inscus_{MOD2} \times Relcli_{MOD4})_{CUSTOMER}$ . . . . . 057

3.7 Grafo de Dependência de Dados interna entre as unidades do Módulo de Programa $Mod_3$ ilustrada com os grafos de programas das UPs . . . . . 058

4.1 Exemplo do Fluxo de Dados Intra-Modular completo. . . . . 063

4.2 Fluxo de Dados *inter-modular* de uma ABDR composta por 4 Módulos de Programas e 8 tabelas que constituem a base de dados. . . . . 065

4.3 Exemplo de um Procedimento que contém todas operações da SQL. . . . . 067

4.4 Exemplo 1 para análise de inclusão . . . . . 080

4.5 Exemplo 2 para análise de inclusão . . . . . 081

4.6 Árvore de inclusão parcial dos critérios de teste de unidade de ABDR. . . . . 082

4.7 Árvore de inclusão dos critérios de teste de unidade de ABDR para caminhos executáveis . . . . . 082

4.8 Exemplo para Análise de Inclusão dos Critérios (*Todos os t-usos-intra* ou inter)× (*Todos os dtu-caminhos-intra* ou *inter*). . . . . 083

4.9 Árvores de Inclusão dos Critérios de integração. O Critério *todos os dtu-caminhos (intra* ou *inter*) inclui parcialmente o critério *todos os t-usos (intra ou inter)*. . . . . 084
4.10 Grafo de fluxo de controle que maximiza o número de combinações do critério *todos os t-usos* . . . . . 085
4.11 Grafo de fluxo de controle com estrutura que maximiza o critério todos os t-usos, onde **a** é o número de nós em cada estrutura e **k** é o número de comandos *if-then-else*. . . . . 087
5.1 Modelo de fluxo de controle e instrumentação associado aos comandos da SQL comando declarativo de controle de erros WHENEVER. . . . . 093
5.2 O modelo de fluxo de controle para os comandos executáveis da **SQL** gerado pela ação do comando declarativo WHENEVER. Na descrição é apresentado o Modelo de instrumentação referente a cada ação nos comandos executáveis da **SQL**. . . . . 094
5.3 Sintaxe parcial e LI para o comando INSERT . . . . . 096
5.4 Sintaxe parcial e LI para o comando DELETE. . . . . 097
5.5 Sintaxe parcial e LI para o comando UPDATE . . . . . 098
5.6 Sintaxe parcial e LI para o comando SELECT. . . . . 099
5.7 Instrumentação dos comandos de manipulação da SQL para a ação DO <*função*> . . . . . 102
5.8 Arquivo de auxílio à instrumentação das tuplas para as tabelas: EMPLOYEE, DEPARTMENT, CUSTOMER e SALES_ORDER . . . . . 104
5.9 Arquivo *tab_ch_pr.tes*: chaves primárias das tabelas usadas no programa *Mod$_3$.pc*. . . . . 105
5.10a Programa original para teste . . . . . 105
5.10b Programa instrumentado para o teste de Unidade. . . . . 106
5.10c Programa instrumentado para os critérios de integração intra-modulares e inter-modulares. . . . . 107
5.11 Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Manipulação da Base de Dados (SQL). . . . . 109

5.12 Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Seleção da Base de Dados (SQL) . . . . . 110
5.13 Arquivo *tabdeftuso*.tes para a tabela EMPLOYEE no módulo $Mod_3.pc$ . . . . . 111
5.14 Arquivo tabidtu.tes para a tabela EMPLOYEE nos módulos $Mod_3.pc$ e $Mod_6.pc$ 112
6.1a Diagrama de Entidades e Relacionamentos . . . . . 118
6.1b Tabelas da base de dados e fluxo de dependência . . . . . 119
6.2 *Fluxo de dados inter-modular* da ABDR composta por 4 Módulos de Programa e 8 tabelas que constituem a base de dados . . . . . 120
6.3 Visualização do teste de unidade *Insemp*. . . . . 125
6.4 Organização dos arquivos utilizados para o teste de *integração intra-modular* . 128
6.5 Arquivos *Pathint.tes* e *Keyint.tes* informações sobre a execução do teste de integração *intra-modular*. . . . . 131
6.6 Teste de integração *inter-modular*: Unidades *Insemp* do Módulo $Mod_3$ e Unidade *RelGer* do Módulo $Mod_6$. . . . . 134
6.7 Arquivos *Pathint.tes* - caminhos executados e *Keyint.tes* - seqüência de tuplas . 135
A1 Projeto da Ferramenta POKE-TOOL. . . . . 155
A2 Apresenta uma proposta de modificação do projeto da ferramenta POKETOOL para atender programas de ABDR na etapa de teste de unidade . . . . . 158
B1 Comandos da **SQL** embutida, agrupados por tipos. Os comandos com * não existem no modo interativo . . . . . 160
C1 Quadro geral do programa $Mod_3.pc$ utilizado no exemplo . . . . . 173

# Lista de Tabelas


4.1 Mapa de definição e uso das *variáveis tabela* mostradas nos Módulos de Programas da Figura 4.2. . . . . 065

6.1 Ocorrências de definição persistente e uso das variáveis tabela nos Módulos de Programas (D: definição, U: uso). . . . . 120

6.2 Quadro geral dos Módulos de Programas executados no teste de unidade . . . . . 137

6.3 Testes para o critério de integração *todos os t-usos-ciclo1-intra* para o Módulo $Mod_3$ . . . . . 137

6.4 Elementos requeridos pelo critério *todos os t-usos-inter* para os módulos $Mod_2$, $Mod_3$ e $Mod_6$. . . . . 138

6.6 Associações entre os comandos da SQL c.r.a. ***t*** . . . . . 140

# 1 Introdução

Neste capítulo são apresentados brevemente o contexto, a motivação e os objetivos deste trabalho – Teste Estrutural de Software de Aplicação de Banco de Dados Relacional. A organização da tese é apresentada na última seção deste capítulo.

## 1.1 Teste de Software

Nas últimas duas décadas os profissionais da área de Engenharia de Software têm demonstrado uma grande preocupação em incorporar qualidade no processo de desenvolvimento de seus produtos de software. Pressman [PRE97] apresenta várias formas de tratar essa qualidade ao longo do processo de desenvolvimento do produto, relacionando requisitos tanto funcionais como de desempenho, documentações padronizadas e características que permitam um planejamento do software desenvolvido profissionalmente como um todo. Segundo o mesmo autor, o processo de desenvolvimento de software se divide em três fases genéricas: definição, desenvolvimento e manutenção. A fase de definição é caracterizada pela identificação dos requisitos básicos do sistema e do software. A fase de desenvolvimento tem por objetivo definir como os requisitos do sistema serão satisfeitos; nela ocorrem três etapas distintas: *projeto*, *codificação* e *teste de software*. Este trabalho concentra-se na etapa de teste de software. A fase de *manutenção* concentra-se nas modificações feitas sobre o software devido às atividades como: *correção*, *melhoria* e *adaptação*.

Os termos: "*defeito*", "*erro*" e "*falha*" correspondem aos termos em inglês: "*fault*", "*error*" e "*failure*", respectivamente. Seus significados são: i) ***Defeito***: é uma deficiência mecânica ou algorítmica que, se ativada, pode levar a uma falha; diz-se que o defeito é inerente ao programa e é geralmente inserido por um "*engano*"("*mistake"*) do programador ou do analista; ii) ***Erro***: é um item de informação ou estado inconsistente do programa; considerado como um item dinâmico que geralmente surge após a ativação de um defeito; iii) ***Falha***: é o evento notável no qual o programa viola suas especificações; uma condição indispensável para a ocorrência de uma falha é que pelo menos um defeito seja ativado (neste caso dizemos que a execução exercitou um ou mais defeitos) [VIL98].

Teste de Software é uma das atividades de garantia de qualidade e, segundo Chusho [CHU87], é a chave para melhorar sua produtividade e sua confiabilidade. Fazer o teste de modo exaustivo é em geral impraticável. Por exemplo, um programa cujas entradas sejam duas variáveis inteiras X e Y em um computador com registradores de 32 bits, teria $2^{64}$ pares de entradas (X, Y), tornando assim impraticável o teste com cada par [HUA75]. O teste é o processo de executar um programa ou sistema com a finalidade de detectar defeitos. O teste pode também ser usado para avaliar o quanto o sistema atende às necessidades especificadas [MYE79].

As atividades de teste consomem em média 50% do tempo e do custo de desenvolvimento de um produto de software [GOD75, MYE79, PRE97]. Esse gasto é em parte resultante da escassez de ferramentas de auxílio ao teste que permitam uma forma planejada e segura para sua execução e para a avaliação de seus resultados.

De uma forma geral o teste de software envolve as seguintes atividades: planejamento, projeto de *casos de teste*[1], execução e avaliação dos resultados dos testes. Os métodos utilizados para testar software são baseados essencialmente nas técnicas funcional, estrutural e baseada em defeitos. A técnica funcional de teste visa a estabelecer requisitos de teste derivados da especificação funcional do software (*teste caixa preta*) enquanto a técnica estrutural de teste apóia-se em informações derivadas diretamente da implementação (teste caixa branca) [WHI87]. Cumpre observar que este trabalho concentra-se em teste de software baseado na técnica estrutural.

Hetzel [HET87] observa que a técnica funcional de teste enfoca apenas as respostas obtidas para cada dado de teste. Esta técnica ignora os detalhes de implementação. A técnica baseada em defeitos (e.g., semeadura de erros e análise de mutantes) [BUD81, DeM78, NTA84] utiliza informações sobre os defeitos típicos de desenvolvimento para derivar os requisitos de teste.

Usualmente, caracterizam-se três fases (ou níveis) de teste em uma estratégia de teste: *teste de unidade*, *teste de integra*ção e *teste de sistema*. O teste de *unidade* concentra

[1] Um *caso de teste* é o conjunto formado por: i) um dado de entrada para um programa e ii) a saída esperada para esse dado de entrada. Em geral, procura-se projetar casos de teste que tenham a maior probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço.

esforços na menor parte executável do programa, conhecida como *módulo*[2], visa a identificação de defeitos de lógica e de implementação. O teste de *integração* é uma técnica sistemática utilizada para integrar as unidades que compõem o programa, visa a detectar defeitos de integração entre elas. O teste de *sistema* é conduzido após a integração do sistema, visa a identificar defeitos de função e/ou características de desempenho. A grande maioria dos critérios estruturais de teste é aplicada na fase do teste de *unidade* [MAL91, VIL98].

Existem vários tipos de aplicação que carecem de técnicas apropriadas para o teste de software. Uma delas é o desenvolvimento de programas de Aplicações de Banco de Dados Relacional (ABDR) que utilizam a linguagem de consulta **SQL** embutida em linguagens procedimentais tais como: C, Pascal, PL/I, etc. Em levantamentos bibliográficos realizados observou-se que o mais comum é utilizar a técnica de teste funcional, na qual ocorrem várias etapas de testes de usabilidade e das principais funcionalidades do programa. Existem casos que aplicam um conjunto de operações de comandos **SQL,** explorando as principais consultas existentes no programa.

## 1.2 Motivação

Nos últimos anos as grandes empresas têm investido maciçamente na elaboração de novas técnicas de desenvolvimento de software em busca da qualidade de seus produtos de software. Contudo, existem muitas áreas de desenvolvimento de software em que as iniciativas e o esforço têm sido muito incipientes. Em particular, muito pouco existe no que se refere ao suporte de teste de programas de ABDR, mesmo no âmbito internacional.

Em geral, a aplicação de Banco de Dados Relacional pode ser composta por vários *Programas* codificados nas linguagens C, Pascal (Delphi), Fortran, Ada, PL/I, Cobol, etc. A característica principal é a manipulação de dados através de comandos de linguagem de consulta **SQL** à bases de dados controladas por um Sistema Gerenciador de Banco de Dados Relacional (SGBDR). Existem no mercado vários ambientes Gerenciadores de Banco de Dados que são usados dos pequenos aos grandes centros de desenvolvimento de sistemas. Apesar da existência de diferentes dialetos **SQL**, um esforço de padronização

[2] Muito usado na literatura para representar a unidade em teste, podendo ser qualquer procedimento ou função do programa em teste.

dessa linguagem resultou no padrão **SQL** [ISO92]. Considerando essa padronização, a construção de uma ferramenta de apoio ao teste de programas de ABDR deve inicialmente atender os comandos padrões de **SQL**.

Defeitos estão presentes na maioria dos produtos de software produzidos e liberados, apesar dos avanços significativos obtidos na área de Engenharia de Software; o teste de software é uma das técnicas utilizada para revelar defeitos, pois cobre classes distintas de defeitos [MAL91]. Técnicas específicas para tratar de ABDR respondem a uma necessidade emergente. A partir dessas técnicas será possível criar suporte automatizado para dar apoio à obtenção de qualidade durante o desenvolvimento de programas de ABDR.

Neste caso, estudar os principais problemas existentes em programas de ABDR com SQL embutida como, por exemplo, o fluxo de informações existentes entre a base dados ocasionado pelos diferentes programas que compõem uma ABDR e como testar todo o sistema tendo em vista sua complexidade devido a grande quantidade de informações que são armazenadas e consultadas. Porém, defeitos de implementação, defeitos de lógica de programação e outros provenientes de falha humana durante a fase de desenvolvimento do programa só poderão ser tratados com as técnicas de teste estrutural para programas de ABDR. A idéia, portanto, é adaptar as principais técnicas de teste estrutural de software para aplicá-las em programas de ABDR com **SQL** embutida em linguagens hospedeiras, visando a atender essas necessidades.

## 1.3 Objetivos da Tese

Este trabalho tem como objetivo principal estudar métodos de teste estrutural e adaptá-los para programas de ABDR. Esses estudos englobam o teste de unidade e o teste de integração de modo a atender às condições específicas de uma aplicação de banco de dados relacional envolvendo comandos mais comuns de **SQL** usados para manipular bases de dados.

A partir desses estudos visa-se a estabelecer uma abordagem para o teste estrutural de programas de ABDR. Essa abordagem deve incorporar métodos de teste baseados em fluxo de dados e métodos de teste estrutural que exercitem os tipos específicos de variáveis de programas de ABDR (variáveis utilizadas em programas de aplicação). Objetiva-se

também propor a implementação desses métodos de teste em uma ferramenta automatizada para apoiar a sua aplicação no teste de programas de ABDR.

Especificamente, critérios de teste, baseados em análise de fluxo de dados, são a base dos métodos de teste estrutural. Estudar novas estratégias de teste estrutural para programas de ABDR é o principal alvo deste trabalho e análise. Através do estudo e análises pretende-se desenvolver critérios que tornam efetivo o teste estrutural de programas de ABDR visando a explorar os diferentes tipos de fluxo de dados que ocorrem entre as tabelas da base de dados a partir dos diferentes programas que compõem a aplicação. Nestes estudos pretende-se apresentar uma proposta completa que atenda todas as etapas de teste de um programa até atingir toda a ABDR.

A proposta de teste estrutural para programas de ABDR apresentada nesta tese é uma importante contribuição para a área de desenvolvimento de software para Sistemas que manipulam grandes volumes de dados armazenados em arquivos específicos ou até mesmo em Sistemas Gerenciadores de Banco de Dados Relacional. Abrindo assim novas pesquisas e condições de auxiliar na obtenção da qualidade de software na etapa de desenvolvimento.

## 1.4 Organização da tese

Neste capítulo são apresentados o contexto, a motivação e os objetivos principais deste trabalho. No Capítulo 2 são apresentados terminologia e os conceitos básicos utilizados nos Critérios de Teste Estrutural baseados em análise de Fluxo de Controle e análise de Fluxo Dados. Em seguida são apresentados os conceitos básicos de Banco de Dados Relacional.

No Capítulo 3 são apresentados os conceitos básicos e a terminologia adotada para o teste de programas de ABDR, segundo duas fases de teste: teste de unidade e teste de integração. O teste de integração engloba duas abordagens distintas: integração baseada no grafo de chamada (idêntica à abordagem para programas convencionais) e integração baseada em dependência de dados das *variáveis tabela* da Base de Dados. Também são apresentados os tipos de fluxo de dados *intra-modular* e *inter-modular* de programas de ABDR.

No Capítulo 4 são apresentados os critérios propostos e a análise de suas propriedades; são descritos os tipos de fluxo de dados dos programas de aplicação: *intra-*

*modular* e *inter-modular*; são descritos os critérios de teste *intra-modular* e os critérios de integração *inter-modular* e é apresentada a análise de propriedades dos critérios.

No Capítulo 5 são abordados os vários aspectos relacionados ao projeto da ferramenta de suporte à aplicação dos critérios de teste em ABDR; são descritos o modelo de fluxo de controle, os modelos de instrumentação e os modelos de fluxo de dados; é discutida a descrição dos elementos requeridos.

O Capítulo 6 ilustra a utilização dos critérios propostos para o teste de programas de ABDR. Inicialmente são apresentados a descrição das tabelas, as dependências e o fluxo de dados para o exemplo de aplicação. Em seguida são apresentados resultados obtidos nos exemplos de utilização dos critérios. A partir do exemplo são apresentadas algumas comparações referentes à utilização dos critérios e são feitas algumas observações quanto aos resultados do teste dos programas do exemplo de aplicação.

No Capítulo 7 são apresentadas as conclusões desta tese; são discutidas as principais contribuições e as sugestões de trabalhos futuros.

Três apêndices completam este trabalho: Apêndice A – contém uma breve descrição da ferramenta POKE-TOOL; Apêndice B – contém uma descrição de comandos da linguagem **SQL**; e Apêndice C – contém exemplos de ilustração da aplicação dos critérios de teste propostos.

# 2 Conceitos Básicos e Terminologia

Na primeira parte deste capítulo são apresentados os conceitos básicos e terminologia usados para definir os critérios de teste estrutural baseados em análise de Fluxo de Dados. Na segunda parte são apresentados conceitos básicos de Banco de Dados Relacional.

## 2.1 Teste Estrutural Baseado em Fluxo de Dados

A técnica estrutural de teste de programas é baseada no conhecimento da estrutura interna da implementação do software e visa a caracterizar um conjunto de componentes elementares do software que devem ser exercitados pelo teste (também conhecido como teste de caixa branca). Diz-se que um programa *P* é correto com respeito a uma especificação *S* se, para qualquer item de dado *d* pertencente ao domínio de entrada *D* do programa *P*, o comportamento do programa estiver de acordo com o especificado em *S* [PRE97]. Para o teste de um programa pressupõe-se que existe um *oráculo* (mecanismo automático ou manual) que possa determinar, para qualquer dado de entrada, se a saída do programa é a saída esperada conforme a especificação. Um *caso de teste* é composto por uma entrada de dados e a correspondente especificação da saída esperada.

Apesar das desvantagens decorrentes das limitações inerentes às atividades de teste estrutural de programas levantadas por vários autores [FRA88, HOW87, LAS83, MAL91, NTA88, RAP85], a técnica estrutural de teste é vista como complementar à técnica funcional, uma vez que cobre classes distintas de erros [HOW87, LAS83, MYE79, MAL91, PRE97]. Ainda, as informações obtidas com a aplicação da técnica estrutural de teste têm sido consideradas relevantes para as atividades de manutenção e depuração de software [MAL91].

No teste estrutural, um programa *P* pode ser decomposto em um conjunto de blocos disjuntos de comandos. Um bloco de comandos é a seqüência de um ou mais comandos com a propriedade de que, sempre que o primeiro comando do bloco é executado, todos os demais comandos do bloco também são executados e não existem desvios para o meio do bloco [HUA75].

Um programa *P* é representado por um grafo de fluxo de controle (ou grafo de programa) *G=(N, A, e, s)* onde *N* é o conjunto de nós, *A* é o conjunto de arcos (ou arestas), *e* é o único nó de entrada e *s* é o único nó de saída. Todo grafo de programa é um grafo dirigido conexo. Um nó $n \in N$ representa uma instrução simples (comando) ou uma seqüência de instruções executadas como um bloco de comandos. Cada arco $a \in A$ é um par ordenado $(n, m)$ de nós de *N* que representa uma possível transferência de controle do nó *n* para o nó *m* [HER76, LAS83, WHI87].

Um *caminho* do programa é representado por uma seqüência finita de nós $(n_1, n_2, .., n_k)$, $k \geq 2$, tal que, para todo nó $n_i$, $1 \leq i \leq k-1$, existe um arco $(n_i, n_{i+1})$ para $i = 1, 2, .., k-1$. Um *caminho completo* é um caminho cujo nó inicial é o nó de entrada e cujo nó final é o nó de saída. Um *caminho simples* é um caminho cujos nós, exceto possivelmente o primeiro e o último, são distintos. Um caminho é *livre de laço* se todos os nós pertencentes a ele são distintos [RAP85].

Vários critérios de teste estrutural de programas foram propostos ao longo dos últimos 20 anos; os primeiros critérios propostos são baseados unicamente no fluxo de controle de programas, sendo os mais conhecidos: o critério *todos-nós*; o critério *todos-arcos*; e o critério *todos-caminhos*. O critério *todos-nós* requer que cada comando do programa seja executado pelo menos uma vez; o critério *todos-arcos*, um dos critérios mais utilizados, requer que toda transferência de controle seja exercitada pelo menos uma vez. O critério *todos-caminhos* requer que todos os caminhos do programa sejam exercitados [HOW87, LAS83, NTA84]. Enquanto os critérios *todos-nós* e *todos-arcos* são considerados pouco exigentes, o critério *todos-caminhos* é em geral impraticável na presença de laços no programa, criando assim uma distância (lacuna) muito grande entre os critérios *todos-arcos* e *todos-caminhos* [RAP82, NTA84]; novos critérios foram introduzidos com o intuito de estabelecer uma hierarquia de critérios capaz de preencher a lacuna existente entre os critérios *todos-arcos* e *todos-caminhos* [LAS83, RAP82, NTA84, RAP85, MAL91].

Surgiram então os critérios de teste estrutural baseados em análise de fluxo de dados do programa. A análise estática do programa fornece informações sobre as ações

executadas a respeito das variáveis do programa e efeitos de tais ações nos vários pontos do programa [HEC77]. Em geral uma variável pode sofrer as seguintes ações no programa: (*d*) definição, (*i*) indefinição; ou (*u*) uso. Uma *definição* de variável ocorre quando um valor é armazenado em uma posição de memória. Uma ocorrência de variável é uma *definição* se ela está: *i)* no lado esquerdo de um comando de atribuição; *ii)* em um comando de entrada; ou *iii)* em chamadas de procedimentos como parâmetro de saída [MAL91]. A ocorrência de uma variável como *uso* se dá quando a referência a esta variável, em um comando executável, não a estiver definindo, ou seja, há uma *recuperação* de um valor em uma posição de memória associada a esta variável. Dois tipos de usos são distinguidos – *c-uso* e *p-uso*. O *c-uso* (uso computacional) afeta diretamente uma *computação* que está sendo realizada ou permite observar o valor de uma variável que tenha sido definida anteriormente - nestes casos o *uso* está associado a um nó do grafo do programa; o *p-uso* (uso predicativo) afeta diretamente o *fluxo de controle* do programa - este *uso* está associado a um arco do grafo.

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

Um caminho $(i, n_1, .., n_m, j)$, $m \geq 0$ com uma definição da variável $\boldsymbol{x}$ no nó $i$ e que não contenha nenhuma redefinição de $\boldsymbol{x}$ nos nós $n_1, .., n_m$ é chamado de *caminho livre de definição* com respeito a (c.r.a.) $\boldsymbol{x}$ do nó $i$ ao nó $j$ e do nó $i$ ao arco $(n_m, j)$. Neste caso, pode existir uma redefinição de $\boldsymbol{x}$ no nó $j$.

Um nó $i$ possui uma *definição global* de uma variável $\boldsymbol{x}$ se ocorre uma definição de $\boldsymbol{x}$ no nó $i$ e existe um *caminho livre de definição* de $i$ para algum nó ou para algum arco que contém um c-uso ou um p-uso, respectivamente, da variável $\boldsymbol{x}$. Um c-uso da variável $\boldsymbol{x}$ em um nó $j$ é um *c-uso global* se não existir uma definição de $\boldsymbol{x}$ no mesmo nó $j$ precedendo este c-uso; caso contrário, é um *c-uso local*.

Vários conceitos e definições foram introduzidos por Rapps e Weyuker [RAP82, RAP85] para estabelecer a Família de Critérios de Fluxo de Dados (FCFD); um deles é o grafo def-uso ('*def-use graph*') que consiste em uma extensão do grafo de programa, incorporando-se informações semânticas do programa a este grafo. O grafo def-uso é obtido a partir do grafo de programa associando-se a cada nó $i$ os conjuntos *c-uso(i)* =

{variáveis com c-uso global no nó *i*} e *def(i)* = {variáveis com definições globais no nó *i* }, e a cada arco *(i, j)* o conjunto p-uso*(i,j)* = {variáveis com p-usos no arco *(i, j)*}. Dois conjuntos foram definidos: *dcu(x,i)* = {nós *j* tal que $\boldsymbol{x} \in$ *c-uso (j)* e existe um caminho livre de definição c.r.a. $\boldsymbol{x}$ do nó *i* para o nó *j*} e *dpu(x,i)* = {arcos (j, k) tal que $\boldsymbol{x} \in$ *p-uso (j, k)* e existe um caminho livre de definição c.r.a. $\boldsymbol{x}$ do nó *i* para o arco *(j, k)*}. Adicionalmente, foram definidos os elementos requeridos de um programa: *du-caminho*, *associação-c-uso*, *associação-p-uso* e *associação*.

Um caminho ($n_1$, $n_2$, .., $n_k$) é um *du-caminho* c.r.a. variável $\boldsymbol{x}$ se $n_1$ tiver uma definição global de $\boldsymbol{x}$ e: (1) ou $n_k$ tem um c-uso de $\boldsymbol{x}$ e ($n_1$, $n_2$, .., $n_j$, $n_k$) é um caminho simples livre de definição c.r.a. $\boldsymbol{x}$; ou (2) ($n_j$, $n_k$) tem um p-uso de $\boldsymbol{x}$ e ($n_1$, $n_2$, .., $n_j$, $n_k$) é um caminho livre de definição c.r.a. $\boldsymbol{x}$ e $n_1$, $n_2$, ..., $n_j$ é um caminho livre de laço [MAL91].

Uma *associação definição-c-uso* é uma tripla (*i, j,* $\boldsymbol{x}$) onde *i* é um nó que contém uma definição global de $\boldsymbol{x}$ e $j \in$ *dcu(x,i)*. Uma *associação definição-p-uso* é uma tripla *(i, (j,k),* $\boldsymbol{x}$*)* onde *i* é um nó que contém uma definição global de $\boldsymbol{x}$ e *(j, k)* $\in$ *dpu(x,i)*. Uma *associação* é uma *associação definição-c-uso*, uma *associação definição-p-uso* ou um *du-caminho* [RAP82, RAP85].

A Família de Critérios de Fluxo de Dados (FCFD), proposta por Rapps e Weyuker, exige a ocorrência explícita de um uso de variável nos elementos requeridos; já a Família de Critérios Potenciais Usos (FCPU), proposta por Maldonado, Chaim e Jino [MAL88, MAL91], não exige a ocorrência explícita de um *uso* de variável; esta é uma pequena, mas fundamental, modificação nos conceitos apresentados por Rapps e Weyuker. Os elementos requeridos pela Família de Critérios Potenciais Usos são decorrentes dos caminhos livres de definição alcançados para cada variável definida no programa. Não importa o uso real desta variável mas os pontos onde pode existir um uso; se uma variável $\boldsymbol{x}$ é definida em um nó *i* e pode existir um uso em um nó *j* ou em um arco *(j, k)*, então existe uma *potencial associação* com respeito à variável $\boldsymbol{x}$.

Os critérios baseados em fluxo de dados utilizam informações decorrentes das seqüências de ações executadas sobre as variáveis ao longo de um caminho percorrido [NTA84], auxiliando na detecção de classes de defeitos decorrentes dos usos indevidos das variáveis nos comandos ao longo do programa. A característica comum desses critérios é a

de requerer interações que envolvam definições de variáveis e seus subseqüentes usos em outros pontos do programa [HER76, LAS83, RAP82, RAP85, NTA84, MAL91]. Os critérios baseados em análise de fluxo de dados baseiam-se na idéia de que não se deve confiar em um resultado de uma computação se o ponto onde ela foi realizada e o subseqüente ponto onde seu valor é usado nunca foram executados conjuntamente [RAP85, MAL91].

### 2.1.1 Teste de Unidade

O teste de unidade concentra-se na execução e exame da menor parte do software, que pode ser uma função de uma linguagem de programação procedimental. Os resultados baseados na especificação (*resultados esperados*) são confrontados com os resultados obtidos durante a execução de um caso de teste para saber se algum defeito foi revelado. Isso é feito para cada unidade testada isoladamente.

Muitas vezes, para executar cada unidade isoladamente, é necessária a construção de *drivers* e *stubs* de testes. O *driver* recebe os dados de teste, passa-os como parâmetros para a unidade de programa em teste e apresenta os resultados produzidos para que o testador possa avaliá-los. O *stub* serve para substituir as partes subordinadas às unidades em teste. Esses *stubs* são, em geral, unidades que simulam o comportamento de unidades reais de programas através de um mínimo de computação ou manipulação de dados [VIL98]. A Figura 2.1 mostra uma unidade em teste.

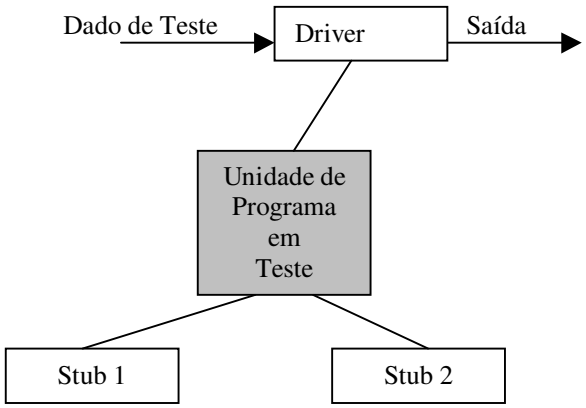



*Figura 2.1 – Ambiente para o teste de unidade*

A criação desses *stubs* e *drivers* é geralmente uma tarefa que merece muita atenção

e é muitas vezes demorada, dificultando ainda mais a fase de teste. Mas é uma tarefa importante para o teste de unidade.

No teste de unidade o programa pode ser avaliado segundo a visão funcional do *software* e a visão da estrutura da implementação (visão estrutural). É importante notar que as duas técnicas mencionadas não são alternativas, isto é, não se utiliza apenas uma ou outra técnica: elas são complementares. Geralmente, uma técnica proporciona a descoberta de classes de defeitos diferentes das que a outra proporciona. Por isso, para que o teste seja o mais completo possível, deve-se utilizar as duas técnicas.

Historicamente o teste estrutural vem sendo aplicado principalmente no teste de unidade. Vários trabalhos propõem sua aplicação no teste de integração [HAR91, CLA89, VIL98].

### 2.1.2 Teste de Integração

O teste de integração é usualmente aplicado depois que todas as unidades que compõem um programa foram devidamente testadas isoladamente (*teste de unidade*). O teste de integração pode ser visto como uma técnica sistemática para a construção da estrutura do software, procurando revelar defeitos associados à interação entre os procedimentos testados na etapa anterior.

A integração das unidades pode ser feita de maneira não incremental, numa abordagem conhecida como "integração *big-bang*" [PRE97]. Neste caso todas as unidades são colocadas para interagir juntas, de uma só vez, e o programa é testado como um todo. Geralmente, o resultado é uma situação caótica, onde um número grande de problemas é identificado tornando a sua correção mais difícil. Muitos autores recomendam que se utilizem abordagens incrementais para o teste de integração.

Pressman [PRE97] diz que as estratégias incrementais para o teste de integração são melhores, devido ao fato de que o programa é construído e testado em partes, onde os defeitos são facilmente isolados e corrigidos, as interfaces são exercitadas de maneira completa e uma abordagem sistemática é aplicada de maneira mais controlada.

No modo incremental, a integração denominada "*top-down*" inicia-se a partir do módulo principal de controle do programa (*procedimento principal*) e prossegue em

direção aos módulos inferiores na hierarquia de controle. Uma das vantagens desta estratégia de integração é testar os principais pontos de decisão da hierarquia de controle antes dos demais, visando a detectar problemas sérios de controle.

Uma terceira estratégia de integração é denominada de integração "*Bottom-Up*" que consiste em integrar as unidades do programa a partir da base da estrutura do software em direção ao seu topo. Nesse caso, como as unidades são integradas de baixo para cima, ao integrar um determinado nível, as unidades subordinadas (do nível inferior) estão devidamente testadas e disponíveis, eliminando assim a necessidade de elaboração de *stubs*. Existem dois tipos de integração *Bottom-Up*:

a) integração por grupos de unidades ou *"clusters"*;
b) integração por pares de unidades (*duas-a-duas* ou *"pairwise"*).

### 2.1.3 Trabalhos Relacionados ao Teste Estrutural

Vários trabalhos propõem critérios baseados em análise de fluxo de dados e em fluxo de controle, contribuindo para o avanço tecnológico na obtenção da qualidade de software através do teste estrutural. Os critérios baseados em fluxo de dados tiveram como motivação preencher a lacuna entre o critério *todos os ramos* e o critério *todos os caminhos*. O principal problema do teste de caminhos reside principalmente no fato de que, para muitos programas, o número de caminhos é extremamente grande devido à ocorrência de estruturas de iteração no programa. Vários critérios têm sido propostos para a seleção de caminhos com o objetivo de introduzir critérios mais rigorosos do que o critério todos os ramos, porém menos restritivos e dispendiosos que o critério todos os caminhos.

Os critérios baseados em análise de fluxo de dados utilizam a informação de fluxo de dados para derivar os requisitos de teste e requerem que as interações que envolvem definições de variáveis de programa e subseqüentes referências a essas variáveis sejam exercitadas. Esses critérios baseiam-se, portanto, para a derivação de casos de teste, nas associações entre a definição de uma variável e seus possíveis usos subseqüentes.

Segundo Frankl e Weyuker [FRA86], uma das propriedades que deve ser satisfeita por um bom critério de teste é a *aplicabilidade* [FRA88, FRA93]; diz-se que um critério *C* satisfaz a propriedade de *aplicabilidade* se para todo programa *P* existe um conjunto de

casos de teste *T* que seja *C-adequado* para *P*, ou seja, o conjunto Π de caminhos executados por *T* inclui cada elemento requerido pelo critério *C*.

Estudos comparativos entre os critérios baseados em fluxo de dados têm sido conduzidos, apoiados principalmente por uma relação de inclusão e pelo estudo da complexidade dos critérios [CLA85, RAP85, WEY84, NAT88]. Maldonado [MAL91] define a complexidade de um critério *C* como o número de casos de teste requerido pelo critério no pior caso; ou seja, dado um programa *P* qualquer, se existir um conjunto de casos de teste *T* que seja *C–adequado* para *P*, então existe um conjunto de casos de teste $T_1$, tal que a cardinalidade de $T_1$ é menor ou igual à complexidade do critério *C*. Vários resultados apresentados indicam que a maioria dos critérios de teste baseados em fluxo de dados tem complexidade de ordem exponencial e que alguns deles têm complexidade polinomial [WEY84]; no entanto, Maldonado [MAL91] provou que todos têm complexidade de ordem exponencial, $O(2^n)$.

A relação de inclusão estabelece uma *ordem parcial* entre os diversos critérios. Diz-se que um critério de teste $c_1$ *inclui* um critério de teste $c_2$ se, para qualquer grafo de fluxo de controle *G* (qualquer programa *P*), qualquer conjunto de caminhos completos *T* que seja $c_1$ – adequado para *P* é $c_2$ – adequado para *P*; ou seja, se *T* satisfaz $c_1$ também satisfaz $c_2$. O critério $c_1$ *inclui* estritamente um critério $c_2$, denotado por $c_1 \Rightarrow c_2$, se $c_1$ inclui $c_2$ e para algum grafo *G* existe um conjunto de caminhos completos *T* de *G* que inclui $c_2$ mas não satisfaz $c_1$. Se $c_1$ não inclui $c_2$ ($c_1 \not\Rightarrow c_2$) nem $c_2$ inclui $c_1$ ($c_2 \not\Rightarrow c_1$) diz-se que $c_1$ e $c_2$ são *incomparáveis*.

A seguir apresentaremos alguns trabalhos relacionados à fase de teste de unidade e outros trabalhos relativos ao teste de integração, segundo a estratégia de teste estrutural de programas.

- **Teste de Unidade**

Huang [HUA75] realizou um estudo de geração de dados de teste a partir da análise de predicados que ocorrem nos arcos de um grafo de programa, abordando os predicados isoladamente e através de combinações entre eles; tais critérios estabelecem a necessidade de tornar os critérios mais exigentes, isto é, mais exigentes do que o critério todos os ramos.

Herman [HER76] apresenta várias definições baseadas em análise de fluxo de dados aplicadas em teste de programas; seus critérios requerem que toda referência de uma variável seja exercitada pelo menos uma vez, a partir de pontos do programa em que essa variável foi definida. Essas definições deram início à estratégia de teste estrutural baseada na análise de fluxo de dados.

Laski e Korel apresentam outros critérios de teste que usam informações de fluxo de dados: o critério *ambiente de dados* (*"data environment"*), o critério *contexto elementar de dados* (*"elementary data context"*) e o critério *contexto ordenado de dados* (*"ordered data context"*) [LAS83].

Ntafos [NTA84] introduz uma família de critérios denominada *k-tuplas requeridas* (*"required K-tuples"*); essa família de critérios requer que todas as seqüências de (K-1) ou menos interações *definição-uso* (*2-dr interactions*) sejam exercitadas.

Rapps e Weyuker [RAP82, RAP85] definem as propriedades apresentadas na Seção 2.1, com informações semânticas da programação procedimental.

Para considerar a existência de *caminhos não executáveis*, Frankl e Weyuker [FRA86, FRA88] propõem modificações em suas definições originais. Um caminho completo é *executável* ou *factível* se existir um conjunto de valores que possam ser atribuídos às variáveis de entrada do programa que causa a execução desse caminho. Caso contrário, diz-se que ele é não executável. Um sub-caminho é executável se estiver contido em um caminho completo executável. Uma associação def-uso é executável se existir um caminho livre de definição executável que cubra essa associação; caso contrário, é não executável. Frankl e Weyuker [FRA88] definem os seguintes conjuntos factíveis:

*(a)* $fdcu(x,i) = \{j \in dcu(x,i) \mid$ a associação $[i, j, \mathbf{x}]$ é executável$\}$;

*(b)* $fdpu(x,i) = \{(j, k) \in dpu(x,i) \mid$ a associação $[i, (j, k), \mathbf{x}]$ é executável$\}$.

A Família de Critérios Baseados em Fluxo de Dados (FCFD) é composta pelos seguintes critérios:

- *todas-definições*: requer que cada definição de uma variável *x* seja exercitada pelo menos uma vez, associando-a por pelo menos um c-uso ou um p-uso, através de um caminho livre de definição c.r.a *x* do nó onde ocorre a definição ao nó ou arco onde ocorre um uso;

- *todos-usos*: requer que todas as associações entre cada definição de variável e subseqüentes *c-usos* e *p-uso*s dessa definição sejam exercitadas pelo menos uma vez;
- *todos-p-usos*: requer que todas as associações entre cada definição de variável e subseqüentes *p-usos* dessa definição sejam exercitadas pelo menos uma vez;
- *todos-c-usos*: requer que todas as associações entre cada definição de variável e subseqüentes *c-usos* dessa definição sejam exercitadas pelo menos uma vez;
- *todos-p-usos/algum-c-uso*: requer que todas as associações entre cada definição de variável e subseqüentes *p-usos* sejam exercitadas; caso não haja *p-uso*, que pelo menos um *c-uso* associado a essa definição seja exercitado;
- *todos-c-usos/algum-p-uso*: requer que todas as associações entre cada definição de variável e subseqüentes *c-usos* sejam exercitadas; caso não haja *c-uso*, que pelo menos um *p-uso* associado a essa definição seja exercitado.
- *todos-du-caminhos*: requer que todas as associações entre cada definição de variável e subseqüentes *p-usos* ou *c-usos* dessa definição sejam exercitadas por todos os caminhos livres de definição e livres de laços que cubram essas associações; ou seja, requer que *todos-du-caminhos* sejam exercitados.

Frankl e Weyuker [FRA88] fazem uma análise de inclusão entre os critérios FCFD, concluindo que os critérios da FCFD estabelecem uma hierarquia entre os critérios *todos-ramos* e *todos-caminhos*. Ntafos [NTA88] também apresenta uma análise de critérios fazendo uma comparação mais abrangente, envolvendo vários outros critérios baseados essencialmente em fluxo de controle.

Ntafos [NTA88] prova que os critérios de Laski e Korel incluem o critério todos os usos e são *incomparáveis* com o critério *todos-du-caminhos* e que o critério de Herman (*"dr interation"*) [HER76] não cobre o critério *todos-ramos*.

Os critérios *todos-usos* da FCFD [RAP85], um dos critérios de Ntafos [NTA84] e um dos critérios de Laski e Korel [LAS83] são similares ao critério de Herman. Uma das exigências mínimas de um bom critério de teste é cobrir o critério *todos os arcos*.

Maldonado, Chaim e Jino [MAL88, MAL88b, MAL91] introduzem a Família de Critérios Potenciais Usos e para isso usam as seguintes definições:

*(i)* *defg(i)* = {variável $\boldsymbol{x}$ | $\boldsymbol{x}$ é definida no nó *i*};

*(ii)* *pdcu(v,i)* = {nós $j \in N$ | existe um caminho livre de definição c.r.a. $\boldsymbol{x}$ do nó *i* para o nó *j* e $\boldsymbol{x} \in$ *defg(i)*};

*(iii)* *pdpu(v,i)* = {arcos *(j,k)* | existe um caminho livre de definição c.r.a $\boldsymbol{x}$ do nó *i* para o arco *(j,k)* e $x \in$ *defg(i)*};

*(iv)* *potencial-du-caminho* c.r.a. $\boldsymbol{x}$ é um caminho livre de definição $(n_1, n_2, ..., n_j, n_k)$ c.r.a. $\boldsymbol{x}$ do nó $n_1$ para o nó $n_k$ e para o arco $(n_j, n_k)$, onde o caminho $(n_1, n_2, ..., n_j)$ é um caminho livre de laço e no nó $n_1$ ocorre uma definição de $\boldsymbol{x}$;

*(v)* *associação potencial-definição-c-uso* é uma tripla [*i, j,* $\boldsymbol{x}$] onde $\boldsymbol{x} \in$ *defg(i)* e $j \in$ *pdcu(x,i)*;

*(vi)* *associação potencial-definição-p-uso* é uma tripla [*i,(j,k),*$\boldsymbol{x}$] onde $\boldsymbol{x} \in$ *defg*(i) e $(j, k) \in$ *pdpu*(*x*,i);

*(vii)* *associação potencial-uso é uma associação potencial-definição-c-uso, uma associação potencial-definição-p-uso* ou um *potencial-du-caminho*.

Maldonado [MAL91] define como conjuntos factíveis:

*(viii)* *fpdcu(x,i)={j ∈ pdcu(x,i)* | a potencial-associação *[i, j,* $\boldsymbol{x}$*]* é executável};

*(ix)* *fpdpu(x,i) = {(j, k) ∈ pdpu(x,i)* | a potencial-associação *[i, (j, k),* $\boldsymbol{x}$*]* é executável}.

A família de critérios de teste estrutural proposta por Maldonado et al [MAL88, MAL88b, MAL91] está baseada na potencial associação que uma variável $\boldsymbol{x}$ pode ter a partir de um nó do grafo de programa G em que ocorre uma definição global dessa variável:

*(i)* *todos-potenciais-usos*: para todo nó $i \in G$ tal que *defg(i)*$\neq \phi$ (onde $\phi$ é o conjunto vazio) e para toda variável $\boldsymbol{x}$ tal que $\boldsymbol{x} \in$ *defg(i)* requer que todas as *associações potenciais-definição-p-uso* e todas as *associações potenciais-definição-c-uso* sejam exercitadas pelo menos uma vez

*(ii)* *todos-potenciais-du-caminhos*: para todo nó $i \in G$ tal que *defg(i)*$\neq \phi$ e para toda variável $\boldsymbol{x}$ tal que $\boldsymbol{x} \in$ *defg(i)* requer que todos os *potenciais du-caminho* c.r.a. $\boldsymbol{x}$ sejam exercitados pelo menos uma vez;

*(iii)* *todos-potenciais-usos/DU:* para todo nó $i \in G$ tal que $defg(i) \neq \phi$ e para toda variável $\boldsymbol{x}$ tal que $\boldsymbol{x} \in defg(i)$ requer o exercício das mesmas potenciais associações de fluxo de dados c.r.a. $\boldsymbol{x}$, mas colocando a restrição de que tal associação deverá ser exercitada por um du-caminho, a partir do ponto onde ocorreu a definição até o ponto onde poderá ocorrer um possível uso.

Maldonado et al. [MAL92] apresentam uma relação de inclusão entre os critérios da FCPU e os critérios da FCFD de Frankl e Weyuker. Nesse mesmo artigo foi demonstrado que os critérios da família Potenciais Usos mantém uma hierarquia entre os critérios *todos-ramos* e o critério *todos-caminhos*, mesmo na presença de caminhos não executáveis; e que nenhum outro critério baseado em análise de Fluxo de Dados inclui os critérios da FCPU.

A aplicação dos critérios baseados em análise de fluxo de dados, sem o apoio de ferramentas automatizadas, fica limitada a programas muito simples [KOR85]. A importância de tais ferramentas tem sido discutida por vários autores [DEM80, NTA88, FRA87, WEY84, PRI87, LAS83, MAL91, MAR96] e tem sido feito algum esforço para implementar tal classe de ferramentas [FRA85a, FRA85b, KOR85, WHI85, CHA91, OST91, HOR91, DEL99, CRU99].

Os critérios de teste estrutural tratados em [LAS83, RAP82, NTA84, RAP85, MAL91] foram desenvolvidos com o intuito de serem aplicados no teste de unidade; contudo, seus conceitos e terminologias podem ser usados tanto para o teste de unidade como para o teste de integração.

- **Teste de Integração**

Linnenkugel e Müllerburg [LIN90] propõem adaptações de critérios do teste de unidade para o teste de integração de programas. O modelo de integração proposto representa um programa por um *grafo de chamada*. A abordagem concentra-se em analisar relações e interfaces entre as unidades. As relações são determinadas pelos comandos de chamada dentro dos procedimentos e as interfaces são determinadas pelos dados usados nos procedimentos (*chamador* e *chamado*). Os critérios baseados em fluxo de controle (*todos-nós*, *todos-ramos* e *todos-caminhos*) foram adaptados para testar as relações entre os procedimentos; os critérios baseados em fluxo de dados de Rapps e Weyuker [RAP85] e

outros, foram adaptados para testar interfaces.

Para o teste de integração é definido um conjunto de critérios baseados no grafo de chamada do programa, idênticos aos critérios baseados em fluxo de controle para o teste de unidade. Os critérios de integração baseados no grafo de chamada são denominados: i) *todos os módulos*: requer que cada procedimento seja executado pelo menos uma vez (equivale ao critério todos os nós); ii) *todas as relações*: requer que cada relação de chamada entre procedimentos seja exercitada pelo menos uma vez; iii) *todas relações múltiplas*: requer que cada chamada entre procedimentos seja exercitada pelo menos uma vez (equivale ao critério todos os arcos); iv*) todas seqüências simples descendentes*: requer que cada seqüência descendente de chamada sem repetição de chamadas, seja exercitada pelo menos uma vez; v) *todas seqüências simples descendentes sem laço*: requer que cada seqüência descendente sem laço seja executada pelo menos uma vez; vi) *todas seqüências de chamadas*: requer que cada seqüência descendente de chamadas seja executada pelo menos uma vez. Considerando as várias argumentações de que esses critérios não eram suficientes já que eles não requerem mais do que a execução de chamadas, insuficiente para testar a interface entre os procedimentos, e que as interfaces entre os dois procedimentos são determinadas pelos dados usados entre elas, Linnernkugel e Müllerburg aplicam os conceitos do teste baseado em análise de fluxo de dados para definir critérios para o teste de integração. Esses critérios de teste de integração são semelhantes aos da Família de Critérios de Rapps e Weyuker (*"int-all-defs", "int-all-c-uses/some-p-uses", "int-all-p-uses/some-c-uses", "int-all-uses", "int-all-du-paths"*).

Harrold e Soffa [HAR91] apresentam uma abordagem para o teste de integração que estende critérios de teste estrutural baseado em análise de fluxo de dados para o teste *interprocedimental*. Os procedimentos são analisados separadamente, de modo a determinar as posições de definições e usos das variáveis globais e das variáveis passadas por referência através de parâmetros, que podem possuir um efeito interprocedimental. É proposta uma técnica de análise que computa as associações *definição-uso* interprocedimentais requeridas (para dependência direta e indireta) e um critério de teste que usa essa informação para selecionar e executar sub-caminhos através das interfaces dos procedimentos.

Nesta mesma linha, Vilela [VIL98] define um programa como uma coleção de unidades, que pode ser representada por um multi-grafo direcionado denominado grafo de chamada GC(P) = (*N*, *A*, *s*), onde as *unidades de programas* (*UPs*) são representadas pelos nós $\eta \in N$ e os possíveis fluxos de controle entre as unidades estão associados a arcos $a \in A \subseteq N \times N$. Como GC(P) é um *multi-grafo*, podem existir mais de um arco entre dois nós; se existir pelo menos um arco entre duas unidades ($\eta_C$ *e* $\eta_R$), existe uma conexão entre essas duas unidades e o arco é representado pelo par $(\eta_C, \eta_R)$, onde $\eta_C$ é a unidade chamadora (superior) e $\eta_R$ é a unidade chamada (subordinada). Assume-se que qualquer nó do grafo pode ser executado a partir do nó *s*, chamado de nó raiz. Vilela [VIL98] define o caminho de integração entre duas unidades de programa como sendo formado pela concatenação $\boldsymbol{p}_{\mathbf{in}}.\ \boldsymbol{q}.\ \boldsymbol{p}_{\mathbf{out}}$, onde $\boldsymbol{p}_{\mathbf{in}}$ é o subcaminho inicial da unidade chamadora (que vai do nó de entrada da unidade $\eta_C$ até o nó que contém o comando de chamada da unidade $\eta_R$), $\boldsymbol{q}$ é o caminho que vai do nó inicial da unidade $\eta_R$ até o nó de saída de $\eta_R$ (quando as unidades forem analisadas duas a duas), e $\boldsymbol{p}_{\mathbf{out}}$ é o subcaminho final de $\eta_C$ que vai do nó seguinte[3] ao nó que contém o ponto de chamada de $\eta_R$ até o nó de saída do grafo de programa da unidade chamadora ($\eta_C$). O conjunto de todos os caminhos de integração entre duas unidades de programa é representado por:

$$_{i}Path(\eta_C, \eta_R) = \{_{i}\boldsymbol{P}_{\boldsymbol{1}}, {}_{i}\boldsymbol{P}_{\boldsymbol{2}}, \ldots, {}_{i}\boldsymbol{P}_{\boldsymbol{n}}\}_{C,R}$$

onde $\boldsymbol{i}$ significa integração entre as unidades $\eta_C$ e $\eta_R$. O conjunto de todos os caminhos no grafo de chamada GC(P) é denotado por $_{GC}Path(P)$ [VIL98].

Vilela [VIL98] propõe uma Família de Critérios Potenciais Usos de Integração adaptados dos critérios básicos e dos critérios executáveis da Família de Critérios Potenciais Usos aplicados ao teste de unidade. Os critérios propostos em [VIL98] consideram a integração das unidades feita dois a dois ("*pairwise*") e seus requisitos de teste são derivados para cada par de unidades envolvidas. Os critérios propostos são: *i)* critério INT-Todos-Potenciais-Usos; *ii)* critério INT-Todos-Potenciais-DU-Caminhos; e *iii)* critério INT-Todos-Potenciais-Usos/DU.

[3] *Todo bloco de comandos que tiver um comando de chamada deverá ser encerrado logo após o comando de chamada, incluindo ai um novo bloco.*

A partir da equivalência de classes de objetos de um programa, Harrold e Rothermel [HAR94] apresentam uma abordagem de teste de integração entre os métodos e classes de um programa orientado a objetos. Essa abordagem testa os métodos e classes segundo três níveis: i) *teste intra-método*: testa os métodos (operações) individualmente (equivalente ao teste de unidade); ii) *teste inter-mét*odos: testa um método público junto com outro método em sua classe que o chama direta ou indiretamente (equivale ao teste de integração interprocedimental); iii) *teste intra-classe*: testa as interações de métodos públicos quando são chamados em várias seqüências. Este trabalho motivou a abordagem aqui proposta para o teste de programas de ABDR, discutida no próximo capítulo.

A seguir são apresentados os conceitos básicos e a terminologia necessária para o entendimento deste trabalho no que se refere à área de Banco de Dados Relacional (BDR).

## 2.2 Banco de Dados Relacional

O Modelo de Dados Relacional foi introduzido por Codd em 1970 [COD70]. O modelo é baseado em uma estrutura de dados simples e uniforme denominada "*relação*" e tem uma fundamentação teórica apoiada na álgebra relacional (conjunto de operações para manipular as relações e especificar as consultas) [DAT90, ELM94]. Esta seção contém uma breve introdução das principais características e conceitos básicos do modelo relacional que serão utilizados no teste estrutural de programas de Aplicação de Banco de Dados Relacional (ABDR).

Na Seção 2.2.1 são definidos os principais conceitos do modelo relacional. A Seção 2.2.2 define Aplicação de Banco de Dados Relacional (ABDR). Os Sistemas Gerenciadores de Banco de Dados Relacionais usam comandos de **SQL** (*Structured Query Language*) para manipular os dados armazenados na relação da base de dados, cujos conceitos são apresentados na Seção 2.2.2.1. Na Seção 2.2.2.2 são discutidos os principais comandos da linguagem **SQL** usados em programas de ABDR. Finalizando, na Seção 2.2.3 são discutidos alguns trabalhos relacionados ao teste em BDR.

### 2.2.1 Modelo de Dados Relacional

O modelo relacional representa uma base de dados como uma coleção de *relações*.

Informalmente, cada *relação* pode ser considerada como uma *tabela*. Existem importantes diferenças entre relações e arquivos; a principal delas é a formação estruturada de seus componentes. Quando uma *relação* é tratada como uma *tabela* de valores, cada *linha* na tabela representa uma coleção de valores de dados relacionados. Esses valores podem ser interpretados como *fatos* que representam uma *entidade* do mundo real ou um relacionamento. O nome da tabela e os nomes das colunas são usados para ajudar a interpretar os valores em cada linha da tabela. Por exemplo, uma *tabela* com nome **EMPLOYEE** armazena valores relacionados aos empregados de uma empresa, onde cada linha contém os dados sobre um determinado empregado. Os nomes das colunas – *emp_id, emp_fname, emp_lname* – interpretam os valores de dados específicos para cada linha, baseados nos valores de cada coluna. Todos os valores de uma coluna devem ser do mesmo tipo de dados.

Na terminologia do modelo relacional formal, uma *linha* é denominada de uma *tupla*, um título de uma *coluna* na tabela é um *atributo* e a tabela pode representar uma *relação* [KRO98]. Os tipos de dados descrevem os tipos de valores de cada coluna e esses tipos de valores são chamados de *domínios* dos atributos.

Um *domínio* D é um conjunto de valores *atômicos*[4]. Um domínio é uma descrição lógica e física dos valores permitidos do atributo. Ele pode ser especificado por um tipo de dado cujos valores condizem com sua representação. Um domínio é estabelecido por um nome, um tipo de dado e um formato.

Um *esquema* de *relação* R, denotado por $R(A_1, A_2, ..., A_n)$, é formado pelo *nome* da relação *R* e um conjunto de *atributos* $A_1, A_2, ..., A_n$. Cada *atributo* $A_i$ é o nome de um papel exercido por algum *domínio* $D_i$ no esquema *relacional R*. $D_i$ é chamado o *domínio* de $A_i$ e é denotado por dom($A_i$). Um esquema de relação é usado para *descrever* uma relação. *R* é o nome dessa relação. O *grau* de uma relação é o número de *atributos* ***n*** desse esquema de relação.

Uma relação (ou instância da relação) *r* do esquema de relação $R(A_1, A_2, ..., A_n)$, também denotado por *r(R),* é um conjunto de n-tuplas $r = \{\tau_1, \tau_2, ..., \tau_n\}$. Cada n-tupla $\tau$ é uma lista ordenada de *n* valores $\tau = <v_1, v_2, ..., v_n>$, onde cada valor $v_i$, para $1 \leq i \leq n$, é um

elemento do domínio de $A_i$ ($dom(A_i)$) ou é um valor nulo especial. Cada tupla na relação representa uma entidade particular do esquema de relação. Os valores nulos representam atributos cujos valores são desconhecidos ou não existentes para alguma tupla individual da relação.

Uma relação r(R) é um subconjunto do produto cartesiano dos domínios que definem R:

$$\mathbf{r(R) \subseteq (dom(A_1) \times dom(A_2) \times ... \times dom(A_n))}$$

O produto cartesiano especifica todas as possíveis combinações de valores dos domínios subjacentes. Assim, se denotarmos o número de valores ou a cardinalidade de um domínio D por |D|, e assumirmos que todos os domínios são finitos, o número total de tuplas resultante do produto Cartesiano é:

$$\mathbf{|dom(A_1)| * |dom(A_2)|*...*|dom(A_n)|}$$

Além de todas essas possíveis combinações, uma instância de relação em um dado tempo representa o estado atual da relação e reflete somente as tuplas válidas que representam um estado particular do mundo real. Em geral, como o estado do mundo real se altera, a relação também é transformada para outro estado de relação. Entretanto, o esquema *R* é relativamente estático e não se altera, exceto muito esporadicamente [ELM94].

Um esquema da base de dados relacional ***S*** é um conjunto de esquemas de relação $\boldsymbol{S}=\{R_1, R_2, .., R_m\}$ acompanhado de suas restrições de integridade. Uma instância da base de dados relacional ***DB*** de ***S*** é um conjunto de instâncias de relações $\boldsymbol{DB} = \{r_1, r_2, ..., r_m\}$ tal que cada $r_i$ é uma instância de $R_i$ e tal que a relação $r_i$ satisfaça às suas restrições de integridade. Geralmente, uma aplicação de BDR possui vários esquemas, e cada esquema *S* pode possuir várias relações sendo algumas comuns a diferentes usuários e outras não. No exemplo apresentado no Capítulo 6, adota-se um esquema da base de dados relacional formado pelos seguintes esquemas de relação (*tabelas*):

***S*** = {**EMPLOYEE**, **DEPARTMENT**, **SALES_ORDER**, **SALES_ORDER_ITEMS**, **FIN_CODE**, **FIN_DATE**, **CUSTOMER**, **PRODUCT**, **CONTACT**}

[4] *O termo atômico significa que cada valor no domínio é indivisível. No modelo relacional os valores em uma tupla são considerados atômicos.*

Cada esquema de relação ou tabela possui seus atributos e as respectivas restrições, de modo a estabelecer as regras de integridade entre as relações.

Nem sempre a palavra *Relacional* é bem aplicada aos produtos existentes no mercado. Para qualificar um SGBD relacional genuíno, o sistema deve possuir pelo menos as seguintes propriedades [ELM94]:

1. Armazenar os dados como relações de tal maneira que cada coluna seja identificada independentemente de seus nomes sem que a ordenação das linhas seja importante;
2. As operações disponíveis para o usuário e as operações usadas internamente pelo sistema devem ser operações relacionais verdadeiras (disponíveis para gerar novas relações a partir de relações existentes);
3. Suportar pelo menos uma variante de operação de junção (JOIN).

Algumas vezes uma *relação* é comparada com uma variável composta (ex.:um *vetor*). Uma *tupla* também pode ser considerada como um *registro*. Tais variáveis, quando usadas no teste estrutural, exigem um tratamento especial que veremos com mais detalhes na Seção 2.2.2.2. A seguir apresentaremos os conceitos básicos de Aplicação de BDR e da linguagem **SQL**.

### 2.2.2 Aplicação de Banco de Dados Relacional (ABDR)

As aplicações de Banco de Dados Relacional são formadas por programas de linguagens convencionais como C, Pascal, Fortran, Cobol, ADA, PL/I e outras mais, que permitem o uso de comandos de **SQL** embutidos em seu código. O processo de projeto de banco de dados possui vários passos que precedem a fase de geração da ABDR. Elmasri e Navathe [ELM94] descrevem as seguintes fases do projeto de banco de dados:

*(i)* *coleta e análise de requisitos*: os projetistas fazem várias reuniões com os usuários do Banco de Dados para entender e documentar seus requisitos de dados. Todos os requisitos dos usuários são escritos e posteriormente especificados e detalhados;

*(ii)* *projeto de banco de dados conceitual*: com os requisitos coletados e analisados, cria-se um esquema conceitual para a base de dados usando o

modelo de dados conceitual de alto nível. O esquema conceitual é uma descrição concisa dos requisitos de dados dos usuários e descrições detalhadas dos tipos de dados, relacionamentos e restrições (utilizando os conceitos fornecidos pelo modelo de dados de alto nível);

*(iii)* *projeto de banco de dados lógico ou mapeamento do modelo de dados*: o projeto de banco de dados é a implementação da base de dados usando um SGBD comercial. Seus resultados são os esquemas da base de dados a serem usados na implementação do modelo de dados do SGBD;

*(iv)* *projeto de banco de dados* físico: as estruturas de armazenamento internas e organizações de arquivos são especificadas.

Várias fases são definidas em paralelo às fases apresentadas acima. Em paralelo à fase de especificação dos requisitos de dados é útil especificar os requisitos funcionais da aplicação. Esses requisitos são extraídos das transações ou operações definidas pelos usuários, que serão aplicadas sobre a base de dados e que incluem recuperação e atualização da base de dados. É comum o uso de técnicas tais como diagramas de fluxo de dados para especificar os requisitos funcionais.

Paralelamente ao projeto físico do banco de dados são projetados os programas de aplicação de Banco de Dados e, a partir daí, o projeto é implementado na forma de transações sobre a base de dados, gerando assim, cada programa da Aplicação.

Uma aplicação de Banco de Dados é composta por um ou vários programas (descritos em linguagens de programação procedimental) que suportam comandos da linguagem **SQL**. Para isso, o SGBD deve possuir um pré-compilador que processa o programa fonte gerando um novo programa modificado que será, então, submetido à compilação e geração do programa executável. As linguagens de programação mais comuns como C, Pascal, Fortran, Cobol, e PL/I são aceitas pela maioria dos SGBDs relacionais.

O termo linguagem hospedeira é usado aqui para representar a linguagem existente nos SGBDs que possibilita o uso de comandos de **SQL**; programas hospedeiros são os programas da aplicação que hospedam a linguagem **SQL**.

#### 2.2.2.1 Linguagem de Banco de Dados Relacional – SQL

A linguagem **SQL** foi projetada e implementada como uma interface para um sistema de banco de dados relacional denominado sistema R. Atualmente, **SQL** é a linguagem mais usada pelos SGBDRs comerciais e tem sido bastante modificada pelos diversos fornecedores, o que resultou em diferentes tipos de comandos, muitas vezes incompatíveis. Houve, então, a necessidade de se gerar um padrão para a linguagem **SQL** através da união de esforços entre o ANSI e a ISO [DAT84, ISO92, ELM94].

A **SQL** usa os termos *tabela*, *linha* e *coluna* para representar uma relação, tupla e atributo, respectivamente. Essa distinção entre **SQL** e o modelo relacional formal discutido na seção anterior é caracterizada pelo fato de que **SQL** usa o termo tabela para possibilitar a existência de duas ou mais tuplas idênticas, pois no modelo relacional isso não é permitido. Portanto, uma tabela não é, em geral, um conjunto de tuplas, porque o conjunto de tuplas não permite dois membros idênticos [ELM94]. Desta forma, adotaremos os termos tabela, linha e coluna quando estivermos tratando dos comandos da **SQL** e relação, tupla e atributo quando estivermos referenciando o modelo relacional.

Existem vários SGBDRs tais como o ORACLE, o DB2 da IBM, o INGRES, INFORMIX e outros. Em geral, os SGBDRs permitem o uso de comandos **SQL** em programas de aplicação (embutidos em linguagens de programação), em módulos interativos e em programação a partir de uma linguagem de programação procedimental específica de cada SGBDR.

Os comandos da **SQL** são classificados em dois tipos: declarativos e executáveis. Tais comandos são utilizados pelos sistemas que suportam o uso de aplicações da **SQL** em linguagens hospedeiras. Existem vários comandos considerados declarativos; o sistema da ORACLE utiliza o comando DECLARE para declarar os objetos utilizados na pré-compilação; o comando INCLUDE é utilizado para declarar áreas de ligação entre as duas linguagens; e o comando *WHENEVER <condição> <ação>* declara variáveis de controle, colocadas em uma área de comunicação denominada de *SQLCA* (no caso do Oracle e DB2); esse comando é responsável pelo fluxo de controle dos comandos executáveis da **SQL**, a partir da *<condição>* e *<ação>* adotadas no comando.

Os comandos executáveis são subdivididos em dois blocos de comandos: Comandos

de Definição e Comandos de Manipulação de Dados. Os Comandos de Definição são compostos por: *a*) (ALTER, CREATE, DROP, RENAME) que definem dados das relações; e *b*) (CONNECT, GRANT, LOCK TABLE, REVOKE) que controlam o acesso aos dados dos esquemas da base de dados. Comandos de Manipulação de Dados são compostos por: *a*) (DELETE, INSERT, UPDATE) usados para manipular os dados das relações, também considerados como comandos de alteração das relações; *b*) (CLOSE, FETCH, OPEN, SELECT) usados para recuperar os dados das relações da base de dados; *c*) (COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION) usados para processar as transações das relações da base de dados; e *d)* (DESCRIBE, EXECUTE, PREPARE) usados para aplicação com **SQL** dinâmico (que permitem criar consultas em tempo de execução) [ORA90, ISO92].

A **SQL** tem um comando básico que recupera as informações de uma base de dados: o comando SELECT. O comando SELECT possui comandos de consultas (queries) variados que são apresentados no Apêndice B na figura que mostra a gramática do comando SELECT padrão. Existem muitas opções para o comando SELECT na **SQL**.

A base de dados de um SGBDR está disponível para os usuários e programas de aplicações através de um sistema de controle de permissão. Todos os dados são vistos como tabelas. Há dois tipos de Tabelas: *Tabelas Básicas* que existem fisicamente como dados armazenados[5]; e *Tabelas de Visão*, que são tabelas virtuais sem uma identidade física de armazenamento, derivadas de outras tabelas e que só existirão durante a execução do programa. As Tabelas Básicas são usualmente geradas em fases que precedem a fase de implementação. Essas tabelas, quando devidamente autorizadas para seus usuários, são consideradas variáveis globais a todos os programas da ABDR [ELM94]. As Tabelas de Visão são consideradas variáveis locais ao programa que as criou. Elas são criadas através do comando CREATE VIEW e são definidas pelo comando SELECT, podendo ser manipuladas por qualquer procedimento pertencente ao Programa; são destruídas pelo comando DROP VIEW [ORA90].

[5] Os dados armazenados como relação ou "*Tabelas*" possuem mecanismos próprios do SGBD que gerenciam e controlam as permissões de acesso à *base de dados.* Através do gerenciador de dados, do gerenciador de buffer e do gerenciador lógico, o sistema é capaz de fornecer todas as funções necessárias para a manipulação dos dados armazenados (pesquisar, recuperar e atualizar).

O comando CREATE TABLE <*nome-da-tabela*> é usado para especificar uma nova tabela básica dando a ela um nome e especificando seus atributos e restrições. Os atributos são inicialmente especificados e a cada atributo é atribuído um *nome*, um *tipo de dado* para especificar o seu domínio de valores e possivelmente algumas *restrições* como chaves primárias, estrangeiras, não nulas etc., usadas para reforçar a consistência e integridade das relações. O comando ALTER TABLE <*nome-da-tabela*> possibilita a alteração da estrutura de uma tabela básica existente. Sempre que uma alteração afetar uma ou mais colunas de uma tabela cujos atributos foram designados com alguma restrição de integridade referencial, pode-se requerer a opção CASCADE para fazer uma alteração de todas as demais tabelas e visões relacionadas a esse atributo. Se a opção escolhida for RESTRICT o comando somente será efetuado se nenhuma visão ou restrição de integridade referencial fizer referência a este atributo.

Da mesma forma como se pode criar ou alterar uma tabela básica, pode-se destruir a tabela através do comando DROP TABLE <*nome-da-tabela*>. É esperado, portanto, que esses comandos não sejam utilizados com freqüência como ocorre com os demais comandos executáveis da **SQL**. Durante a fase de desenvolvimento de um sistema esses comandos são geralmente aplicados em etapas de implantação das bases de dados de acordo com o projeto lógico do sistema. Apesar de não ser prática comum o uso desses comandos nos programas de uma aplicação de banco de dados relacional, eles não serão excluídos em nosso trabalho.

Na próxima seção são apresentados os principais comandos da **SQL** em programas hospedeiros e os efeitos ocasionados pelas cláusulas de comandos de controle dos SGBDRs.

#### 2.2.2.2 Comandos da Linguagem SQL embutida

O termo **SQL** embutida refere-se a comandos da **SQL** que podem ser colocados em linguagens de programas procedimentais (como PL/SQL no caso do SGBD da Oracle ou em programas escritos em linguagem C que existe em vários SGBD) [ELM00]. Como os programas alojam os comandos da **SQL**, os programas da aplicação são denominados de *programas host* (hospedeiro), e as linguagens nas quais eles são escritos são denominadas

de *linguagens host*. Por exemplo, o Pro*C é o pré-compilador fornecido pelo SGBDR da Oracle e que permite embutir comandos **SQL** em um programa C denominado de *programa host* [ORA90]. Geralmente, os comandos **SQL** são colocados após a diretiva "EXEC SQL", ou "&SQL (" e terminada com um ";" ou ")", respectivamente. No Sistema Gerenciador de Banco de Dados da Oracle, um comando **SQL** em uma linguagem *host* é apresentado da seguinte forma:

**EXEC SQL <comando da SQL>;**

Os SGBDRs pesquisados possuem algumas diferenças com relação à maneira de tratar os comandos de **SQL**. Em geral, os SGBDRs que dispõem de pré-compiladores de linguagens que hospedam comandos da **SQL** fornecem mecanismos para manipulação de erros possíveis de ocorrer durante uma execução de comandos da **SQL**. Tais mecanismos são gerenciados pelo comando declarativo:

**WHENEVER <condição> <ação>;**

onde as ações estabelecem uma regra de controle de fluxo para os comandos executáveis da **SQL** sempre que a condição for satisfeita. Também são gerados valores de controle (códigos) e armazenados em uma área de comunicação da **SQL**, denominada SQLCA. Essa área de acesso comum serve para a recuperação de informação tanto pelo programa de aplicação como pelo SGBDR.

Cada SGBDR define suas próprias condições e ações aplicadas ao comando WHENEVER. O DB2 da IBM, por exemplo, possui as cláusulas NOTFOUND PERFORM X e SQLERROR GOTO y; PERFORM X e GOTO y são ações possíveis de serem executadas no caso da condição ser satisfeita.

Os SGBDRs possuem em geral os seguintes tratamentos de exceções: SQLWARNING, SQLERROR, NOT FOUND; e as seguintes ações: CONTINUE, STOP, GOTO<statement_label> e DO<função>. São as ações que definem os fluxos de controle para os comandos executáveis da **SQL**, encontrados na seqüência estática (de cima para baixo) a partir do comando WHENEVER, quando a condição for estabelecida. A Figura 2.2 mostra os fluxos de controle dos comandos executáveis da **SQL**.

A principal característica do programa de ABDR com comandos de **SQL** é que os valores das relações da base de dados podem ser manipulados por diferentes usuários, controlados pelo SGBDR que dispõe de recursos próprios para garantir a segurança e estabilidade de transação dos dados armazenados nas relações. Como qualquer programa, podem possuir códigos mal formulados durante a implementação ou até consultas mal formuladas derivando dados incorretos aos usuários.

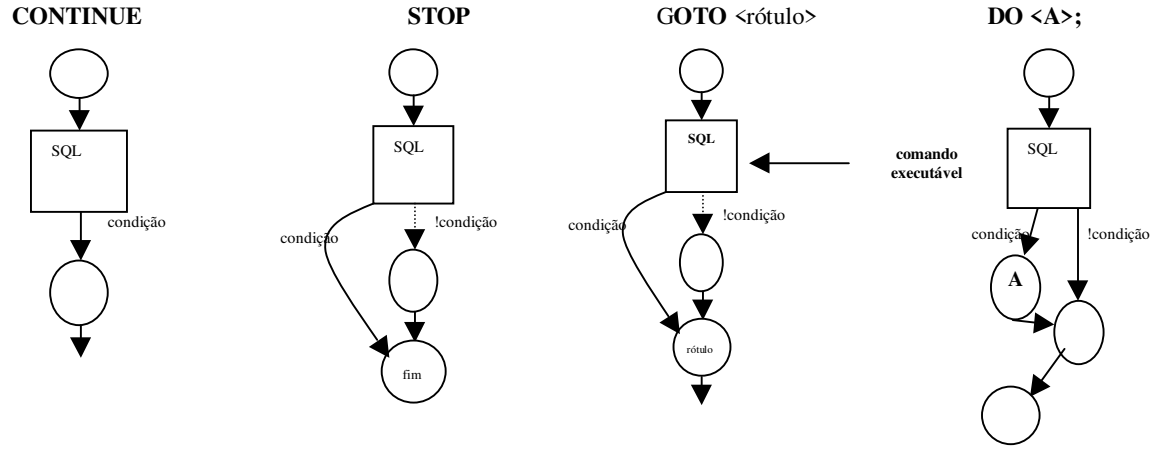



*Figura 2.2 : Representação gráfica do fluxo de controle dos comandos executáveis da **SQL** gerados pelas ações do comando: " WHENEVER <condição> <ação>"*

Quando é utilizado o comando SELECT, entretanto, deve-se definir quantas linhas de dados da tabela queremos que ele retorne. As consultas podem ser classificadas como segue:

- consultas que não retornam nenhuma linha;
- consultas que retornam somente uma linha;
- consultas que retornam mais do que uma linha.

Quando houver necessidade de recuperar mais de uma linha será necessário o uso de cursores declarados explicitamente ou o uso de *variáveis host (conhecidas como variáveis de ligação)* do tipo vetor para controlar o resultado da consulta. Para manipular os cursores explicitamente nos comandos de **SQL**, pode-se utilizar:

| | |
|---|---|
| DECLARE <nome_cursor> | Nomeia um cursor e associa a uma consulta. |
| OPEN | Executa a consulta e identifica o conjunto ativo. |
| FETCH | Avança o cursor, recuperando cada linha no conjunto ativo apontado pelo cursor, um a um. |

CLOSE Desabilita o cursor.

As variáveis *hosts* são fundamentais para estabelecer a comunicação entre as duas linguagens. Elas são o veículo de dados entre as relações da base de dados e as variáveis do programa hospedeiro e vice-versa. Elas podem ser declaradas como um tipo de dados escalar ou vetor. A variável host pode ser usada em qualquer expressão no programa hospedeiro; e nos comandos da **SQL**, elas devem ser prefixadas com um " : " (dois pontos) para os objetos do SGBDR. Para declarar essas variáveis no início do programa (variável global) ou no início de cada procedimento (variável local) deve-se usar as diretivas EXEC SQL BEGIN DECLARE SECTION; após declaradas todas as variáveis, a seção deve ser encerrada com a linha de comando EXEC SQL END DECLARE SECTION;

Nos Exemplos 1 e 2, a seguir, considerem-se as *variáveis host* [*dept_num_value, e_emp_name, e_emp_number, e_salary*], cujos valores foram recuperados pelo cursor *emp_cursor* no *Exemplo1* e passados para as variáveis *host* após a cláusula INTO no *Exemplo2*.

*Exemplo 1:*

```
EXEC SQL OPEN emp_cursor;
EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT    emp_fname, emp_id, emp_sal
        FROM      EMPLOYEE
        WHERE     dept_id = :dept_num_value;
```

*Exemplo 2:*

```
EXEC SQL FETCH    emp_cursor
        INTO  :e_emp_name, :e_emp_number, :e_salary;
```

Os recursos de tratamento de erros ajudam a evitar que alguma operação de manipulação da base de dados seja validada de maneira inconsistente.

Outros pontos importantes tratados nessa abordagem são as *variáveis tabela* que possuem armazenamento em memória secundária. Essas são tratadas como *variáveis persistentes* que possuem várias propriedades definidas pelos sistemas de Banco de Dados. Uma variável é dita *persistente* quando sua definição se mantém viva além da execução do programa que a *definiu*, podendo ser usada na execução de outra unidade de programa.

Uma linguagem de programação possui tipicamente estruturas de dados complexas tais como registros, vetores e outros tipos como os oferecidos pela linguagem Pascal ou nas definições de classes em C++. Os valores dessas variáveis são em geral descartados quando termina a execução do programa. Isso não ocorre com as variáveis persistentes; seus valores ficam armazenados em memórias secundárias (disco, fitas, etc) podendo ser acessados por qualquer outro programa da ABDR [ELM94].

Os dados na base de dados em um momento particular no tempo são chamados de um estado da base de dados (ou conjunto de ocorrências ou instâncias). Cada vez que se insere ou se remove ou se alteram os valores de uma tupla, está-se alterando o estado da base de dados para outro estado. O SGBDR é particularmente responsável por assegurar que cada estado da base de dados seja um estado válido, isto é, que satisfaça a estrutura e as restrições especificadas no esquema da Base de Dados.

## 2.3 Trabalhos Relacionados

Na área de desenvolvimento de Banco de Dados existem poucos trabalhos divulgados que tratam de testes, especificamente teste estrutural. Acredita-se que essa deficiência é ocasionada pela falta de ferramentas e técnicas apropriadas para esse tipo de aplicação. A seguir são apresentados alguns trabalhos que tratam de testes em sistemas de Banco de Dados.

Algumas empresas governamentais e institucionais vêm investindo, nos últimos anos, em testes de aplicações de Banco de Dados com o uso de **SQL**. A Universidade de *Cornell -EUA* desenvolveu uma ferramenta denominada ***SQLBench*** que utiliza um conjunto de testes *Benckmark* AS3AP (ANSI **SQL** *Standard Scalable and Portable*) para sistemas de BDR com **SQL**. Essa ferramenta exercita uma bateria de testes de desempenho para plataformas cruzadas, abrangendo um espectro de operações típicas de banco de dados e pode ser aplicada em sistemas desenvolvidos nas linguagens C, C++, Cobol e **SQL** na plataforma Windows [BUT93].

Alguns fatores como demanda de dados da aplicação em tempo real e dependência de elementos relacionada ao local de trabalho (como número de usuários, freqüência de transações e quantidade de dados envolvidos) são considerados na fase de teste pelo

*SQLBench*. O *SQLBench* trabalha sob a hipótese de que determinar o desempenho de um sistema de aplicação de BDR em estágios mais fáceis (início de projeto), que precedem a implementação, ajuda a corrigir fraquezas detectadas antes da implementação, evitando maiores desgastes no futuro. Isso é possível se estimarmos as condições de carga de dados esperadas durante uma operação real, visando a resolver possíveis problemas de "gargalos", resultantes do modelo estrutural de dados ou da estrutura de acesso à base de dados provocados, por exemplo, pela execução de transações.

Na Universidade da Califórnia (EUA) [DAV97] foi desenvolvido um projeto que detalha o processo de desenvolvimento de um sistema de Banco de Dados, envolvendo as fases de planejamento, análise, projeto, construção e manutenção. Apesar de ser um trabalho completo, não foram divulgadas informações detalhadas sobre a realização dos testes durante o desenvolvimento do sistema de Banco de Dados.

Yan [YAN91] trata de teste declarativo de banco de dados lógico composto por fatos e regras. No teste de programa, os dados de teste são considerados corretos, bastando apenas verificar a corretitude do programa; já no teste de Banco de Dados é necessário verificar a corretitude de ambos, do Banco de Dados e dos predicados usados nas consultas. Yan parte do princípio de que tanto a base de dados como as consultas podem estar incorretas. O intuito é verificar se a base de dados ou as consultas estão inconsistentes ou incompletas, através da interpretação da base de dados refletida pela especificação. Tais verificações também são aplicadas na fase de elaboração do projeto da aplicação do BD lógico. Os dados de teste utilizados durante a fase de teste declarativo do BD lógico contribuem para a geração de dados de testes baseados nas consultas e na base de dados lógica.

Mannila [MAN89] descreve uma técnica que possibilita criar uma amostra simplificada da base de dados real. Em alguns casos, quando as relações da base de dados de uma ABDR são muito grandes, pode-se ter alguma dificuldade para aplicar os testes. Uma forma de contornar esse problema é gerar uma base de dados para o teste, considerada completa, utilizando as próprias consultas existentes no programa [MAN89]. Isso é possível, utilizando operações de *Select Project Join* (SPJ) extraídas das consultas do programa para gerar uma base de dados pequena que represente a base de dados total. Essa

geração da base de dados para teste é um mecanismo que depende do conhecimento de toda a base de dados necessária para cada programa da aplicação, bem como das consultas envolvidas no programa.

Chays et al. [CHA00] apresentam uma proposta de como testar sistemas de Banco de Dados e como testar programas de Aplicação de Banco de Dados Relacional. Eles apresentam um "*framework*" para teste de Aplicação de Banco de Dados analisando os principais aspectos para a correção de um sistema de Banco de Dados. Nesse artigo também são apresentados alguns tipos de restrições que podem ser avaliadas: domínio, unicidade, valores de atributos, integridade referencial e integridade semântica. Também é levantada a importância da linguagem **SQL** como linguagem de definição de dados (DDL) e linguagem de manipulação de dados (DML) em aplicações de Banco de Dados Relacional. Uma das observações apresentadas está relacionada ao estado da Base de Dados, que não pode ser ignorado, além dos aspectos do ambiente e da aplicação. Assim, o ideal é controlar o estado da base de dados antes e depois da execução de cada caso de teste.

O trabalho desenvolvido por Aranha [ARA00] tem como objetivo testar o esquema da base de dados que será usado em uma ABDR. Os elementos da base de dados a serem testados são os atributos e as restrições de integridade; os critérios propostos no trabalho requerem o exercício desses elementos através de operações da linguagem **SQL**. Várias Famílias de Critérios de testes foram propostas nesse trabalho, visando a exercitar as diferentes ocorrências relacionadas aos atributos da base de dados. Foram definidas as seguintes famílias de critérios: Atributos sem Restrições Explícitas, Atributos com Restrições Explícitas, Atributos com Restrições de Integridade Referencial (como chave primária e como chave estrangeira), Atributos com Potenciais Restrições de Integridade Referencial (como chaves primárias e como chaves estrangeiras). Esse trabalho é complementar ao trabalho que será apresentado nos próximos capítulos. A aplicação desses critérios propostos antecede a aplicação dos critérios de teste que serão apresentados neste trabalho.

## 2.4 Considerações Finais

Neste capítulo, são apresentados as técnicas e os conceitos básicos de teste estrutural de programas. A partir dessas técnicas e desses conceitos são definidos a terminologia e conceitos para o teste estrutural de programas de Aplicação de Banco de Dados Relacionais.

Também são apresentados os conceitos básicos de Banco de Dados Relacionais usados no restante desta tese. Os trabalhos relacionados indicam que existem poucas publicações sobre teste em ABDR e que, apesar de existirem várias técnicas de teste estrutural já desenvolvidas elas não contemplam totalmente as necessidades do teste de Programas de ABDR.

No próximo capítulo são apresentados os conceitos básicos e a terminologia adotados neste trabalho.

# 3 Teste Estrutural de Aplicações de Banco de Dados Relacional: Conceitos e Terminologia

Neste capítulo são apresentados os conceitos básicos e a terminologia adotada para o teste de programas de ABDR, segundo duas fases de teste: teste de unidade e teste de integração. O teste de integração engloba duas abordagens distintas: integração baseada no grafo de chamada (idêntica à abordagem para programas convencionais) e integração baseada na dependência de dados existente entre as *variáveis tabela* da Base de Dados; a segunda abordagem é uma das contribuições deste trabalho ao teste estrutural de programas de ABDR. A abordagem de teste proposta para programas de ABDR considera dois tipos de fluxo de dados denominados *intra-modular* e *inter-modular*.

## 3.1 Teste estrutural de ABDR: conceitos básicos e terminologia

O teste estrutural de *programas de ABDR com **SQL** embutida*[6] apresenta características similares as do teste de programas convencionais. Assim, as técnicas convencionais de teste podem ser aplicadas em programas de aplicação a partir de algumas adaptações necessárias, tendo em vista a presença de comandos SQL e *variáveis tabela* e *variáveis host*, não existentes em programas convencionais. Os conceitos básicos apresentados nesta seção são os relativos aos critérios estruturais de teste de programas convencionais, modificados e estendidos para os critérios estruturais de testes utilizados em aplicações de Banco de Dados Relacionais. Na elaboração da abordagem de teste para programas de aplicação foi necessário: *i)* adaptar os conceitos e critérios de testes de unidade e de integração de programas convencionais para programas de ABDR; e *ii)* criar novos critérios de teste estrutural específicos para o escopo de Banco de Dados Relacional (BDR).

Uma Aplicação de Banco de Dados Relacional (ABDR) é composta por um ou mais *Módulos de programa*, onde cada *Módulo*[7] é composto por um conjunto de *Unidades de Programa* (*UP*) que são os procedimentos e funções do programa. Para não confundir com

[6] A partir deste ponto, programa de ABDR com SQL embutida passa-se a chamar de ***programas de aplicação***.
[7] A palavra Módulo usada nesta abordagem não tem o mesmo significado de [MAL91], onde significa a menor unidade de teste; aqui um Módulo representa um programa de aplicação de BDR.

a terminologia adotada na literatura o *módulo* que representa a menor parte executável de um programa será denominado, no contexto desta tese, de *unidade de programa* ou simplesmente *unidade*. Os Módulos geralmente podem ser codificados em diferentes linguagens procedimentais com **SQL** embutida, dependendo do Sistema Gerenciador de Banco de Dados (SGBD) adotado. A Figura 3.1 apresenta uma idéia geral de uma aplicação de BDR.

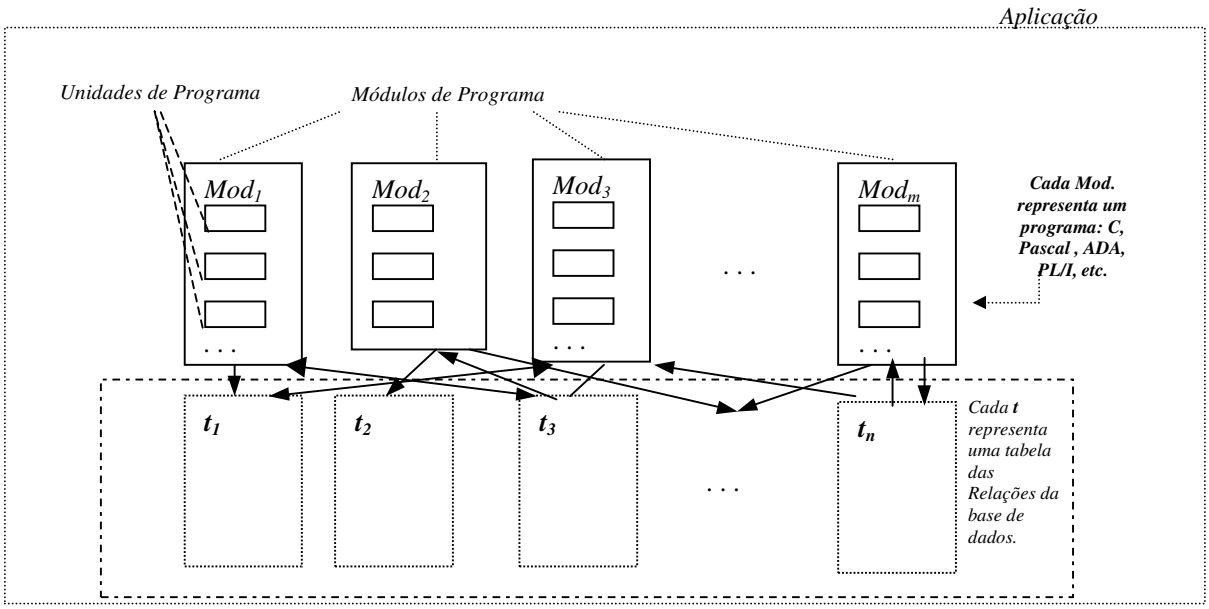



*Figura 3.1. – Uma Aplicação de Banco de Dados Relacional*

Devido à existência dos comandos de **SQL** em programas hospedeiros de aplicações de Banco de Dados relacional, a definição de grafo de programa foi alterada para acomodar os comandos executáveis da **SQL** em nós especiais, um em cada nó. A representação gráfica de uma unidade de programa é a mesma adotada em [CLA89, MAL91], estendida para ABDR com uso de **SQL**. O Módulo $Mod_i$ pode ser decomposto em várias *UP*s que representam as funções ou subrotinas de $Mod_i$, onde $1 \leq i \leq m$. Cada *UP* é uma seqüência finita de comandos da linguagem hospedeira e comandos da linguagem **SQL**. Na representação gráfica, os comandos da linguagem hospedeira e os comandos declarativos da **SQL** podem ser acomodados em blocos de comandos, tal como foi definido no Capítulo 2. Os comandos executáveis da linguagem **SQL** (INSERT, DELETE, UPDATE, SELECT, COMMIT, ROLLBACK, entre outros) [ORA90] são acomodados isoladamente. Como notação gráfica adota-se nós circulares para representar os blocos de comandos (linguagem

hospedeira e comandos declarativos da **SQL**) e nós retangulares para representar os comandos executáveis da **SQL**. As setas, denominadas de arcos, representam possíveis transferências de controle entre os nós.

O grafo de programa para representar uma *UP* de uma ABDR é denotado por $\mathbf{G}(UP)=(N^{BD}, E, n_{in}, n_{out})$ onde $N^{BD} = N_h \cup N_S$, onde $N_h$ é o conjunto de nós provenientes da linguagem hospedeira e $N_S$ é o conjunto de nós provenientes da linguagem **SQL**; $E$ é o conjunto de arcos onde $E \subseteq N^{BD} \times N^{BD}$. O nó $n_{in} \in N_h$ é o nó de entrada e $n_{out} \in N^{BD}$ é o nó de saída do grafo de programa. O grafo de programa estabelece uma correspondência entre os comandos do programa representados pelos conjuntos de nós $N_h$ e $N_S$, indicando os possíveis fluxos de controle entre os nós através dos arcos [SPO97]. Os conjuntos $Arc_{in}(i)$ e $Arc_{out}(i)$ representam os conjuntos de arcos que respectivamente chegam e saem do nó $i$. O exemplo da Figura 3.2 mostra o grafo de programa para a $UP_{Selemp}$, extraída do Módulo $Mod_3$ (usado no exemplo de aplicação, descrito no *Apêndice C*). Os arcos *(4,15), (7,15), (10,15) e (13,15)* $\in Arc_{in}(15)$ e os arcos *(3,4) e (3,18)* $\in Arc_{out}(3)$.

Um *caminho* é uma seqüência finita de nós $(n_1, n_2,..,n_k)$, $k \geq 2$, tal que, para todo $i$, $1 \leq i \leq k-1$, existe um arco $(n_i, n_{i+1}) \in E$. Um caminho é *completo* quando o primeiro nó é o nó $n_{in}$ de entrada (*nó 1 no exemplo da Figura 3.2*) e o último nó é o nó $n_{out}$ de saída (*nó 19 no exemplo da Figura 3.2*). Um caminho é *simples* se todos os nós que o compõem, exceto possivelmente o primeiro e o último, são distintos. Se todos os nós são distintos, diz-se que o caminho é um *caminho livre de laço*. O comando declarativo *"EXEC SQL WHENEVER SQLERROR GOTO end16;"* utilizado para tratamento de erro nos comandos **SQL** foi colocado no nó 1 (junto com os comandos da linguagem C) e ele ocasiona a ocorrência dos arcos *(3, 18)*, *(6, 18)*, *(9,18)* e *(12,18)* com a condição de um desvio condicional para o rótulo *"end16"* colocado no nó *18*.

A abordagem de teste estrutural para programas de ABDR considera dois tipos de fluxos de dados: *i)* fluxo de dados *intra-modular*; e *ii)* fluxo de dados *inter-modular*. O fluxo *intra-modular* é usado para o teste estrutural de cada *Módulo de Programa* da aplicação. O fluxo *intra-modular* é usado no teste de unidade e no teste de integração *intra-modular*; este último é efetuado para a integração das *unidades de programa*.

```
int selemp  (line)
RECORD *line;
/*  1 */  {
            FILE * path = fopen("selemp/path.tes","a");
            static int printed_nodes = 0;
/*  1 */    char string[10], strfield[4];
/*  1 */    int  field;
/*  1 */    EXEC SQL BEGIN DECLARE SECTION;
               int     cod_value;
               VARCHAR
                  lname_value    [20],
                  address_value  [40],
                  rg_value       [11];
            EXEC SQL END DECLARE SECTION;
            ponta_de_prova(1);
/*  1 */    EXEC SQL WHENEVER SQLERROR GOTO end16;
/*  1 */    printf(" \n 1-Codigo Empregado  2- Sobrenome
                                              3- Endereco 4-RG ");
/*  1 */    gets (strfield);
/*  1 */    field = atoi (strfield);
/*  1 */    switch(field)
/*  1 */       {
/*  2 */          case 1   :
/*  2 */           {
                  ponta_de_prova(2);
/*  2 */              printf ("\n Digite o codigo do Empregado: ");
/*  2 */              gets (line->e_emp_id);
/*  2 */              cod_value = atoi (line->e_emp_id);
                     ponta_de_prova(3);
/*  3 */              EXEC SQL SELECT  TO_CHAR(emp_id),
                              TO_CHAR(manager_id),
                                      emp_fname,
                                     emp_lname
                              INTO   :he1,
                                     :he2,
                                     :he3,
                                     :he4
                              FROM   EMPLOYEE
                              WHERE  emp_id = :cod_value;
                             ponta_de_prova(4);
/*  4 */                 break;
/*  4 */              }
/*  5 */              case 2     :
/*  5 */              {
                        ponta_de_prova(5);
/*  5 */                  printf ("\n Digite o Sobrenome do Empregado:");
/*  5 */                  gets (line->e_emp_lname);
/*  5 */                  strcpy (lname_value.arr, line->e_emp_lname);
/*  5 */                  lname_value.len = strlen (lname_value.arr);
                        ponta_de_prova(6);
/*  6 */                  EXEC SQL SELECT TO_CHAR(emp_id),
                               TO_CHAR(manager_id),
                                       emp_fname,
                                       emp_lname
                              INTO   :he1,
                                     :he2,
                                     :he3,
                                     :he4
                              FROM   EMPLOYEE
                              WHERE  emp_lname = :lname_value;

/*  7 */                  printf("\n Sem erro: ");
                        ponta_de_prova(7);
/*  7 */                  break;
/*  7 */              }
/*  8 */              case 3 :
/*  8 */              {
                        ponta_de_prova(8);
/*  8 */                  printf ("\n Digite o Endereco do Empregado:");
/*  8 */                  gets (line->e_street);
/*  8 */                  strcpy (address_value.arr, line->e_street);
/*  8 */                  address_value.len = strlen (address_value.arr);
                        ponta_de_prova(9);
/*  9 */                  EXEC SQL SELECT TO_CHAR(emp_id),
                                      emp_fname,
                                     emp_lname
                              INTO   :he1,
                                     :he2,
                                     :he3,
                                     :he4
                              FROM   EMPLOYEE
                              WHERE  street = :address_value;
/* 10 */                  printf("\n Sem erro:");
                        ponta_de_prova(10);
/* 10 */                  break;
/* 10 */          }
/* 11 */           case 4 :
/* 11 */           {
                        ponta_de_prova(11);

/* 11 */                printf ("\n Digite o Numero do RG do
Empregado:");
/* 11 */                gets (line->e_ss_number);
/* 11 */                strcpy(rg_value.arr, line->e_ss_number);
/* 11 */                rg_value.len = strlen (rg_value.arr);
                        ponta_de_prova(12);
/* 12 */                EXEC SQL SELECT TO_CHAR(emp_id),
                                    TO_CHAR(manager_id),
                                    emp_fname,
                                   emp_lname
                                INTO   :he1,
                                       :he2,
                                       :he3,
                                       :he4
                                FROM   EMPLOYEE
                                WHERE  ss_number = :rg_value;

/* 13 */                    printf("\n Sem erro:");

                            ponta_de_prova(13);
/* 13 */                    break;
/* 13 */               }
/* 14 */              default:
/* 14 */              {
                       ponta_de_prova(14);
/* 14 */                 sqlca.sqlcode=-1;
/* 14 */              }
/* 15 */            } /* switch */
                    ponta_de_prova(15);
/* 15 */            printf ("code = %d\n", sqlca.sqlcode);
/* 15 */            if(sqlca.sqlcode == 0)
/* 16 */            {
                       ponta_de_prova(16);
/* 16 */                if(sqlca.sqlerrd[2] == 1)
/* 17 */                {
                           ponta_de_prova(17);
/* 17 */                    itoa(he1, line->e_emp_id);
/* 17 */                    itoa(he2, line->e_manager_id);
/* 17 */                    he3.arr [he3.len] = '\0';
/* 17 */                    strcpy (line->e_emp_fname, he3.arr);
/* 17 */                    he4.arr [he4.len] = '\0';
/* 17 */                    strcpy (line->e_emp_lname, he4.arr);
/* 17 */                }
/* 18 */            }
/* 18 */ end16:
                    ponta_de_prova(18);
                    ponta_de_prova(19);
                    fclose(path);
/* 18 */            return (sqlca.sqlcode);
/* 19 */        }
```

***G*** (*Selemp*)

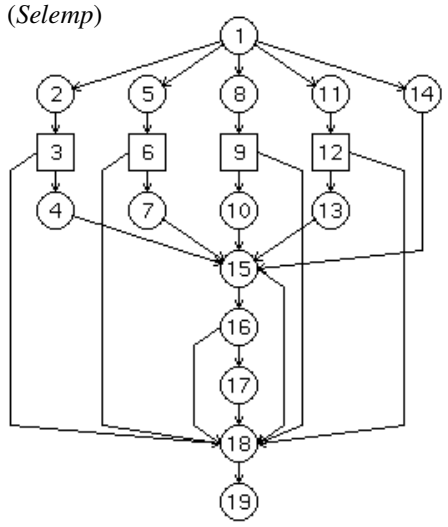



*Figura 3.2: Grafo de Programa da Unidade de Programa Selemp do Módulo de Programa $Mod_3$.*

O fluxo *inter-modular* é utilizado para o teste estrutural considerando o fluxo de dados entre os Módulos de Programas que compõem a ABDR. Esse fluxo de dados envolve as *variáveis tabela* da base de dados. O teste baseado neste tipo de fluxo de dados foi

denominado de teste de integração *inter-modular*, efetuado para integrar os diferentes Módulos de Programas existentes na aplicação.

Qualquer critério de teste estrutural apresentado no Capítulo 2 pode ser aplicado no teste de unidade dos programas de aplicação. Em particular, neste trabalho utiliza-se como exemplos os critérios da FCPU. O teste de integração *intra-modular* compõe-se de duas abordagens distintas: teste de integração baseado no grafo de chamada e teste de integração baseado no fluxo de dados das *variáveis tabela* da base de dados. As definições dos critérios de teste baseados nos fluxos *intra-modular* e *inter-modular* são apresentadas no Capítulo 4.

A seguir são apresentados os conceitos básicos de teste estrutural para programas de aplicação e a terminologia adotada neste trabalho.

### 3.1.1 Teste de unidade

Numa aplicação de Banco de Dados Relacional são usados três tipos de variáveis: *variáveis de programa,* $\boldsymbol{P = \{p_1,p_2,..,p_m\}}$, (variáveis da linguagem hospedeira); variáveis de ligação ou *variáveis host,* $\boldsymbol{H = \{h_1, h_2, ..,h_n\}}$, (que guardam os valores da base de dados); e as *variáveis tabela,* $\boldsymbol{T = \{t_1, t_2, ..,t_k\}}$, que são as tabelas da base de dados relacional manipuladas pelo programa. As *variáveis tabela de visão* são extraídas das *variáveis tabela* da base de dados da aplicação. A ocorrência de uma *variável host* ou de uma *variável de programa* pode ser: uma *definição* de variável, um *uso* de variável ou uma *indefinição*. As *variáveis tabela* são variáveis globais para todos os Módulos de Programas $Mod_i$ de uma ABDR.

As *variáveis tabela* são criadas pelo comando CREATE TABLE <*tabela*> e essa criação é feita uma única vez durante o projeto do banco de dados. A partir de sua criação, o SGBD passa a controlar os acessos às tabelas da ABDR, permitindo que seus usuários possam modificá-las e/ou atualizá-las a partir dos programas da aplicação. Assim, as *variáveis tabela* não são declaradas como as *variáveis host* e *variáveis de programa* nos programas de aplicação. Desse modo, adota-se a ocorrência de uma *variável tabela* ***t*** como sendo *definição* ou *uso* e considera-se que toda *variável tabela* referenciada por algum programa implica a ocorrência de uma definição anterior (até por outro módulo de programa). Não há, portanto, nenhuma exigência sintática em declará-la antes de uma

definição ou antes de um uso; para as demais variáveis pode ocorrer um erro de compilação ou uma "*anomalia*" [ORA90, ELM94].

Uma variável é considerada *definida* quando um valor é armazenado em uma posição de memória. No caso das *variáveis de programa* e *variáveis host*, quando estiver nos comandos na linguagem hospedeira, uma ocorrência de variável é uma *definição* se ela estiver:

*i)* no lado esquerdo de um comando de atribuição[8];

*ii)* em um comando de entrada; ou

*iii)* em chamadas de procedimento como parâmetro de saída.

Nos comandos da **SQL**, uma ocorrência das *variáveis host* é uma definição se ela estiver:

*iv)* dentro da cláusula INTO de um comando SELECT ou FETCH.

Uma *variável tabela* é considerada definida quando um valor é armazenado em memória secundária modificando assim o estado da *variável tabela* de $e_0$ para $e_1$. Uma ocorrência de *variável tabela* em um programa é uma *definição* se ela estiver:

I. em uma cláusula INTO do comando INSERT;

II. em uma cláusula FROM do comando DELETE;

III. do lado esquerdo da cláusula SET do comando UPDATE; e

IV. no caso de uma visão, quando estiver no lado direito da cláusula CREATE VIEW.

Apesar de esses comandos da **SQL** caracterizarem a ocorrência de definição da *variável tabela*, as ocorrências I, II e III só são efetivadas quando são executadas juntas com o comando COMMIT, quando o valor é armazenado em memória secundária. Para distingui-la das demais definições (em memória interna), denomina-se de *definição persistente* de ***t*** sempre que ocorre uma alteração em seu estado físico (em memória secundária). Isto só é possível se houver um comando de manipulação da **SQL** seguido do comando COMMIT.

[8] Em caso de variáveis do tipo *array*, *x[i]=0*, implicam na definição apenas da variável *x* e não do índice *i*. O mesmo ocorre com variáveis do tipo registro. No caso de C++ e Java, que permitem comandos do tipo *y*++ que implica no uso de *y* seguido de uma definição de *y* e outros comandos que são tratados de maneira especial.

Por exemplo, na *Figura 3.3* existe a definição da variável DEP somente se a seqüência <2, 4> faz parte do caminho. Assim, para o caminho (1, 2, 5, 6) não existe a definição persistente de DEP. Como conseqüência, para estabelecer uma associação definição-uso através de uma *definição persistente* requer-se que os dois nós que caracterizam a definição persistente façam parte do caminho; para isso adotou-se a notação $<\ell,i>$ para denotar o par de nós que caracteriza a *definição persistente*. Caso contrário, dizemos que o nó $\ell$ possui a *definição por referência* (a variável foi referenciada) e o nó $i$ possui a *definição por valor* (um valor foi atribuído a ela) da *variável tabela*.

Em geral, o comando COMMIT deveria seguir cada um dos comandos que caracterizam uma definição da *variável tabela* durante a manipulação de dados (INSERT, DELETE, UPDATE), mas não podemos garantir que isso sempre acontece. No caso de não existir pode-se adotar a abordagem de associar ao nó de saída do grafo ($n_{out}$) o comando COMMIT esperado. No Exemplo da Figura 3.3 seria o nó 6.

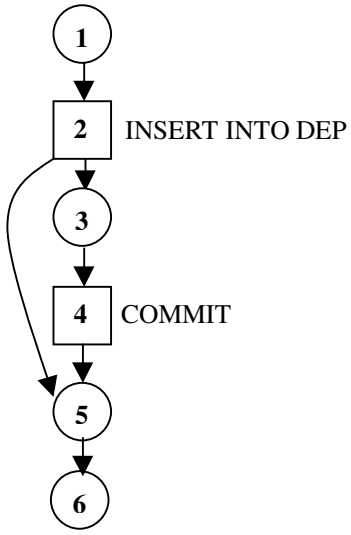



*Figura 3.3 - Exemplo de um grafo de programa com **SQL** embutida.*

Devido à existência de tratamento de erro, é comum a existência de dois arcos de saída dos nós da **SQL** que contêm um comando de execução. No caso de tratamento de erros, dizemos que uma *definição* de ***t*** em um nó $i \in N_S$ só será efetivada, na prática, quando a execução tomar o caminho que incluir o arco de saída pertencente a $ARC_{out}(i)$, cujo

predicado contém um erro nulo (*SQLERROR=0*) e que leva à execução do comando COMMIT.

A ocorrência de uma variável é um uso quando a referência a essa variável não a estiver definindo, ou seja, há a recuperação de um valor em uma posição de memória associada a esta variável [NTA84]. Em geral, uma ocorrência de *variável de programa* ou *variável host* nos comandos da linguagem hospedeira é um uso se ela estiver:

*i)* no lado direito de um comando de atribuição;

*ii)* em um comando de saída;

*iii)* em chamada de procedimento como parâmetro de entrada;

*iv)* em um predicado de um comando condicional.

Para uma variável *host,* nos comandos da **SQL**, uma ocorrência da variável é um uso se ela estiver:

*v)* na condição da cláusula WHERE dos comandos SELECT, DELETE e UPDATE;

*vi)* do lado direito do comando de atribuição da cláusula SET do comando UPDATE;

*vii)* na cláusula VALUE do comando INSERT.

Uma ocorrência de uma *variável tabela* é um *uso* quando ela estiver:

I. na cláusula FROM dos comandos SELECT e DELETE;

II. em uma cláusula INTO do comando INSERT;

III. do lado esquerdo da cláusula SET do comando UPDATE;

IV. na cláusula WHERE dos comandos executáveis DELETE, UPDATE e SELECT.

Quando se tratar de uma *variável tabela de visão*, a ocorrência será um uso quando estiver:

V. no operando do comando DROP VIEW.

Para distinguir os tipos de usos definidos na Seção 2.1, são introduzidos os seguintes conceitos: *a)* Uma referência a uma variável ***v*** é um *s-uso* (uso **SQL**) se ***v*** afeta o resultado de uma consulta ou de uma computação em um comando executável da **SQL** (este uso está associado ao nó da **SQL**); *b)* O uso de uma *variável tabela* ***t*** (*persistente*) ocorre quando estiver em algum nó $j \in N_S$ com um dos comandos executáveis da **SQL** que

caracterizam o uso da variável ***t*** (SELECT, INSERT, UPDATE ou DELETE), o *uso persistente* será associado aos arcos $ARC_{out}(j)$ e será denominado de ***t*-uso**.

O *s-uso* ocorre dentro de algum nó $n \in N_S$ e pode influenciar o fluxo de controle nos arcos de saída (***t*-uso**) dos nós pertencentes ao conjunto de $N_S$ quando houver tratamento de erros; um *s-uso* pode envolver as *variáveis host* e *variáveis tabela* e o ***t*-uso** envolve apenas o *uso persistente* da *variável tabela* ***t***.

O tratamento da definição e uso de vetores, registros e apontares merecem uma atenção especial. Em geral, diz-se que um tratamento proposto é conservador [MAL91] se ele não deixa de requerer nenhum fluxo de dados que possa existir; ou seja, a seleção de caminhos e associações é a mais rigorosa dentre as possíveis.

No programa de aplicação, um caminho $(i, n_1, n_2, ..., n_m, j)$ para $m \geq 0$, com definição da variável ***v*** no nó $i$ e que não contém nenhuma redefinição de ***v*** nos nós $n_1, n_2, ..., n_m$ é chamado de *caminho livre de definição* com respeito a (c.r.a) ***v*** do nó $i$ ao nó $j$ e do nó $i$ ao arco $(n_m, j)$. Essa definição é válida para: todos os nós em $N^{BD}$, todos os arcos de E e todas as variáveis de uma ABDR.

Um nó $i \in N^{BD}$ possui uma definição global de uma variável ***v*** se toda vez que ocorre uma definição de ***v*** no nó $i$ existe um caminho *livre de definição* de $i$ para algum nó ou para algum arco que contém um uso c.r.a. ***v***. Um *c-uso* da variável ***v*** em um nó $j$ é um *c-uso global* se não existir uma definição de ***v*** no nó $j$ precedendo este *c-uso*; caso contrário, é um *c-uso local*. Geralmente, todo *s-uso* é um *s-uso global*.

Dizemos que o caminho $(n_\ell,...,n_i, ..., n_j, n_k)$ é um caminho *livre de definição persistente* de $<n_\ell, n_i>$ até o nó $k$, se não existe outro par de nós $<n_q, n_m>$ onde ocorre uma *redefinição persistente* de ***t*** em $<n_q, n_m>$ do nó $n_i$ até o nó $n_k$ onde o par $<n_\ell, n_i>$ antecede o par $<n_q, n_m>$. Lembrando que na ausência de um comando COMMIT, após um comando executável da **SQL** até o último nó de um grafo, adota-se que o COMMIT estará no último nó do grafo.

Rapps e Weyuker [RAP85] definem que os comandos podem possuir sucessores e antecessores físicos (seqüência de escrita) ou de execução (fluxo de execução). Um comando é sintaticamente alcançável se existe uma seqüência de comandos $s_1, ..., s_n$ tal que

$s_1$ é o comando de partida e $s_n$ é o comando de chegada e cada $s_i$ precede a execução de $s_{i+1}$, para $i = 1, ..., n-1$. Requer-se, portanto, que cada comando no programa seja sintaticamente alcançável.

A partir dos conjuntos definidos por Rapps e Weyuker [RAP82, RAP85], apresentados na Seção 2.1, são definidos os conjuntos a seguir, utilizados para descrever os conceitos básicos dos critérios de testes para programas de aplicação:

*(i)* *s-uso*(i)={variáveis com *s*-uso no nó $i \in N_s$};

*(ii)* $defT<\ell,i>$ = {Variável ***t*** | ***t*** possui uma *definição persistente* pela *concatenação* dos nós $\ell$ e *i, representado por* $<\ell, i>$ onde $\ell$ e $i \in N_S$ .}

*(iii)* *dsu(v,i)*={nós $j \in N_S$ | ***v*** $\in$ *s-uso(j)*, existe um caminho livre de definição c.r.a. ***v*** do nó *i* para o nó *j* e $v \in defg(i)$ (onde *defg(i)* representa o conjunto de todas variáveis com definição global no nó *i*)}.

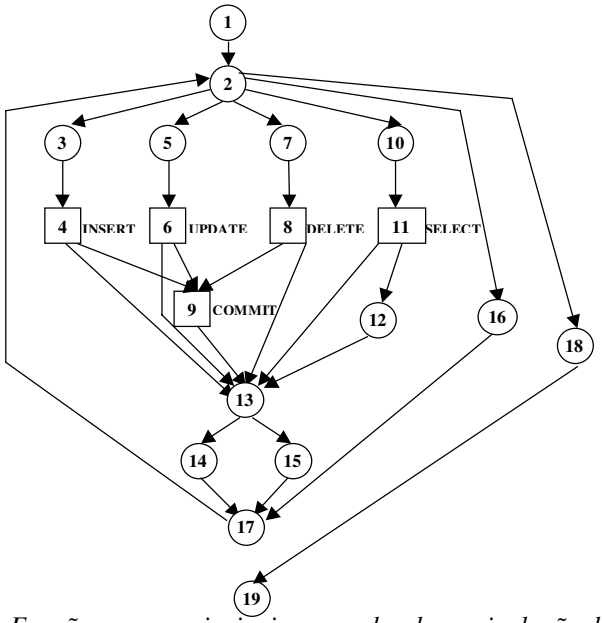



*Figura 3.4: Função com os principais comandos de manipulação da **SQL** para uma tabela **t**.*

Em *(ii)* observa-se que existe a *definição persistente* em *variáveis tabela* ***t*** em um caminho se e somente se ambos os nós $\ell$ e $i \in N_S$ pertencem ao caminho. No exemplo da *Figura 3.4* o caminho (1, 2, 3, 4, 13, 14, 17, 2, 7, 8) não contém definição persistente da variável tabela ***t*** visto que, apesar de o nó 4 conter o comando INSERT, o comando

COMMIT não faz parte do caminho. Somente os caminhos contendo os pares de nós <4, 9>, <6, 9> ou <8, 9> contêm *definição persistente* da variável ***t*** (conjuntos *defT<4,9>, defT<6,9>, defT<8,9>*). Em geral, o nó de saída de um grafo não pertence ao conjunto $N_S$. Na ausência sintática do comando COMMIT em um procedimento que contém um comando de manipulação da **SQL** (INSERT, DELETE, UPDATE), considera-se que o comando COMMIT ocorre no nó de saída do grafo (por "*default*"). Se o grafo da Figura 3.4 não tivesse um nó com o comando COMMIT, as *definições persistentes* de ***t*** seriam, por exemplo: *defT<4,19> e defT<6,19> e defT<8,19>*. Para efeito de implementação dos critérios da FCPU para o teste de unidade foi considerado que existe uma definição de ***t*** (na memória interna) nos nós que contêm um comando de manipulação da **SQL** (INSERT, UPDADE ou DELETE) (nós 4, 6 e 8 da Figura 3.4) e que existe uma definição persistente de ***t*** no nó da **SQL** que contém o comando COMMIT (no grafo da Figura 3.4 é o nó *9*).

Para os programas de aplicação, um caminho $(n_1, n_2, ..., n_j, n_k)$ é um *du-caminho* c.r.a ***v*** se $n_1$ tiver uma definição global de ***v*** e: *i*) $n_k$ tem um *c*-uso ou *s*-uso de ***v*** e $(n_1, n_2, ...,n_j, n_k)$ é um caminho simples livre de definição c.r.a ***v***; ou ii) o arco $(n_j, n_k)$ tem um *p*-uso de ***v*** e $(n_1, n_2, ...,n_j, n_k)$ é um caminho livre de definição c.r.a ***v*** e $n_1, n_2, ...,n_j$ é um caminho livre de laço. Uma *associação definição-c-uso* é uma tripla [*i, j,* ***v***] onde *i* é um nó que contém uma definição global de ***v*** e $j \in dcu(v,i)$. Uma associação *definição-p-uso* é uma tripla [*i,(j,k),v*] onde *i* é um nó que contém uma definição global de ***v*** e o arco $(j, k) \in dpu(v, i)$. Uma associação é uma *associação-c-uso*, uma *associação-p-uso* ou um *du-caminho*.

As definições apresentadas na Seção 2.1.3 (*teste de unidade*) podem ser estendidas para programas de aplicação no que se refere às *variáveis host* e *variáveis tabela.* Considere-se que toda *variável host* e *variável tabela* pode ter uma definição global em um nó *i*, isto é, *defg(i)* = {variável ***v*** | ***v*** é uma variável do conjunto de todas as variáveis usadas em programas de aplicação}.

Considerando que as *variáveis host* e *variáveis tabela* possuem uma definição global no nó *i*, os critérios da FCPU ou da FCFD podem ser aplicados no teste de unidade. As seguintes definições complementam os conceitos básicos para a definição dos critérios de teste de programas de ABDR:

*(a) Associação definição-s-uso* é uma tripla [*i, j,* ***v***] onde ***v*** ∈ *defg(i)* e *j* ∈ *dsu(v,i)*;

*(b) Associação definição-t-uso* é uma tripla *[<ℓ,i>, (j,k),* ***t***] onde ***t*** ∈ *defT<ℓ,i>*, *j* ∈ *dsu(**t**,i)*, o arco *(j,k)* é o arco de saída do nó j.

*(c) dtu-caminho* é um caminho livre de *definição persistente* ($n_\ell$, ..., $n_i$, ..., $n_j$, $n_k$) c.r.a. ***t*** dos nós $<n_\ell , n_i>$ até o nó $n_k$ ou até o arco ($n_j$, $n_k$ ) onde ocorre um uso de ***t*** e o caminho ($n_\ell$, ..., $n_i$, ..., $n_j$, $n_k$), onde $n_i < n_j$ e $n_i < n_k$ é um caminho livre de laço e no par de nós $<n_\ell, n_i>$ ocorre uma *definição persistente* de ***t***.

Um *dtu-caminho* é um *du-caminho* com relação a *variável tabela* no enfoque de variável persistente. Uma associação pode ser uma associação *definição-c-uso*, uma associação *definição-p-uso*, uma associação *definição-s-uso*, uma associação *definição-t-uso*, um *du-caminho* ou um *dtu-caminho*. Toda associação é uma potencial associação no enfoque de Potencial Uso [MAL91].

***Observação***: o conceito de potencial associação, além de possibilitar a identificação de defeitos devido ao uso incorreto de variáveis, inclui também a possibilidade de identificar defeitos devido à ausência do uso, que também é um defeito [MAL91]. Partindo dessa premissa, as *variáveis tabela* jamais irão requerer associações com os nós pertencentes ao conjunto de nós $N_h$, isto é, nós da linguagem hospedeira, pelo fato de ser impossível o uso de uma tabela nestes nós. Essa situação é verdadeira tendo em vista que tanto a definição como o uso de uma *variável tabela* só é possível ocorrer em um comando executável da **SQL**, conforme foi especificado no início desta seção.

O grafo de fluxo de dados definido por Maldonado [MAL91] não necessita referenciar os pontos onde há ocorrência de um *uso* de uma variável, mas apenas os pontos onde existe uma definição global de uma variável; daí a denominação de *grafo-def*[9]. O *grafo-def* pode ser facilmente adaptado para: *1)* programas de aplicação, acrescentando as informações específicas de comandos de **SQL** e observando as operações de controle do SGBDR (responsáveis pelos fluxos após os comandos **SQL**) e, *2)* os três tipos de variáveis existentes nos programas de aplicação (programa, host e tabela) para o teste de unidade.

[9] Os autores [RAP82, NTA84, RAP85] definem o grafo de fluxo de dados como grafo *def-uso*, que contém informações de ocorrências de c-uso, p-uso e definições.

Desta forma, para cada nó *i* tal que *defg(i)* $\neq \phi$ é associado um grafo denominado *grafo(i)* que contém todos os *potenciais-du-caminhos* com início no nó *i*. Cada nó *k* do *grafo(i)* é completamente identificado pelo número do nó do grafo de programa que lhe deu origem e pelo conjunto *deff*(*k*) – o conjunto de variáveis definidas no nó *i*, mas não redefinidas num caminho do nó *i* para o nó *k* [MAL91]. Maldonado [MAL88b] propôs um algoritmo para a obtenção desses grafos. A partir desses grafos e do conceito de arco primitivo apresentado em [MAL91], estabelece-se a base para o Modelo de Descrição dos Elementos Requeridos para os critérios da FCPU [MAL91, CHA91]. Adotou-se a FCPU para exemplificar o teste de unidade neste trabalho por causa da disponibilidade da ferramenta POKE-TOOL que apóia a aplicação desses critérios.

Uma associação é denotada por *[i, (j,k),{$v_1$, ...,$v_n$}]*, onde *i* é o nó onde ocorre a definição das variáveis $v_1$, ...,$v_n$ e existe um caminho livre de definição c.r.a $v_1$, ...,$v_n$ do nó *i* ao arco *(j,k)*. Cada *grafo(i)* contém todos os caminhos livres de definição c.r.a as variáveis $\boldsymbol{v}$ definidas em *i* que podem satisfazer a potencial associação do nó *i* com os demais arcos alcançáveis com caminhos livres de definição c.r.a $\boldsymbol{v}$. As variáveis $\boldsymbol{v}$ representam o conjunto de todas as variáveis $\boldsymbol{P} \cup \boldsymbol{H}$ de uma aplicação de BDR. As definições *defg(i)* (definição lógica) e *defT*$<\ell,i>$ (definição física) são tratadas de maneira distinta nos critérios de teste propostos nesta tese. O conjunto *defg(i)* é usado apenas no teste de unidade e no teste de integração baseado no grafo de chamada. O conjunto *defT*$<\ell,i>$ pode ser usado em todos os tipos de testes estruturais propostos nesta tese, mas será dada ênfase para o seu uso no teste de integração baseado na dependência de dados (*intra-modular* e *inter-modular*).

A *cobertura* de uma associação por um caminho completo $\pi$ ocorre se $\pi$ incluir um caminho livre de definição c.r.a $\boldsymbol{v}$ do nó *i* (onde ocorre a definição de $\boldsymbol{v}$) ao nó *j* ou arco (*j,k*) onde se estabeleceu a associação. Um conjunto de caminhos completos $\Pi$ cobre uma associação se algum dos elementos do conjunto o fizer. Essa definição é válida para todos os critérios de teste estrutural baseados em análise de fluxo de dados.

A partir do conceito de caminho executável estabelecido em [FRA88] pode-se dizer que uma associação é executável se existir um caminho completo executável que cubra essa associação; caso contrário, ela não é executável.

Definimos como conjunto factível:

*(d)* $fdsu(v, i) = \{j \in dsu(v,i) \mid$ a associação $[i, j, v]$ onde $j \in N_S$, é executável$\}$;

Observe que nas definições apresentadas na Seção 2.1.3 (*teste de unidade*), o nó *j* pertence ao conjunto $N_h$ onde $N_S \cup N_h = N^{BD}$.

Alguns conceitos vistos em [NTA84, RAP85, FRA88, MAL91, CHA91, HAR91] foram aproveitados e adaptados para a proposta de teste estrutural de programas de aplicação apresentados nesta tese. Ao teste de unidade pode ser aplicado qualquer critério de teste estrutural, ou seja, o teste de programas de ABDR pode ser suportado por ferramentas, adequadamente modificadas, que implementem a utilização desses critérios.

A ferramenta de teste POKE-TOOL e a ferramenta de visualização ViewGraph *v.2.0* foram utilizadas na realização de um exemplo de aplicação para ilustração de nossa proposta. A ferramenta POKE-TOOL, que apóia o Teste de Unidade para programas convencionais, foi modificada para o teste dos programas de aplicação. A ferramenta *ViewGraph* (*v.2.0*) [VIL94, VIL97, CRU99] permite a visualização gráfica e de informações dos resultados obtidos do teste de cada *UP* dos Módulos de Programas de uma ABDR.

### 3.1.2 Teste de Integração

O teste de integração é aplicado depois que todas as unidades que compõem um programa foram devidamente testadas isoladamente (*teste de unidade*). O teste de integração pode ser visto como uma técnica sistemática para a construção da estrutura do software, procurando revelar defeitos associados à interação entre as *Unidades do Programa* (*UP*) ou entre os *Módulos de programa*s (*Mod*).

A dependência de dados é um aspecto fundamental associado à interação entre Unidades de Programas e Módulos de Programas. As dependências de dados existentes entre as *UPs* de programas convencionais são ocasionadas pelas variáveis globais ou por variáveis passadas por parâmetros através de comandos de chamadas. Os programas de aplicação possuem dependências de dados baseadas nos comandos de chamada, como acontece nos programas convencionais, e dependências de dados baseadas nas tabelas da base de dados. As dependências de dados entre as *UPs* e os Módulos de Programas de uma ABDR dão ensejo a novas abordagens de integração, descritas na Seção 3.1.2.2.

A seguir são apresentados os conceitos básicos do teste de integração baseado nas chamadas de procedimentos em programas de aplicação.

### 3.1.2.1 Teste de integração baseado no grafo de chamadas

Algumas características do teste de unidade podem ser aproveitadas e adaptadas para o teste de integração. Assim como no teste de unidade, também podemos utilizar qualquer critério de teste de integração estrutural para exercitar as variáveis ***v*** existentes nos programas de aplicação que tenham uma definição global em algum nó do grafo da unidade chamadora $\eta_C$ e que alcancem um nó que contém um comando de chamada para a unidade $\eta_R$, por um caminho livre de definição c.r.a. ***v***; pode existir uma associação com relação a qualquer ponto da unidade chamada.

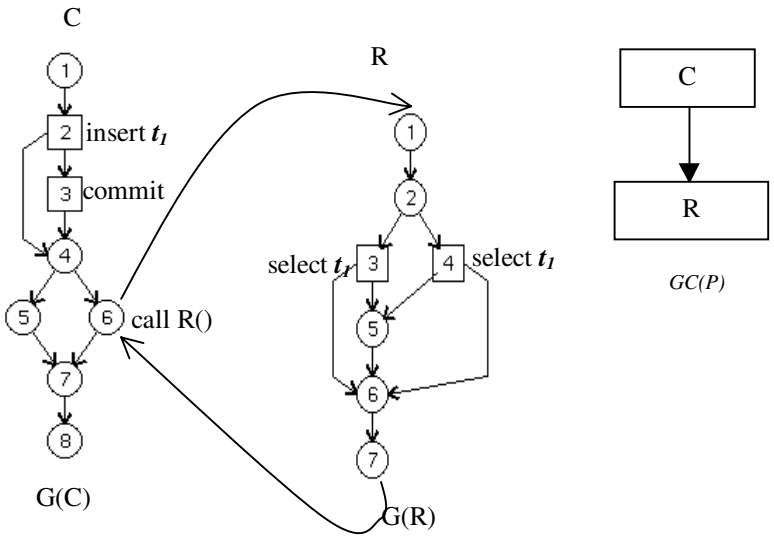



*Figura 3.5: Exemplo de um grafo de chamada entre duas UPs (C e R)*

Para as definições de Vilela [VIL98], a Figura 3.5 mostra duas unidades de programa que são relacionadas por um ponto de chamada e apresenta um grafo de chamada contendo apenas dois nós (*C* e *R*).

O conjunto ${}_iPath(C,R)$, definido na Seção 2.1.3, contém os seguintes caminhos:

${}_i\boldsymbol{P_1}$ = ($1_C$, $2_C$, $3_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $3_R$, $5_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

${}_i\boldsymbol{P_2}$ = ($1_C$, $2_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $3_R$, $5_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

${}_i\boldsymbol{P_3}$ = ($1_C$, $2_C$, $3_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $3_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

${}_i\boldsymbol{P_4}$ = ($1_C$, $2_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $3_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

$_i\boldsymbol{P_5}$ = ($1_C$, $2_C$, $3_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $4_R$, $5_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

$_i\boldsymbol{P_6}$ = ($1_C$, $2_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $4_R$, $5_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

$_i\boldsymbol{P_7}$ = ($1_C$, $2_C$, $3_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $4_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

$_i\boldsymbol{P_8}$ = ($1_C$, $2_C$, $4_C$ , $6_C$), ($1_R$, $2_R$, $4_R$, $6_R$, $7_R$), ($7_C$, $8_C$);

Vários conceitos tratados em [CLA89, VIL98], destacando-se a das associações definição-uso, aplicadas para variáveis globais e variáveis passadas por parâmetros, podem ser adaptados para programas de aplicação. As variáveis de interesse particular de um programa de aplicação (*variáveis host* e as *variáveis tabela*) ocorrem nos nós que contêm comandos da **SQL**. As *variáveis tabela* são globais para todos os Módulos de Programa, enquanto as *variáveis host* e *variáveis de programa* são locais ou globais dependendo do escopo em que são declaradas. O conjunto de variáveis usadas como entrada ou saída de uma chamada $\boldsymbol{c}$ da unidade de programa $UP_1$ para $UP_2$, denotado $in_c$, é o conjunto de parâmetros reais, ou variáveis globais, usados como entrada na chamada $\boldsymbol{c}$ da $UP_1$; $out_c$ é o conjunto de parâmetros reais ou variáveis globais usadas como saída na chamada $\boldsymbol{c}$ da $UP_1$ à $UP_2$; o conjunto de todas as variáveis, $in_c \cup out_c$, é denominado de variáveis de comunicação [VIL98] entre as *unidades de programas*. Vilela [VIL98] define que um parâmetro formal correspondente a um parâmetro real $x$ de uma chamada $c$ é denotado por $formal_c(x)$; no caso de uma variável global $x$, $formal_c(x)$ mapeia a variável $x$ para ela mesma; $formal_c(x) = x$. Cabe lembrar que não vamos tratar das variáveis aliases (ponteiros).

Visando a englobar todas as variáveis de um Programa de ABDR, estendemos as notações tratadas em Vilela [VIL98], acrescentando:

i) *associação-definição-s-uso interprocedimental de chamada*: é uma quádrupla [*i, j, v, c*] onde $c$ é uma chamada de uma unidade $UP_1$ para outra $UP_2$; $\boldsymbol{v} \subseteq in_c$; $i \in N^{BD}$ é um nó de $UP_1$ que contém definições globais de variáveis de $\boldsymbol{v}$; $j \in N_S$ é um nó de $UP_2$ que contém um *s-uso* global de $formal_c(\boldsymbol{v})$ e existe um sub-caminho livre de definição c.r.a $\boldsymbol{v}$ de $i$ para $n_c$ e c.r.a $formal_c(\boldsymbol{v})$ de $n_{in}(UP_2)$ para $j$.

ii) *associação-definição-s-uso interprocedimental de retorno*: é uma quádrupla [*i, j, v, c*] onde $c$ é uma chamada de uma unidade $UP_1$ para outra $UP_2$; $\boldsymbol{v} \subseteq out_c$; $i \in N^{BD}$ é um nó de $UP_2$ que contém uma definição global de $formal_c$

(***v***); $j \in N_S$ é um nó de $UP_1$ que contém um *s-uso* global de ***v*** e existe um sub-caminho livre de definição c.r.a $formal_c(\boldsymbol{v})$ de $i$ para $n_{out}(UP_2)$ e c.r.a ***v*** de $n_c$ para $j$ em $UP_1$.

iii) *associação-definição-t-uso interprocedimental de chamada*: é uma *quádrupla* $[<\ell, i>, (j, k), \boldsymbol{t}, c]$ onde $c$ é uma chamada de uma unidade de programa $UP_1$ para a $UP_2$ no nó $n_c$ do $G(UP_1)$; $\boldsymbol{t} \subseteq in_c$; $\ell$ e $i \in N_S$ são nós de $G(UP_1)$ que contêm uma definição persistente de ***t***, na concatenação $<\ell,i>$; $(j, k)$ é um arco de saída do nó $j \in N_S$ em $UP_2$ que contém um *uso* de ***t*** (*t-uso*) e contém um sub-caminho livre de definição *persistente* c.r.a ***t*** de $i$ para $n_c$ e c.r.a ***t*** de $n_{in}(UP_2)$ até $(j, k)$.

iv) *associação-definição-t-uso interprocedimental de retorno*: é uma *quádrupla* $[<\ell, i>, (j, k), \boldsymbol{t}, c]$ onde $c$ é uma chamada de uma unidade de programa $UP_1$ para outra $UP_2$ no nó $n_c$ de $G(UP_1)$; $\boldsymbol{t} \subseteq out_c$; $\ell$ e $i \in N_S$ são nós de $G(UP_2)$ que contém uma definição persistente de ***t*** em $<\ell, i>$; $j \in N_S$ e é um nó de $G(UP_1)$ que contém um *t-uso* e contém um sub-caminho livre de definição *persistente* c.r.a ***t*** de $i$ para $out_c$ e c.r.a ***t*** de $n_c$ até $j$.

As associações *i*) e *ii)* tratam das *variáveis host* que podem ser exercitadas por qualquer critério estrutural de teste de integração [CLA89, VIL98], por serem variáveis que podem ser definidas e usadas em quaisquer nós dos grafos *G(UP1)* e *G(UP2)* quando da chamada de uma Unidade de Programa $UP_1$ por outra $UP_2$. O mesmo não ocorre com as *variáveis tabela* que possuem um tratamento exclusivo e diferente das demais variáveis (*host* e *programa*).

Uma *associação-definição-t-uso* interprocedimental baseada no grafo de chamadas é formada pela *quíntupla* $[UP_A, <\ell, i>, UP_B, (j, k), \{ \boldsymbol{t} \}]$, onde $<\ell, i>$ representa o par de nós de $N_S$ que caracterizam uma definição persistente de ***t*** e são nós da $UP_A$, isto é, $\boldsymbol{t} \in defT<\ell,i>$ e $(j, k)$ é um arco da $UP_B$ cujo nó $j$ contém um uso de ***t*** e o arco $(j,k) \in Arc_{out}(j)$. O exemplo mostrado na *Figura 3.6a* contém as seguintes associações *def-t-uso* inter-procedimental ou *inter-unidade*:

- <*Insdep,<2,3>, Seldep,(3,4), {DEPARTMENT}*>;
- <*Insdep,<2,3>, Seldep,(3,16),{DEPARTMENT}*>;
- <*Insdep,<2,3>, Seldep,(6,7),{DEPARTMENT}*>;
- <*Insdep,<2,3>, Seldep,(6,16),{DEPARTMENT}*>;
- <*Insdep,<2,3>, Seldep,(9,10),{DEPARTMENT}*>;
- <*Insdep,<2,3>, Seldep,(9,16),{DEPARTMENT}*>;

O caminho de integração interprocedimental $_iPath(A, B)$ é composto pela composição dos caminhos completos $\mathbf{p_A}$ e $\mathbf{p_B}$, onde $\mathbf{p_A}$ é o caminho completo da unidade $\eta_A$ que contém o ponto de definição persistente de ***t*** e o caminho $\mathbf{p_B}$ é o caminho completo da unidade $\eta_B$ que contém o ponto onde ocorre o uso de ***t*** a partir da *definição persistente* em $\mathbf{p_A}$. No exemplo dado na Figura 3.6a temos:

$iPath(Insdep, Seldep) = \{(1, 2, 3, 4, 5)_{\text{Insdep}} \,\|\, (1, 2, 3, 4, 12, ..., 17)_{\text{Seldep}}, (1, 2, 3, 4, 5)_{\text{Insdep}} \,\|\, (1, 2, 3, 16, 17)_{\text{Seldep}}, ..., (1, 2, 3, 4, 5)_{\text{Insdep}} \,\|\, (1, 2, 9, 16, 17)_{\text{Seldep}} \}$ onde $(1, 2, ...,n)_A \,\|\, (1,2, ...,m)_B$ representa um caminho de integração entre as unidades $UP_A$ e $UP_B$ , ou seja, "‖" é a concatenação do caminho completo percorrido na $UP_A$ com o caminho completo percorrido na $UP_B$. ***Observação:*** o caminho (*1, 2, 3, 4, 12, ..., 17*) implica que qualquer trajeto de 12 a 17 satisfaz a associação.

Uma associação inter-unidade <***A***,<$\ell$, *i*>,***B***,*(j,k),{**t**}*> será satisfeita se e somente se existe um caminho $iPath(A, B) = \boldsymbol{p_A} \,\|\, \boldsymbol{p_B}$ tal que $\boldsymbol{p_A}$ contém a definição persistente de ***t*** em <$\ell$, *i*>, $\boldsymbol{p_B}$ contém um uso de ***t*** no arco *(j, k)* e existe um caminho livre de definição persistente de <$\ell$, *i*> até $n_{out}$ de $UP_A$ e de $n_{in}$ de $UP_B$ até cada arco $\in ARC_{out}(j)$ de $UP_B$. Desta forma, será mapeado o conjunto de associações definição-t-uso para cada variável ***t*** envolvida na integração a partir do grafo de chamada do Módulo de Programa.

Uma observação é feita neste caso quando ocorrer a ausência do comando COMMIT em uma Unidade $UP_1$ onde contem um comando de SQL que define a *variável tabela* ***t***, e existe um nó que contem um comando de chamada, o comando COMMIT será considerado neste nó, visando assim possibilitar a ocorrência de uma *associação definição-t-uso interprocedimental de chamada*, o mesmo ocorre no retorno para o último nó do grafo.

### 3.1.2.2 Teste de integração baseado na dependência de dados

A principal característica de programas de aplicação, que os distingue dos programas convencionais é a *persistência* dos dados da base de dados. As unidades de programas e os módulos de programa, se autorizados, podem ter acesso a esses dados a qualquer momento. A *persistência* é uma propriedade que não depende exclusivamente de Banco de Dados Relacional (BDR), podendo ser estendido para outros programas com as mesmas características.

No caso de BDR, a mesma tabela da base de dados pode estar disponível para diferentes módulos de programa da aplicação e, conseqüentemente, para diferentes *UPs* de um mesmo módulo. Neste caso, duas unidades de programas podem possuir uma dependência de dados entre elas, mesmo não existindo chamadas entre elas.

Duas unidades de programa A e B possuem uma dependência de dados de A para B com relação à *variável tabela* ***t***, se a unidade A possuir um ponto no programa que define a *variável tabela* ***t*** (isto é, altera o estado da tabela de $e_i$ para $e_{i+1}$) e a unidade de programa B possuir um ponto no programa que usa a *variável tabela* ***t*** (com o estado de ***t*** idêntico a $e_{i+1}$). Para uma *variável tabela* ***t*** com uma *definição persistente* em uma unidade $UP_A$ (passando a variável ***t*** para um estado $e_A$) estará em um estado consistente de A (representado por $e_A$) se e somente se não existir nenhuma *redefinição persistente* de ***t*** em pelo menos um subprograma $UP_A$ no nó de saída. Existe uma dependência de dados c.r.a. *variável tabela* ***t*** se ***t*** tem uma definição persistente em uma unidade $UP_A$ e existe um caminho livre de definição c.r.a. ***t*** incluindo o sub-caminho de $UP_A$ a partir da definição persistente até o nó de saída e um sub-caminho do nó de entrada de uma unidade $UP_B$ até um *s-uso* de ***t***.

***Observação***: notar que por esta definição, $UP_A$ e $UP_B$ podem ser a mesma unidade de programa, ou seja, a dependência de dados é entre a *definição persistente* (par de nós) até a saída da Unidade que a definiu ($UP_A$) e o t-uso precedente a qualquer definição persistente a partir da entrada na Unidade que usa ($UP_B$).

São considerados dois tipos de dependência de dados c.r.a ***t***:

i) *Dependência interna ou intra-modular*: ocorre quando existir uma dependência de dados entre unidades de programa de um mesmo Módulo de

Programa, com relação a uma ou mais tabelas, mesmo quando não existir um ponto de chamada entre elas, (*Figura 3.6a*).

ii) *Dependência externa ou inter-modular*: ocorre quando existir uma dependência de dados entre duas ou mais *UPs* de Módulos de Programas diferentes c.r.a ***t*** . Isto é, ***t*** deve estar disponível para os dois Módulos de Programa, (Figura 3.6b).

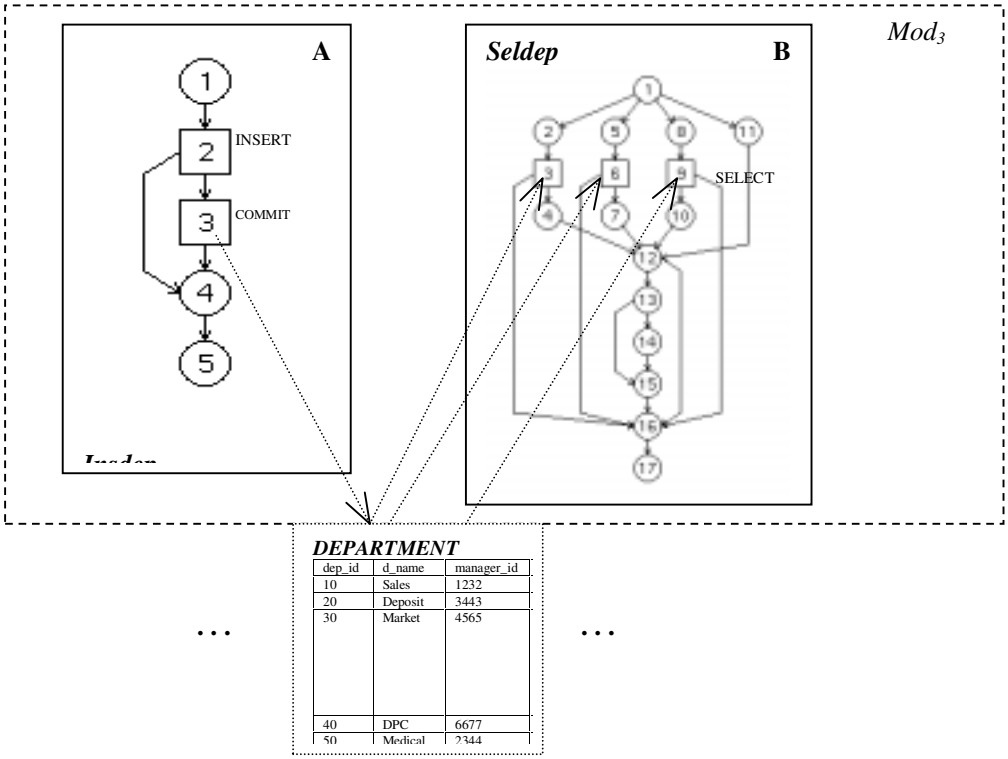



*Figura 3.6a: Grafo de dependência Interna de dados c.r.a DEPARTMENT.*
$GD_{PA}(Insdep_{Mod3} \times Seldep_{Mod3})_{DEPARTMENT}$

Considere o grafo de dependência de dados de um programa de ABDR, $GD_{PA}(A_{Modi} \times B_{Modj})_T = ([G(A_i) \times G(B_j)], E_{int}, T) = ([(N^{BD}, E, n_{in}, n_{out})_A \times (N^{BD}, E, n_{in}, n_{out})_B], E_{int}, T])$, onde $G(A_i) \times G(B_j)$ representa a união dos conjuntos de arcos e nós do grafo de programa da unidade **A** do módulo de programa ***i*** e do grafo de programa da unidade **B** do módulo de programa ***j***; *E* é o conjunto de arcos (setas pontilhadas) que mostram o fluxo de dados das unidades de programa para a tabela e vice-versa, e *T* é o conjunto de tabelas envolvidas na dependência de dados. $GD_{PA}(A_{Modi} \times B_{Modj})_T$ é um multi-grafo direcionado

que representa o fluxo de dados baseado na dependência de dados extraída da associação def-uso c.r.a. ***t***. Um exemplo desse grafo é apresentado na Figura 3.6b.

Uma *seta* (pontilhada) direcionada da *UP* para a tabela caracteriza a ocorrência de uma *definição* da variável tabela ***t*** pela *UP*. Uma seta direcionada no sentido da tabela ***t*** para a *UP*, caracteriza a ocorrência de um *uso* da tabela ***t*** pela *UP*. O Exemplo mostrado na *Figura 3.6a*. é um grafo de dependência *Interna*, mas se as *UPs A* e *B* pertencerem a Módulos de Programas distintos tratar-se-á de grafo de dependência *Externa* (*Figura 3.6.b*). No *exemplo da Figura 3.6.a*. a *UP Insdep* define a variável tabela DEPARTMENT (nó 3) e a *UP Seldep* usa a variável tabela DEPARTMENT em vários pontos do programa (nós 3, 6 e 9). Já no exemplo da *Figura 3.6.b.* a *UP Inscus* do Módulo de Programa $Mod_2$ define a variável tabela CUSTOMER no nó 3 e a *UP Relcli* do Módulo de Programa $Mod_4$ usa essa mesma variável no nó 2.

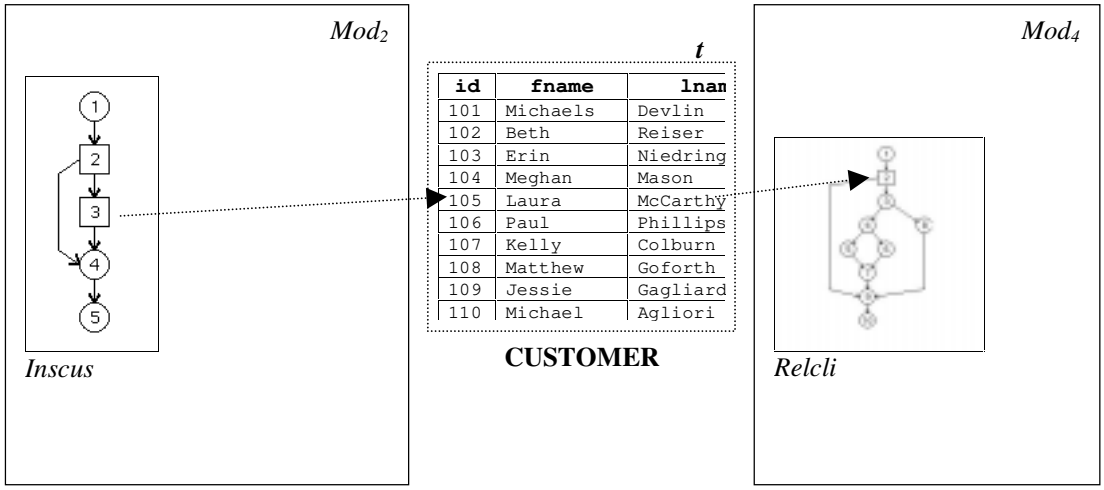



*Figura 3.6b: Grafo de dependência Externa de dados c.r.a CUSTOMER entre as UPs Inscus ($Mod_2$) e Relcli ($Mod_4$).*
$GD_{PA}(Inscus_{Mod2} \times Relcli_{Mod4})_{CUSTOMER}$

O Grafo geral de dependência de dados é um multi-grafo composto por todos os grafos de programas das *UPs* que possuem uma dependência de dados c.r.a alguma variável tabela ***t***. Assim, o grafo de dependência geral é denotado por $\boldsymbol{GD_G = (G_{UP}, E_{int}, T_{BD})}$, onde $\boldsymbol{G_{UP}}$ são os grafos das Unidades de Programa, $E_{int}$ são os arcos de integração que indicam o fluxo de dados entre as Unidades de Programas e as Tabelas da base de dados e $\boldsymbol{T_{BD}}$ são as tabelas da base de dados pontilhadas, que caracterizam as dependências de dados. O grafo geral tem o objetivo de dar um panorama completo das *UPs* que definem e/ou usam alguma

*variável tabela*, ajudando a visualizar os grafos de dependência de dados $G_{PA}(A_{Modi} \times B_{Modj})_t$, para o teste de integração intra-modular. A *Figura 3.7* mostra o grafo geral de dependência de dados, para o Módulo de Programa $Mod_3$. Um grafo de dependência externa de dados geral pode ser também construído para cada *variável tabela* permitindo, assim, organizar melhor todas as relações existentes entre as *UPs* contidas nos módulos de programas da aplicação. Para simplificar a *Figura 3.7*, foram colocadas as três unidades que contêm uma *definição persistente* em uma mesma representação gráfica relacionada a

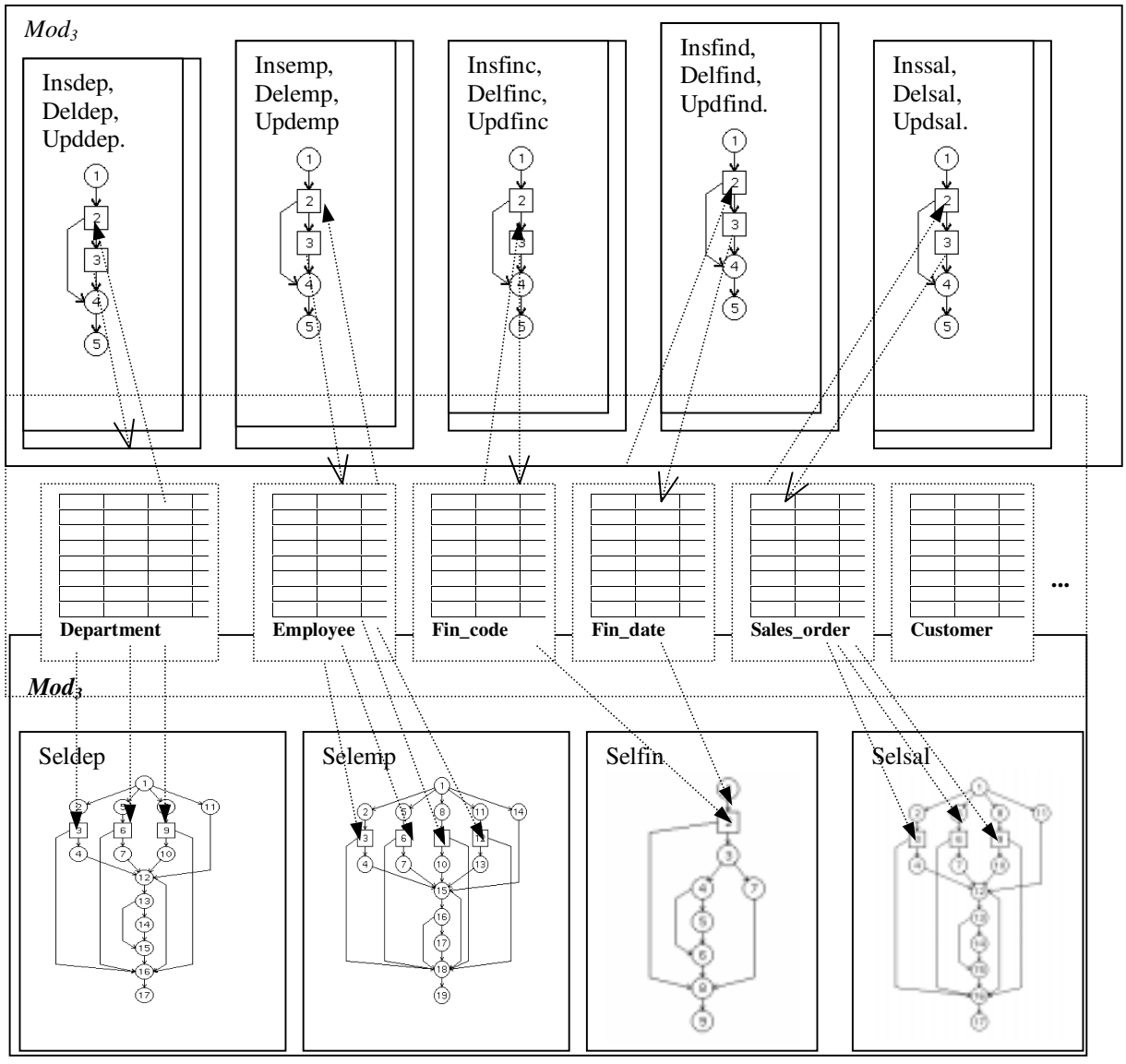



*Figura 3.7: Grafo de Dependência de dados interna entre as Unidades do Módulo de Programa $Mod_3$ . Ilustrada com os grafos de programas das UPs.*

cada *variável tabela*. Podemos observar também que a variável tabela CUSTOMER não possui nenhum relacionamento com as Unidades deste programa.

O teste de integração baseado na dependência de dados existente entre as Unidades de Programa é um teste caracterizado por exercitar as *associações definição-uso* c.r.a. variável ***t***, entre duas ou mais Unidades de Programas (internas ou externas). Essa técnica de teste é complementar à técnica de teste de unidade; sua finalidade é detectar defeitos associados a *variáveis tabela* através das dependências de dados entre as UPs (classes de defeitos não tratadas nos outros tipos de técnicas de teste).

## 3.2 Considerações Finais

Neste capítulo foram introduzidos os conceitos de teste estrutural aplicado em programas de aplicação. Os trabalhos apresentados em [RAP85, WEY86, FRA88, LIN90, MAL91, VIL98] foram estendidos para a abordagem de teste estrutural de Banco de Dados Relacional, adaptando alguns conceitos aos programas de aplicação e criando novos conceitos, específicos para programas de aplicação.

A característica essencial de uma ABDR é a existência de comandos da **SQL** que manipulam as *variáveis tabela,* tratadas como *variáveis persistentes*, e que possibilitam a criação de novas abordagens de teste estrutural. Foram propostos novos conceitos que relacionam a ocorrência de uso de variáveis dentro dos comandos de manipulação da **SQL** (denominados de *s-usos*). Propõe-se uma nova técnica de teste de integração baseada na dependência de dados, que associa uma variável definida em uma unidade com unidades que usam a mesma variável.

Esses conceitos formam uma base para a definição dos critérios de teste estrutural baseados em fluxo de dados *intra-modular* e *inter-modular* que são propostos no próximo capítulo.

# 4 Critérios de Teste: Definição e Análise de Propriedades

Neste capítulo são apresentados os critérios propostos e a análise de suas propriedades. Na Seção 4.2 são descritos os tipos de fluxo de dados dos programas de aplicação: *intra-modular* e *inter-modular*; na Seção 4.3 são descritos os critérios de teste *intra-modular*; na Seção 4.4 são descritos os critérios de integração *inter-modular*; a Seção 4.5 contém a análise de propriedades dos critérios.

## 4.1 Considerações iniciais

Foram propostos para o teste de unidade somente os critérios de teste que exercitam as *variáveis persistentes* (*tabelas*), tendo em vista que qualquer critério estrutural baseado em fluxo de dados pode ser usado para exercitar as *variáveis de programa* e *variáveis host*. O teste de integração *intra-modular* considera duas abordagens: *i)* teste de integração baseado no grafo de chamadas, segundo as definições de Vilela [VIL98]; e *ii)* teste de integração baseado na dependência de dados (da *variável tabela*). Esta segunda abordagem visa a exercitar *associações definição-t-uso* derivadas do fluxo de dados entre as unidades com comandos que caracterizam ocorrência de definição e as unidades com comandos que caracterizam ocorrência de uso.

## 4.2 Tipos de Fluxos de Dados

A análise do fluxo de dados baseia-se nos tipos de ocorrência das variáveis de um programa de aplicação. Conforme foi discutido no capítulo anterior, existem três tipos de variáveis: *i) variáveis de programa*: são as variáveis do programa hospedeiro; *ii) variáveis host*: são as variáveis que estabelecem os fluxos de dados entre a base de dados e o programa; e *iii) variáveis tabela*: são as *variáveis tabela* básicas (*variáveis persistentes* que compõem a base de dados) e as *variáveis tabela de visão* (tabelas virtuais derivadas das tabelas básicas e tratadas como variáveis locais ao *Módulo de Programa*). Cada tipo de variável tem o seu escopo de validade. Uma *variável de programa* é processada apenas no âmbito da linguagem hospedeira, uma *variável host* é processada tanto na linguagem

hospedeira como na linguagem **SQL** e uma *variável tabela* é processada apenas na linguagem **SQL**.

Considera-se que o projeto lógico das tabelas da aplicação está correto e que elas foram definidas e testadas anteriormente à implementação dos programas de aplicação.

### 4.2.1 Fluxo Intra-modular

O fluxo de dados *intra-modular* é utilizado para o teste de cada Módulo de Programa. Inicialmente é feito o teste de cada *UP* que compõe o Módulo de Programa (teste *intra-unidade ou teste de unidade*). Depois que todas as Unidades foram devidamente testadas, realiza-se o teste de integração para as *UPs* do Módulo de Programa (teste *inter-unidades* ou *teste de integração intra-modular*).

A partir do grafo de programa *G(UP)* de cada *UP* do Módulo é construído o grafo de dependência de dados que representa os fluxos de dados entre as *UPs* que compõem o Módulo. O grafo de dependência pode ser construído baseado na hierarquia de chamada entre as *UPs* ou baseado na dependência de dados entre as *UPs* determinada pelas *variáveis tabela* presentes no Módulo.

A Figura 4.1 apresenta um exemplo de dependência de dados entre as unidades de um mesmo módulo ($Mod_3$). O Módulo $Mod_3$ possui comandos de **SQL** com definições e usos *persistentes* de cinco das oito *variáveis tabela* mostradas na figura. Toda seta direcionada da *variável tabela* ***t*** para o nó de um grafo representa um uso de ***t*** naquele nó e toda seta apontada do nó para a *variável tabela* ***t*** representa uma *definição persistente* de ***t*** naquele nó. Toda unidade que tem um *uso* de uma variável ***t*** possui uma dependência de dados com as unidades que têm uma *definição persistente* da mesma variável ***t***.

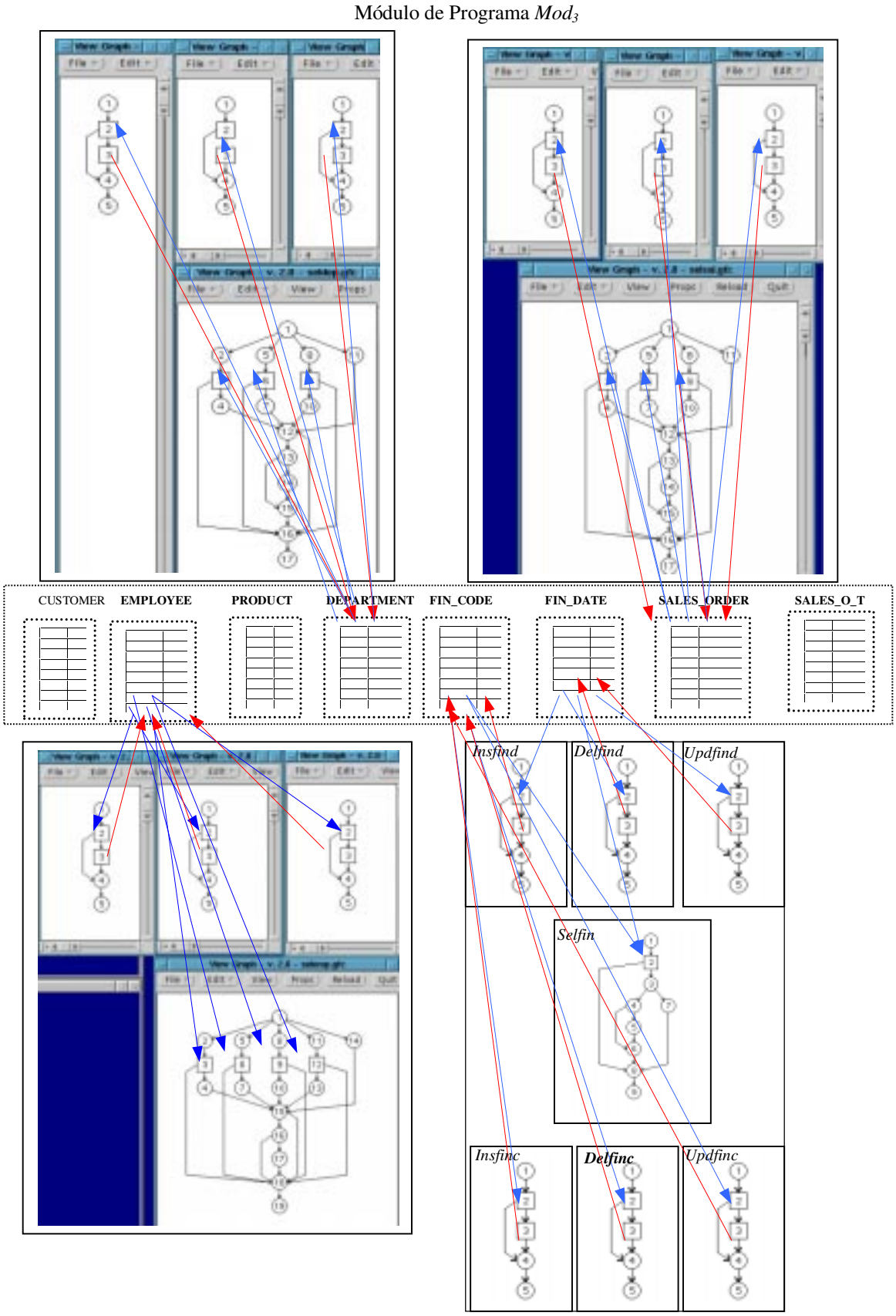



*Figura 4.1: Exemplo do Fluxo de Dados Intra-Modular completo.*

### 4.2.2 Fluxo Inter-modular

O fluxo de dados *inter-modular* é determinado a partir das dependências de dados ocasionadas pelas tabelas da base de dados comuns aos Módulos da ABDR. Uma $UP_i$ de um *Módulo* $Mod_k$ pode estar *definindo* uma *variável tabela* ***t*** enquanto outra $UP_j$ contida em outro *Módulo* $Mod_W$ pode estar *usando* a mesma *variável tabela* ***t***, determinando assim, um fluxo de dados entre os módulos $Mod_k$ e $Mod_w$. Assim, é possível definir associações *definição-uso* c.r.a ***t*** *(*denominada de *associação definição-t-uso-inter*) entre os módulos de programas criando um grafo de dependência externa de dados entre a $UP_i$ do Módulo $Mod_k$ que possui uma *definição persistente* da variável ***t*** e a $UP_j$ do Módulo $Mod_w$ que possui um *uso persistente* da variável ***t***. O modelo de fluxo de dados *inter-modular* é composto por nós retangulares contínuos que representam os *Módulos de Programa* e nós retangulares pontilhados representando as *variáveis tabela* que compõem a base de dados da aplicação. As setas indicam os fluxos de dado entre os nós. Tendo em vista que um Módulo de Programa é visto como um conjunto de Unidades de Programa $Mod_a = \{UP_1, .., UP_m\}$, temos:

a) Seta do *Módulo de Programa* ($Mod_i$) para tabela ***t***: indica que existe uma $UP_\alpha \subset Mod_i$ tal que $UP_\alpha$ tem uma definição *persistente* da tabela ***t***;
b) Seta da tabela ***t*** para o *Módulo de Programa* $Mod_i$: indica que existe uma $UP_\alpha \subset Mod_i$ tal que $UP_\alpha$ tem um uso *persistente* da tabela ***t***.

A execução de cada módulo de programa é feita isoladamente não havendo, portanto, rotinas de controle entre eles. De acordo com o exemplo apresentado na *Figura 4.2*, o módulo de programa $Mod_2$ define as tabelas CUSTOMER e SALES_ORDER. Da mesma forma, o módulo $Mod_2$ usa as tabelas CUSTOMER e SALES_ORDER. Deste modo, é possível formar todas as relações que determinam associações *definição-uso* para cada *Módulo de Programa* (*Mod*); essas informações são extraídas das *UPs* pertencentes a cada *Módulo de Programa* que possuírem um par $<\ell, i>$, da *definição persistente* de ***t***, com as *UPs* que possuírem um comando da **SQL** que usa a tabela ***t***.

Para entendermos melhor a Figura 4.2 é apresentado um mapa que relaciona para cada Módulo de Programa, as ações de ocorrência (*definição* e *uso*) para cada *variável tabela* (numeradas da esquerda para a direita).

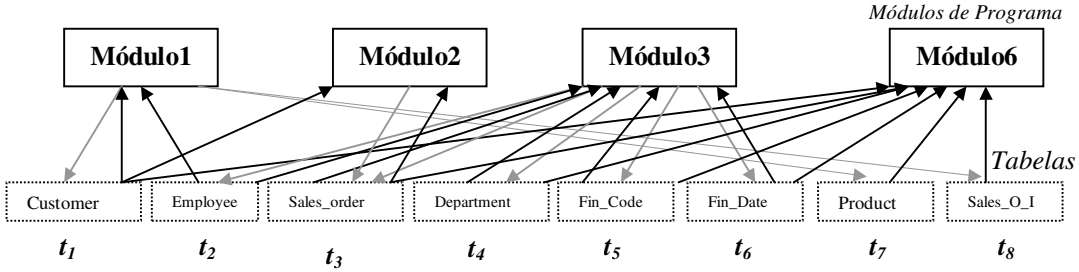



*Figura 4.2:* Fluxo de dados *inter-modular* de uma ABDR composta por 4 Módulos de Programa e 8 tabelas que constituem a base de dados

*Tabela 4.1:Mapa de definição e uso das variáveis tabela mostradas nos Módulos de Programa da Figura 4.2.*

| Módulos de Programa | Tabelas *definidas* | | | | | | | | Tabelas *usadas* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ |
| *Modulo1* | X | X | | | | | X | X | X | | | | | | | |
| *Modulo2* | X | | X | | | | | | X | | X | | | | | |
| *Modulo3* | | X | X | X | X | X | | | | X | X | X | X | X | | |
| *Modulo6* | | | | | | | | | X | X | X | X | X | X | X | X |

## 4.3 Os critérios intra-modulares

Os critérios de teste *intra-modulares* são aplicados no teste de cada *UP* do Módulo (teste de unidade) e no teste de integração das *UPs* do Módulo. Para o teste de unidade podemos utilizar qualquer critério de teste estrutural apresentado no Capítulo 2; os critérios para o teste de integração são divididos em dois subgrupos: critérios de testes baseados no *grafo de chamada* e critérios de testes baseados na *dependência de dados* das tabelas.

Os critérios de teste propostos para associar as *variáveis persistentes* devem ser satisfeitos com a mesma tupla, forçando o testador a gerar casos de testes específicos para exercitá-la. Deste modo, o teste deve ser executado de maneira controlada, não sendo suficiente usar qualquer tupla de ***t***. Por outro lado, com a exigência do uso da mesma tupla para satisfazer uma *associação definição-t-uso*, pode aumentar o número de elementos requeridos não factíveis (isto é, não executáveis com a mesma tupla).

### 4.3.1 Critérios aplicados ao teste de unidade

No teste de unidade pode ser aplicado qualquer critério de teste estrutural. Neste trabalho usou-se a Família de Critérios Potenciais Usos [MAL91] por razões já apresentadas.

Entretanto, os critérios para o teste de unidade não contemplam o teste de *variáveis persistentes*. Para tratar as *variáveis persistentes,* são propostos os critérios descritos a seguir. Considere os conceitos apresentados na Seção 3.1.1, que trata do teste de unidade para *variáveis persistentes*.

Seja ***G(UP)*** um grafo de programa e $\Pi$ um conjunto de caminhos completos de ***G(UP)***. Seja $\Gamma$ um conjunto de tuplas $\{\tau_1, \tau_2, ..., \tau_c\}$ pertencentes a uma dada tabela ***t***:

- **Todos os *t-usos***: requer para todas as associações *definição-t-uso* [$<\ell,i>$, *(j,k),* ***t***] que pelo menos um caminho $\pi=(n_{in}, .., \ell, .., i, .., j, k, .., n_{out}) \in \Pi$, livre de *definição persistente* de $<\ell,i>$ até *(j, k)* c.r.a. ***t***, seja exercitado pelo menos uma vez, para a mesma *tupla* $\tau \in \Gamma$.
- **Todos os *dtu-caminhos***: requer que todos os *dtu-caminhos* $(n_\ell, ..., n_i, ..., n_j, n_k)$ sejam executados pelo menos uma vez. A associação deve ser exercitada para a mesma *tupla* $\tau \in \Gamma$.

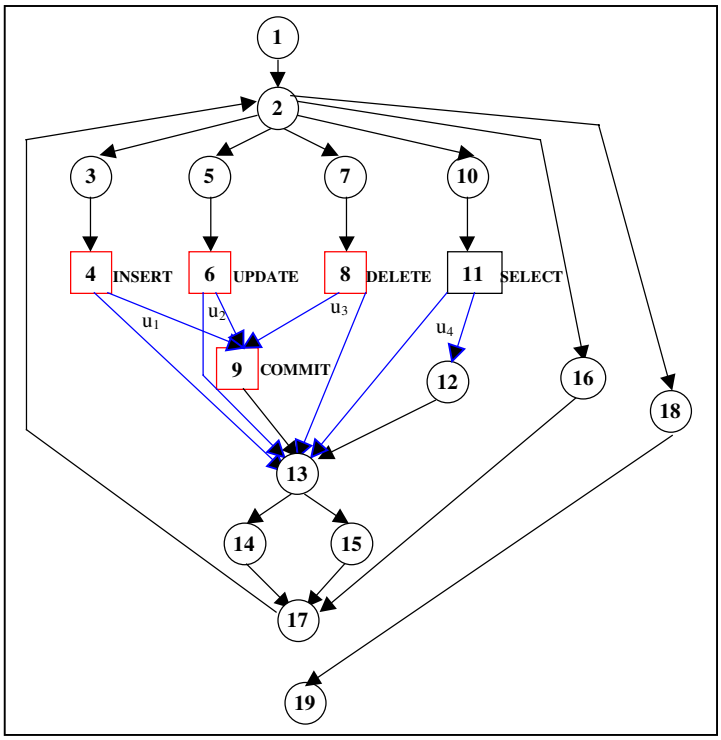
Assim, um conjunto de caso de testes ***Tc***, que resulta em $\Pi$ e $\Gamma$:

- Satisfaz o critério (*Todos os t-usos*) se, para todo par $<\ell, i>$ de nós $\in$ *G(UP)* com $defT<\ell,i> \neq \phi$ para toda variável $\boldsymbol{t} \in defT<\ell,i>$, $\Pi$ incluir todas as associações [$<\ell,i>$, *(j,k),* ***t***] onde $<\ell, i>$ *contém uma* $defT<\ell,i>$ e existir um ***t-uso*** em *(j, k)* sendo $j \in fdsu(\boldsymbol{t}, i)$ e $j \in N_S$; e existe um caminho livre de *definição persistente* de $<\ell,i>$ até *(j, k)*. A associação será satisfeita se e somente se for exercitada para a mesma *tupla* $\tau$.
- Satisfaz o critério (*todos os dtu-caminhos*) se, para todo par $<\ell, i>$ de nós $\in$ *G(UP)* com $defT<\ell,i> \neq \phi$, $\Pi$ incluir todos os *dtu-caminhos* de $<\ell,i>$ até *(j, k)*, c. r. a. cada variável $\boldsymbol{t} \in defT<\ell,i>$ para todas as associações [$<\ell,i>$, *(j, k),* ***t*** ] tal que $<\ell , i>$ *contém uma* $defT<\ell,i>$ e existe um ***t-uso*** em *(j, k)* onde $j \in fdsu(\boldsymbol{t}, i)$ e $j \in N_S$; e existe um

caminho livre de *definição persistente* de <ℓ,*i*> até *(j, k)*. A associação será satisfeita se e somente se for exercitada para a mesma *tupla* $\tau$.

Esses critérios têm como objetivo requerer que todo sub-caminho que inicia no nó ℓ da **SQL** (INSERT ou DELETE ou UPDATE) e passa pelo nó *i* (COMMIT), onde ocorre a *definição persistente* de ***t***, e alcança um arco de saída do nó da **SQL**, onde ocorre o uso de ***t*** (***t-uso***) com os comandos (INSERT, DELETE, UPDATE ou SELECT), sejam executados pelo menos uma vez para a mesma *tupla* $\tau$.

Como exemplo, considere o Grafo *G(UP)* da Figura 4.3, uma *UP* que contém as quatro operações da **SQL**. Neste exemplo, $u_1$, $u_2$, $u_3$ e $u_4$ representam os arcos que contêm um uso de ***t*** (*t-uso*) e $d_1$, $d_2$ e $d_3$ representam os pares de nós que caracterizam a *definição persistente* de ***t*** (sempre representado pelo par <ℓ,*i*>). O critério *todos os t-usos* para o teste de unidade só terá elementos requeridos se existirem comandos de repetição envolvendo comandos de manipulação da Base de Dados na *UP* ou repetição dos comandos de




| | *Todos-t-usos* | ocorrências |
|---|---|---|
| **1.** | [<4, 9>, (4, 9), ***t***] | $d_1u_1\ d_1$. |
| **2.** | [ <4, 9>, (4, 13), ***t***] | $d_1\ u_1$.. |
| **3.** | [ <4, 9>, (6, 9), ***t***] | $d_1u_2d_2$.. |
| **4.** | [<4, 9>, (6, 13), ***t***] | $d_1u_2$.. |
| **5.** | [<4, 9>, (8, 9), ***t***] | $d_1u_3d_3$.. |
| **6.** | [<4, 9>, (8, 13), ***t***] | $d_1u_3$... |
| **7.** | [<4, 9>, (11, 12), ***t***] | $d_1u_4$... |
| **8.** | [<4, 9>, (11, 13), ***t***] | $d_1u_4$... |
| **9.** | [<6, 9>, (4, 9), ***t***] | $d_2\ u_1d_1$. |
| **10.** | [<6, 9>, (4, 13), ***t***] | $d_2\ u_1$.. |
| **11.** | [<6, 9>, (6, 9), ***t***] | $d_2u_2d_2$.. |
| **12.** | [<6, 9>, (6, 13), ***t***] | $d_2u_2$.. |
| **13.** | [<6, 9>, (8, 9), ***t***] | $d_2u_3d_3$.. |
| **14.** | [<6, 9>, (8, 13), ***t***] | $d_2u_3$... |
| **15.** | [<6, 9>, (11, 12), ***t***] | $d_2u_4$... |
| **16.** | [<6, 9>, (11, 13), ***t***] | $d_2u_4$... |
| **17.** | [<8, 9>, (4, 9), ***t***] | $d_3\ u_1d_1$. |
| **18.** | [<8, 9>, (4, 13), ***t***] | $d_3\ u_1$.. |
| **19.** | [<8, 9>, (6, 9), ***t***] | $d_3u_2d_2$.. |
| **20.** | [<8, 9>, (6, 13), ***t***] | $d_3u_2$.. |
| **21.** | [<8, 9>, (8, 9), ***t***] | $d_3u_3d_3$.. |
| **22.** | [<8, 9>, (8, 13), ***t***] | $d_3u_3$... |
| **23.** | [<8, 9>, (11, 12), ***t***] | $d_3u_4$... |
| **24.** | [<8, 9>, (11, 13), ***t***] | $d_3u_4$... |

*Figura 4.3: Exemplo de um procedimento que contém todas operações da SQL.*

manipulação na seqüência de execução da *UP*. No caso da *variável tabela* a ocorrência de *anomalias* de fluxo de dados não é necessariamente um defeito. Uma seqüência de ocorrências que define uma *variável tabela* não precisa ser usada antes de uma outra definição da mesma variável. Outro tipo de anomalia, que não é necessariamente um defeito, é o uso de uma variável ***t*** em uma *UP* sem uma definição da mesma no programa.

Lembrando que os critérios de teste que exercitam as *variáveis persistentes* somente são satisfeitos se a associação *definição-t-uso* for exercitada com a mesma *tupla*; o objetivo é exercitar as associações de ***t*** em memória secundária.

No exemplo da Figura 4.3 temos os seguintes *dtu-caminhos*:

**1.** 4 9 13 14 17 2 3 4 9
**2.** 4 9 13 14 17 2 3 4 13
**3.** 4 9 13 14 17 2 5 6 9
**4.** 4 9 13 14 17 2 5 6 13
**5.** 4 9 13 14 17 2 7 8 9
**6.** 4 9 13 14 17 2 7 8 13
**7.** 4 9 13 14 17 2 10 11 12
**8.** 4 9 13 14 17 2 10 11 13
**9.** 4 9 13 15 17 2 3 4 9
**10.** 4 9 13 15 17 2 3 4 13
**11.** 4 9 13 15 17 2 5 6 9
**12.** 4 9 13 15 17 2 5 6 13
**13.** 4 9 13 15 17 2 7 8 9
**14.** 4 9 13 15 17 2 7 8 13
**15.** 4 9 13 15 17 2 10 11 12
**16.** 4 9 13 15 17 2 10 11 13
**17.** 6 9 13 14 17 2 3 4 9
**18.** 6 9 13 14 17 2 3 4 13
**19.** 6 9 13 14 17 2 5 6 9
**20.** 6 9 13 14 17 2 5 6 13
**21.** 6 9 13 14 17 2 7 8 9
**22.** 6 9 13 14 17 2 7 8 13
**23.** 6 9 13 14 17 2 10 11 12
**24.** 6 9 13 14 17 2 10 11 13
**25.** 6 9 13 15 17 2 3 4 9
**26.** 6 9 13 15 17 2 3 4 13
**27.** 6 9 13 15 17 2 5 6 9
**28.** 6 9 13 15 17 2 5 6 13
**29.** 6 9 13 15 17 2 7 8 9
**30.** 6 9 13 15 17 2 7 8 13
**31.** 6 9 13 15 17 2 10 11 12
**32.** 6 9 13 15 17 2 10 11 13
**33.** 8 9 13 14 17 2 3 4 9
**34.** 8 9 13 14 17 2 3 4 13
**35.** 8 9 13 14 17 2 5 6 9
**36.** 8 9 13 14 17 2 5 6 13
**37.** 8 9 13 14 17 2 7 8 9
**38.** 8 9 13 14 17 2 7 8 13
**39.** 8 9 13 14 17 2 10 11 12
**40.** 8 9 13 14 17 2 10 11 13
**41.** 8 9 13 15 17 2 3 4 9
**42.** 8 9 13 15 17 2 3 4 13
**43.** 8 9 13 15 17 2 5 6 9
**44.** 8 9 13 15 17 2 5 6 13
**45.** 8 9 13 15 17 2 7 8 9
**46.** 8 9 13 15 17 2 7 8 13
**47.** 8 9 13 15 17 2 10 11 12
**48.** 8 9 13 15 17 2 10 11 13

Como exemplo, o quinto *dtu-caminho* (**4 9 13 14 17 2 7 8 9**) será satisfeito se e somente se a tupla usada na *definição persistente* no par <*4, 9*> for a mesma usada no arco de saída do nó da **SQL** (8, 9) onde ocorre um ***t-uso*** e existir um caminho livre de *definição persistente* c.r.a. ***t*** do par <*4, 9*> até o arco *(8, 9)* e um caminho livre de laço a partir do nó *9* até o arco (*8, 9*). De maneira análoga essas regras são válidas a todos *dtu-caminhos*.

### 4.3.2 Critérios aplicados ao teste de integração intra-modular

Os critérios de teste de integração *intra-modular* são divididos segundo duas abordagens de teste: **Baseada no grafo de chamada** - para exercitar as variáveis definidas na memória interna pode ser usado qualquer critério de teste de integração como os descritos em [VIL98]; **Baseada na dependência de dados** - para exercitar as *variáveis persistentes* são propostos critérios específicos para programas de aplicação que requerem o exercício de associações *def-uso* de *variáveis persistentes*.

Os critérios de integração *intra-modular* visam a exercitar as *associações definição-t-uso* determinadas pelos comandos de manipulação da base de dados. Tanto os critérios de integração baseados no grafo de chamadas como os critérios baseados na dependência de dados determinada pelas tabelas da base de dados visam a exercitar as associações referentes às *variáveis de programa* (***P***), *variáveis host* (***H***) ou *variáveis tabela* (***T***), de acordo com o enfoque estabelecido pelos critérios.

#### 4.3.2.1 Critérios de Teste de integração baseados no grafo de chamadas

Os critérios definidos nesta seção estão baseados na mesma idéia dos critérios Potenciais Uso de Integração definidos e analisados em Vilela [VIL98]. O grafo de chamada é um multi-grafo, podendo haver mais de uma chamada entre duas *Unidades de Programa*, o que corresponde à ocorrência de mais de um arco ligando os respectivos nós do grafo. Essa abordagem considera a integração das *Unidades de Programa (UPs)*, feita *duas-a-duas* ("*pairwise*") e os requisitos de teste são, consequentemente, derivados para cada par de *UPs*.

Para estender os critérios de integração propostos por Vilela, são definidos dois critérios que têm como finalidade exercitar as associações envolvendo as *variáveis tabela* ***t***. Os critérios de teste de integração em [VIL98, OST91, LIN90, HAR91] são derivados dos critérios de teste de unidade, visando a associar pontos de definição e uso para as variáveis globais ou variáveis passadas por parâmetros que se mantêm vivas ao longo das unidades envolvidas na integração, controladas pelo grafo de chamada do programa. Nesse caso, são consideradas as *variáveis de programa* e as *variáveis host*. Os critérios propostos nesta seção têm como objetivo exercitar associações definição uso para as *variáveis tabela* já que no caso das *variáveis host*, quando forem tratadas como variáveis globais, podem ser exercitadas por qualquer critério de teste estrutural de integração. Por exemplo, os critérios de Vilela possibilitam considerar as *variáveis host*, com definição global em algum nó *i* de uma unidade $UP_a$ que, a partir de um caminho livre de definição, alcance algum comando de chamada para outra unidade $UP_b$. Os critérios Potenciais-Usos de integração contemplam as *variáveis host* e *tabelas de visão*, existentes em programas de aplicação (casos *i* e *ii* da Seção 3.1.2.1).

As definições dadas em *iii* e *iv* da Seção 3.1.2.1 contemplam as *variáveis persistentes* e dão origem aos dois critérios: *todas as associações-definição-t-uso-int* e *todos os dtu-caminhos-int* (de integração), apresentados abaixo. Esses critérios exercitam as associações baseadas no grafo de chamada, para as *variáveis tabela* como variáveis *persistentes*. Lembre-se que o *t-uso* ocorre nos arcos de saída do nó da **SQL** com uso de ***t***.

*i)* ***Todos os t-usos*** **interprocedimentais de chamada**: requer que *todas as associações* $[<\ell,i>, (j, k), \mathbf{t}, c]$ sejam satisfeitas, onde *c* é uma chamada de uma unidade $UP_a$ para outra unidade $UP_b$ no nó $n_c$ do $G(UP_a)$; $\mathbf{t} \in in_c$; onde $defT<\ell,i> \neq \phi$, e $\ell \in N_S$ e $i \in N_S$ e são nós de $G(UP_a)$ que contêm uma definição *persistente* de ***t***; $j \in N_S$ e é um nó de $G(UP_a)$, existe um *t-uso* global de ***t*** em $(j, k) \in Arc_{out}(j)$ e contém um sub-caminho livre de definição c.r.a ***t*** de *i* para $n_c$ e c.r.a ***t*** de $n_{in}(UP_b)$ para *j*; a associação é satisfeita se e somente se for exercitada através da mesma tupla $\tau$.

Uma *associação-definição-t-uso interprocedimentais de chamada* é representada pela sextupla:

$$[UP_a, <\ell, i>, UP_b, (j, k), \mathbf{t}, c]$$

*ii)* ***Todos os t-usos*** **interprocedimentais de retorno**: requer que *todas as associações* $[<\ell,i>, (j, k), \mathbf{t}, r]$ sejam satisfeitas, onde *c* é uma chamada de uma unidade $UP_a$ para outra unidade $UP_b$ no nó $n_c$ de $G(UP_a)$; $\mathbf{t} \in out_c$; onde $defT<\ell,i> \neq \phi$, e $\ell \in N_S$ e $i \in N_S$ e são nós de $G(UP_b)$ que contêm uma *definição persistente* de ***t***; $j \in N_S$ e é um nó de $G(UP_a)$, existe um *t-uso* global de ***t*** em $(j, k) \in Arc_{out}(j)$ e contém um sub-caminho livre de definição c.r.a ***t*** (tratada em [VIL98]) de *i* para $out_c$ e c.r.a ***t*** de $n_c(UP_a)$ para *j* e a associação é satisfeita se e somente se for exercitada através da mesma tupla $\tau$.

Uma *associação-definição-t-uso interprocedimental de retorno* é representada pela sextupla:

$$[UP_a, <\ell, i>, UP_b, (j, k), \mathbf{t}, r]$$

*iii)* ***Todos os dtu-caminhos interprocedimentais de chamada****:* requer que *todos os dtu-caminhos* ($n_\ell$, ..., $n_i$, ..., $n_c$, ..., $n_j$, $n_k$) sejam satisfeitos, onde c é uma chamada de uma unidade $UP_a$ para outra unidade $UP_b$ no nó $n_c$ do G($UP_a$); $\boldsymbol{t} \in in_c$; onde $defT\langle \ell, i\rangle \neq \phi$, $\ell \in N_S$ e $i \in N_S$ e são nós de $G(UP_a)$ que contém uma *definição persistente* de ***t***; onde $j \in N_S$ em $UP_b$, onde $j \in fdsu(\boldsymbol{t}, i)$ e contém um ***t****-uso* em (*j, k*), e existe um sub-caminho livre de definição persistente c.r.a. ***t*** de *i* para $n_c$ c.r.a. ***t*** de $n_{in}(UP_b)$ para (*j,k)*; o *dtu-caminho* é satisfeito se for exercitado através da mesma tupla $\tau$.

*iv)* ***Todos os dtu-caminho interprocedimentais de retorno***: requer que todos os *dtu-caminhos* ($n_\ell$, ..., $n_i$, ..., $n_r$, ..., $n_j$, $n_k$) sejam satisfeitos, onde *r* é o retorno da unidade $UP_b$ para $UP_a$ no nó $n_r$ de $G(UP_b)$; $\boldsymbol{t} \in out_r$; $defT\langle \ell, i\rangle \neq \phi$, $\ell \in N_S$ e $i \in N_S$ são nós de $G(UP_b)$ que contêm uma *definição persistente* c.r.a. ***t***; ; onde $j \in N_S$ em $UP_a$ onde $j \in fdsu(\boldsymbol{t}, i)$ e contém um ***t****-uso* em (*j, k*) e existe um sub-caminho livre de definição c.r.a ***t*** de *i* para $out_c$ e c.r.a ***t*** de $n_c(UP_a)$ para (*j,k)*; o *dtu-caminho* é satisfeito se for exercitado através da mesma tupla $\tau$.

As estruturas das unidades $UP_a$ e $UP_b$ (que chama e que é chamada, respectivamente) têm uma ligação direta entre elas, com a existência do comando de chamada.

Os critérios aplicados ao teste de integração são complementares aos do teste de unidade e visam a exercitar classes de erros distintas. Embora nem sempre seja possível exercitar tais critérios dois a dois, pode-se estender os critérios de *a* a *d* para ciclos de chamadas indiretas (usando mais de uma *UP* para satisfazer a associação), contanto que se respeitem as propriedades de caminhos livres de definição e a mesma *tupla* seja usada na *definição persistente* de ***t*** e no *uso* de ***t***, para satisfazer a associação. Os casos abordados nesta seção serão estudados com mais profundidade em trabalhos futuros, juntamente com o trabalho desenvolvido por Vilela [VIL98], não sendo, portanto, o principal enfoque desta tese.

A seguir são apresentados os critérios de teste de integração baseados na dependência de dados ocasionada pelas tabelas da base de dados, tendo em vista a persistência das *variáveis tabela*.

#### 4.3.2.2 Critérios de teste de integração baseados na dependência de dados

O fluxo de dados ocasionado pelas *variáveis tabela* é considerado o mais importante em ABDR, tendo em vista a sua *persistência.* A *variável tabela* definida por uma unidade de programa mantém-se viva ao longo do tempo, determinando a existência de uma associação *def-t-uso* entre duas *UPs* executadas em momentos distintos, sem a necessidade de um comando de chamada de uma para outra. Dizemos que uma associação *def-t-uso* existe, se uma variável ***t*** definida em $UP_A$ tem um *t-uso* em $UP_B$.

*Exemplo:* Sejam duas unidades $UP_A$ e $UP_B$ pertencentes ao mesmo Módulo $Mod_i$. Um dado colocado em uma tabela ***t*** pela unidade $UP_A$ pode ocasionar um resultado não esperado na unidade $UP_B$ (em outro momento de execução), durante um uso de ***t***; esse tipo de falha só é observável pela execução seqüenciada de $UP_A$ e $UP_B$, exigida pelos critérios de integração baseados em dependência de dados .

As associações *definição-t-uso* de integração são exercitadas com seqüências de execuções para pares de unidades ($UP^d$, unidade que define ***t***; $UP^u$, unidade que usa ***t***). Neste caso, é importante definir uma estratégia que exercite essas associações utilizando os mesmos valores, tanto para a execução de $UP^d$ como para a de $UP^u$.

Depois que todas as unidades de programa são associadas *aos pares*, para o exercício das variáveis persistentes, algumas associações podem não ter sido exercitadas, devido à dependência de dados existente entre uma variável tabela ***t*** e outra variável tabela ***t'***. Neste caso, existe a necessidade de acrescentar uma unidade que define a variável ***t'*** exigindo-se um ciclo a mais de execução para exercitar a associação referente à variável ***t***.

- **Critérios baseados no fluxo de dados intra-modular**

Estes critérios baseiam-se nas dependências de dados existentes entre os procedimentos de um mesmo *Módulo de Programa* com relação a uma *variável tabela* da base de dados e requerem a mesma *tupla* para satisfazer a associação *definição-t-uso*. Cada unidade $UP_A$ que contém uma *definição persistente* da variável ***t*** é associada a outra

unidade $UP_B$ que contém um *uso* de ***t*** ***(t-uso)***; os pares ($UP_A$, $UP_B$) são requeridos pelo critério com um ciclo de execução (denomina-se de *ciclo1*)

Considere que $G(UP_A)$ e $G(UP_B)$ sejam os grafos das unidades A e B, respectivamente, envolvidas na integração e $\Pi$ seja o conjunto dos caminhos completos em $G(UP_A)$ e $G(UP_B)$ e $\Gamma$ o conjunto de tuplas. $(\pi_a, \pi_b) \in \Pi$ implica que existe a concatenação $\pi_a, \pi_b$, onde $\pi_a \in G(UP_A)$ e $\pi_b \in G(UP_B)$.

***Todas os t-usos-ciclo1-intra***: $\Pi$ e $\Gamma$ satisfazem o critério (*todos os t-usos-ciclo1-intra*) para o par de unidades $UP_A$ e $UP_B$ pertencentes ao mesmo módulo se, para todos os pares de caminhos $(\pi_a, \pi_b) \in \Pi$, $\pi_a$ inclui o par de nós $<\ell,i> \in G(UP_A)$ tal que $defT<\ell,i> \neq \phi$ c.r.a. ***t***, e existir um caminho livre de definição c.r.a. ***t*** de $<\ell, i>$ até $n_{out}$ de $G(UP_A)$; e $\pi_b$ inclui o arco $(j, k) \in G(UP_B)$ onde $j \in fdsu(\mathbf{t},i)$ e existe um ***t-uso*** em $(j, k)$, e existir um *caminho livre de definição persistente* c.r.a. ***t*** de $n_{in}$ até o arco $(j, k)$; a associação é satisfeita se for exercitada através da mesma tupla $\tau$.

Uma *associação-t-uso-ciclo1-intra* é representada pela quíntupla:

$$[^{z}UP_A\ ,\ <\ell, i>,\ ^{z}UP_B\ ,\ (j, k)\ ,\ \mathbf{t}]$$

***Todos os dtu-caminhos-intra***: $\Pi$ e $\Gamma$ satisfazem o critério (*todos os dtu-caminhos-intra*) para o par de unidades $UP_A$ e $UP_B$ pertencentes ao mesmo módulo se, para todos os pares de caminhos $(\pi_a, \pi_b) \in \Pi$, $\pi_a$ incluir um *dtu-caminho* do nó $\ell$ até o nó $n_{out}$ de $G(UP_A)$ passando pelo par de nós $<\ell, i> \subseteq G(UP_A)$ onde $defT<\ell,i> \neq \phi$ c.r.a. ***t***, através da tupla $\tau$ e $\pi_b$ incluir um *dtu-caminho* c.r.a ***t*** do nó $n_{in}$ de $G(UP_B)$ *até o arco* $(j, k) \in G(UP_B)$ sendo $j \in fdsu(\mathbf{t},i)$ e existe um ***t-uso*** em $(j, k)$, através da mesma tupla $\tau$ e existir um *caminho livre de definição persistente* que vai do par $<\ell, i>$ em $UP_A$ até o arco $(j, k)$ em $UP_B$ c.r.a ***t*** na composição dos caminhos $\pi_a . \pi_b$.

O elemento requerido pelo critério *todos os dtu-caminhos-intra* é representado pela n-upla abaixo, sendo $z$ o número do módulo que contém as unidades *A* e *B*:

$$[^{z}UP_A\ \ \ell\ a_1\ ...\ a_n\ \ i\ ...\ \ n_{out}\ -\ ^{z}UP_B\ \ n_{in}\ ...\ j\ \ k]$$

Onde $^{z}UP_A$ é a unidade onde ocorre a definição de ***t*** e $\ell\ a_1\ ...\ a_n\ \ i\ ...\ \ n_{out}$ é o *dtu-caminho* de $<\ell\ ,\ i>$ até o nó de saída, podendo ou não passar por um sub-caminho ($a_1$ ...

$a_n$) livre de laço; $^{z}UP_B$ é a unidade onde ocorre o uso de ***t*** e $n_{in}$ ... $j$ $k$ é o *dtu-caminho* que vai do nó inicial até o arco *(j, k)* onde existe um *t-uso* c.r.a. ***t***.

Existem dependências que não podem ser exercitadas com a execução de apenas duas unidades (uma de *definição* e outra de *uso*), mas necessitam da execução de outra unidade para satisfazer a associação. Isso ocorre quando existem dependências múltiplas, ou seja, para definir a variável ***t*** é necessário estar definida a variável ***t'*** e, para isso, pode ser necessário executarmos a unidade que define a variável ***t'*** para depois executarmos a unidade que define ***t*** e finalmente executarmos a *UP* que usa ***t***. Observe que podem existir dependências múltiplas exigindo mais de dois ciclos de execução. A observação mais importante é que mesmo executando três unidades: $UP_1$, $UP_2$ e $UP_3$, a associação é de $UP_1$ para $UP_3$ ou $UP_2$ para $UP_3$. A unidade $UP_1$ ou a unidade $UP_2$ tem a *definição persistente* de uma variável ***t*** que é usada em $UP_3$. Outro caso ocorre quando existem usos múltiplos em $UP_3$ e para satisfazê-los temos que executar duas ou mais unidades para definir as variáveis tabelas referenciadas em $UP_3$, ocasionando associações múltiplas ($UP_1$, $UP_3$) e ($UP_2$, $UP_3$) e que podem ser satisfeitas com o critério de *Ciclo2*.

***Todos os t-usos-ciclo2-intra***: $\Pi$ e $T$ satisfazem o critério (*todos os t-usos-ciclo2-intra*) para as unidades $UP_A$, $UP_B$ e $UP_C$ pertencem ao mesmo módulo se, para todos os caminhos ($\pi_a$, $\pi_b$, $\pi_c$) $\in \Pi$, onde $\pi_a$ é o caminho que contém um par de nós $<\ell',i'>$ que *define* a variável ***t'*** pela *tupla* $\tau$'; o caminho $\pi_b$ contém um par de nós $<\ell,i>$ que *define* ***t*** pela tupla $\tau$ podendo ou não ter uma dependência de ***t'***, *e* o caminho $\pi_c$ possuir um nó que contém um *uso* de ***t*** e ou ***t'*** pela mesma *tupla* $\tau$ *e/ou* $\tau$', respectivamente, e o caminho $\pi_a$ passar pelo par de nós $<\ell',i'> \in$ G($UP_A$) com $defT<\ell',i'> \neq \phi$ c.r.a. ***t'***; existir um *caminho livre de definição* c.r.a. ***t'*** de $i$ até $n_{out}$ de $G(UP_A)$; o caminho $\pi_a$ passar pelo par de nós $<\ell,i> \in$ G($UP_B$) onde $defT<\ell,i> \neq \phi$ c.r.a. ***t***; existir um *caminho livre de definição* c.r.a. ***t*** de $i$ até $n_{out}$ de $G(UP_B)$; o caminho $\pi_b$ passar pelo arco $(j,k) \in G(UP_C)$/ existe um ***t-uso*** em $(j,k)$ e $j \in fdsu(\mathbf{t},i)$ (sendo ***t*** e/ou ***t'***) e existir um *caminho livre de definição* c.r.a. ***t*** e/ou ***t'*** de $n_{in}$ até o arco $(j,k)$. A associação [$<\ell',i'>$, $UP_A$, $<\ell,i>$, $UP_B$, $(j,k)$, $UP_C$, {***t'***, ***t***}] é satisfeita se e somente se a mesma *tupla* $\tau$ ( $\tau$') usada para satisfazer a *definição* de ***t*** ( ***t'***) é também usada para satisfazer o *uso*.

Neste caso, os elementos requeridos são representados por uma sétupla do tipo:

$$[^{z}UP_A\ ,\ <\ell, i>,\ ^{z}UP_B,\ <\ell', i'>,\ ^{z}UP_C\ ,\ (j, k)\ ,\ \{\boldsymbol{t}, \boldsymbol{t'}\}]$$

Dois enfoques podem ser considerados: Utilizar as mesmas tuplas para satisfazer os pares *definição-uso* na integração ou utilizar tuplas diferentes para satisfazer os elementos requeridos não cobertos com a mesma tupla. O primeiro enfoque é mais conservador e mostrou, na execução de um exemplo de aplicação, ser melhor para a detecção de defeitos do que o segundo.

## 4.4 Critérios de integração inter-modular

Os critérios de integração *inter-modular* são semelhantes aos critérios de integração *intra-modular* com a diferença de que as unidades associadas devem pertencer a Módulos de Programa distintos. Esses critérios visam a exercitar associações de *variáveis tabela* que são definidas em unidades de um Módulo $\alpha$ e são usadas em unidades de outro Módulo $\beta$. Neste caso, a associação *definição-t-uso* é uma n-upla $[_{Mod1}UP_A\ ,<\ell,i>,\ _{Mod2}UP_B\ ,\ (j,k),\ \boldsymbol{t}\ ]$ onde a *definição persistente* em $<\ell,i> \in G(UP_A)_{Mod1}$ c.r.a. $\boldsymbol{t}$ é associada a um *uso* de $\boldsymbol{t}$ em $(j, k) \in G(UP_B)_{Mod2}$.

- **Critérios baseados no fluxo de dados inter-modular**

Considere duas unidades $UP_A$, pertencente a um módulo $Mod_x$, e $UP_B$, pertencente a outro módulo $Mod_y$. Considere também os mesmos conjuntos $\Pi$ e $\Gamma$ definidos anteriormente. Temos:

***Todos os t-usos-ciclo1-inter***: Este critério é idêntico ao critério *todos os t-usos-ciclo1-intra*, com a única diferença que as unidades $UP_A$ e $UP_B$ pertencem a Módulos distintos.

Neste caso o elemento requerido é representado pela quíntupla:

$$[^{z}UP_A\ ,\ <\ell, i>,^{w}UP_B\ ,\ (j, k)\ ,\ \boldsymbol{t}]$$

onde $z$ representa o módulo $Mod_z$ e $w$ representa $Mod_w$.

***Todos os dtu-caminhos-inter***: Este critério é idêntico ao critério *todos os dtu-caminhos-intra*, com a única diferença que a unidade onde existe uma definição persistente ($UP_A$) pertence a um Módulo e a unidade onde existe um ***t-uso*** pertence a outro Módulo.

Neste caso o elemento requerido é representado pela quíntupla:

$$[^{z}UP_A \ \ell \ a_1 \ldots a_n \ i \ldots \ n_{out} - {}^{w}UP_B \ n_{in} \ldots j \ k]$$

onde $z$ representa o módulo $Mod_z$ e $w$ representa $Mod_w$.

***Todos os t-usos-ciclo2-inter***: É idêntico ao critério *todos os t-usos-ciclo2-intra*, com a diferença que as unidades *associadas* devem pertencer a Módulos distintos.

Neste caso, os elementos requeridos são representados por uma sétupla do tipo:

$$[^{z}UP_A, <\ell, i>, {}^{y}UP_B, <\ell', i'>, {}^{w}UP_C, (j, k), \{\mathbf{t, t'}\}]$$

## 4.5 Análise de Propriedades

Estudos de propriedades e características de teste estrutural têm sido conduzidos em trabalhos anteriores [FRA88, MAL91, ZHU96] visando a comparar os diferentes critérios. Critérios incomparáveis analiticamente podem ter suas forças relativas avaliadas empiricamente para determinar quais são os mais exigentes com relação ao número de casos de teste necessários para satisfazê-los. Comparações entre os critérios da FCPU e os critérios de teste de integração baseados na dependência de dados *intra-modular* foram realizadas com os exemplos de aplicação descritos resumidamente no Capítulo 6; os exemplos são apresentados detalhadamente no *Apêndice C*.

A seguir é apresentada uma análise de inclusão entre os critérios propostos neste trabalho. Na Seção 4.5.2, é apresentado um estudo da complexidade dos critérios de teste propostos neste trabalho.

### 4.5.1 Análise de inclusão entre os critérios

O resultado da análise de inclusão de critérios é uma ordem parcial entre esses critérios. Um critério de teste $C_1$ inclui um critério de teste $C_2$ se, para qualquer fluxo de controle *G(UP)* (qualquer *Programa P*), um conjunto de *caminhos completos* $\Pi$ que seja

$C_1$ – *adequado* para $P$ é também $C_2$-*adequado* para $P$; ou seja, se $\Pi$ satisfaz $C_1$ também satisfaz $C_2$. Dizemos que um critério $C_1$ inclui estritamente um critério $C_2$ ($C_1 \Rightarrow C_2$), se $C_1$ inclui $C_2$ e, para algum grafo $G(UP)$, existe um conjunto de *caminhos completos* $\Pi$ de $G(UP)$ que satisfaz $C_2$, mas não satisfaz $C_1$. E dizemos que dois critérios são incomparáveis quando nem $C_1 \Rightarrow C_2$ e nem $C_2 \Rightarrow C_1$. Um critério de teste $C_{BD}$ é um critério que para ser satisfeito, exige um conjunto $\Pi$ de caminhos completos de $G(UP)$ e as mesmas tuplas utilizadas para exercitá-los. Nesse caso, um critério de teste $C$ para software convencional não inclui um critério $C_{BD}$ pois não é necessário a determinação de tuplas para satisfazer um critério de teste $C$.

Para simplificar as demonstrações adotou-se a notação $\boldsymbol{T_C}$, como sendo um conjunto de casos de teste; $C$ é um critério de teste para programas convencionais (FPU e FCFD); $C_{BD}$ é um critério exclusivo para Banco de Dados; $C_{BDint}$ é um critério de teste de integração (*intra-modular* e *inter-modular*); $\tau$ representa o conjunto de tuplas usadas na execução dos casos de testes ao exercitar os critérios $C_{BD}$ e $C_{BDint}$; $\boldsymbol{T_C}^{\tau}$ são os casos de testes controlados por tuplas $\tau$; $UP^d$ (que define) e $UP^u$ (que usa) representam as unidades que definem uma *variável tabela* e usam uma *variável tabela*, respectivamente; $\boldsymbol{t}$ é a *variável tabela*; $\Pi$ é um conjunto de caminhos completos e; $\pi$ é um caminho completo.

**Teorema 1:** Os critérios da FCPU são incomparáveis com os critérios *todos os t-usos e todos os dtu-caminhos*.

***Prova*:** Os critérios *todos os t-usos* e *todos os dtu-caminhos* exercitam as *variáveis persistentes*, existentes em programas de aplicação, não contempladas pelos critérios tradicionais. A *Figura 4.4* mostra um exemplo que não permite comparar os critérios da FCPU com os critérios gerados especialmente para exercitar as variáveis persistentes.

- *(todos os potenciais-usos)* $\not\Rightarrow$ *(todos os t-usos)*
- *(todos os t-usos)* $\not\Rightarrow$ *(todos os potenciais-usos)*

*i)* A primeira condição é verdadeira devido ao fato de que o critério *todos os t-usos* requer a mesma tupla para exercitar a *associação definição-t-uso* c.r.a $\boldsymbol{t}$ e o critério *todos os potenciais-usos* não.

*ii)* A segunda condição é verdadeira por existirem partes do programa que não envolvem nós de **SQL** e, sendo assim, o *critério todos os t-usos* não requer nenhum

elemento requerido nessas partes, enquanto os critérios *todos os potenciais-usos* requerem várias associações potenciais usos. No exemplo da Figura 4.8 os caminhos (1, 10, 11, 13, 14) e (1, 10, 12, 13, 14) são requeridos para satisfazer o critério *todos os potenciais-usos* e não são exigidos pelo critério *todos os t-usos* por não existir nenhum comando de manipulação de dados da **SQL**, nesta parte do programa que caracterize uma *definição persistente* de qualquer *variável tabela*.

*iii)* As condições *i*) e *ii)* se estendem aos critérios *todos os potenciais-du-caminhos* e *todos os dtu-caminhos*, respectivamente.

**Teorema 2:** O critério *todos os dtu-caminhos* inclui parcialmente o critério *todos os t-usos*. Na presença de caminhos não executáveis eles são incomparáveis. As Figuras 4.6 e 4.7 mostram as respectivas árvores de inclusão relacionadas a estes critérios.

**Prova:** Um conjunto de casos de testes $\boldsymbol{T_C}$ é $C_{BD}$ – adequado para P se existir um caminho executável que vai de um ponto do programa onde ocorre uma *definição persistente* c.r.a. ***t*** até um ponto do programa onde há um uso de ***t***. O que é preciso provar, neste caso, é que mesmo na presença de laços entre o par de nós $<\ell, i>$ onde ocorre a *definição persistente* até o arco *(j, k)* onde existe um uso de ***t*** em *j*, o critério *todos os dtu-caminho* cobre parcialmente o critério *todos os t-usos*.

*i)* Suponha que o *caminho completo* $(I, ..., n_1, n_2, ..., \ell, ..., i, ..., n_l, ..., n_l, n_s, ..., j, k, ..., F)$ satisfaz o critério *todos os t-usos*. Se existir um *caminho executável* que não entra no laço a partir de $\ell\,(I, ..., n_1, n_2, .. , \ell, ..., i, ..., n_l, n_s, ..., j, k, ..., F)$, que satisfaz o critério *todos os dtu-caminhos*, este caminho também irá satisfazer o critério *todos os t-usos*. Isto só é verdadeiro quando todos os caminhos são executáveis.

*ii)* No caso de existir mais de um caminho entre o ponto onde ocorreu a *definição persistente* $<\ell, i>$ até o arco *(j, k)*, o critério *todos os dtu-caminhos* requerem todos os caminhos usados para satisfazer o critério *todos os t-usos*, além de outros caminhos entre $<\ell, i>$ até o arco *(j,k)* ainda não executados. A Figura 4.5 mostra uma situação semelhante a esta.

*iii)* Os critérios de teste da FCPU exigem que as *associações potenciais usos* das variáveis que são definidas nos nós 1, 10, 11 e 12 sejam exercitadas pelo menos uma vez, enquanto o *critério todos os t-usos* (usado para exercitar as *variáveis tabela*), não

requer nenhuma associação relacionada a esses nós, uma vez que requer apenas associações de *variáveis persistentes* (que no caso de *variáveis tabela* são definidas e usadas apenas em nós da **SQL**). Por outro lado, o critério de teste *todos os t-usos* é incomparável ao critério *todos os potenciais-usos*, conforme mostra o exemplo da *Figura 4.4*, quando existem partes da estrutura (nós 10, 11, 12 e 13) do programa que são tratadas pelo critério *Todos os Potenciais Usos* e que não são tratadas pelo critério de teste *todos os t-usos* devido à ausência de comandos da **SQL** que contêm comandos de manipulação.

Em qualquer unidade de programa $UP^d$ onde há uma *definição persistente* de $\boldsymbol{t}$ e que possua um *caminho livre de definição persistente* do par de nós $\langle \ell, i \rangle$ até o nó de saída $n_{out}(UP^d)$, existe uma associação *definição-t-uso* com outra $UP^u$ com $\boldsymbol{t}$*-uso* e onde existe um caminho livre de *definição persistente* c.r.a. $\boldsymbol{t}$ do nó $n_{in}(UP^u)$ até o arco *(j, k)* onde ocorre o *t-uso*.

Em qualquer par de unidades de programa $UP^d$ e $UP^u$ que caracterizam uma ou mais *associações persistentes* c.r.a. $\boldsymbol{t}$, existe um conjunto de casos de testes $\boldsymbol{T_C}^{\tau}$ que é $C_{BDint}$ – adequado para $UP^D$ e $UP^U$ onde $C_{BDint}$ representa os critérios de testes *todos os t-usos-intra* e *todos os dtu-caminhos-intra*, se ambas as unidades pertencem ao mesmo Módulo ou representa os critérios *todos os t-usos-inter* e *todos os dtu-caminhos-inter*, se ambas as unidades são de Módulos distintos.

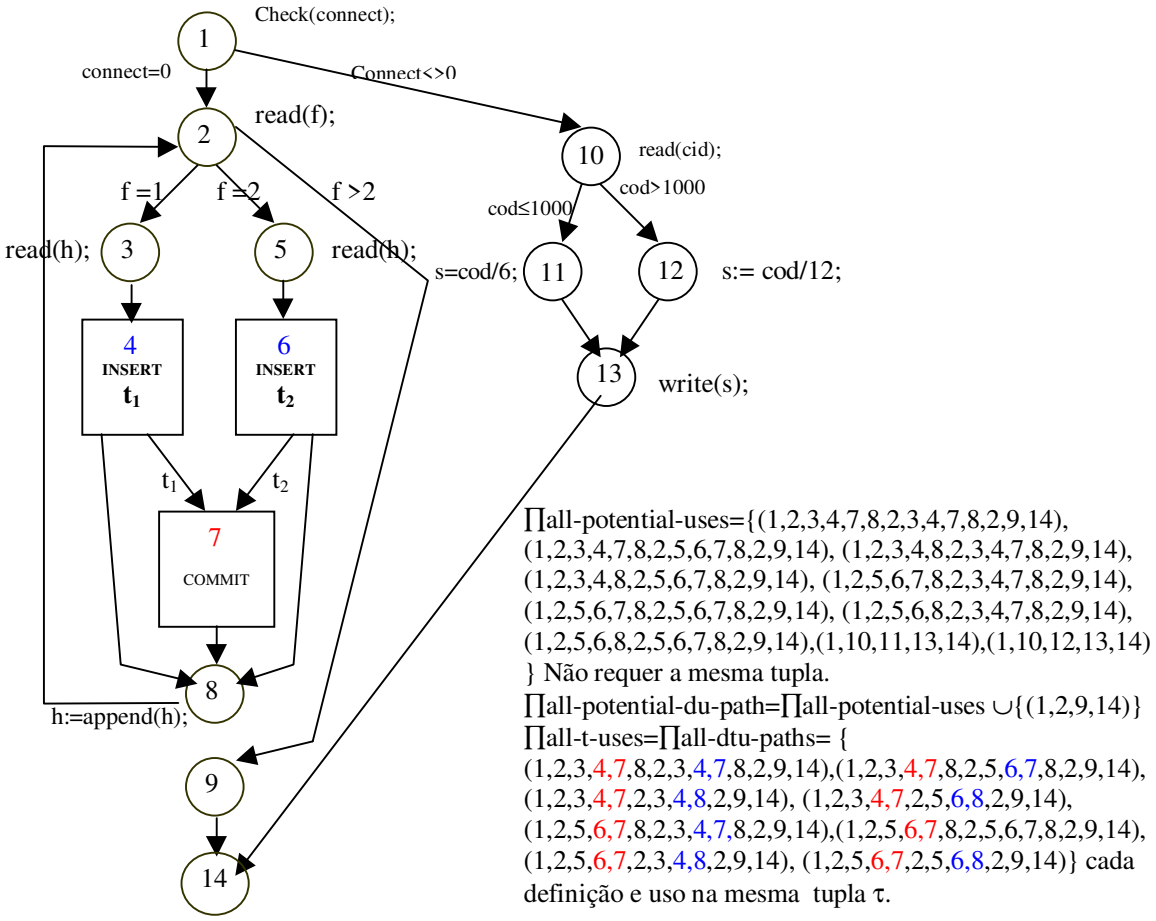



*Figura 4.4 – Exemplo 1 para análise de Inclusão.*

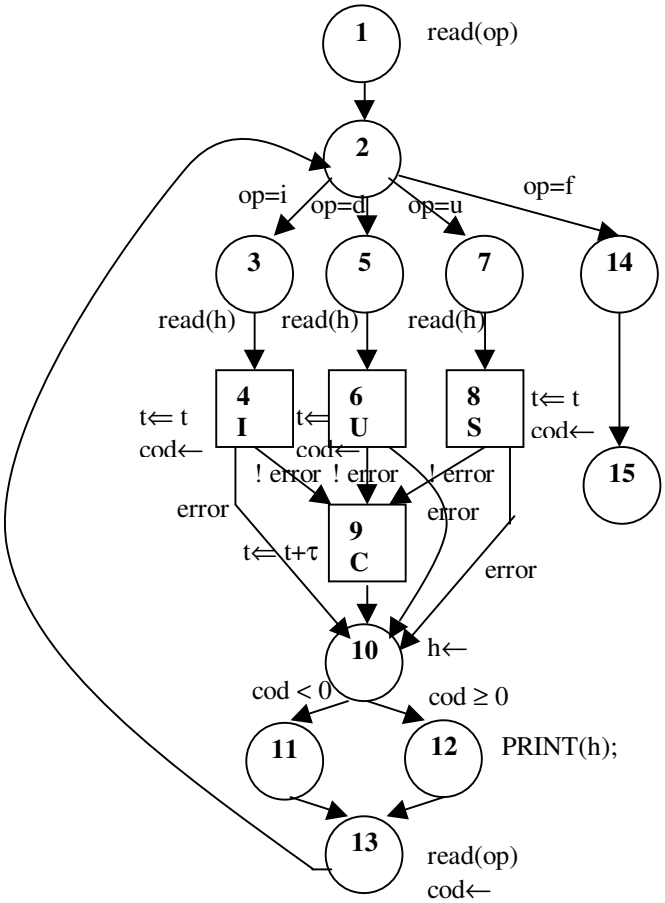



*Figura 4.5 Exemplo 2 para análise de inclusão.*

**OBS**: Os nós 4, 6 e 8 possuem respectivamente as letras I, U e S para representar os comandos *Insere*, *Update* e *Select* da **SQL**.

∏all-potential-t-uses:
{(1,2,3,4,9,10,12,13,2,3,4,9,10,12,13,2,14,15),
(1,2,3,4,9,10,12,13,2,3,4,10,11,13,2,14,15),
(1,2,3,4,9,10,12,13,2,5,6,9,10,12,13,2,14,15),
(1,2,3,4,9,10,12,13,2,5,6,10,11,13,2,14,15),
(1,2,3,4,9,10,12,13,2,7,8,9,10,12,13,2,14,15),
(1,2,3,4,9,10,12,13,2,7,8,10,11,13,2,14,15),

**PRINT(**

(1,2,5,6,9,10,12,13,2,3,4,9,10,12,13,2,14,15),
(1,2,5,6,9,10,12,13,2,3,4,10,11,13,2,14,15),
(1,2,5,6,9,10,12,13,2,5,6,9,10,12,13,2,14,15),
(1,2,5,6,9,10,12,13,2,5,6,10,11,13,2,14,15),
(1,2,5,6,9,10,12,13,2,7,8,9,10,12,13,2,14,15),
(1,2,5,6,9,10,12,13,2,7,8,10,11,13,2,14,15),
(1,2,7,8,9,10,12,13,2,3,4,9,10,12,13,2,14,15),
(1,2,7,8,9,10,12,13,2,3,4,10,11,13,2,14,15),
(1,2,7,8,9,10,12,13,2,5,6,9,10,12,13,2,14,15),
(1,2,7,8,9,10,12,13,2,5,6,10,11,13,2,14,15),
(1,2,7,8,9,10,12,13,2,7,8,9,10,12,13,2,14,15),
(1,2,7,8,9,10,12,13,2,7,8,10,11,13,2,14,15)}

∏all-potential-dtu-paths=∏all-potential-t-uses ∪
{(1,2,3,4,9,10,11,13,2,3,4,9,10,12,13,2,14,15),
(1,2,3,4,9,10,11,13,2,3,4,10,11,13,2,14,15),
(1,2,3,4,9,10,11,13,2,5,6,9,10,12,13,2,14,15),
(1,2,3,4,9,10,11,13,2,5,6,10,11,13,2,14,15),
(1,2,3,4,9,10,11,13,2,7,8,9,10,12,13,2,14,15),
(1,2,3,4,9,10,11,13,2,7,8,10,11,13,2,14,15),
(1,2,5,6,9,10,11,13,2,3,4,9,10,12,13,2,14,15),
(1,2,5,6,9,10,11,13,2,3,4,10,11,13,2,14,15),
(1,2,5,6,9,10,11,13,2,5,6,9,10,12,13,2,14,15),
(1,2,5,6,9,10,11,13,2,5,6,10,11,13,2,14,15),
(1,2,5,6,9,10,11,13,2,7,8,9,10,12,13,2,14,15),
(1,2,5,6,9,10,11,13,2,7,8,10,11,13,2,14,15),
(1,2,7,8,9,10,11,13,2,3,4,9,10,12,13,2,14,15),
(1,2,7,8,9,10,11,13,2,3,4,10,11,13,2,14,15),
(1,2,7,8,9,10,11,13,2,5,6,9,10,12,13,2,14,15),
(1,2,7,8,9,10,11,13,2,5,6,10,11,13,2,14,15),
(1,2,7,8,9,10,11,13,2,7,8,9,10,12,13,2,14,15),
(1,2,7,8,9,10,11,13,2,7,8,10,11,13,2,14,15)}
Requer que a mesma tupla seja usada na definição e no uso c.r.a. ***t***

**Teorema-3:** O critério de integração *todos os dtu-caminhos-intra (dtu-caminhos-inter)* inclui parcialmente o critério *todos os t-usos-intra (todos os t-usos-inter)*. Na presença de caminhos não executáveis eles são incomparáveis. A Figura 4.8 mostra a árvore de inclusão relacionada a estes critérios.

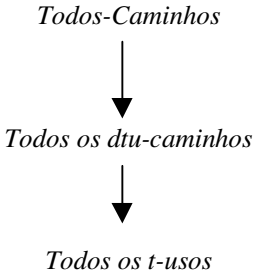



*Figura 4.6 - Árvore de inclusão parcial dos critérios de teste de unidade de ABDR*

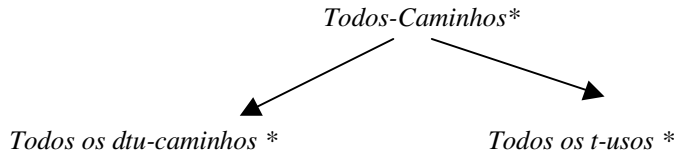



*Figura 4.7 Árvore de inclusão dos critérios de teste de unidade de ABDR para caminhos executáveis*

***Prova***: Um conjunto de casos de testes controlados pela mesma tupla $\tau$ ($\boldsymbol{T_C}^{\tau}$) é $C_{BDint}$ – adequado para $UP^d \times UP^u$ se existe um *caminho executável* que vai do par de nós $<\ell,\ i>$ da unidade $UP^D$ onde ocorre a *definição persistente* c.r.a. ***t***, até o nó final (em $UP^d$ ) e existe um ou mais caminhos executáveis que vai do nó de entrada $n_{in}(UP^u)$ até o arco *(j, k)* onde ocorre um ***t****-uso*. O que é preciso provar é que, mesmo na presença de laços entre o par de nós $<\ell,\ i>$ onde ocorre a *definição persistente* em $UP^d$ até o arco onde ocorre um ***t***-uso em $UP^u$ (arco *(j, k)*) o critério *todos os dtu-caminhos* cobre parcialmente o critério *todos os t-usos* (de integração). Assim como os resultados obtidos no Teorema 2, na associação, o que altera é apenas a concatenação dos caminhos $\pi_d$ e $\pi_u$ resultantes da execução das unidades $UP^d$ e $UP^u$.

*i)* Suponha que os pares de caminhos $(..., \ell, ..., i, ..., nl, nl+1, ..., nl, ns, ..., n_{out})^d \times (n_{in}, ..., j, k)^u$ ou $(..., \ell, ..., i, ..., n_{out})^d \times (n_{in}, ..., nl, nl+1, ..., nl, ns, ..., j, k)^u$ satisfazem o

critério *todos os t-usos-intra (todos os t-usos-inter)* eles satisfazem o critério *todos os dtu-caminhos-intra (todos os dtu-caminhos-inter)* se existe um caminho executável que não pertença ao laço. Neste caso, este caminho também satisfaz o critério *todos os t-usos-intra (todos os t-usos-inter)*. Portanto, o critério *todos os dtu-caminhos-intra (todos os dtu-caminhos-inter)* cobre parcialmente o critério *todos os t-usos-intra (todos os t-usos-inter)*.

*ii)* A Figura 4.8 mostra um caso onde o critério *todos os dtu-caminhos-intra (todos os dtu-caminhos-inter)* cobre o critério *todos os t-usos-intra (todos os t-usos-inter)*, mas o critério *todos os t-usos-intra (todos os t-usos-inter)* não cobre o critério *todos os dtu-caminhos-intra (todos os dtu-caminhos-inter)*.

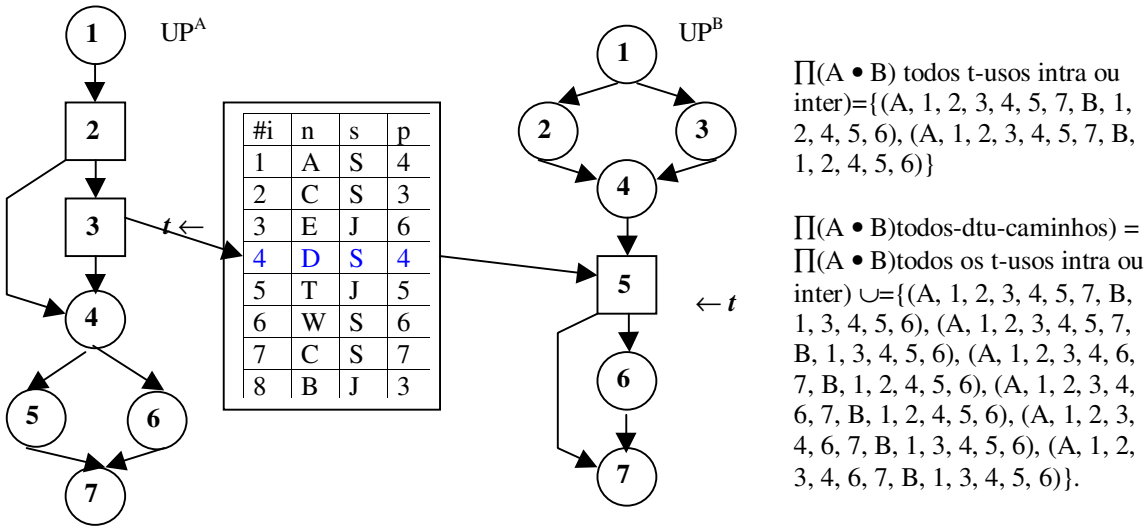



*Figura 4.8 Exemplo para análise de inclusão dos critérios (todos os t-usos-intra ou inter)x(todos os dtu-caminhos-intra ou inter)*

Tendo em vista que os critérios *todos os t-usos-intra* e os critérios *todos os dtu-caminhos-intra* exigem que as associações sejam exercitadas com a mesma tupla da base de dados e que o critério da FCPU não possui a mesma exigência, foram observados no caso *i)* apenas os caminhos percorridos; caso contrário, os casos de testes gerados no teste de unidade com relação aos critérios *todos os t-usos* não seriam satisfeitos.

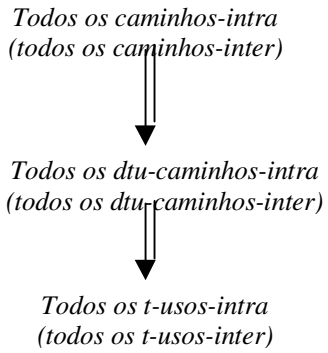



*Figura 4.9 Arvore de inclusão dos critérios de integração. O critério todos os dtu-caminhos (intra ou inter) inclui parcialmente o critério todos os t-usos ( intra ou inter)*

### 4.5.2 Análise de Complexidade

A complexidade de um critério C é definida como o número de casos de teste requeridos pelo critério no pior caso, para exercitar todos os elementos requeridos pelo critério de teste. Dado um Módulo de Programa $Mod_i$, existe um conjunto de casos de teste $\boldsymbol{T_C}$ tal que a cardinalidade de $\boldsymbol{T_C}$ é menor ou igual à complexidade do critério C. Deve-se considerar a distribuição de ocorrência de variáveis em qualquer grafo de fluxo de controle de uma Unidade de Programa *G(UP)*. Os critérios da FCPU foram analisados em [MAL91]; os resultados mostram que todos os critérios baseados em fluxo de dados têm complexidade de ordem exponencial e, portanto, do ponto de vista teórico, estes critérios não seriam aplicáveis na prática. No entanto, estudos empíricos têm demonstrado que, para programas reais, esses critérios demandam um pequeno número de casos de teste. A complexidade de um critério é um fator importante para as atividades de teste, principalmente na estimativa e determinação de custos associados a essas atividades. As análises são sempre focadas nos piores casos, embora na prática seja muito improvável sua ocorrência. No caso dos critérios de testes para programas de ABDR, apresentou-se alguns grafos de fluxo que ilustram a complexidade dos critérios de teste *todas as associações-definição-t-usos* e *todos os dtu-caminhos* para a mesma *tupla* de ***t***.

A complexidade de um critério é analisada a partir de uma estrutura de fluxo de controle, considerando o fluxo de dados que maximiza o número de elementos requeridos pelo critério. O grafo de fluxo de controle apresentado na Figura 4.10 maximiza o número

de operações da **SQL** para cada tabela, aumentando o número de associações *definição-t-usos* requeridas. Considere que temos ***n*** tabelas e um comando requer 3*4 associações para cada tabela, resultando em ***n*3*4*** elementos requeridos pelos critérios *todos os t-usos* e *todos os dtu-caminhos* (onde 3 representa o número de comandos da **SQL** que geram uma definição de ***t*** e 4 representa o número de comandos que usam as variáveis ***t***). Para cada definição de ***t*** os critérios *todos os dtu-caminhos* e *todos os t-usos* associam apenas os nós que realmente possuem um uso de ***t*** e não requerem associações nos nós onde a variável tabela ***t*** não estiver sendo relacionada. Em exemplos de [MAL91], programas com estruturas de controle consistindo de seqüências de comandos IF-THEN-ELSE requerem $2^k$ casos de teste. No caso de programas de aplicação é possível aplicar comandos de **SQL** em comandos IF-THEN-ELSE exigindo um número bem acentuado de elementos requeridos para satisfazer os critérios *todos os dtu-caminhos* e *todos os t-usos,* de modo a requerer $2^k$ casos de testes para satisfazer esses critérios, sendo *k* o número de comandos de decisão na *UP*.

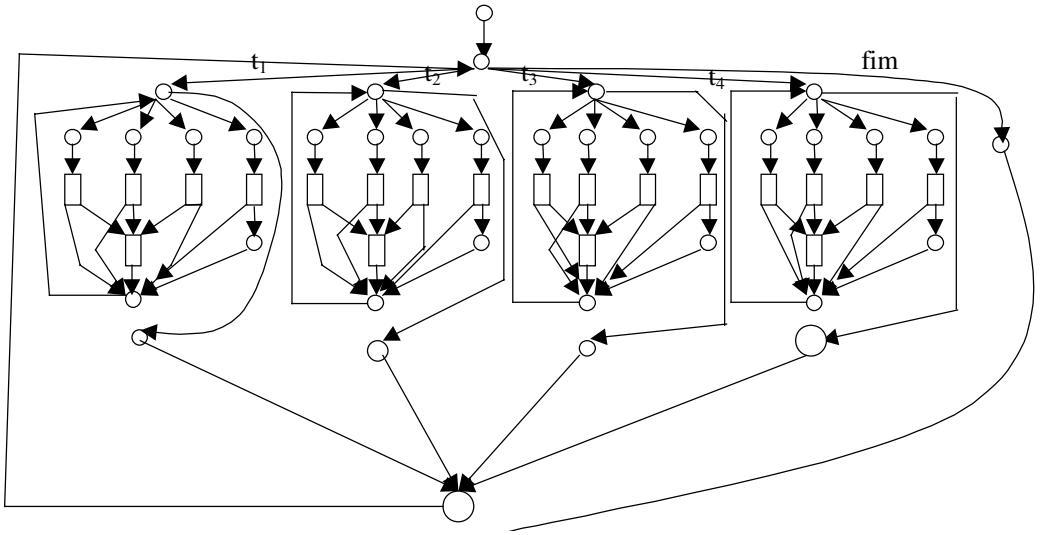



*Figura 4.10: Grafo de fluxo de controle que maximiza o número de combinações do critério todos os t-usos.*

A *Figura 4.11* mostra uma *UP* que possui *k* comandos de decisão e que maximiza o número de casos de teste para satisfazer os critérios de teste estrutural para programas de ABDR.

Os critérios de teste de integração *todos os t-uso*s (*intra-modular* e *inter-modular*) extraem seus elementos requeridos das possíveis combinações existentes entre os comandos

da **SQL** (de manipulação da base de dados) c.r.a *variável tabela*. Sendo assim, existe um fator relevante que se reflete diretamente na complexidade do critério; este fator está relacionado à quantidade de tabelas envolvidas na Aplicação. Geralmente, quanto mais tabelas existirem maior será o número de comandos de manipulação da **SQL** que contribuem diretamente com o acréscimo da quantidade de elementos requeridos em cada critério de integração.

A complexidade do critério *todos os t-usos*, a partir desse tipo de estrutura, é dada pela soma de todas as combinações possíveis que vão de uma definição até um ***t-uso***, alcançado por um caminho livre de *definição persistente* e livre de laço.

Neste caso a complexidade é dada pela equação:

$$[\boldsymbol{n}*4*2]*\boldsymbol{m}$$

sendo ***m*** o número de tabelas associadas em cada opção; ***n*** o número de comandos que possuem uma definição de ***t*** (no pior caso, foram considerados os três comandos juntos); 4 representa o número de distintos comandos executáveis da **SQL** que possuem um uso de ***t*** e 2 representa o *uso* nos arcos de saída dos nós da **SQL** (no pior caso) que contêm um ***t-uso***. Ao invés de exigir-se uma potencial associação como ocorre nos critérios FCPU, foi exigido apenas a *associação definição-t-uso* entre os comandos que usam a mesma *variável tabela*, evitando-se uma explosão de elementos requeridos não executáveis (devido à exigência de executar a associação com a mesma *tupla*).

Na estrutura apresentada na Figura 4.11, para cada nó que contém um comando *if-then-else* existe uma entrada de dados que decide a execução dos laços *then* ou *else* do comando *if*. Em cada laço *then* e *else* é colocada uma estrutura que *define* e *usa* uma ou mais tabelas diferentes ou, no pior caso, todas as tabelas usando a estrutura *case* mostrada na *Figura 4.11*; com essa combinação de estruturas, temos o maior número de elementos requeridos pelo critério de teste *todos os t-usos*, $[(3*4*2)*m*2^k]$, onde $m$ é o número de tabelas envolvidas na estrutura do programa e $k$ é o número de comandos *if-then-else*.

A complexidade deste critério é de ordem exponencial em relação ao número de caminhos de decisão como os demais critérios da FCPU [MAL91].

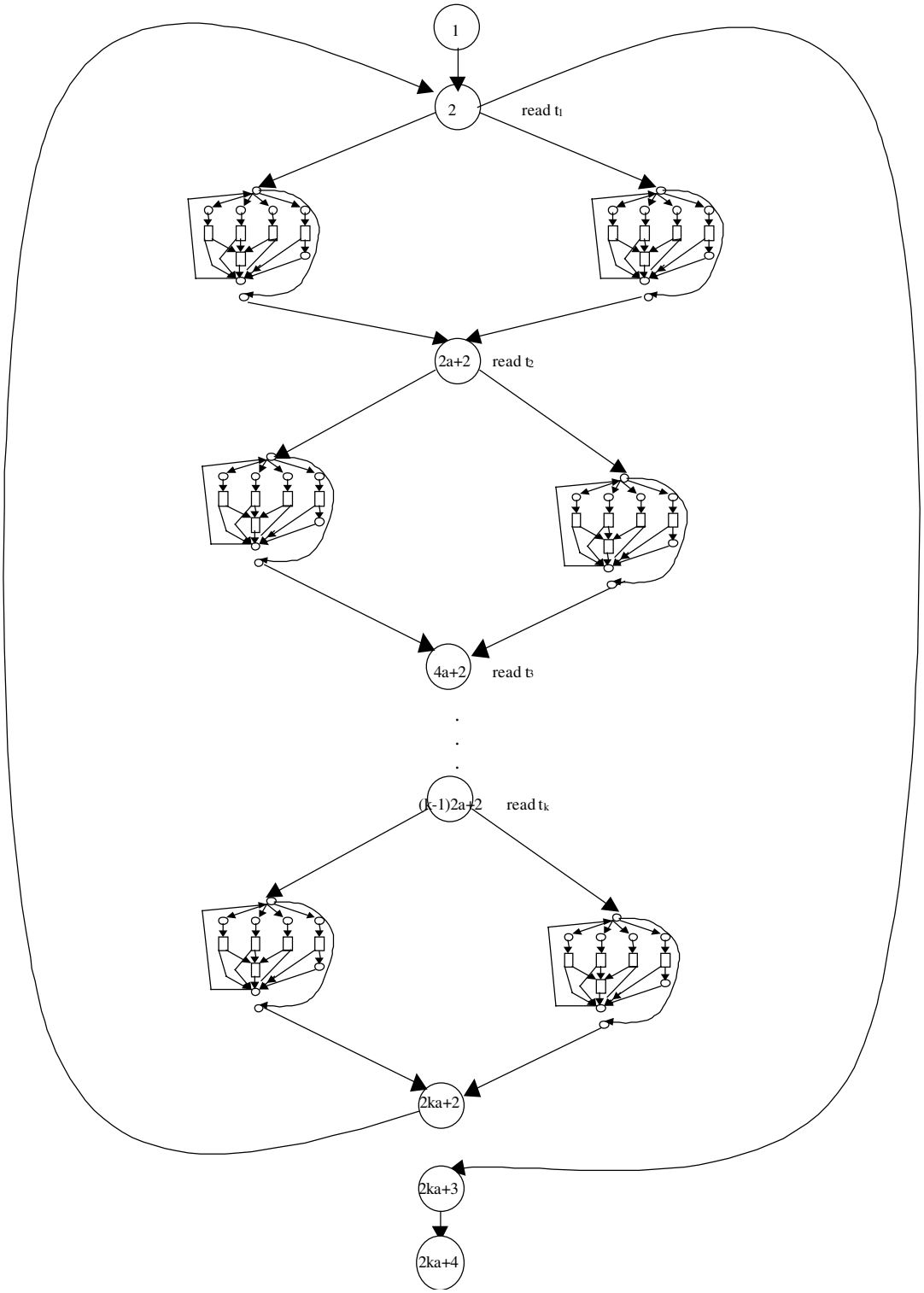



*Figura 4.11: Grafo de fluxo de controle com estrutura que maximiza o critério todos os t-usos, onde **a** é o número de nós em cada estrutura e **k** é o número de comandos if-then-else.*

O mesmo ocorre para os critérios de teste de integração *intra-modular e inter-modular* que, apesar de serem associados a duas unidades de cada vez e terem algumas diferenças em relação aos critérios de teste de unidades estudados em [MAL88 e MAL91], têm estruturas de grafo de programa com números de casos de teste que alcançam a ordem exponencial no pior caso. Assim, os critérios *todos os t-usos-intra (todos os t-usos-inter)* e *todos os dtu-caminhos-intra (todos os dtu-caminhos-inter)* também têm complexidade de ordem $O(2^k)$, onde $k$ representa o número de comandos de decisão que exigem uma entrada de dados nas unidade de programa onde ocorre a definição, $UP^D$, e o uso de ***t***, $UP^U$. A partir dessas *UPs* são formadas as possíveis combinações de pares *definição-uso persistentes* c.r.a. ***t***; na prática, o número de combinações pode não ser de ordem exponencial, tornando-se assim viável no caso geral. No caso de ABDR, tratando-se de associações que relacionam apenas nós de **SQL** com a mesma *variável tabela* ***t*** e que sejam alcançáveis por um caminho *livre de definição* c.r.a. ***t***, acredita-se que o número de combinações não atinja ordem exponencial.

Um estudo mais detalhado pode ser realizado em trabalhos futuros visando a estudar outros fatores que podem elevar a ordem de complexidade dos critérios de teste estrutural de programas de ABDR; por exemplo, a quantidade de tabelas envolvidas, o número de Módulos existentes, entre outros, que afetam diretamente a geração dos elementos requeridos dos critérios propostos neste trabalho.

## 4.6 Considerações Finais

Neste capítulo foi apresentada a definição dos critérios de teste estrutural para programas de aplicação para o teste de unidade e teste de integração *intra-modular* e integração *inter-modular* que requerem a mesma *tupla* para satisfazer a associação *definição-uso* c.r.a. ***t***. Uma característica marcante desses critérios é a abordagem de integração entre duas unidades de programas sem a existência de um comando de chamada entre elas.

Foi apresentada uma análise de inclusão entre os critérios de teste propostos. Tendo em vista que esses critérios exercitam as *variáveis tabela*, a análise foi feita apenas para os trechos de programas cujo caminho inclui algum nó da **SQL**, atendendo ao objetivo

proposto inicialmente (que era o de complementar os critérios de teste estrutural de programas convencionais).

Em relação à análise de complexidade foi mostrado que todos os critérios de teste baseados em fluxo de dados têm complexidade de ordem exponencial e, do ponto de vista teórico, estes critérios não seriam aplicáveis na prática; entretanto, estudos empíricos podem mostrar que esses critérios demandam um pequeno número de casos de teste. Para os programas de aplicação, maximizar o número de comandos de **SQL** pode maximizar o número de associações definição *t-uso* c.r.a. ***t***, aumentando assim a complexidade; na prática, isso não é comum ocorrer.

A análise de complexidade dos critérios envolvidos na etapa de integração deve ser reavaliada em trabalhos futuros buscando, assim, uma estimativa mais adequada para os casos reais; embora o pior caso seja exponencial, no caso de programas de ABDR não são apenas os comandos de decisão que contribuem com esta ordem de grandeza, mas também o número de *variáveis tabela* envolvido no relacionamento entre as unidades de um ou mais módulos da aplicação. Esse estudo tem a importância de mostrar que apesar da ordem exponencial dos critérios de testes, na prática é viável a sua utilização. Isso pode ser mostrado através de experimentos.

# 5 Os Modelos de Implementação dos Critérios de Teste

Neste capítulo são abordados vários aspectos relacionados ao projeto da ferramenta de suporte à aplicação dos Critérios de Teste em ABDR. Na Seção 5.2 é descrito o modelo de fluxo de controle; na Seção 5.3 são descritos os modelos de instrumentação; na Seção 5.4 são descritos os modelos de fluxo de dados; e na Seção 5.5 discute-se a descrição dos elementos requeridos.

## 5.1 Considerações Iniciais

Uma proposta de modificação da ferramenta POKE-TOOL é apresentada, visando à inclusão dos critérios de integração propostos neste trabalho. A aplicação de um critério de teste estrutural envolve a análise de cobertura para avaliar o grau de satisfação do critério, obtida pela execução dos casos de teste. A análise de cobertura é feita através da instrumentação de cada unidade do programa em teste, pela monitoração dos caminhos executados pelo conjunto de casos de teste e pela determinação dos elementos requeridos executados e não executados. Para os critérios específicos para banco de dados, a cobertura é analisada a partir dos caminhos percorridos e das *tuplas* que exercitam as associações *definição-t-uso*. Para isso, necessita-se incluir *pontas de provas* que identifiquem o número da Unidade de Programa em teste e das *tuplas* que foram utilizadas para exercitar as unidades de programa envolvidas no teste. Essa instrumentação é utilizada para avaliar a cobertura dos critérios baseados na dependência de dados entre as *variáveis persistentes* (nas etapas de teste *intra-modular*).

Neste capítulo são apresentados os modelos propostos para a implementação de uma ferramenta de apoio ao teste de programas de ABDR. No modelo de fluxo de controle apresentou-se os comandos da **SQL** que ocasionam fluxos distintos de controle e também apresentou-se uma extensão da Linguagem Intermediária (LI) para os comandos **SQL**. A LI é uma linguagem utilizada na ferramenta POKE-TOOL para torná-la *multi-linguagem*, ou seja, parte do processamento da ferramenta é feito sobre a LI e parte sobre a linguagem do programa testado [CHA91].

A POKE-TOOL possui características que permitem a incorporação de novos critérios de teste estrutural englobando os quatro modelos: Fluxo de Controle,

Instrumentação, Fluxo de dados e Descrição dos Elementos Requeridos [MAL91]. Os modelos básicos necessários para os nossos critérios: O Modelo de Grafo de Fluxo de Controle; O Modelo de Instrumentação; O Modelo de Dados; e O Modelo de Descrição dos Elementos Requeridos, são discutidos a seguir.

## 5.2 O Modelo de Fluxo de Controle

No modelo de fluxo de controle adotado, um programa **P** é representado por um grafo dirigido ***G(UP)*** = ($N^{BD}$, *E*, $n_{in}$, $n_{out}$), definido no Capítulo 3 como grafo de fluxo de controle ou grafo de programa. Cabe lembrar que os nós de $N_h$ representam blocos de comandos da linguagem hospedeira e que os comandos declarativos da **SQL** são colocados juntos com esses blocos de comandos. Os comandos executáveis da **SQL** são representados pelos nós $N_S$ e são colocados um em cada nó.

Maldonado [MAL91] apresenta as construções básicas dos comandos da linguagem LI. Diferentemente, na **SQL** o fluxo de controle dos comandos executáveis é definido pelo comando declarativo WHENEVER ***<condição>*** ***<ação>***, no qual pode-se especificar uma ação a ser tomada quando for detectada uma condição de "*error*", uma condição de "*warning"* ou uma condição de "*not found*" em um comando executável da **SQL**. As ações incluem condições de: parada (*stop*), continuação (*continue*), desvio condicional (*goto <rótulo>*) ou execução de uma função (*do <function>*). Os SGBDs utilizam esses comandos para tratamento de erros nos comandos executáveis da **SQL;** assim sendo, o comando declarativo WHENEVER deve anteceder os comandos executáveis da **SQL**. É importante observar que o comando WHENEVER tem sua ação controlada de cima para baixo com relação às linhas de comandos do programa fonte; assim, o esquecimento de um comando declarativo antes de um comando executável da **SQL** pode causar um erro de fluxo. A Figura 5.1 apresenta a construção básica do grafo de fluxo de controle para o comando WHENEVER.

A linguagem LI representa os comandos executáveis da linguagem **SQL**. Um grafo de programa de uma ABDR é formado pela simples concatenação das construções básicas apresentadas em [MAL91] para os comandos da linguagem hospedeira e das construções básicas dos comandos executáveis da linguagem **SQL**. Observe que o modelo de fluxo de

controle é importante para refletir os aspectos semânticos da LI e para a instrumentação das unidades em teste.

As condições podem ser SQLWARNING, SQLERROR ou NOT FOUND sendo que cada tipo de *condição* pode gerar informações de controle nos comandos executáveis da **SQL** durante a execução. Essas informações são passadas para as variáveis correspondentes, que compõem a área de comunicação da **SQL** (SQLCA). Conforme mostrado na Figura 5.1, existem quatro tipos de *ações* que determinam o fluxo de controle após cada comando executável da **SQL**: CONTINUE, STOP, GOTO <*rótulo*> e DO <*função*>. A *ação* CONTINUE faz com que o fluxo passe para o próximo comando do programa independentemente do resultado obtido na condição adotada, conforme mostra a Figura 5.2. A *ação* STOP faz com que o fluxo de controle após o comando executável da **SQL** tenha dois arcos, um apontando para o próximo comando do programa indicando que a condição não foi satisfeita (FALSE) e outro para o final do programa, caso a condição tenha sido satisfeita (TRUE), mostrado na Figura 5.2. A ação GOTO <*rótulo*> faz com que existam dois fluxos diferentes após o comando executável da **SQL** sendo um para o próximo comando do programa, caso a condição não tenha sido satisfeita (FALSE) e o outro para o ponto do programa onde o rótulo foi colocado, veja a Figura 5.2. A *ação* DO <*função*> faz com que o fluxo de controle após um comando executável da **SQL**, tenha um

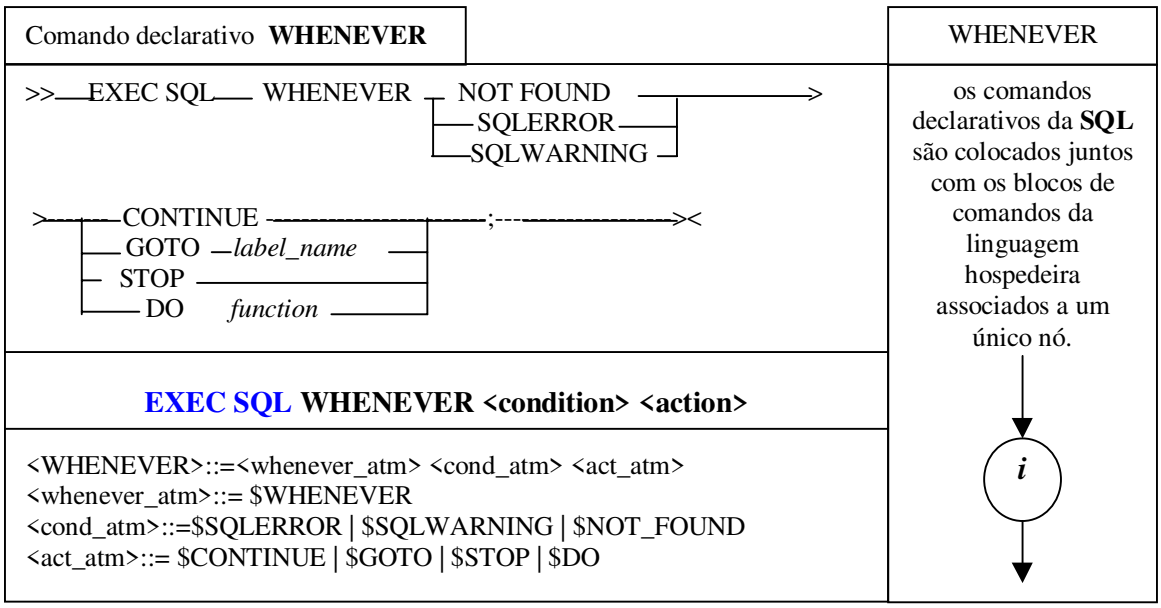



Figura 5.1 Modelo de fluxo de controle e instrumentação associado aos comandos da *SQL*: comando declarativo de controle de erros WHENEVER

arco para um comando de chamada da função, caso a condição tenha sido satisfeita (TRUE), ou para o próximo comando, caso a condição não tenha sido satisfeita (FALSE),

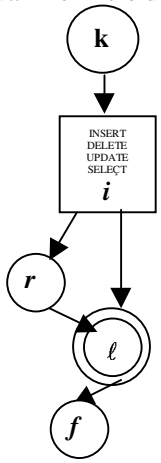
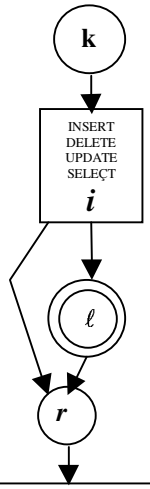
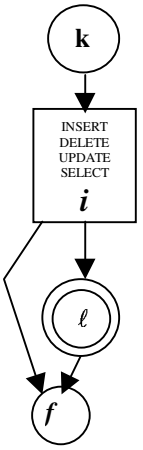
| ***DO*** *<função>* | *GOTO <label>* | **STOP** | **CONTINUE** |
|---|---|---|---|
| **k** representa um subgrafo que antecede o comando executável SELECT. Seja *s* um nó que contém um comando declarativo da **SQL** cuja ação é DO <FUNC>. *Instrumentação*: Antes do nó *i* (que contém um comando executável da **SQL**) é inserida uma ponta de prova *i;* seja *j* o nó de entrada do subgrafo ℓ, é inserida uma ponta de prova j no início do subgrafo ℓ; e seja *r* o nó que contém o comando de chamada da função FUNC colocada na ação DO, é colocada uma ponta de prova *r* no início deste nó.   | ***k representa um subgrafo que antecede o comando executável da SQL. Seja s um nó que contém um comando declarativo da SQL cuja ação é GOTO <label>. Instrumentação: Antes do nó i (que contém um comando executável da SQL) é inserida uma ponta de prova i; seja j o nó de entrada do subgrafo ℓ, é inserida uma ponta de prova j no início do subgrafo ℓ, e seja r o nó que contém o rótulo da ação GOTO, é colocada uma ponta de prova r no início deste nó.***   | **k** representa um subgrafo que antecede o comando executável da **SQL**. Seja *s* um nó que contém um comando declarativo da **SQL** cuja ação é STOP. *Instrumentação*: Antes do nó *i* é inserido a ponta de prova *i;* Seja *j* o nó de entrada do subgrafo ℓ, é inserida uma ponta de prova j no início do subgrafo ℓ e é inserida uma ponta de prova *f* no início do nó final.   | **k** representa um subgrafo que antecede o comando executável da **SQL**. Seja *s* um nó que contém um comando declarativo da **SQL** cuja ação seja CONTINUE. *Instrumentação*: Antes do nó *i* é inserida a ponta de prova *i;* Seja *j* o nó de entrada do subgrafo ℓ, é inserida a ponta de prova *j* no início do nó *j*.   |

*Figura 5.2*: O modelo de fluxo de controle para os comandos executáveis da **SQL** gerado pela ação do comando declarativo WHENEVER. Na descrição é apresentado o Modelo de instrumentação referente a cada ação nos comandos executáveis da **SQL**.

mostrada na Figura 5.2.

As ações utilizadas nos comandos declarativos de tratamento de erros que determinam as regras de fluxo de controle a partir dos comandos executáveis da **SQL** e os comandos de desvios condicionais e incondicionais apresentados em [MAL91] introduzem grandes modificações no fluxo de controle e, conseqüentemente, no modelo de instrumentação de uma ABDR mostrada na Figura 5.2. Para cada nó da **SQL** executável

coloca-se uma *ponta_de_prova* (uma informação do número do nó) antes do nó da **SQL**, marcando o número do nó da **SQL** (válido para todos nós da **SQL**) e, para cada ação existe um arco de saída do nó da **SQL** para representar a respectiva ação. Para a ação GOTO <*rótulo*> existe um arco que liga o nó da **SQL** ao próximo nó (ou bloco) (para representar a *condição* = *false*) e outro arco para o nó (bloco) que contém o *rótulo* associado à última ação GOTO (para representar a *condição* = *true*) (Figura 5.2). Da mesma forma, para a ação STOP existe um arco saindo do nó da **SQL** que contém um comando executável da DML, para o último nó (bloco) do programa e o outro para o próximo bloco do programa (próximo nó) (mostrado pela Figura 5.2). Para a ação CONTINUE existe apenas um arco de saída do nó da **SQL,** que contém um comando executável da DML, para o próximo nó (bloco) (mostrado pela Figura 5.2). Para a última ação DO <função> é adotada a mesma estratégia do comando IF_THEN onde, no bloco THEN existe uma chamada de função (a função colocada na ação DO <*função*>) e aí é colocada uma *ponta_de_prova* marcando o número deste nó e, caso a condição não seja satisfeita, há um arco para o próximo comando na seqüência do programa após o comando **SQL** onde existe uma *ponta_de_prova* no início do bloco. Neste caso, além da *ponta_de_prova* colocada antes do nó, é colocada outra *ponta_de_prova* do próximo número do nó executável dentro da função usada na ação e outra *ponta_de_prova* logo após o nó da **SQL** para marcar o bloco (caso seja um *if_then*) (mostrado pela Figura 5.2).

As Figuras 5.3 a 5.6 mostram a sintaxe parcial dos comandos executáveis da DML, o comando INSERT na Figura 5.3, o comando DELETE na Figura 5.4, o comando UPDATE na Figura 5.5 e o comando SELECT na Figura 5.6. A proposta aqui apresentada para a instrumentação foi feita manualmente para exercitar o exemplo de aplicação que é mostrado no próximo capítulo. Como trabalho futuro pretende-se transformar a instrumentação uma operação automatizada e segura quanto ao acréscimo das informações.

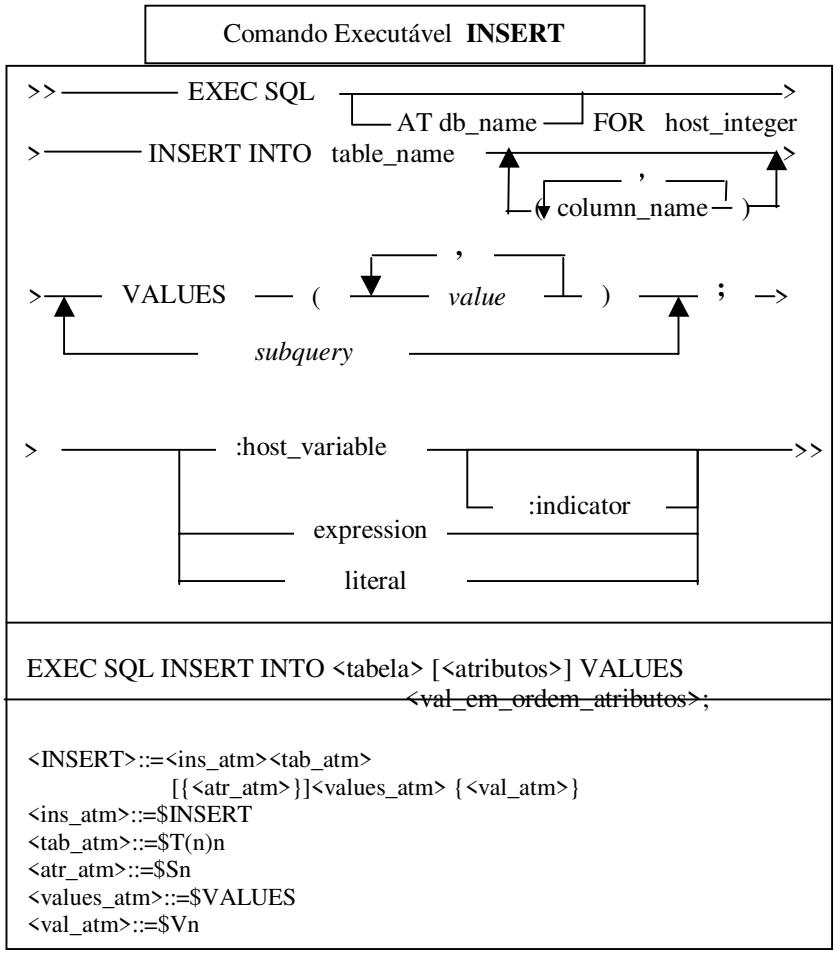



*Figura 5.3 Sintaxe parcial e LI para o comando INSERT.*

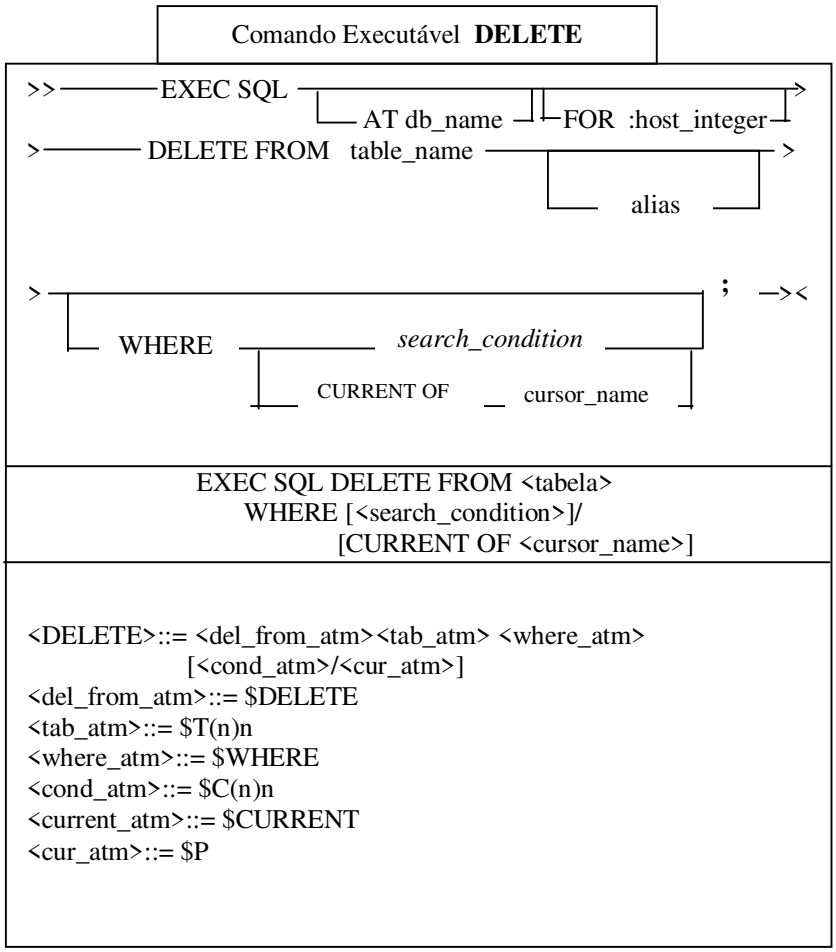



*Figura 5.4 Sintaxe parcial e LI para o comando DELETE..*

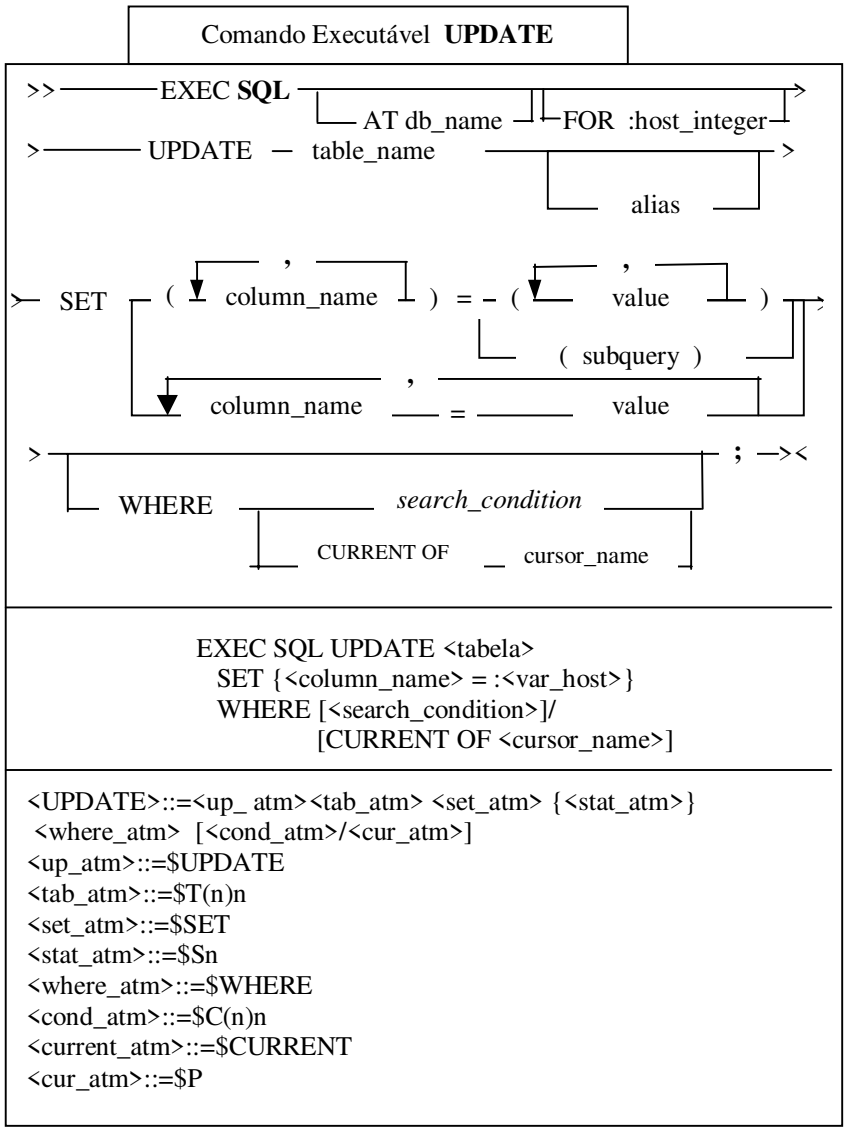



*Figura 5.5 Sintaxe parcial e LI para o comando UPDATE.*

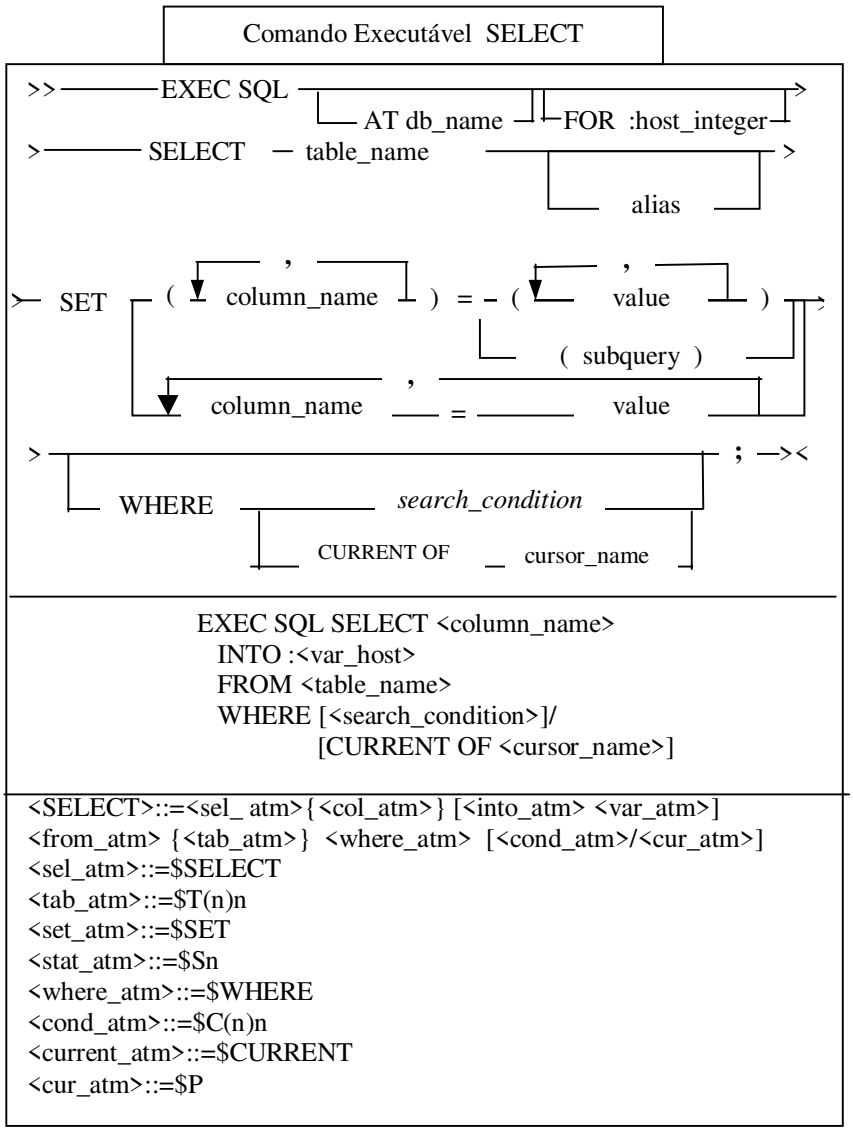



*Figura 5.6 Sintaxe parcial e LI para o comando SELECT.*

A seguir é apresentado um modelo de instrumentação para o teste estrutural de programas de aplicação. A instrumentação tem como objetivo dar condições necessárias para a avaliação dos critérios *intra-modular* e *inter-modular*.

## 5.3 Os Modelos de Instrumentação

A instrumentação do programa original tem como objetivo capturar informações sobre a execução de casos de teste para a análise de cobertura dos elementos requeridos

pelos diversos critérios de teste. No caso de programas de aplicação, a instrumentação é dividida em duas etapas: *i)* modificar o programa fonte inserindo comandos e informações, gerando uma nova versão do programa, usualmente denominada de unidade instrumentada para o teste de Unidade; *ii)* a outra etapa de instrumentação captura informações referentes à identificação das unidades em teste e sobre as tuplas envolvidas na execução dos comandos da **SQL**, para os testes *intra-modular* e *inter-modular*. No caso da POKE-TOOL, a unidade instrumentada é armazenada no arquivo denominado TESTEPROG.C. Os detalhes e exemplos são mostrados no Apêndice C.

A instrumentação consiste, essencialmente, em inserir *pontas de prova* para indicar o número de cada bloco de comandos da linguagem hospedeira ou um comando executável da linguagem **SQL**, possibilitando a identificação dos caminhos executados pelos casos de teste usados durante o teste. A *ponta de prova* é um comando de escrita do número do nó em um arquivo produzindo a seqüência de execução dos nós em cada caso de teste durante o teste (no caso da POKE-TOOL este arquivo é denominado de PATH.TES).

No teste de integração um número identifica o Módulo de Programa e a *Unidade de Programa*. Esta informação é escrita no início do arquivo PATH.TES, representada pelo número *muuu* (*m* é o número do módulo e *uuu* é o número da unidade). A unidade $UP_A$ do módulo $Mod_3.pc$ tem o número 3001 onde 3 representa o número do módulo e 001 representa o número da unidade no módulo $Mod_3.pc$.

Para os critérios que tratam das associações *definição-t-uso* usadas tanto no teste de unidade como no teste de integração (*intra-modular* e *inter-modular*), é utilizada uma instrumentação especial no programa, cujo objetivo é informar as tuplas utilizadas durante a execução dos comandos de manipulação da **SQL**. Essas informações são gravadas em um arquivo denominado *Keyint.tes* indicando o número de cada caso de teste juntamente com as informações sobre as tuplas utilizadas na execução do respectivo caso de teste. Essa instrumentação difere da primeira tendo em vista que sua informação depende da composição dos campos que compõem a tupla.

A partir do programa instrumentado, cada execução dos casos de testes resulta em uma escrita de todos os nós percorridos no programa referente à Unidade instrumentada, gerando um arquivo $Path_n.tes$ onde *n* representa o número do caso de teste. Tal escrita facilita a implementação de outras funções; por exemplo: determina quantas vezes um

comando foi executado bem como auxilia na depuração após a ocorrência de uma falha durante a fase de teste. Exemplo e detalhes podem ser vistos no Apêndice C.

### 5.3.1 Instrumentando o programa original

A instrumentação reflete a semântica dos comandos da LI de forma a viabilizar a avaliação exata dos caminhos efetivamente executados. O fluxo de controle adotado para cada comando é, entretanto, representado pela instrumentação que reflete o fluxo correlato no momento da execução das unidades instrumentadas. Quanto aos comandos da **SQL**, cuidados especiais devem ser tomados com relação ao comando declarativo de tratamento de erros que é o responsável pelo fluxo de controle dos comandos executáveis da **SQL**. A ferramenta POKE-TOOL instrumenta programas escritos em linguagem **C** e os transformam em uma linguagem intermediária LI [MAL91, CHA91].

Para os comandos executáveis da **SQL** existem cuidados relativos às ações de controle de erros, sendo que a ação denominada *DO <função>* faz com que o gerador do programa instrumentado modifique alguns comandos do programa. No caso da ação *DO <função>* ocorrem as seguintes modificações: *i)* alterar o comando declarativo "**EXEC SQL WHENEVER** <*condição*> **DO** <função>" para o comando "**EXEC SQL WHENEVER** <*condição*> **CONTINUE**" e; *ii)* incluir o comando "**if** *sqlerror<0* {ponta de prova (*n*); <*função*>}" após os comandos de manipulação da **SQL** que sucedem ao comando declarativo de manipulação de erro; *n* representa o número do bloco de comando onde o comando de chamada da função foi acrescentado. Isto implica que quando a operação do comando de manipulação resultar em um erro de execução, o comando "*função()*" colocado após o comando de manipulação da **SQL** deve ser executado, conforme mostra a *Figura 5.7*.

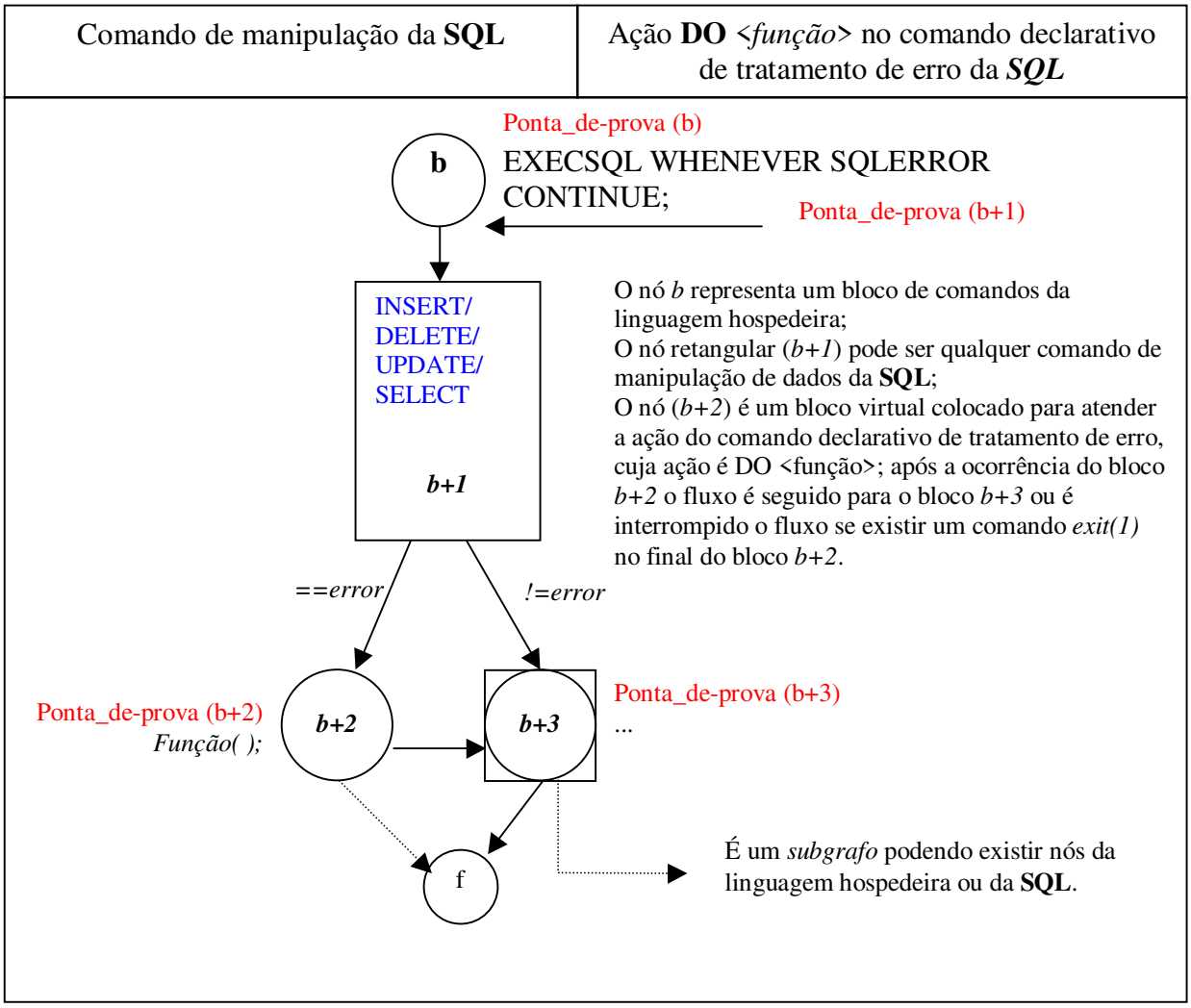



*Figura 5.7 Instrumentação dos comandos de manipulação da SQL para a ação DO <função>*

A ação **STOP** do comando declarativo de tratamento de erro faz com que o gerador de instrumentação provoque várias modificações no programa: *i)* alterar o comando declarativo "**EXEC SQL WHENEVER** <*condição*> **STOP**" para o comando "**EXEC SQL WHENEVER** <*condição*> **CONTINUE**"; e *ii)* incluir o comando "**if** (sqlca.sqlcode < 0) {*ponta de prova(f)*; *exit*(1); }" após os comandos de manipulação da **SQL** que sucederem ao comando declarativo de manipulação de erro; *f* representa o número do último comando do programa (último nó).

A instrumentação para todos os blocos de comandos é sempre anotada no início de cada bloco e, nos casos dos comandos executáveis da **SQL**, ela ocorre antes de cada comando. No caso da ação *GOTO* <*rótulo*> não ocorre nenhuma modificação nos

comandos; apenas são acrescentadas as pontas de provas no início de cada bloco de comando e antes de cada comando executável da **SQL**. O mesmo ocorre para a ação CONTINUE.

### 5.3.2 Identificando as tuplas

A instrumentação para a inclusão das informações sobre as tuplas utilizadas durante a execução de cada comando de manipulação da **SQL**, em uma definição ou um uso persistente de uma tabela ***t***, é feita em duas etapas. A primeira etapa é preenchida pelo próprio testador que fornece várias informações ao analisador sintático, responsável pela instrumentação do programa. A segunda etapa é gerar um arquivo que informa todas as *variáveis tabela* e suas respectivas chaves primárias: cada tipo e quais comandos devem ser incluídos no programa para gerar a informação referente às tuplas (conforme mostra a *Figura 5.8*).

As informações iniciais se restringem ao nome de cada *variável tabela* do programa e sua respectiva chave primária. Essas informações são armazenadas em um arquivo denominado *tab_ch_pr.tes* mostrado na Figura 5.9. Em seguida são informados os nomes das *variáveis host* utilizadas para relacionar as chaves primárias de cada variável tabela. Dessa forma, o analisador sintático incluirá no programa novos comandos que indicarão as chaves primárias, colocadas pelo testador no arquivo *tab_ch_pr.tes* (*Figura 5.8*), que são usadas no comando de manipulação da **SQL**. Para cada comando que caracteriza uma definição persistente da tabela ***t***, existe um comando específico para indicar a tupla utilizada. O comando SELECT é o comando que necessita de mais informações. Sempre que a chave primária está ausente no comando, é necessário incluí-la. A Figura 5.9 mostra o arquivo de auxílio para a instrumentação das informações sobre as tuplas referentes a cada *variável tabela* no Módulo de Programa *$Mod_3$.pc* do experimento realizado. Este arquivo auxilia a etapa de instrumentação. Ele é composto por comandos de atribuição das chaves primárias para cada *variável tabela* no Módulo da aplicação. Esses comandos dão auxílio à instrumentação para aplicar os critérios de teste que requerem as associações das *variáveis persistentes*.

Arquivo **inst_ch_pr.tes**

```
EXEC SQL BEGIN DECLARE SECTION;
  int  mhp1,mhsi1,mhd1,mhe1,mhc1,mhso1;
  char mhfd1[5];
  VARCHAR mhfd2[7], mhfd3[5], mhfc1[4];
EXEC SQL END DECLARE SECTION;
char me_emp_id[11], mso_id[11], md_dept_id[11], mfc_code[7],
     mfd_year[6], mfd_quarter[5], mfd_code[7], mp_prod_id[11],
     msi_id[11], mc_cust_id[11];

/* Conversoes */

NUMBER(11) p.prod_id in PRODUCT for int :mhp1 ->char mp_prod_id[11];
NUMBER(11) si.id in SALES_ORDER_ITEMS for  int :mhsi1 ->char msi_id[11];
NUMBER(11) d.dept_id in DEPARTAMENTO for int :mhd1 ->char md_dept_id[11];
NUMBER(11) e.emp_id in EMPLOYEE for int :mhe1 ->char me_emp_id[11];
NUMBER(11) c.cust_id in CUSTOMER for int :mhc1 ->char mc_cust_id[11];
NUMBER(11) so.id in SALES_ORDER for int :mhso1 ->char mso_id[11];
CHAR(4) fd.year, VARCHAR2(7) fd.quarter,VARCHAR2(4) fd.code in FIN_DATE for char :mhfd1,
VARCHAR :mhfd2,:mhfd3 ->char mfd_year[6], mfd_quarter[5],mfd_code[7];
VARCHAR2(4) fc.code in FIN_CODE for VARCHAR :mhfc1 ->char mfc_code[7];

PRODUCT:
mhp1=hp1; /* INSERT, DELETE, UPDATE */

itoa(mhp1, mp_prod_id); /* int para char */
fprintf(key, "PRODUCT: %s    \n",mp_prod_id);

SALES_ORDER_ITEMS:
mhsi1=hsi1; /* INSERT, DELETE, UPDATE */

itoa(mhsi1,msi_id); /* int para char */
fprintf(key, "SALES_ORDER_ITEMS: %s    \n",msi_id);

DEPARTAMENTO:
mhd1=hd1; /* INSERT, DELETE, UPDATE */

itoa(mhd1,md_dept_id); /* int para char */
fprintf(key, "DEPARTAMENTO: %s    \n",md_dept_id);

EMPLOYEE:
mhe1=he1;  /* INSERT, DELETE, UPDATE */

itoa(mhe1,me_emp_id); /* int para char */
fprintf(key, "EMPLOYEE: %s    \n",me_emp_id);

CUSTOMER:
mhc1=h1;  /* INSERT, DELETE, UPDATE */

itoa(mhc1, mc_cust_id); /* int para char */
fprintf(key, "CUSTOMER: %s    \n",mc_cust_id);

SALES_ORDER:
mhso1=hso1;  /* INSERT, DELETE, UPDATE */

itoa(mhso1,mso_id); /* int para char */
fprintf(key, "SALES_ORDER: %s    \n",mso_id);
```

*Figura: 5.8: Arquivo de auxílio à instrumentação das tuplas para as tabelas: EMPLOYEE, DEPARTAMENTO, CUSTOMER E SALES_ORDER .*

```
Arquivo Tab_ch_pr.tes

Relação das chaves primárias e dos tipos de dados para cada variável tabela.

Tabela Employee
he1    emp_id NUMBER(11)

Tabela Sales_Order
hso1   id NUMBER(11)

Tabela Department
hd1    dept_id NUMBER(11)

Tabela Fin_Code
hfc1   code VARCHAR2(4)

Tabela Fin_Date
hfd1   year CHAR(4)
hfd2   quarter VARCHAR2(7)
hfd3   code VARCHAR(4)
```

*Figura 5.9: Arquivo tab_ch_pr.tes: chaves primárias das tabelas usadas no programa Mod3.pc*

Nos exemplos apresentados nas Figuras *5.10a, 5.10b e 5.10c* são mostrados os arquivos utilizados na elaboração da instrumentação de uma *UP* do Módulo de Programa *$Mod_3.pc$*, usado nos exemplos de utilização dos critérios; a Figura 5.10a mostra o Programa Fonte usado no teste; o *Exemplo* da Figura 5.10b mostra o arquivo gerado na primeira instrumentação (nós e caminhos) e o *Exemplo* da Figura 5.10c mostra o arquivo gerado na segunda instrumentação (tuplas).

```
          Arquivo testemod3.pc (inicial)
int insemp (line)
RECORD *line;
{
       he1 = atoi (line->e_emp_id);
       he2 = atoi (line->e_manager_id);

       EXEC SQL WHENEVER SQLERROR GOTO end1;
       EXEC SQL INSERT INTO EMPLOYEE
                           ( emp_id,
                             manager_id)
                      VALUES ( :he1,
                               :he2);
       EXEC SQL COMMIT;
       end1:
       return (sqlca.sqlcode);
}
```

**Figura**

```
       Arquivo testeprog3.pc (Instrumentação dos caminhos)

int insemp (line)
RECORD *line;
{
FILE * path =fopen("insemp/path.tes","a");
       static int printed_nodes = 0;
       ponta_de_prova(1);

/*  1 */  he1 = atoi (line->e_emp_id);
/*  1 */ he2 = atoi (line->e_manager_id);

/*  1 */    EXEC SQL WHENEVER SQLERROR GOTO end1;
           ponta_de_prova(2);
/* 2 */ EXEC SQL INSERT INTO EMPLOYEE
                           ( emp_id,
                             manager_id)
                      VALUES ( :he1,
                               :he2);
           ponta_de_prova(3);
/* 3 */    EXEC SQL COMMIT;
/* 4 */  end1:
           ponta_de_prova(4);
           ponta_de_prova(5);

           fclose(path);
/*  4 */   return (sqlca.sqlcode);
/* 5 */   }
```

*Figura 5.10b Programa instrumentado para o teste de Unidade*

Arquivo **Mod3_int.pc** (Instrumentação dos caminhos)

```
int insemp (line)
RECORD *line;
{
FILE * path =fopen("insemp/path.tes","a");
FILE * key = fopen("insemp/key.tes","a");
       static int printed_nodes = 0;
       ponta_de_prova(3001);  /* número do programa insemp */
       ponta_de_prova(1);

/* 1 */  he1 = atoi (line->e_emp_id);
/* 1 */  he2 = atoi (line->e_manager_id);

/* 1 */    EXEC SQL WHENEVER SQLERROR GOTO end1;
           ponta_de_prova(2);
/* 2 */  EXEC SQL INSERT INTO EMPLOYEE
                          ( emp_id,
                            manager_id)
                    VALUES ( :he1,
                             :he2);

      mhe1=he1; /* INSERT, DELETE, UPDATE */
      itoa(mhe1,me_emp_id); /* int para char */
      fprintf(key, "EMPLOYEE: %s    \n",me_emp_id);

           ponta_de_prova(3);
/* 3 */    EXEC SQL COMMIT;
/* 4 */  end1:
           ponta_de_prova(4);
           ponta_de_prova(5);
           fclose(key);
           fclose(path);
/* 4 */   return (sqlca.sqlcode);
/* 5 */  }
```

*Exemplo 5.10c: Programa instrumentado para os critérios de integração intra-modulares e inter-modulares.*

## 5.4 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 desse grafo que, posteriormente, é utilizado para determinação de associações requeridas pelos critérios de teste. Para programas de ABDR o modelo de fluxo de dados é dividido em dois tipos de fluxos: *intra-modular* e *inter-modular*.

Existem três tipos de ocorrências das variáveis: *definição*, *uso* e *indefinição*. Essas ocorrências são aplicadas às *variáveis de programa* e *variáveis host*. Já para as variáveis tabela, adotou-se apenas as ocorrências de *definição* e *uso* por razões já apresentadas.

O modelo *intra-modular* é aplicado no teste de unidade e no teste de integração das unidades de um Módulo. O modelo *inter-modular* é aplicado na integração entre os Módulos da aplicação.

### 5.4.1 Modelo de Fluxo de Dados Intra-Modular

Para o teste de unidade foram escolhidos para ilustração os critérios Potenciais Usos [MAL91], implementados na POKE-TOOL. O g*rafo_def* é gerado pela POKE-TOOL a partir da ocorrência de *definição* das variáveis do programa de aplicação. O *grafo_def* informa as *definições* e *referências* ocorridas em cada nó do programa c.r.a. *variáveis de programa* e *variáveis host*, cujas ocorrências foram estudadas na Seção 3.2.1.

A passagem de valores entre procedimentos através de parâmetros pode ser feita por: valor, referência ou nome. Se a variável for 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 do *grafo(i)* [MAL91], ou seja, na determinação dos *caminhos livres de definição* para cada variável definida em *i*.

Da mesma forma, as ocorrências das *variáveis tabela* foram estudadas na Seção 3.2.1. Para a construção do g*rafo_def* é apenas indicada a ocorrência da definição por referência de ***t*** no nó onde existir um comando de manipulação da **SQL**. As Figuras 5.11 e 5.12 sintetizam os conceitos e hipóteses para a determinação do *grafo_def* a partir do modelo de fluxo de controle dos principais comandos da **SQL** na linguagem LI.

Comandos de Manipulação da **SQL**
INSERT, DELETE, UPDATE

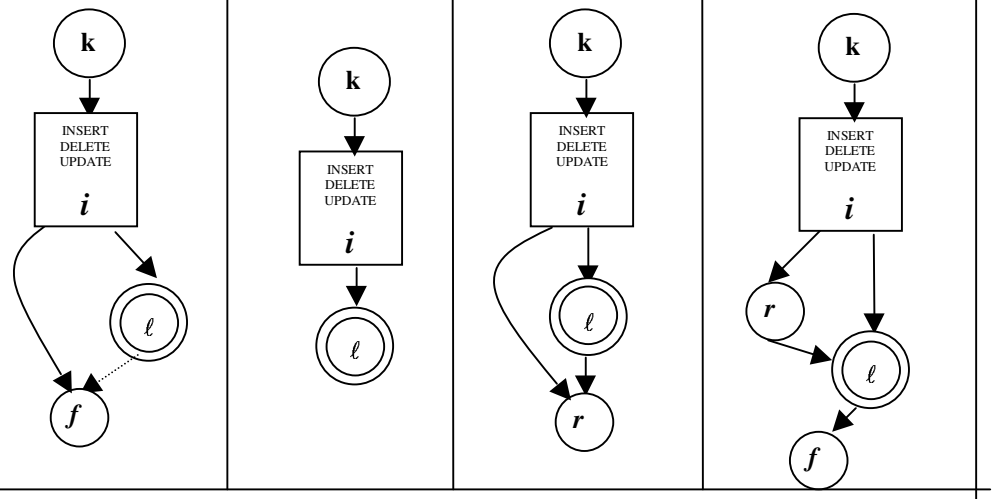
<WHENEVER>::=<whenever_atm> <cond_atm> <**act_atm**>

<INSERT>::=<ins_atm><tab_atm>
[{<atr_atm>}]<values_atm> {<val_atm>}

<DELETE>::= <del_from_atm><tab_atm> <where_atm>
[<cond_atm>/<cur_atm>]

<UPDATE>::=<up_ atm><tab_atm> <set_atm> {<stat_atm>}
<where_atm> [<cond_atm>/<cur_atm>]

**STOP** **CONTINUE** **GOTO** <*rotulo*> **DO** <*FUNC*>




O nó *k* contém o conjunto de variáveis definidas no bloco de comandos associados a este nó que envolve o comando declarativo da **SQL** WHENEVER, o qual é responsável pelo fluxo de controle no comando de manipulação da **SQL** colocado no nó *i*. O nó ℓ contém o conjunto de variáveis definidas no bloco de comandos da linguagem hospedeira, associado a este nó. O nó ***i*** contém o conjunto de *variáveis tabela* associadas a um dos comandos de manipulação da **SQL** (INSERT, DELETE, UPDATE), contendo uma definição por referência neste nó, a qual é atribuída uma definição por valor se o comando COMMIT estiver no subgrafo ℓ conforme mostram os fluxos de controles acima.

*Figura 5.11: Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Manipulação da Base de Dados (**SQL**)*

Comandos de Manipulação da **SQL**
SELECT

<WHENEVER>::=<whenever_atm> <cond_atm> <**act_atm**>

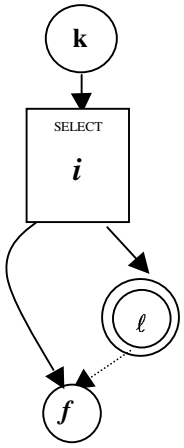
<SELECT>::=<sel_ atm>{<col_atm>}

[<**into_atm**> <var_atm>]

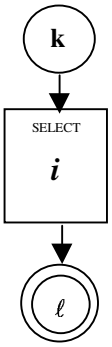
<from_atm> {<tab_atm>}

<where_atm> [<cond_atm>/<cur_atm>]

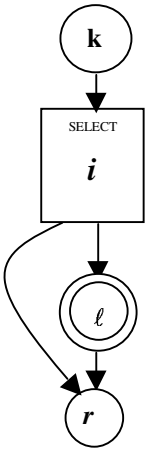
**STOP**



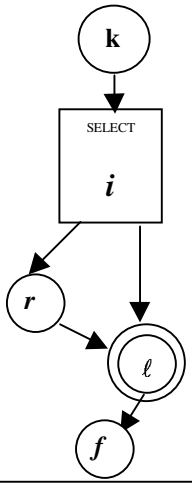

**CONTINUE**




**GOTO** <*rotulo*>




**DO** <*FUNC*>




O nó *k* contém o conjunto de variáveis definidas no bloco de comandos associados a este nó que envolve o comando declarativo da **SQL** WHENEVER, o qual é responsável pelo fluxo de controle no comando de seleção da **SQL** colocado no nó *i*. O nó ℓ contém o conjunto de variáveis definidas no bloco de comandos, da linguagem hospedeira, associado a este nó. O nó ***i*** contém também o conjunto de *variáveis host* associadas ao comando de seleção da **SQL** (SELECT) na cláusula <*into*> <*var_host*>.

*Figura 5.12: Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Seleção da Base de Dados (**SQL**)*

Para obter as informações referentes aos critérios que exercitam as *variáveis persistentes*, foi necessário criar uma relação que identifique os pontos do programa onde ocorrem os comandos de manipulação da **SQL** (INSERT, DELETE, UPDATE) seguido do

comando COMMIT para caracterizar a *definição persistente* de uma tabela ***t*** e dos comandos que caracterizam um *uso persistente* da tabela ***t*** (INSERT, DELETE, UPDATE e SELECT). Este arquivo, denominado *Tabdtu.tes*, é gerado para cada tabela utilizada no programa da aplicação. Este arquivo indica o número da Unidade de Programa que contém um uso ou uma definição da tabela ***t*** mostrando o número do nó onde ocorre uma *definição* e um *uso persistente* de ***t***. A partir deste arquivo são construídas as *associações definição-t-uso* para cada tabela envolvida no programa. Essas informações são utilizadas para a geração dos elementos requeridos para o teste de unidade e integração referentes aos critérios exclusivos de ABDR.

Para exemplificar foi utilizado o Módulo *Mod₃.pc* que relaciona 5 *variáveis tabela*. Para a variável EMPLOYEE foi gerado o arquivo *Tabdtu.tes* mostrado na Figura 5.13. O número (3001) do arquivo *Tabdtu.tes* representa o número da Unidade. *Defp* indica os números dos nós onde ocorrem as *definições persistentes*; o primeiro número representa o nó $\ell$ referente ao comando de manipulação da **SQL** e o número que antecede o zero representa o nó *i* referente ao comando COMMIT da **SQL**. Já os números de *uso* representam os nós que contêm um dos comandos de manipulação ou de alteração da **SQL** que caracterizam um *t-uso*. O número zero (0) indica que não existe nenhum nó para *defp* ou *uso*.

Arquivo *Tabdtu.tes* - EMPLOYEE

```
3001
defp:  2  3  0
uso:  2  0
3006
defp:  2  3  0
uso:  2  0
3011
defp:  2  3  0
uso:  2  0
3016
defp:  0
uso:  3  6  9  12  0
```

*Figura 5.13:* Arquivo *tabdeftuso.tes* para a tabela EMPLOYEE no Módulo *Mod₃.pc*

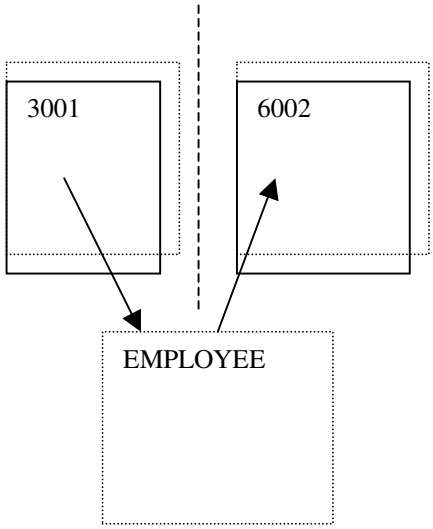
### 5.4.2 Modelo de Fluxo de Dados Inter-Modular

As principais características para o teste estrutural de uma ABDR, no teste de integração entre os Módulos, estão relacionadas ao fluxo de dados ocasionado entre os diferentes Módulos que definem e usam uma *variável tabela*, ocasionando assim, um fluxo de dados entre os Módulos. Neste caso, para tratar dos critérios *Todos os t-usos-ciclo1-int*, *Todos os dtu-caminhos-int* e *Todos os t-usos-ciclo2-int* definidos na *Seção 4.4,* são construídos arquivos que relacionam os módulos que definem e usam uma mesma tabela. Isso é feito para cada *variável tabela* presente no Módulo, conforme é mostrado na *Figura. 5.14* para a tabela EMPLOYEE. O arquivo é denominado de *Tabidtu.tes*. A *Figura 5.14* mostra também um grafo de fluxo para duas *UPs* referentes aos Módulos *Mod$_3$.ps* e *Mod$_6$.ps*.

A partir deste arquivo são extraídos todos os elementos requeridos pelos critérios de

Arquivo *Tabidtu.tes* - EMPLOYEE

```
3000

 3001
 defp:  2  3  0
 uso:  2  0
 3006
 defp:  2  3  0
 uso:  2  0
 3011
 defp:  2  3  0
 uso:  2  0
3016
 defp:  0
 uso:  3  6  9  12  0

 6000

 6002
 defp:  0
 uso:  2  5  0
6003
 defp:  0
 uso:  2  5  0
6004
 defp:  0
 uso:  2  5  0
```




*Para cada Unidade de um Módulo que define a tabela Employee existe uma associação com outra Unidade de outro Módulo. Esta combinação é feita para cada tabela. E assim é construído o elemento requerido para satisfazer o critério todos os t-usos-ciclo1-inter e para o critério Todos-dtu-caminhos-inter.*

*Figura 5.14*: Arquivo *Tabidtu.tes* para a tabela EMPLOYEE nos Módulos *Mod$_3$.*pc e *Mod$_6$.*pc

teste *inter-modular*, utilizados na integração dos Módulos da aplicação.

## 5.5 Modelo de Descrição dos Elementos Requeridos

O modelo de descrição dos elementos requeridos para os critérios propostos nesta tese não é apresentado neste trabalho, tendo em vista que não foi desenvolvida nenhuma ferramenta para apoiar esses critérios. O modelo implementado na POKE-TOOL está fundamentado no conceito de arco primitivo [CHU87] e no conceito de *grafo(i)* [MAL88b]. O conceito de arco primitivo se baseia no fato de existirem *arcos*[10] dentro de um grafo de programa que são sempre executados quando outro arco é executado. Desta forma, a FCPU passa a requerer associações com base nos arcos primitivos, eliminando os demais arcos denominados de *arcos herdeiros*. Para os critérios de ABDR a utilização de arcos primitivos no contexto de fluxo de dados continua sendo a mesma de Maldonado [MAL91].

Sendo assim, a implementação dos descritores dos critérios abordados nesta tese é mais simples, considerando apenas os caminhos que possuem os nós da **SQL**. Para os critérios de teste de unidade e teste de integração, que requerem *associações definição-t-uso*, foi criado o *grafo*$<\ell,i>$ (extraído do conceito de *grafo(i)*, definido em [MAL91]). O *grafo*$<\ell,i>$ é construído a partir de cada par de nós $<\ell,i>$, com *definição persistente* c.r.a. ***t***, que alcançar: ou um ***t-uso*** (nó de **SQL**), ou um o nó de saída do grafo de programa, com um caminho livre de definição c.r.a. ***t*** e livre de laço.

O modelo de descrição dos elementos requeridos para os critérios de programas de aplicação pode ter como ponto de partida o modelo de implementação apresentado em [MAL91]. Nossa expectativa é implementar uma ferramenta que apóie os critérios de ABDR e complemente os demais critérios estruturais (como os da FCPU ou FCFD).

Nos exemplos utilizados para ilustrar a aplicação dos critérios, os descritores dos elementos requeridos foram gerados manualmente, separados em diretórios referentes a cada *variável tabela* envolvida na associação. No caso do critério *todos os t-usos-ciclo1-intra* é criado um arquivo que armazena todas as *associações definição-t-uso* referentes a uma *variável tabela*. Durante a execução dos casos de testes (execução do teste) são

[10] *Esses arcos são considerados não essenciais para a análise de cobertura. Já os arcos primitivos são considerados essenciais para a análise de cobertura por garantirem a execução dos demais arcos do grafo de programa.*

gerados dois arquivos: *1) pathint.tes* que armazena o número do caso de teste, mais o número da unidade exercitada e o respectivo caminho executado; e *2) keyint.tes* que armazena o número de cada caso de teste, o número da unidade executada e a tupla utilizada para exercitar o caso de teste (em alguns casos pode ser a própria *chave primária*).

Depois que todos os casos de teste são executados, a avaliação da cobertura do critério é feita manualmente, avaliando cada caso de teste, tuplas e quais unidades foram envolvidas, para depois avaliar quais associações foram satisfeitas. Essa avaliação pode ser feita utilizando um autômato finito que avalie o $grafo_d\langle \ell,i \rangle$ que inicia na definição persistente e no $grafo_u\langle \ell,i \rangle$ que inicia no nó de entrada do grafo de programa (da unidade associada) e termina nos nós que contêm um ***t-uso***. No Apêndice C, é mostrado um exemplo completo que apresenta as *associações definição-t-uso* e os casos de testes que executaram a associação.

## 5.6 Considerações Finais

Os modelos teóricos aqui apresentados visam a fornecer maior precisão e uniformidade à automatização dos critérios de testes baseados em fluxo de dados para Aplicações de Banco de Dados Relacional. Uma das propostas de continuidade deste trabalho é incorporar à ferramenta POKE-TOOL os critérios de teste de integração *intra-modular* e *inter-modular*, de modo a auxiliar o teste de programas de aplicação.

Os modelos de fluxo de controle, de instrumentação e de fluxo de dados são fundamentais para completar os estudos de comparação empírica dos critérios aqui propostos. Para a implementação desses modelos na ferramenta POKE-TOOL deve-se inicialmente acrescentar informações referentes aos comandos da **SQL** (estender a LI), de modo a possibilitar o teste de unidade em programas de aplicação. Vários programas devem ser modificados para a geração dos descritores e dos elementos requeridos referentes aos novos critérios.

A partir da descrição dos elementos requeridos para os critérios de teste de unidade usada em [MAL91] (conceitos de arcos primitivos, de *grafo_def* e de *grafo(i)*), podemos estendê-los para extrair o $grafo\langle \ell,i \rangle$ para os critérios que exercitam as *variáveis*

*persistentes*. Os estudos iniciados neste capítulo dão suporte à implementação de um sistema automatizado para a aplicação dos critérios de testes propostos para ABDR.

Para o teste de integração *intra-modular* e *inter-modular*, acredita-se que é conveniente a preparação de um ambiente que controle e organize os elementos requeridos separando-os para cada *variável tabela* envolvida nos Módulos de Programa. Pode até mesmo ser mais conveniente a construção de uma nova ferramenta de apoio ao teste de integração.

# 6 Exemplos de Utilização dos Critérios de Teste

Este capítulo ilustra a utilização dos critérios propostos para o teste de programas de ABDR. Na Seção 6.2 são apresentados a descrição das tabelas, dependências e fluxo de dados para o exemplo de aplicação. Na Seção 6.3 são apresentados resultados obtidos nos exemplos de utilização. Na Seção 6.4 são apresentadas algumas comparações referentes à utilização dos critérios. Algumas observações são feitas sobre os resultados do teste dos programas do exemplo de aplicação.

## 6.1 Considerações Iniciais

Para ilustrar a utilização dos critérios foi desenvolvido um exemplo completo de uma aplicação composta por quatro Módulos de Programas em linguagem C com **SQL** embutida, com programas que foram adaptados de uma *aplicação real*. As tabelas escolhidas para compor a aplicação foram extraídas do módulo de demonstração do sistema de Banco de Dados da "*Sybase"*. A construção desse exemplo também teve como objetivo mostrar o comportamento dos fluxos de controle e de dados dos principais comandos da **SQL**. Inicialmente foi construído o projeto da base de dados a partir do módulo SQL/PLUS (que permite usar comandos **SQL** para criar, manipular e consultar as tabelas da base de dados). Com todas as tabelas devidamente criadas e com suas regras definidas (chaves primárias, chaves estrangeiras, e outras regras), os programas foram construídos e submetidos ao teste estrutural para todos os critérios propostos. Para a execução dos exemplos foi necessário fazer algumas modificações nos arquivos gerados pela ferramenta. A POKE-TOOL não aceita **SQL**; a primeira tarefa foi instrumentar o programa sem os comandos de **SQL** e, em seguida, acrescentar os comandos e modificar o programa instrumentado acrescentando os comandos **SQL** e as novas pontas de provas. Após a preparação dos programas e da modificação de todos os arquivos (inclusão de informações referentes aos comandos e variáveis de ABDR) gerados pela ferramenta (como: *tab_vardef.tes*, *função.gfc*, *etc*), foram gerados os descritores (para os critérios FCPU) e, em seguida, iniciou-se o teste de cada unidade.

Para os testes de integração *intra-modular* e *inter-modular*, a instrumentação para a integração e os arquivos necessários foram gerados e modificados manualmente.

O teste é executado em três fases distintas. A primeira, análise estática do programa fonte, consiste em gerar o programa instrumentado, a versão LI do programa e os arquivos referentes às variáveis. No programa instrumentado são colocadas várias informações que são usadas na última fase do teste. Na segunda fase do teste, executam-se os casos de teste para satisfazer os critérios.

Para o teste de integração, foram modificados os programas que controlam a execução dos casos de teste (de *pokeexec* para *pokeexec_int1*e *pokeexec_int2*) para contemplar os dois ciclos de execução (critérios de *ciclo1* e *ciclo2*). A última fase do teste consiste na análise dos resultados através da avaliação da cobertura dos elementos requeridos pelos critérios.

Esses exemplos serviram para avaliar a dificuldade de gerar os casos de teste (uso da mesma tupla, controle do estado das tabelas, etc) que exercitem os elementos requeridos. Como a POKE-TOOL não possui recursos para o teste de integração, a organização dos arquivos, a preparação dos resultados e a avaliação são feitas manualmente.

## 6.2 Descrição das Tabelas, Dependências e Fluxo de Dados

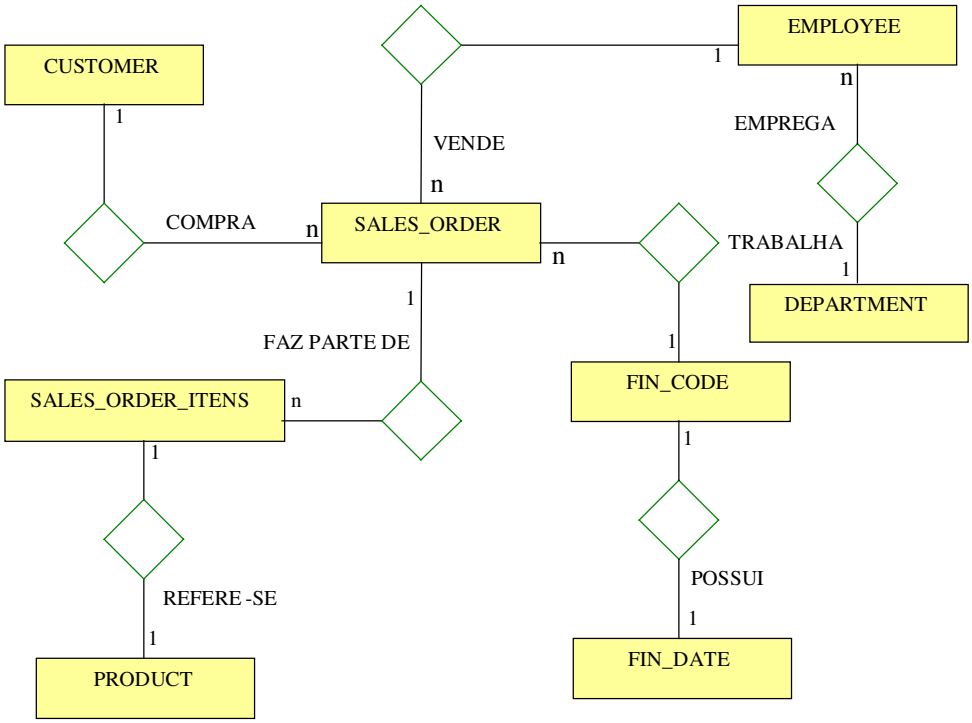



*Figura 6.1a – Diagrama de Entidades e Relacionamentos*

A *Figura 6.1a* mostra o Diagrama de Entidades e Relacionamentos (*DER*) da base de dados dos exemplos de aplicação, e a *Figura 6.1b* mostra o fluxo de dependência entre as tabelas com suas respectivas *chaves primárias* e *chaves estrangeiras*. Foram utilizadas nove tabelas; note-se que a tabela CONTACT não possui relacionamento com as demais tabelas.

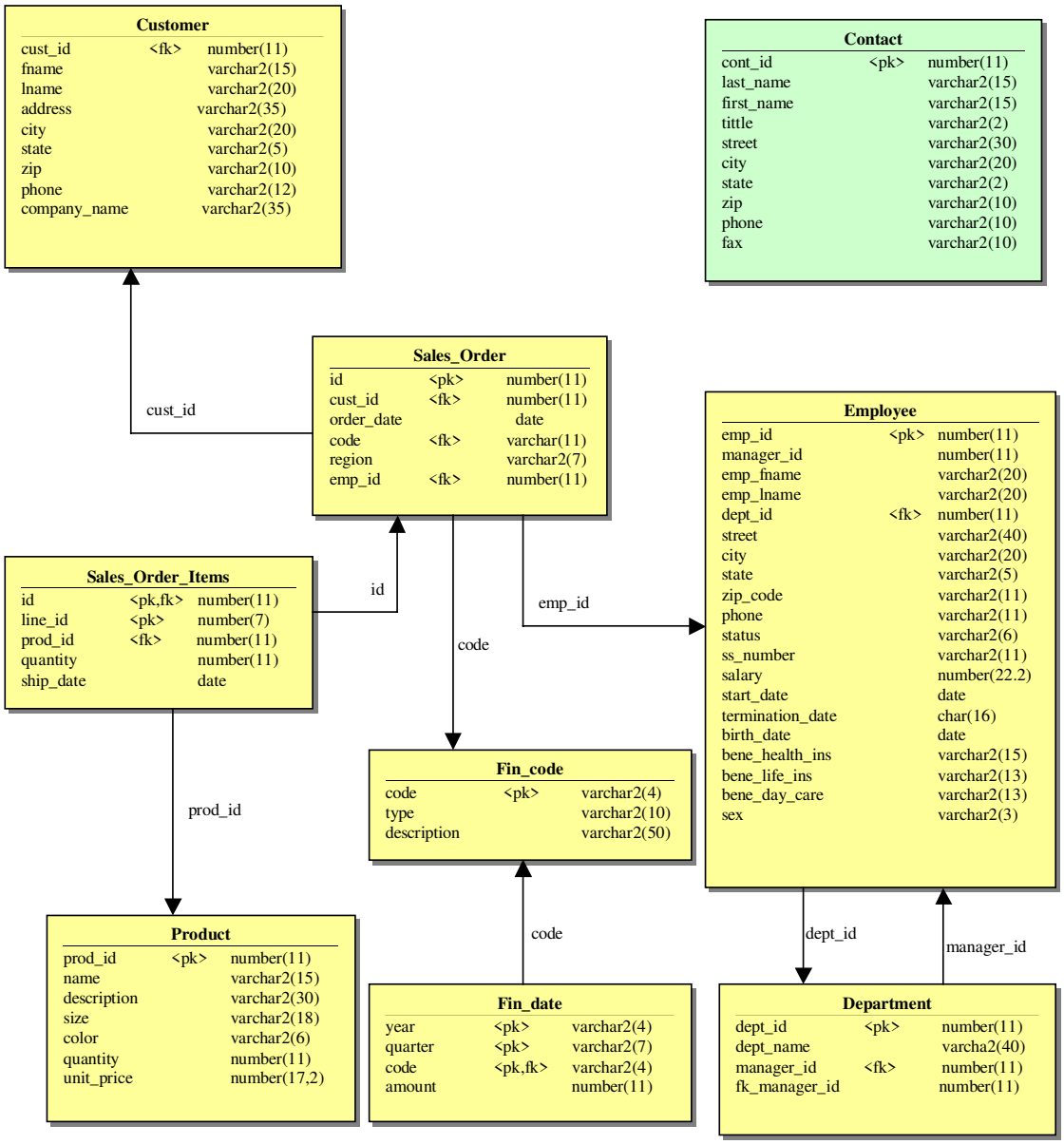



*Figura 6.1b: Tabelas da base de dados e fluxo de dependência.*

Os exemplos de utilização dos critérios foram desenvolvidos com os objetivos de: *i)* mostrar que os critérios de teste baseados em análise de fluxo de dados, em particular os

critérios Potenciais Usos, podem ser adaptados para programas de aplicação; *ii)* estudar o fluxo de controle dos comandos **SQL** e o fluxo de dados ocasionados pelas *variáveis tabela* e *variáveis host*; *iii)* estudar os fluxos de dados *intra-modular* e *inter-modular* c.r.a. *variáveis tabela*.

As tabelas da base de dados são acessadas pelos programas dos exemplos. A aplicação consiste de quatro Módulos compostos por unidades com comandos **SQL** que possuem diferentes fluxos de dados. A *Figura 6.2* apresenta um diagrama de composição da aplicação, mostrando os fluxos de dados existentes entre os Módulos e as *variáveis tabela*.

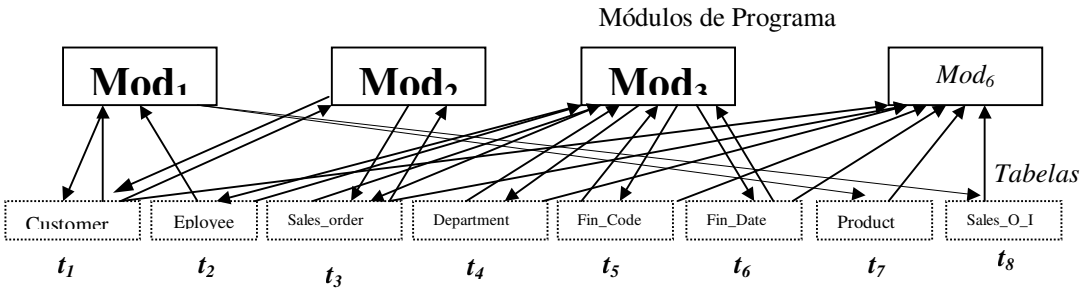



*Figura 6.2*: Fluxo de dados *inter-modular* da ABDR composta por 4 Módulos de Programa e 8 tabelas que constituem a base de dados

A *Tabela 6.1* explica a *Figura 6.2*; as letras **D** e **U** indicam que o Módulo ($Mod_i$) possui pelo menos uma Unidade que define a *variável tabela* (**D**) e/ou que existe pelo menos uma Unidade que usa a *variável tabela* (**U**). Sendo assim, existe um fluxo do Módulo para a Tabela quando existir **D** e um fluxo da Tabela para o Módulo quando existir **U**.

*Tabela 6.1:* Ocorrências de *definição persistente* e uso das *variáveis tabela* nos Módulos de Programas *(D: Definição / U: Uso).*

| ***Tabelas*** | $Mod_1$ | $Mod_2$ | $Mod_3$ | $Mod_6$ |
|---|---|---|---|---|
| **$t_1$ – Customer** | D / U | D / U | - | U |
| **$t_2$ – Employee** | D / U | - | D / U | U |
| **$t_3$ – Sales_order** | - | D / U | D / U | U |
| **$t_4$ – Department** | - | - | D / U | U |
| **$t_5$ – Fin_Code** | - | - | D / U | U |
| **$t_6$ – Fin_Date** | - | - | D / U | U |
| **$t_7$ – Product** | D / U | - | - | U |
| **$t_8$ – Sales_Order_items** | D / U | - | - | U |
| **$t_9$ – Contact** | D / U | - | - | U |

## 6.3 Resultados dos Exemplos de Utilização

A seguir são mostrados os resultados obtidos nos exemplos de utilização dos critérios de teste para fluxos de dados *intra-modular* e *inter-modular*, apresentados no Capítulo 4.

### 6.3.1 Teste intra-modular

O teste *intra-modular* é o teste de cada Módulo da aplicação, envolvendo o teste de unidade e o teste de integração baseado no fluxo de dados entre as unidades. Para ilustrar a utilização dos critérios de teste *intra-modular* as situações possíveis de ocorrer em um sistema de Banco de Dados Relacional foram consideradas. Os programas em C com **SQL** embutida desenvolvidos espelham uma amostra real das operações básicas de **SQL** que ocorrem na prática. Os resultados dos exemplos foram usados para análise e estudo dos critérios de testes, visando a obter indicações quanto à aplicabilidade dos critérios.

#### 6.3.1.1 Teste de unidade

Tendo em vista que cada programa utilizado no exemplo supera mil linhas de código, serão apresentadas apenas as principais partes dos Módulos utilizadas para ilustrar o teste e seus respectivos critérios; um exemplo completo está no Apêndice C para complementar as informações sobre como utilizar os critérios.

A seguir é apresentada uma seqüência de exemplos de resultados parciais do teste de unidade para o Módulo *Mod$_3$.pc* - Unidade *Insemp*.

*Programa Original: Modulo$_3$.pc* – Unidade: *Insemp*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct { char e_emp_id [11];
                 char e_manager_id[11];
                 . . .
                       } RECORD;
EXEC SQL BEGIN DECLARE SECTION;
int        h1,he1;
VARCHAR    h2[15];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE sqlca;
. . .
```

```
int insemp (line)
RECORD *line;
{
      he1 = atoi (line->e_emp_id);
      he2 = atoi (line->e_manager_id);
      . . .
      EXEC SQL WHENEVER SQLERROR GOTO end1;

      EXEC SQL INSERT INTO EMPLOYEE
            ( emp_id,
              manager_id,
     VALUES ( :he1,
              :he2,
             . . .
       EXEC SQL COMMIT;
 end1:
    return (sqlca.sqlcode);
  }
....
main()
{ ...
}
```

Programa *Mod3.pc Instrumentado* - Unidade: *Insemp*

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d
",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
.... /* manteve o mesmo do original */ ....

int insemp (line)
RECORD *line;
/*  1 */        {
                   FILE * path = fopen("insemp/path.tes","a");
                   static int printed_nodes = 0;
                   ponta_de_prova(1);
/*  1 */           he1 = atoi (line->e_emp_id);
/*  1 */           he2 = atoi (line->e_manager_id);
                   . . .
/*  1 */           EXEC SQL WHENEVER SQLERROR GOTO end1;
                   ponta_de_prova(2);
/* 2 */            EXEC SQL INSERT INTO EMPLOYEE
                                ( emp_id,
                                  manager_id,
                                  . . .
                                      )
                         VALUES ( :he1,
                                  :he2,
                                  . . .
                                     );
                      ponta_de_prova(3);
/* 3 */               EXEC SQL COMMIT;
```

```
/* 4 */  end1:
                    ponta_de_prova(4);
                    ponta_de_prova(5);
                    fclose(path);
/*  4 */            return (sqlca.sqlcode);
/*  5 */        }
```

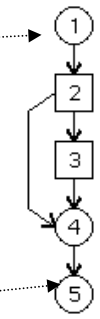
Além da instrumentação do programa em teste, a POKE-TOOL gera os seguintes arquivos relativos à Unidade *Insemp* (para o teste de unidade): *Tabvardef.tes* - contém informação sobre todas as variáveis globais e locais utilizadas no programa instrumentado *Mod₃.pc* para a unidade *Insemp*; arquivo *Insemp.gfc* – contém a descrição do grafo de programa da Unidade *Insemp*.

Arquivo *Tabvardef.tes* do Módulo *Mod₃.pc* da Unidade *Insemp*

```
# Tabela de Variaveis Definidas do Modulo insemp

# Globais
@ 137 he13
  . . .
@ 102 he2
@ 101 he1
@ 100 hfd4

# Locais
@ 3 sqlca
@ 2 strcpy
@ 1 atoi
@ 0 line
@@

# Grafo Def Sintetico do Modulo insemp

@@  1
Defs:  101  102  103  111  112  113  114  115  116  117  118  119  120
121  122  123  133  134  135  137 @
Refs: @
@@  2
Defs: @
Refs:  109 @
@@  3
Defs: @
Refs:  110 @
@@  4
Defs: @
Refs: @
@@  5
Defs: @
Refs: @
```

*Insemp.gfc: Grafo de programa* da Unidade *Insemp*.

```
5          Total de nós no grafo
1
   2 0
2
   3 4 0
3
   4 0
4
   5 0
5
   0
```




*Insemp.sql*: Nós de SQL na unidade *Insemp*.

```
2          Total de nós SQL
2          Nº do nó SQL
3
```

A partir do arquivo *Insemp.gfc* e do arquivo *Insemp.sql* a ferramenta ViewGraph 2.0 [CRU99] gera a visualização do grafo com os nós da **SQL**; essa visualização é mostrada na Figura 6.3.

O arquivo *Arcprim.tes* contém os arcos primitivos referentes ao grafo de programa. A ferramenta ViewGraph oferece a opção de desenhar o grafo de programa os arcos primitivos utilizando o arquivo *Arcprim.tes*.

```
ARCOS PRIMITIVOS da Unidade Insemp

  1) arco ( 2, 3)
  2) arco ( 2, 4)
```

*Exemplo 6.3: Arquivo Arcprim.tes.*

A segunda fase do teste de Unidade é a execução dos casos de teste. Após compilado, o Programa Instrumentado (mostrado no Exemplo 6.2) é utilizado para a execução dos casos de testes. Para ilustrar esta etapa, foram apresentados os exemplos referentes à unidade *Insemp* do *Exemplo 6.1*. A fase de execução dos casos de teste é feita a partir do programa *pokeexec* da POKE-TOOL.

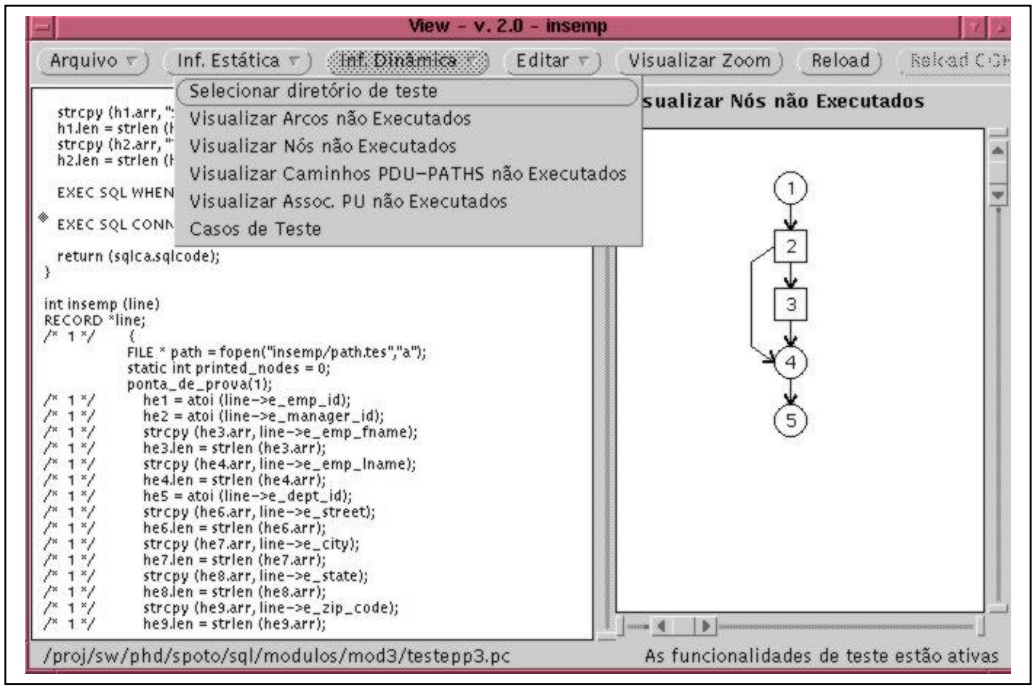



Figura 6.3: Visualização do teste da unidade Insemp

A seguir é apresentado um dado de entrada para exercitar a unidade *Insemp*; os dados foram colocados em arquivos de entrada numerados (*finp01, finp02 etc).* No Exemplo 6.4 é apresentado o arquivo *finp01* com os valores separados por vírgula (“**,** “).

```
i, 1, 1903, 1576, Carlos, Germanni, 400, Av Maracanã, Rio de janeiro, RJ,
21060009, 02198882, A, 118998234, 34000.000, 12-JUL-90, (NULL), 09-DEC-
60, Y, Y, N, M
```

*Exemplo 6.4: Ilustra um arquivo de entrada (finp01) para o* Módulo *Mod*$_3$

O resultado da execução é um arquivo de saída para cada caso de teste executado. Esses arquivos são armazenados no diretório *output;* cada arquivo é numerado como *output1*, *output2,* etc (onde 1 e 2 referem-se ao número de cada caso de teste). Os arquivos de entrada são armazenados no diretório *keyboard* ou no diretório *input*, que mostram as entradas utilizadas durante a execução dos casos de testes. Na execução é gerado um arquivo referente ao caminho percorrido no grafo de programa; esses arquivos são numerados como *path1*, *path2* etc, conforme o número de cada caso de teste. No exemplo, a execução do dado de teste *finp01* gerou o arquivo *path1* para a função *Insemp*.

```
1   2   3   4   5
```

*Exemplo 6.5: Arquivo path1 – caminho de execução do dado de teste do Exemplo 6.4*

Esse arquivo é utilizado posteriormente para a análise de cobertura dos critérios *todos-potenciais-usos*.

Para avaliar os critérios que exercitam as *variáveis persistentes* é necessário gerar o arquivo *keyint.tes* onde são armazenadas as tuplas utilizadas durante a execução dos casos de teste. No Exemplo 6.5, não existe nenhuma *associação-t-uso* na mesma unidade. Sempre que existir pelo menos uma unidade que possibilite a ocorrência de uma *associação definição-t-uso*, com relação a alguma *variável tabela*, é necessário fazer a re-instrumentação desta unidade. Para indicar se existe ou não a *associação definição-t-uso* na unidade, foi criado um arquivo denominado *dtuassoc.tes* onde são armazenadas todas as associações da unidade. A *re-instrumentação* é feita a partir do programa intrumentado do teste de unidade. Para o exemplo de utilização dos critérios a instrumentação foi feita manualmente.

### 6.3.1.2 Teste de integração baseado na dependência de dados

O teste de integração baseado na dependência de dados entre as Unidades envolvidas nesta etapa é preparado com a re-instrumentação dos programas utilizados no teste de unidade.

A re-instrumentação para esta fase acrescenta informações pertinentes ao número de cada unidade e às tuplas, mantendo as demais informações instrumentadas do teste de unidade. Para ilustrar um programa re-instrumentado é apresentado o Exemplo 6.6, o programa do Exemplo 6.2 (unidade *Insemp*).

A re-instrumentação acrescenta informações sobre as chaves primárias usadas para representar as tuplas das *variáveis tabela*, exigidas pelos critérios propostos neste trabalho, e um número que é usado para indicar qual unidade foi exercitada durante um caso de teste. Os elementos requeridos pelos critérios de teste de integração são gerados para cada *variável tabela*. Os elementos requeridos dos critérios *todos os t-usos-ciclo1-intra* e *todos os dtu-caminhos-intra* são gerados para todos os pares de Unidades que possuem comandos

de manipulação da **SQL** de uma mesma tabela e são colocados em diretórios denominados *ciclo1* e organizados pelo nome de cada tabela.

```
int insemp (line)
RECORD *line;
/*  1 */        {
                  FILE * path =       fopen("insemp/pathint.tes","a");
                  FILE * key = fopen("insemp/keyint.tes","a");
                  static int printed_nodes = 0;
                  ponta_de_prova(3001);
                  fprintf(key,"3001 \n");
                  ponta_de_prova(1);
/*  1 */          he1 = atoi (line->e_emp_id);
/*  1 */          he2 = atoi (line->e_manager_id);
                  . . .
/*  1 */          EXEC SQL WHENEVER SQLERROR GOTO end1;
                  ponta_de_prova(2);
                  mhe1=he1; /* INSERT, DELETE, UPDATE */
                  itoa(mhe1,me_emp_id); /* int para char */
                  fprintf(key, "EMPLOYEE: %s    \n",me_emp_id);
/* 2 */           EXEC SQL INSERT INTO EMPLOYEE
                              ( emp_id,
                                manager_id,
                                . . . )
                        VALUES ( :he1,
                                 :he2,
                                 . . . );
                     ponta_de_prova(3);
/* 3 */              EXEC SQL COMMIT;
/* 4 */  end1:
                  ponta_de_prova(4);
                  ponta_de_prova(5);
                  fclose(path);
                  fclose(key);
/*  4 */          return (sqlca.sqlcode);
/*  5 */        }
```



*Exemplo 6.6 Função Insemp do Programa $Mod_3$ re-instrumentada para o teste de integração.*

A organização dos arquivos para o teste de integração do Módulo $Mod_3$ é mostrada na Figura 6.4. Os arquivos *dtuassoc.tes* e *dtupaths_intra.tes* contêm os elementos requeridos dos critérios *todos os t-usos-ciclo1-intra* e *todos os dtu-caminhos-intra*, respectivamente.

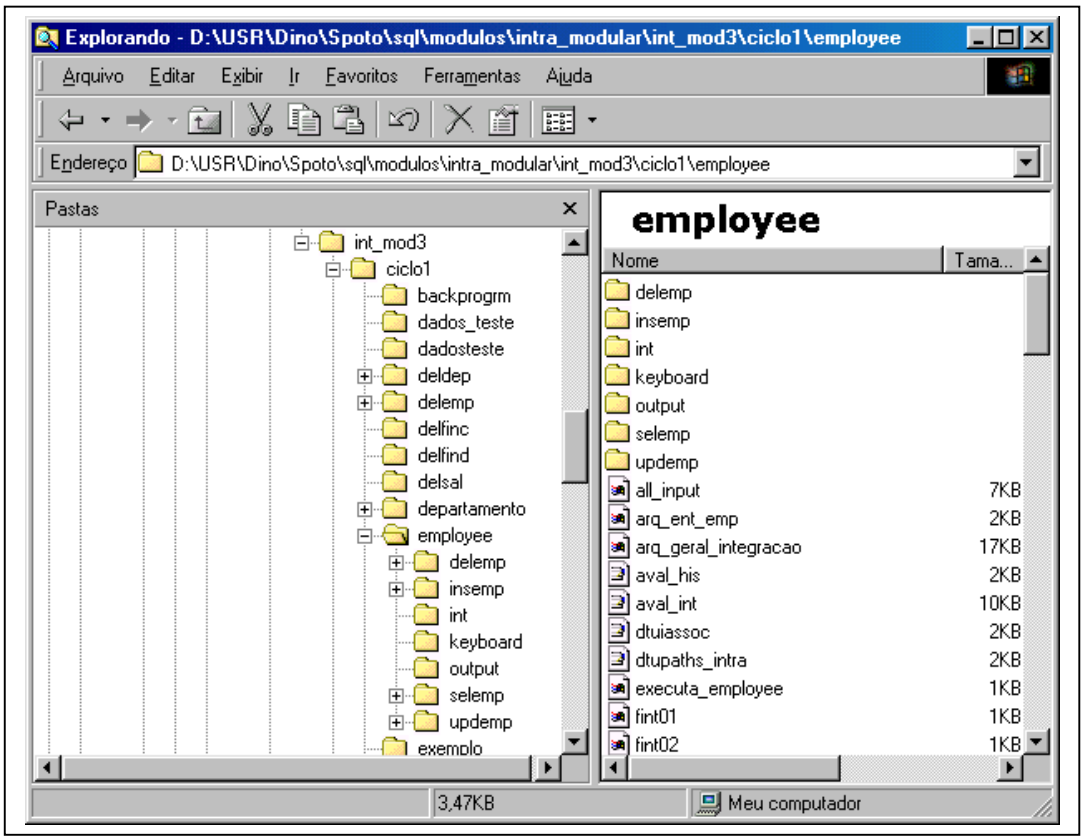
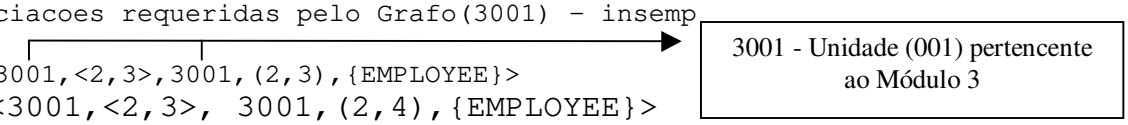



*Figura 6.4* Organização dos arquivos utilizados para o teste de integração *intra-modular.*

Os arquivos que contêm os elementos requeridos dos critérios *todos os t-usos-ciclo1-intra* e *todos os dtu-caminhos-intra* são mostrados no *Exemplo 6.7*.

O *Exemplo 6.7* mostra parte dos elementos requeridos para os critérios *todos os t-usos*; <3001,<2,3>,3016,(2,4),{EMPLOYEE}> é a associação entre as unidades 001 e 016 do Módulo 3. No par <2,3> ocorre a *definição persistente* da variável EMPLOYEE, na Unidade 001, e no arco (2,4) ocorre o uso da variável EMPLOYEE, na Unidade 016. O mesmo ocorre com os elementos requeridos para o critério *todos os dtu-caminhos* representados pela seguinte seqüência 3011 2 3 4 5 - 3016 1 11 12 18, onde a primeira seqüência 3011 2 3 4 5 é o *dtu-caminho* (2, 3, 4, 5) da Unidade 011 do Módulo 3 referente à *definição-persistente* de alguma *variável tabela*, e a seqüência **3016** 1 11 12 18 é o *dtu-caminho* (1, 11, 12, 18) referente ao uso da *variável persistente*.

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODAS DEF-T-USOS-INTRA EMPLOYEE

Associacoes requeridas pelo Grafo(3001) - insemp
```




```
1) <3001,<2,3>,3001,(2,3),{EMPLOYEE}>
2) <3001,<2,3>, 3001,(2,4),{EMPLOYEE}>
. . .
14) <3001,<2,3>, 3016,(12,18),{EMPLOYEE}>

Associacoes requeridas pelo Grafo(3006)

15) <3006,<2,3>,3001,(2,3),{EMPLOYEE}>
16) <3006,<2,3>,3001,(2,4),{EMPLOYEE}>
. . .
28) <3006,<2,3>,3016,(12,18),{EMPLOYEE}>

Associacoes requeridas pelo Grafo(3011)

29) <3011,<2,3>,3001,(2,3),{EMPLOYEE}>
30) <3011,<2,3>,3001,(2,4),{EMPLOYEE}>
. . .
42) <3011,<2,3>,3016,(12,18),{EMPLOYEE}>

ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS DTU-PATHS-INTRA -
EMPLOYEE

Associacoes requeridas pelo Grafo(3001) - insemp

01) 3001 2 3 4 5 - 3001 1 2 3
02) 3001 2 3 4 5 - 3001 1 2 4
. . .
14) 3001 2 3 4 5 - 3016 1 11 12 18

Associacoes requeridas pelo Grafo(3006) - delemp

15) 3006 2 3 4 5 - 3001 1 2 3
16) 3006 2 3 4 5 - 3001 1 2 4
. . .
28) 3006 2 3 4 5 - 3016 1 11 12 18

Associacoes requeridas pelo Grafo(3011) - updemp

29) 3011 2 3 4 5 - 3001 1 2 3
30) 3011 2 3 4 5 - 3001 1 2 4
. . .
42) 3011 2 3 4 5 - 3016 1 11 12 18
```

*Exemplo 6.7: Arquivos dos elementos requeridos pelos critérios todos os t-usos-ciclo1-intra e todos os dtu-caminhos-intra.*

As informações de re-instrumentação são utilizadas no teste de integração *intra-modular* e no teste de integração *inter-modular*, além das informações sobre as tuplas utilizadas na execução de cada caso de teste.

No caso específico da *variável tabela* EMPLOYEE foram executados 65 casos de teste para satisfazer os elementos requeridos pelo teste de integração baseado na dependência de dados.

A *Figura 6.5* mostra os arquivos *Pathint.tes* e *Keyint.tes*, gerados pela execução dos casos de teste referentes à tabela EMPLOYEE no teste de integração baseado na dependência de dados *intra-modular*. Esses arquivos armazenam as informações referentes aos caminhos e tuplas (para cada unidade envolvida no teste de integração), utilizadas na avaliação da cobertura dos elementos requeridos pelos critérios de teste de integração.

A avaliação da cobertura dos elementos requeridos pelos critérios propostos nesta tese foi realizada manualmente. Os elementos não exercitados são examinados isoladamente, analisando-se a possibilidade de serem exercitados por outros casos de teste. Neste caso, novos casos de teste são incluídos.

Sempre que executamos um mesmo conjunto de casos de teste em programas convencionais obtém-se a mesma cobertura. No caso do teste de programas de ABDR, devido à persistência da *variável tabela*, a execução de um mesmo conjunto de casos de teste pode ocasionar coberturas distintas. Assim, é necessário restaurar os estados iniciais das tabelas envolvidas no teste antes de iniciarmos a execução dos casos de teste. O estado inicial de cada tabela é denominado de estado "*start"*, que é o estado em que a tabela deve se encontrar no início do teste. O estado "*start*" da tabela é preparado com o menor número de elementos que represente todas as dependências e restrições do projeto lógico da base de dados. Para retornar ao estado "*start"*, após uma sessão de teste é preciso recuperar os valores de todos os elementos que foram modificados durante o teste.

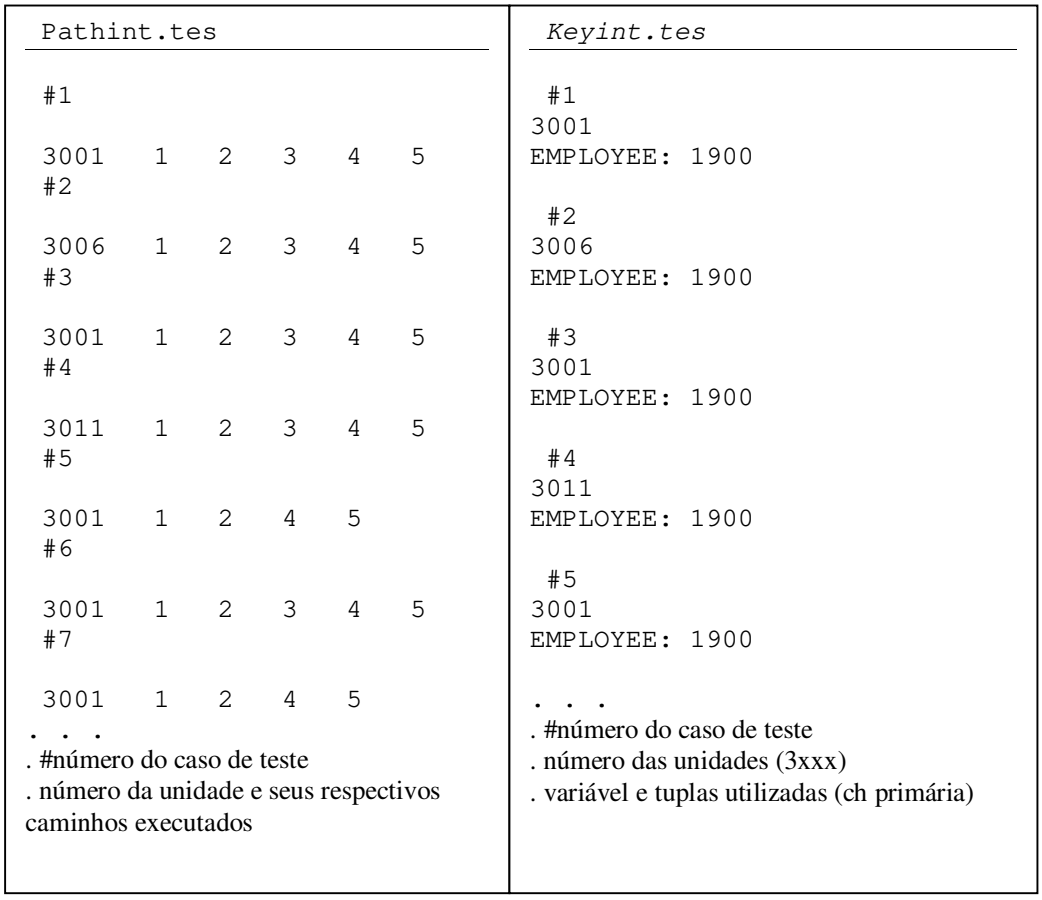



Figura 6.5: *Arquivos* Pathint.tes e Keyint.tes *informações sobre a execução do teste de integração* intra-modular.

A cobertura de cada critério é avaliada manualmente. Para simplificar a análise de cobertura foi criado um arquivo denominado *aval_int* que apresenta todos os elementos requeridos de um critério. A partir dos arquivos da Figura 6.5 é feita uma avaliação de quais elementos requeridos (do arquivo *aval_int*) foram exercitados. O arquivo *aval_his* contém as informações relacionadas à cobertura dos elementos requeridos e quais resultados foram obtidos para compará-los com os resultados esperados. O objetivo desse arquivo é simplesmente armazenar a avaliação de cobertura e os resultados do teste. No caso do teste de ABDR, existem elementos requeridos que não são exercitados com a mesma tupla; esses elementos podem ser descartados após algumas tentativas. Esses elementos forçam o testador a exercitar situações que podem revelar defeitos de implementação. Por exemplo, no comando DELETE ao eliminar uma tupla que é chave estrangeira (possui uma referência a outra tabela) neste caso a tupla não é removida.

O critério *todos os t-usos-ciclo2-intra*, que exercita as dependências múltiplas, foi aplicado ao Módulo $Mod_3$, para a Unidade *Selfin* (3019) que emite um relatório referente às tabelas FIN_CODE e FIN_DATE. Os elementos requeridos pelo critério de *ciclo2-intra*, apresentados no Exemplo 6.8, exercitam as associações ocasionadas pelas variáveis FIN_CODE e FIN_DATE em relação às unidades *Insfinc* (3005), *Insfind* (3006), *Delfinc* (3009), *Delfind* (3010), *Updfinc* (3014) e *Updfind* (3015).

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODAS DEF-T-USOS-CICLO2-INTRA
FIN_CODE/ FIN_DATE

Associacoes requeridas pelo Grafo(3004/3005)

1) <3004, <2,3>, 3005, <2,3>, 3019, (2,3), {FIN_CODE,FIN_DATE}>
2) <3004, <2,3>, 3005, <2,3>, 3019, (2,8), {FIN_CODE,FIN_DATE}>

Associacoes requeridas pelo Grafo(3009/3010)

3) <3009, <2,3>, 3010, <2,3>, 3019, (2,3), {FIN_CODE,FIN_DATE}>
4) <3009, <2,3>, 3010, <2,3>, 3019, (2,8), {FIN_CODE,FIN_DATE}>

Associacoes requeridas pelo Grafo(3014/3015)

5) <3014, <2,3>, 3015, <2,3>, 3019, (2,3), {FIN_CODE,FIN_DATE}>
6) <3014, <2,3>, 3015, <2,3>, 3019, (2,18), {FIN_CODE,FIN_DATE}>
```

*Exemplo 6.8: Arquivo dtuassoc.tes elementos requeridos do critério Todos os t-usos-ciclo2-intra*

Este critério também pode ser aplicado quando existe a necessidade de se definir uma tabela para conseguir a *associação-definição-t-uso* de outra *variável tabela*.

### 6.3.2 Teste de integração inter-modular

A integração *inter-modular* é realizada com base nas dependências de dados entre as tabelas e os Módulos da aplicação. Várias associações *inter-modular* são requeridas pelos critérios para o exemplo de aplicação.

Nos módulos $Mod_1$, $Mod_2$, $Mod_3$ e $Mod_6$, existem várias combinações de associações com relação a diferentes *variáveis tabela*. Para continuar a ilustrar o critério *inter-modular*, apresenta-se uma associação entre os Módulos $Mod_3$ e $Mod_6$ em relação à variável ***Employee***. A Figura 6.6 mostra uma tela do teste de integração *inter-modular* dos Módulos $Mod_3$ e $Mod_6$ (envolvendo as Unidades *InsEmp* (3001) do Módulo $Mod_3$ e a Unidade *RelGer* (6002) do Módulo $Mod_6$).

O Exemplo 6.9 apresenta os elementos requeridos do critério *todos os t-usos-ciclo1-inter* para a tabela **Employee** e Módulos $Mod_3$ e $Mod_6$.

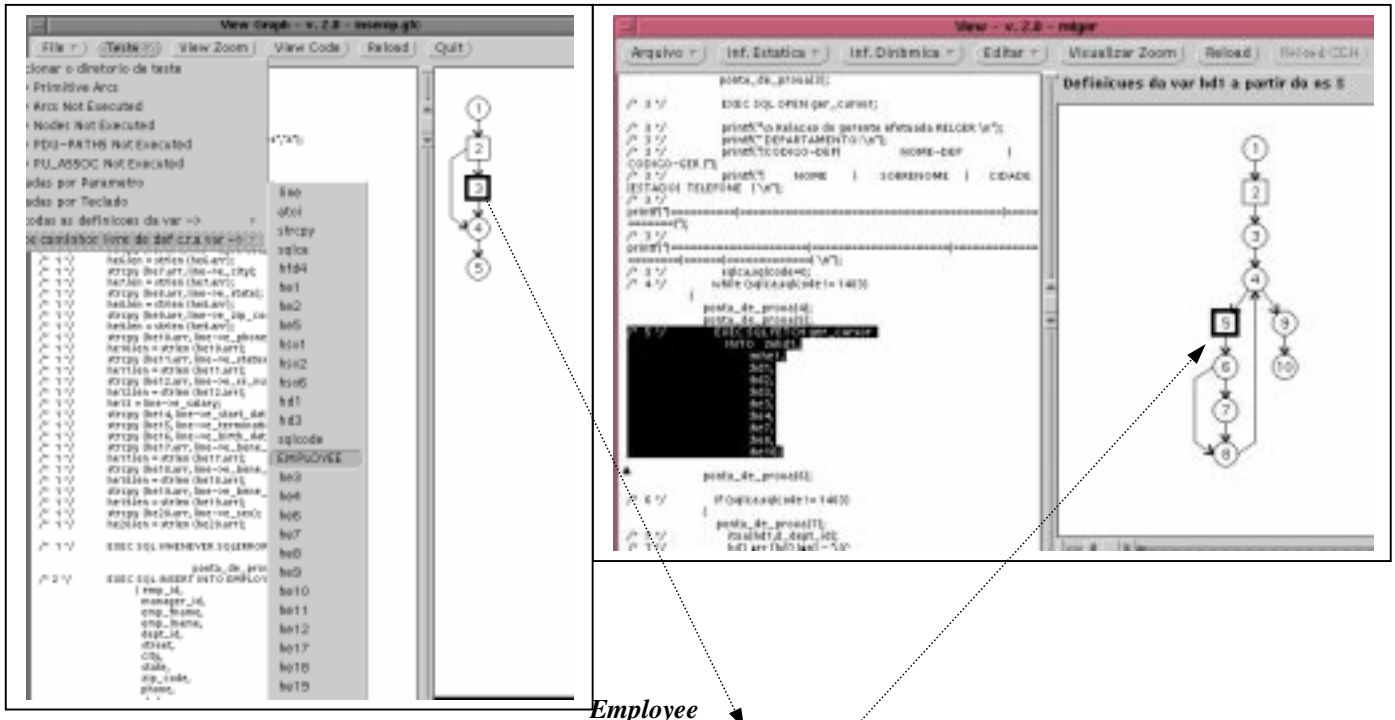
```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS DE INTEGRACAO TODOS_T_USOS
INTER_MODULAR: MOD3 X MOD6 -- CICLO1 - MESMA TUPLA - EMPLOYEE

Associacoes originadas de Insemp (3001)

01- <3001, <3,4>, 6002, (2,3),EMPLOYEE>
02- <3001, <3,4>, 6002, (5,6),EMPLOYEE>
. . .
06- <3001, <3,4>, 6004, (5,6),EMPLOYEE>

Associacoes originadas de Delemp (3006)

07- <3006, <3,4>, 6002, (2,3),EMPLOYEE>
08- <3006, <3,4>, 6002, (5,6),EMPLOYEE>
. . .
12- <3006, <3,4>, 6004, (5,6),EMPLOYEE>

Associacoes originadas de Updemp (3011)

13- <3011, <3,4>, 6002, (2,3),EMPLOYEE>
14- <3011, <3,4>, 6002, (5,6),EMPLOYEE>
. . .
18- <3011, <3,4>, 6004, (5,6),EMPLOYEE>
```

*Exemplo 6.9: Arquivo dtuassoc_emp.tes;*
Elementos requeridos do critério *todos os t-usos-ciclo1-inter.*

Este é um exemplo montado para exercitar a integração das unidades do Módulo $Mod_6$, que não têm nenhuma *definição persistente* das *variáveis tabela* referenciadas neste Módulo. Sendo assim, não existe nenhum elemento requerido pelos critérios de integração *intra-modular* para o Módulo $Mod_6$. Com a integração *inter-modular* foi possível exercitar as unidades do Módulo $Mod_6$ integrando-o com outros Módulos ($Mod_2$ e $Mod_3$). A POKE-TOOL não apóia o teste de integração; para executar o teste de integração *inter-modular* o programa *pokeexec* foi substituído pelo programa *pokeexec_inter*, que executa os casos de teste para a integração *inter-modular*. O programa *pokeexec_inter* é decomposto em dois programas, visando a satisfazer os critérios de *ciclo1* e *ciclo2* para o teste *inter-modular*. Os critérios *todos os t-usos-ciclo1-inter* e *todos os dtu-caminhos-inter* foram aplicados através do programa *pokeexec_inter_1* e o critério *todos os t-usos-ciclo2-inter* foi aplicado através

do programa *pokeexec_inter_2*. O Exemplo 6.10 mostra os arquivos de saída *pathint.tes* e *keyint.tes*, gerados pela execução dos casos de testes.

Esses arquivos são usados na avaliação da cobertura dos critérios *todos os t-usos-ciclo1-inter* e *todos os dtu-caminhos-inter*.



***Employee***

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *3* | *4* | *y* | *r* | *e* | *7* | *8* | *e* | *w* | *1* | *y* | *u* | *f* | *r* | *5* | *7* | *3* |
| *4* | *5* | *R* | *R* | *d* | *5* | *7* | *t* | *q* | *4* | *r* | *d* | *j* | *e* | *7* | *8* | *0* |
| *5* | *4* | *F* | *F* | *c* | *3* | *6* | *r* | *t* | *6* | *w* | *s* | *b* | *h* | *9* | *3* | *1* |
| *2* | *3* | *G* | *R* | *b* | *2* | *5* | *s* | *r* | *5* | *w* | *y* | *s* | *g* | *6* | *7* | *4* |
| *7* | *2* | *T* | *F* | *h* | *7* | *2* | *d* | *u* | *4* | *r* | *j* | *c* | *n* | *5* | *5* | *1* |
| *6* | *7* | *Y* | *R* | *j* | *8* | *8* | *t* | *q* | *4* | *r* | *n* | *j* | *e* | *7* | *8* | *0* |
| *8* | *8* | *N* | *F* | *k* | *1* | *9* | *r* | *t* | *6* | *w* | *s* | *b* | *h* | *9* | *3* | *1* |
| *9* | *9* | *H* | *R* | *l* | *3* | *0* | *f* | *i* | *5* | *r* | *x* | *s* | *g* | *3* | *3* | *1* |
| *0* | *7* | *J* | *T* | *q* | *2* | *1* | *h* | *o* | *6* | *w* | *v* | *c* | *n* | *2* | *2* | *2* |
| *2* | *0* | *M* | *r* | *w* | *5* | *2* | *t* | *q* | *4* | *r* | *D* | *j* | *e* | *7* | *8* | *0* |

*Figura 6.6: Teste de Integração inter-modular:* Unidade *Insemp* do Módulo $Mod_3$ e Unidade *RelGer* do Módulo $Mod_6$.

Os arquivos para a avaliação desses critérios são organizados da mesma forma que os arquivos para avaliação do teste de integração (*intra-modular*), referente aos mesmos critérios.

Os critérios *todos os t-usos-ciclo1-inter* e *todos os t-usos-ciclo2-inter* ($Mod_3 x Mod_6$) para a variável tabela EMPLOYEE requerem 18 (dezoito) associações. Para satisfazer essas associações foram necessários 18 casos de testes, sendo 9 para executar o Módulo $Mod_3$ e 9

para $Mod_6$ no ciclo1 (associações apresentadas no Exemplo 6.9); para o ciclo2 foram necessários 27 casos de testes, sendo 18 para o $Mod_3$ e 9 para o $Mod_6$ (associações apresentadas no Exemplo 6.10).

O principal objetivo do teste estrutural de programas de aplicação é detectar a presença de defeitos relacionados à implementação. Contudo, existem situações em que o teste em ABDR exercita: *i)* a consistência nos comandos de manipulação da **SQL** em relação a um atributo das *variáveis tabela* (por exemplo: se, no comando UPDATE, o programador permitisse uma alteração de um atributo que é chave estrangeira de outra tabela – acusaria o erro de consistência); *ii)* se a segurança das transações entre os comandos de manipulação da base de dados está correta (exemplo: um comando que removesse uma tupla que tem uma regra relacionada a outra tabela – o sistema não deveria permitir a exclusão da tupla). Isso foi observado após a execução dos casos de testes na

```
#1

3001   1   2   3   4   5
#1

6002   1   2   3   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   8   4   9
10
#2

3011   1   2   3   4   5
#2

6002   1   2   3   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   7   8   4   5   6   7   8   4
 5   6   8   4   9  10
#3

3001   1   2   3   4   5
#3

6003   1   2   3   4   5   6   7   8   4
 5   6   8   4   9  10
#4

3011   1   2   3   4   5
```

```
 #1
3001
EMPLOYEE: 1920

 #1
6002
DEPARTAMENTO: 400
EMPLOYEE: 1920

 #2
3011
EMPLOYEE: 1920

 #2
6002
DEPARTAMENTO: 300
EMPLOYEE: 1920

 #3
3001
EMPLOYEE: 1901

 #3
6003
DEPARTAMENTO: 200
EMPLOYEE: 902

 #4
3011
EMPLOYEE: 1576

 #4
6003
DEPARTAMENTO: 400
EMPLOYEE: 1576
```

*Figura 6.7: Arquivos Pathint.tes – caminhos executados e Keyint.tes – seqüência de tuplas.*

integração *intra-modular* e *inter-modular*. Defeitos em comandos de consultas da **SQL** (uso de "*queries*" indevidas) são detectados ao exercitar os elementos requeridos pelos critérios de integração (*intra-modular* e *inter-modular*).

Várias associações com *dependência múltipla* foram detectadas nos Módulos $Mod_3$ e $Mod_6$. Neste caso foi gerado o arquivo *dtu_assoc_inter_mod3x6.tes*, mostrado no Exemplo 6.10. Aqui vemos uma situação onde o Módulo $Mod_6$ não possui nenhum elemento requerido para o teste de integração *intra-modular* e no entanto ele é exercitado no teste de integração *inter-modular*.

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS DE INTEGRACAO TODOS_T_USOS
INTER_MODULAR: MOD3 X MOD6 -- CICLO2 - MESMA TUPLA - EMPLOYEE,
SALES_ORDER E DEPARTAMENTO

Associacoes originadas de Insemp(3001),Inssal(3002) e Insdep(3003)

01- <3003,(3,4),3001,(3,4),6002,(2,3),{DEPARTAMENTO,EMPLOYEE}> * 1
02- <3003,(3,4),3001,(3,4),6002,(5,6),{DEPARTAMENTO,EMPLOYEE}> * 1
. . .
06- <3001,(3,4),3002,(3,4),6004,(5,6),{SALES_ORDER,EMPLOYEE}> * 3

Associacoes originadas de Delemp(3006),Delsal(3007) e Deldep(3008)

07- <3006,(3,4),3008,(3,4),6002,(2,3),{DEPARTAMENTO,EMPLOYEE}>
08- <3006,(3,4),3008,(3,4),6002,(5,6),{DEPARTAMENTO,EMPLOYEE}>
. . .
12- <3007,(3,4),3006,(3,4),6004,(5,6),{SALES_ORDER,EMPLOYEE}>

Associacoes originadas de Updemp(3011), Updsal(3012) e Upddep(3013)

13- <3013,(3,4),3011,(3,4),6002,(2,3),{DEPARTAMENTO,EMPLOYEE}>
14- <3013,(3,4),3011,(3,4),6002,(5,6),{DEPARTAMENTO,EMPLOYEE}>
. . .
18- <3011,(3,4),3012,(3,4),6004,(5,6),{SALES_ORDER,EMPLOYEE}>
```

*Exemplo 6.10: Elementos requeridos pelo critério todos os t-usos-ciclo2-inter.*

As comparações apresentadas visam a avaliar a utilização dos critérios de teste propostos. Um dos principais problemas no teste de integração é a geração do grande volume de dados em cada tipo de integração (*intra-modular* e *inter-modular*), dificultando a elaboração da análise de cobertura. A cobertura neste caso deve ser observada entre as ocorrências das Unidades envolvidas na definição da *variável tabela* com a Unidade que usou a *variável tabela* para a mesma tupla, um exemplo é mostrado no final do Apêndice C.

## 6.4 Comparação da utilização dos critérios

Inicialmente é mostrado o total parcial do número de casos de testes necessários para o teste de unidade e para o teste de integração.

- **Teste de unidade**

A *Tabela 6.2* mostra o número de casos de teste necessários para satisfazer os critérios (da FCPU) de Teste de Unidade para cada Módulo do exemplo.

*Tabela6.2: Quadro geral dos Módulos de Programas executados no teste de Unidade.*

| **Módulo de Programa** | **Número de Casos de testes** | **No. de Ups em teste** |
|---|---|---|
| $Mod_2$ | 168 | 07 |
| $Mod_3$ | 98 | 20 |
| $Mod_6$ | 56 | 06 |

No teste de integração, os casos de testes são escolhidos para cada *variável tabela* envolvida nas associações, devido à exigência de mesma tupla para satisfazer os critérios de teste de integração.

- **Teste de integração**

A *Tabela 6.3* apresenta o total de elementos requeridos (*elem.req.*) pelo critério *Todos* os *t-usos-ciclo1-intra*, para as *variáveis tabela* associadas no Módulo $Mod_3$; apresenta também o total de elementos requeridos não executáveis (*ne*) dando como resultado, a porcentagem de cobertura atingida para este critério.

*Tabela 6.3: Testes para o critério de integração todos os t-usos-ciclo1-intra para o Módulo $Mod_3$*

| ***Tabelas*** | ***Total de elem.req. (ne)*** | ***N$^{os}$ de Casos de testes*** | ***Cobertura*** |
|---|---|---|---|
| Departamento | 36 (9) | 60 | 75% |
| Employee | 42 (10) | 65 | 76,19% |
| Fin | 48(24) | 60 | *50% |
| Sales | 36(10) | 49 | 72,20% |

A Tabela 6.4 mostra o resultado para tabelas associadas nos testes de integração (*intra-modular* e *inter-modular*) dos Módulos para os critérios de *ciclo1* e *ciclo2.*

- **Observações**

1. No teste de integração *intra-modular-ciclo1* (Tabela 6.3) para o Módulo $Mod_3$ foram necessários 234 casos de testes para satisfazer os elementos requeridos das associações entre as *UPs* do Módulo $Mod_3$; no teste de unidade para os critérios da FCPU (Tabela 6.2) foram necessários 98 casos de testes. No caso do Módulo $Mod_6$ não houve nenhuma associação *intra-modular*, devido à ausência de comandos que definem uma *variável tabela*; no teste de unidade (Tabela 6.2) foram necessários 56 casos de testes para satisfazer os elementos requeridos.
2. No teste de integração *intra-modular* não existe nenhum elemento requerido para o Módulo $Mod_6$; na integração *inter-modular* existem 48 elementos requeridos em cada ciclo para o critério *todos os t-usos* (sendo 42 com $Mod_3$ e 6 com o $Mod_2$). Os erros ocasionados pelas queries durante as consultas só foram observados com a integração entre os módulos envolvendo o $Mod_6$.

*Tabela 6.4: Elementos requeridos pelo critério todos os t-usos-inter para os módulos $Mod_2$, $Mod_3$ e $Mod_6$.*

| $Mod_{def}$ | $Mod_{uso}$ | *Elementos Requeridos Todos os t-usos* | | *Casos de testes* | |
|---|---|---|---|---|---|
| | | *Ciclo1* | *ciclo2* | *ciclo1* | *Ciclo2* |
| $Mod_2$ | $Mod_3$ | **9** | **-** | **18** | **-** |
| $Mod_2$ | $Mod_6$ | **6** | **6** | **12** | **18** |
| $Mod_3$ | $Mod_2$ | **9** | **-** | **18** | **-** |
| $Mod_3$ | $Mod_6$ | **42** | **42** | **40** | **60** |

3. Esses resultados mostram que os critérios de teste de unidade, de integração *intra-modular* e de integração *inter-modular*, além de serem incomparáveis teoricamente, não podem tampouco ser comparados quanto à sua força relativa (não se pode concluir sobre a dificuldade relativa de satisfazê-los). Isto se deve ao fato de que cada um deles enfoca aspectos distintos de programas de uma ABDR. Os erros observados pelos critérios da FCPU não destacam os erros

referentes ao uso dos comandos da **SQL**, sendo esses observados somente com os critérios baseados da dependência de dados *intra-modular* e *inter-modular*.

## 6.5 Considerações Finais

O exemplo de aplicação apresentado neste capítulo mostra que é possível utilizar os critérios propostos e analisar os diferentes tipos de fluxos de dados de uma ABDR. Foi necessário adaptar a ferramenta POKE-TOOL, modificando manualmente os arquivos utilizados pela ferramenta. As ferramentas POKE-TOOL e VIEWGRAPH (V.2.0) foram essenciais para a elaboração do exemplo em tempo hábil.

Os exemplos de utilização mostram que os critérios são complementares e importantes no teste de programas de aplicação, por exigirem, características exclusivas de ABDR. O teste *inter-modular* complementa o teste *intra-modular* através da combinação de módulos que somente consultam uma *variável tabela* com módulos que possuem *definição persistente* da mesma tabela.

Durante a realização do teste, notou-se que a exigência de mesma tupla para satisfazer os critérios de integração faz com que exercitemos diferentes tipos de combinações de atributos das *variáveis tabela*; isto força o entendimento das condições do projeto do Banco de Dados (consistência da base de dados em relação às chaves primárias e estrangeiras; restrições ou regras existentes no projeto; etc), dando a oportunidade de detecção de defeitos de implementação dos comandos **SQL**. Em nosso exemplo, defeitos foram detectados no teste de integração *intra-modular* e *inter-modular* relativamente aos comandos **SQL**.

Também foi possível observar a existência de algumas *associações definição-t-uso* que podem ser omitidas ou cuja dificuldade de exercitar pode ser indicada ao testador. A Tabela 6.6 apresenta uma relação entre as possíveis associações dos comandos que definem a variável ***t*** e comandos que usam ***t***. Os comandos colocados na coluna são os responsáveis pela *definição persistente* de ***t***. Os comandos colocados na linha superior da tabela são os responsáveis pelo ***t-uso***. As colunas **Error** correspondem às associações com os arcos de saída que representam o arco de tratamento de erro dado pelo comando **WHENEVER *<condição> <ação>*** e as colunas **!Error** representam o fluxo de saída sem a ocorrência de erro.

*Tabela 6.6: Associações entre os comandos da SQL* c.r.a. ***t***

| ***t*-uso** | **INSERT** | | **DELETE** | | **UPDATE** | | **SELECT** | |
|---|---|---|---|---|---|---|---|---|
| *Definição* ***t*** | **Error** | **!Error** | **Error** | **!Error** | **Error** | **!Error** | **Error** | **!Error** |
| **INSERT** | E | * | * | E | E | E | $E^+$ | $E^+$ |
| **DELETE** | *E | E | E | * | E | E | $E^+$ | $E^+$ |
| **UPDATE** | * | E | * | E | E | E | $E^+$ | $E^+$ |

***** omitir ***E** casos críticos **E** exercitar $\mathbf{E^+}$ exercitar com mais atenção

Observe que 50% das associações cujos ***t*-usos** estão em nós com INSERT e DELETE não precisam ser exercitadas (casos críticos como presença de condição CASCADE devem ser observados – *E). As associações (E+) que possuem um ***t*-uso** no comando SELECT são todos necessários para explorar as situações em que ocorre mais do que um elemento consultado ou casos onde ocorre duplicação de chaves utilizadas no predicado do comando. Neste caso, deve-se forçar as diferentes situações para exercitar os arcos com **Error** ou !**Error** após o comando SELECT.

O exemplo mostra também que os critérios de teste de unidade e de integração não podem ser comparados quanto à sua força relativa devido ao fato de enfocarem aspectos distintos de programas de ABDR e, portanto, classes distintas de defeitos. No caso dos critérios de teste da FCPU visam a exercitar as variáveis que têm definição e uso na memória interna do computador, durante uma mesma execução. Já os critérios específicos para ABDR exercitam as *variáveis persistentes*, que neste caso são representadas pelas *variáveis tabela*.

# 7 Conclusões e Trabalhos Futuros

Na Seção 7.1 deste capítulo são apresentadas as conclusões desta tese; na Seção 7.2, são discutidas as principais contribuições da tese; e, finalmente, na última seção são propostos e discutidos os trabalhos de pesquisas futuros.

## 7.1 Conclusões

Apesar de os critérios de teste estrutural de software desenvolvidos nos últimos anos serem aplicáveis a diversos tipos de softwares, eles não se aplicam a todas as áreas de desenvolvimento de software. O enfoque principal deste trabalho é o desenvolvimento de programas de Aplicação de Banco de Dados Relacional com **SQL** embutida.

É proposta uma abordagem de teste estrutural baseada em dois tipos de fluxos de dados: *intra-modular* e *inter-modular*. O primeiro tipo de fluxo aplica-se ao teste isolado de programas de ABDR com **SQL** embutida; o segundo tipo de fluxo de dados aplica-se à integração de módulos, requerendo a execução de comandos **SQL** que manipulam as *variáveis tabela* de uma ABDR.

Critérios de teste estrutural podem ser usados para o teste de unidade usando o fluxo de dados *intra-modular*; novos critérios (com características exclusivas para exercitar as *variáveis tabela* de programas de ABDR) são propostos neste trabalho. É introduzida uma nova abordagem de teste de integração para exercitar associações *definição-t-uso* de *variáveis persistentes*; essa abordagem pode ser estendida para exercitar outros tipos de *variáveis persistentes* (gravadas e acessadas em meio físico).

Estabeleceu-se também uma terminologia para os critérios de teste propostos, tendo como alvo principal às *associações definição-uso* das *variáveis tabela*. Os critérios propostos para o teste de integração baseado na dependência de dados *intra-modular* e *inter-modular* exigem a mesma tupla da *variável tabela* para exercitar as associações *definição-t-uso*.

Foi apresentada uma proposta com os seguintes modelos de implementação: *i)* Modelo de instrumentação dos comandos de **SQL**; *ii)* Modelo de fluxo de controle; *iii)* Modelo de fluxo de dados; e *iv)* Descrição dos elementos requeridos pelos critérios. Esses modelos são a base de ferramentas de apoio ao teste de programas de ABDR.

A partir do exemplo de aplicação foi possível observar que os critérios possuem exigências distintas por exercitarem aspectos diferentes. Foi possível observar também que nenhum critério é mais forte do que os outros e que eles são complementares por exercitarem tipos diferentes de fluxo de dados.

Os critérios de teste de integração *intra-modular* e *inter-modular* requerem o exercício de todas as possíveis combinações de usos das "*queries*" nos comandos **SQL** para cada *variável tabela*. A realização do teste do exemplo de ABDR mostrou que é importante controlar o estado inicial da *variável tabela* para manter a mesma cobertura quando executarmos um mesmo conjunto de casos de teste. Este fato mostra que o testador deve estar bem atento quanto à escolha dos casos de teste e quanto ao estado inicial da *variável tabela*; isso equivale a estender o conjunto de casos de teste acrescentando o estado inicial de cada *variável tabela*.

O exemplo mostrou que é possível adaptar critérios de teste de unidade para programas de ABDR e que o auxílio das ferramentas de apoio ao teste facilitam a aplicação dos critérios, mesmo para programas extensos. Os objetivos iniciais foram alcançados e o teste estrutural passa a dispor de uma abordagem que pode ser estendida para outras classes de programas que utilizam *variáveis persistentes*.

## 7.2 Contribuições da Tese

As contribuições desta tese são caracterizadas pela proposição e análise de propriedades de critérios de teste para programas de ABDR com **SQL** embutida, além da utilização desses critérios em um exemplo de aplicação derivado de um caso real e do estudo dos resultados da utilização dos critérios. As principais contribuições são apresentadas e discutidas a seguir:

*i)* A primeira contribuição deste trabalho foi observar que os critérios de teste estrutural podem ser usados no teste de unidade; para completá-lo foram propostos outros critérios exclusivos para *variáveis persistentes* (*variáveis tabela*). Dois tipos de fluxos de dados foram identificados para contemplar as associações envolvendo definições e uso de *variáveis tabela*.

*ii)* Os critérios propostos, ao exigirem o uso da mesma tupla para satisfazer as *associações definição-t-uso*, distinguem-se da maioria dos demais critérios de

teste; isto obriga o testador a escolher os dados de teste com mais cuidado e resulta, ao mesmo tempo, em um teste mais efetivo ao que se refere à integridade das tabelas.

*iii)* A dependência de dados por *variáveis persistentes* define uma maneira de integrar as unidades (*fluxo intra-modular*) e integrar os Módulos (*fluxo inter-modular*) podem ser estendidas para outros tipos de programas que utilizam *variáveis persistentes*.

*iv)* Modelos de implementação dos critérios contribuem com a geração de uma possível ferramenta de apoio ao teste de programas de ABDR.

*v)* Resultados da execução de um exemplo completo de utilização dos critérios de teste de integração baseados na dependência de dados das *variáveis tabela* mostram que os critérios são complementares e suas forças relativas são incomparáveis; cada um deles enfoca aspectos distintos de programas de ABDR.

Finalizando, os objetivos iniciais foram atingidos, podendo-se caracterizar contribuições tanto no aspecto teórico relativo aos fluxos de dados *intra-modular* e *inter-modular*, como quanto à análise de propriedades e características dos critérios de teste que exercitam as *variáveis tabela*, tratadas neste contexto como *variáveis persistentes*. Quanto ao aspecto prático, foram utilizados quatro Módulos com dezenas de Unidades que manipulam nove tabelas da base de dados, que possibilitaram exercitar e demonstrar a aplicabilidade dos critérios em programas de ABDR.

A partir desses resultados espera-se que novas atividades de pesquisas sejam desenvolvidas como expandir os critérios aqui apresentados para outras aplicações de Banco de Dados como PL/SQL, DB2 etc, visando a tornar os critérios independentes da linguagem.

## 7.3 Trabalhos Futuros

As abordagens apresentadas nesta tese envolveram duas fases de teste de um programa: teste de unidade e teste de integração. Essas fases estão distribuídas e apoiadas no fluxo de dados *intra-modular* e no fluxo de dados *inter-modular*. Visando um melhor planejamento dessas fases distintas de teste, pode-se visualizar direções de pesquisas

futuras relevantes para a área de teste de programas de aplicações de Banco de Dados Relacional.

A disponibilidade de uma ferramenta de suporte ao teste de programas de ABDR é fundamental para as equipes de desenvolvimento de software de Banco de Dados, tendo em vista a grande quantidade de informações que surgem nas fases de teste de software e do elevado número de associações provenientes das integrações baseadas nas dependências de dados. Sendo assim, um trabalho futuro é adaptar a POKE-TOOL ou uma ferramenta similar para torná-la uma ferramenta de suporte ao teste de unidade para programas de ABDR e incluir o teste de integração baseado nos critérios *todos os t-usos* e *todos os dtu-caminhos intra-modular* e *inter-modular*.

Outra sugestão de trabalho futuro é aplicar os Fluxos de Dados intra-modular e inter-modular e seus respectivos critérios de teste que exercitam as *associações definição-uso* com relação às *variáveis persistentes*, em programas escritos em linguagens de programação (PL) existentes em cada SGBDR. No caso do SGBD da Oracle existe o módulo PL/SQL que possibilita gerar vários programas com **SQL** embutida e que interagem com as tabelas da base de dados; tais programas possuem comandos semelhantes aos dos programas em C, Pascal e PL/I, tornando possível à aplicação desses critérios.

Para qualquer programa que utilize *variável persistente*, pode-se aplicar os conceitos básicos desenvolvidos nesta tese para exercitar tais variáveis. Através da adaptação dos critérios que exercitam as *variáveis tabela*, poderão ser propostos novos critérios de teste para programas que utilizam variáveis com as mesmas características de uma *variável persistente*.

A exigência de se utilizar a mesma tupla para definição e uso de uma *variável persistente* (*tabela*), pode ser estudada como uma possibilidade adicional em outros critérios baseados em fluxo de dados, com o objetivo de estabelecer associações que sejam mais rigorosas. No caso de ABDR a instrumentação das tuplas nos programas deverá ser estudada visando a possibilidade de automatizar esta tarefa para o teste de integração.

Uma linha de trabalho enfocando uma técnica bem diferente é o estudo da utilização da abordagem de análise de mutantes para o teste de programas de ABDR.

Experimentos podem ser realizados com o objetivo de estudar a aplicabilidade e a eficácia dos critérios propostos e estabelecer classes de defeitos que podem ser detectados.

# Referências Bibliográficas


[ARA00] Aranha, C. M. "Sistematização para o Uso de Critérios de Testes de Banco de Dados Relacional", Tese de Doutorado em andamento, DCA/FEEC-UNICAMP, Campinas – SP – Brasil.

[BEI90] Beizer, B. "Software Testing Techniques", Van Nostrand Reinhold, N.Y., USA,1990.

[BER62] Berge, C. "Theory of Graphs and Its Applications", John Willey & Sons, N.Y, 1962.

[BUD81] Budd, T. A. "Mutation Analysis: Ideas, Examples, Problems and Prospects", Computer Program Testing, North-Holand Publishing Company, 1991, pp.129-148.

[BUT93] Butler, B. and Canter, S., "Testes de Performance: Bancos de Dados SQL", PC. Magazine Labs, Dezember 1993, pp.5-9.

[CHA91] Chaim, M. L. "Poke-tool – Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados", Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas – SP – Brasil, April 1991.

[CHA91a] Chaim, M. L., Maldonado, J. C. e Jino, M., "Projeto / Implementação de uma Ferramenta de Teste de Software", Relatório Técnico – DCA/RT/007/91 – DCA/FEEC/UNICAMP - 1991.

[CHA91b] Chaim, M. L., Maldonado, J. C. e Jino, M., "Manual de Configuração da POKE-TOOL", Relatório Técnico DCA/RT/008/91 - DCA/FEEC/UNICAMP - 1991.

[CHA91c] Chaim, M. L., Maldonado, J. C. e Jino, M., "Manual de Usuário da POKE-TOOL", Relatório Técnico DCA/RT/009/91 - DCA/FEE/UNICAMP - 1991.

[CHA97] Chaim, M. L., J. C. Maldonado e M. Jino, "Ferramentas para Teste Estrutural de Software Baseado em Análise de Fluxo de Dados: o caso da POKE-TOOL", Workshop do Projeto Validação e Teste de Sistemas de Operação, WPVTSO97, Águas de Lindóia, Janeiro, 1997.

[CHA00] Chays, D., Dan, S., Frankl, P. G., Vokolos, F. and Weyuker, E. J. "A Framework for Testing Database Applications", ISSTA'00, *ACM*, Oregon, PO, pp. 147-157, 2000.

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

[CLA89] Clarke, L. A. et al. "A Formal Evaluation of Data Flow Path Selection Criteria", IEEE TSE, 15(11), pp. 1318-1332, November 1989.

[COD70] Codd, E. "A Relational Model for Large Shared Data Banks", CACM, June 1970.

[CRU99] Cruzes, D. – "Geração e Visualização de Informações para Suporte a Depuração e Teste de Programas", Dissertação de Mestrado, DCA/FEE/ÚNICAMP, Campinas – SP – Brasil, Julho 1999.

[DAT84] Date, C. "A Critique of the SQL Database Language", ACM SIGMOD Record, 14-3, November 1984.

[DAT90] Date, C. "An Introduction to Database Systems", Fifth Edition, Vol. 1, Addison-Wesley, 1990.

[DAV97] Davis "Application Development Methodology", Univ. of California, USA, 1997 (pesquisa via Internet).

[DEL97] Delamaro, M.E.; Maldonado, J.C.; Interface Mutation: An Approach for Integration Testing, Workshop do Projeto Validação e Teste de Sistemas de Operação, Águas de Lindóia, Janeiro, 1997.

[DEL99] Delamaro, M.E. and Maldonado, J.C., "Interface Mutation: Assessing Testing Quality at Interprocedural Level", International Conference of the Chilean Computer Science Society (SCCC99), Talca - Chile, 1999.

[DEM78] De Millo, R.A., Lipton, R. J. and Syward, F. G. "Hints on Test Data Selection for the Practicing Programmer", Computer, April 1978.

[DEM80] De Millo, R. A. "Mutation Analisys as a Tool For Software Quality Assurance", in Proc. Of COMPASAC 80, Chicago-IL, October 1980.

[ELM94] Elmasri, R. and Navathe, S. B. – "Fundamentals of Database Systems" – Sec. Edition, Addison Wesley, 1994.

[FON93] Fonseca, R.P., "Suporte ao Teste Estrutural de Programas Fortran no Ambiente POKE-TOOL", Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas, SP, Janeiro, 1993.

[FRA85a] Frankl, F. G. and Weyuker, E. J., "ASSET – A System to Select and Evaluate Tests" In Proc. Of the IEEE Conference on Software Tool, April 1985, pp. 72-79.

[FRA85b] Frankl, F. G. and Weyuker, E. J., "Data Flow Testing Tool," In Proc. Softfair II, San Francisco, CA, Dezember, 1985, pp. 46-53.

[FRA86] Frankl, F. G. and Weyuker, E. J., "Data Flow Testing in The Presence of Unexecutable Paths" in Proc. Workshop on Software Testing, Banff, Canada, July 1986, pp. 4-13.

[FRA87] Frankl, F. G. "The Use of Data Flow Information for the Selection and Evaliation of Software Test Data" Phd. Thesis, Courant Institue of Mathematical Sciences, N.Y. University, October 1987.

[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, October 1988, pp. 1483-1498.

[FRA93] Frankl, F. G. and Weyuker, E. J., "Provable Improvements on Branch Testing", IEEE Trans. on Software Eng., Vol. 19, No. 10, October 1993, pp. 962-974.

[GOD75] Goodenough, J. e Gerhart, S. L., "Toward a Theory of Test Data Selection", IEEE Trans. on Software Eng., Vol.1, 1975.

[HAN92] Hanson, E. N. "Rule Condition Testing and Action Execution in Ariel"- ACM SIGMOD, June 1992.

[HAN95] Hanson, E. N. and all "Optimized Rule Condition Testing in Ariel Using Gator Networks"- National Science Foundation, October 1995.

[HAR69] Harary, F. "Graph Theory", Addison-Wesley, Reading, Mass. 1969.

[HAR91] Harrold, M. J. and Soffa, M. L. "Selecting and Using Data for Integration Testing"- IEEE Software, Vol. 8, N. 2, pp.58-65, March 1991.

[HAR93] Harrold, M. J. and Rothermel, G. "A System for Testing and Analysis of C Programs" Proc. Quality Week'93, May 1993.

[HAR94] Harrold, M. J. and Rothermel, G., "Performing Data Flow Testing on Classes", Proc. of the 2nd ACM SIGSOFT Symposium on Foundations of Soft. Eng., Vol. 19, N. 5, Dezember 1994, pp. 154-163.

[HEC77] Hecht, M. S. "Flow Analysis of Computer Programs", North Holland, N.Y., 1977.

[HER76] Herman, P. M. "A Data Flow Analysis Approach to Program Testing" Australian Computer Journal, Vol.8, N. 3, November 1976.

[HET87] Hetzel, W., "Guia Completo ao Teste de Software", Ed. Campus, Rio de Janeiro, 1987.

[HOR88] Horwitz, S., Reps, T. and Binkley, D. "Interprocedural Slicing Using Dependence Graphs", ACM SIGPLAN, Note 23, 7, July 1988, pp-10-44.

[HOR91] Horgan, J.R. and London, S. "Data Flow Coverage and the C Language", Fourth Symp. Soft. Testing, Analysis, and Verification, October 1991, pp. 87-97.

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

[HUA75] Huang, J. C. "An Approach to Program Testing", Computing Surveys, Vol. 7, No. 3, September 1975, pp. 113-128.

[ISO92] ISO/IEC 9075 - "Information Technology - Database Language - SQL", Third Edition 1992(E).

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

[KRO98] Kroenke, D.M., "Database Processing: Fundamentals, Design, and Implementation", 6ª Edição, Prentice Hall, 1998.

[LAS83] Laski, J. W. and Korel, B., "A Data Flow Oriented Program Testing Strategy", IEEE Trans. Software Eng., Vol. 9, No. 3, May 1983, pp. 347-354.

[LEI92] Leitão, P.S.J., "Suporte ao Teste Estrutural de Programas Cobol no Ambiente POKE-TOOL", Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, SP, Agosto, 1992.

[LIN90] Linnenkugel, U. and Müllerburg, M. "Test Data Selection Criteria for (Software) Integration Testing", *In First Intern. Conf. on Syst. Integration*, pp. 709-717, Morristown, N.J., USA, April 1990.

[MAL88] Maldonado, J. C.; Chaim, M. L. and Jino, M. "Seleção de Casos de Testes Baseada nos Critérios Potenciais Usos", II SBES, Canela, RS, Brasil, Outubro, 1988, pp.24-35.

[MAL88b] Maldonado, J. C.; Chaim, M. L. and Jino, M. "Resultados do Estudo de uma Família de Critérios de Teste de Programas baseada em Fluxo de Dados", Relatório Técnico, DCA/FEE/UNICAMP – RT/DCA-001/88 – Campinas – SP- Brasil, 1988.

[MAL89] Maldonado, J. C.; Chaim, M. L. and Jino, M., "Feasible Potential Uses Criteria Analysis", Relatório Técnico - DCA/FEE/UNICAMP – RT/DCA-001/89 – Campinas –SP- Brasil, 1989.

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

[MAL92] Maldonado, J. C.; Chaim, M.L.; Jino, M. - "Bridging the Gap in the Presence of Infeasible Paths: Potential Uses Testing Criteria" II International Conference of the SCCC, 1992, pp. 323-339.

[MAN89] Manilla, H. and Räihä, K.J. - "Automatic Generation of Test Data for Relational Queries"- Journal of Computer and System Sciences, Vol. 38, No.2, 1989, pp.240.

[MAR96] Marx, D. I. S. and Frankl, P. G. "The Path-Wise Approach to Data Flow Testing with Pointers Variables", In ISSTA'96, February 1996.

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

[NAK97] Nakagawa, E. Y., et al, "Aspectos e Implementação de Interfaces Gráficas do Usuário para Ferramentas de Teste de Software", Workshop do Projeto Validação e Teste de Sistemas de Operação, WPVTSO97, Águas de Lindóia, Janeiro, 1997.

[NTA84] Ntafos, S. C., "On Required Element Testing", IEEE Trans. Software Eng., Vol. SE - 10, November 1984, pp. 795-803.

[NTA88] Ntafos, S. C.,, "A Comparison of Some Structural Testing Strategies", IEEE – TSE, Vol. 14, No. 6, June 1988, pp. 868-873.

[ORA90] Oracle Corporation, - Programmer's Guide to the Oracle Precompilers - Version 1.3 - Part No. 5315 - V.1.3, Revised – Dezember 1990.

[OST91] Ostrand, T. J. and Weyuker, E. J. "Data Flow-based Test Adequacy Analysis for Languages With Pointers", In Fourth Symp. Soft. Testing, Analysis, and Verification, October 1991, pp. 74-86.

[PRE97] Pressman, R.S., Software Engineering: A Practitioner's Approach, McGraw-Hill, 4ª Ed. - New York, 1997.

[PRI87] Price, A. M., Garcia, F. e Purper, C.B., "Visualizando o Fluxo de Controle de Programas", *in Proc*. I SBES, Petrópolis, R.J., October 1987.

[RAP82] Rapps, S.and Weyuker, E.J. "Data Flow Analysis Techniques for Test Data Selection", In International Conference on Software Engineering, Tokio – Japan, September 1982, pp. 272-278

[RAP85] Rapps, S.and Weyuker, E.J. "Selection Software Test Data Using Data Flow Information", IEEE, TSE, SE-11, April, 1985, pp. 367-375.

[SPO96] Spoto, E. S.; Jino, M. and Maldonado, J. C. "Teste Estrutural de Software Programas de Aplicação de Banco de Dados Relacional", I *Workshop de Tese e Dissertações em Engenharia de Software – SBES'96*, São Carlos – SP, Brasil, Outubro,1996.

[SPO97] Spoto, E. S.; Jino, M. and Maldonado, J. C. "Teste Estrutural Baseado em Fluxo de Dados de Software Aplicativo de Banco de Dados Relacional", - *Workshop do Projeto Validação e Teste de Sistemas de Operação*, Águas de Lindóia – SP, Brasil, pp.163-176, Janeiro,1997.

[SPO99] Spoto, E. S., "Experimento de Teste Estrutural Baseado em Fluxo de Dados em Programas de Aplicação de Banco de Dados Relacional", *Relatório Técnico*, DCA/FEEC/UNICAMP – Campinas – SP, Julho 1999.

[SPO00] Spoto, E. S.; Jino, M. and Maldonado, J. C. "Teste Estrutural de Software: Uma abordagem para Aplicação de Banco de Dados Relacional", SBES'00, XIV Simpósio Brasileiro de Engenharia de Software, João Pessoa – PB, pp. 243-258, Outubro 2000.

[TAI96] Tai, K. C. "*Theory of Fault-Based Predicate Testing for computer Programs*"- IEEE, TSE, Vol.22, N.8, Agosto, 1996, pp.552-562.

[VIL94] Vilela, P. R., "Ferramenta de Visualização", Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas – SP – Brasil, Março 1994.

[VIL97] Vilela, P.R.; Maldonado, J.C. and Jino, M. "Program Graph Visualization", *Software-Practice and Experience*, 27(11), Novembro, 1997.

[VIL98] Vilela, P. R. S. – "Critérios Potenciais Usos de Integração: Definição e Análise"- Tese de doutorado – FEEC – UNICAMP, Abril, 1998.

[ZHU96] Zhu, H. – "A Formal Analysis of The subsume Relational Between Software Test Adequacy Criteria"- IEEE T.S.E., Vol. 22, No. 4, April 1996..

[WEY84] Weyuker, E. J., "The Complexity of Data Flow Criteria for Test Data Selection", Information Processing Letters, Vol. 19, No. 2, Agosto, 1984, pp. 103-109.

[WEY86] Weyuker, E. J., "Axiomatizing Software Test Data Adequacy", IEEE Trans. on Software Eng. Vol. SE – 12, No. 12, Dezembro, 1986, pp. 1128-1138.

[WEY88] Weyuker, E. J., "An Empirical Study of the Complexity of Data Flow Testing", in Proc. Of Second Workshop on Software, Verification and Analysis, Banff, Canada, Julho, 1988.

[WHI87] White, Lee J., "Software Testing and Verification", Advances in Computers – Vol.26, Ed. Acad. Press, Inc., 1987

[YAN91] Yan, S. Y. "Declarative Testing of Logic Databases"-Computer Mathematical – Vol.22, N.11, 1991, pp.39-45.

# APÊNDICE A

# A Ferramenta POKE-TOOL

Neste apêndice é apresentada uma síntese da ferramenta POKE-TOOL descrita com detalhes em [CHA91a, CHA91b, CHA91c, CHA91].

A POKE-TOOL (Potential Uses Criteria Tool for Program Testing) [MAL89, CHA91] foi desenvolvida na Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas – FEEC/UNICAMP. Essa ferramenta apóia a aplicação dos critérios Potenciais-Usos e de outros critérios estruturais como Todos-Nós e Todos-Arcos. Inicialmente, foi desenvolvida para o teste de unidade de programas escritos em C [CHA91], mas atualmente, devido à sua característica de multi-linguagem, já existem configurações para o teste de programas em Cobol e FORTRAN [LEI92, FON93].

A POKE-TOOL é uma ferramenta interativa cuja operação é orientada para sessão de trabalho. Nesta, o usuário realiza todas as tarefas de teste: análise estática da unidade a ser testada; preparação para o teste; submissão de casos de teste; avaliação e gerenciamento dos resultados de teste. Na POKE-TOOL, a sessão de trabalho é dividida em duas fases: a fase estática e a fase dinâmica [CHA97]. Na fase estática, a ferramenta analisa o código-fonte, obtém as informações necessárias para a aplicação dos critérios e instrumenta o código-fonte, inserindo pontas de prova (instruções de escrita que produzem um "*trace*" do caminho executado), gerando nova versão (versão instrumentada) da unidade em teste, que viabiliza uma posterior avaliação da adequação de um dado conjunto de casos de teste. Ao término dessa fase, a POKE-TOOL apresenta, por solicitação do usuário, entre outras coisas, o conjunto de caminhos requeridos pelo critério *todos-potenciais-du-caminhos* e o conjunto de associações requeridas pelo critério *todos-potenciais-usos* e *todos-potenciais-usos/du*. A partir dessas informações, o usuário pode projetar seus casos de teste a fim de que os mesmos executem os caminhos ou associações exigidas.

A fase dinâmica consiste do processo de execução e avaliação de casos de teste. Antes de executar os casos de teste, porém, é necessário que o programa executável seja gerado, a partir da versão instrumentada da unidade a ser testada. Após a execução dos casos de teste, a POKE-TOOL realiza a avaliação destes de acordo com um dos três

critérios Potenciais Usos. O resultado da avaliação é um conjunto de caminhos ou associações que restam ser executados para satisfazer os critérios e o percentual de cobertura do conjunto de casos de teste. Nesta fase, a ferramenta fornece também o conjunto de caminhos ou associações que foram executados, as entradas e saídas, bem como os caminhos percorridos em cada um dos casos de teste. O processo de execução/avaliação deve continuar até que todos os caminhos ou associações restantes tenham sido satisfeitos, ou seja, tenham sido executados ou tenha sido detectada a sua não executabilidade. O usuário pode interromper a sessão de trabalho sem que tenha atingido uma das situações anteriores. Nesse caso, a POKE-TOOL fornece meios para a interrupção da sessão, armazenamento dos dados gerados e do estado da ferramenta até aquele momento, e permite a recuperação da sessão de trabalho e o recomeço dos testes.

Assim, a POKE-TOOL recebe como entrada um programa escrito em linguagem de programação (no caso da primeira versão da ferramenta, um programa escrito em C), um conjunto de casos de teste e a seleção de um dos critérios (*todos-potenciais-usos, todos-potenciais-usos/Du, todos-potenciais-du-caminhos, todos-nós* e *todos-arcos*).

A estrutura da POKE-TOOL está baseada na composição de vários módulos que se comunicam através de arquivos e implementam as funções ou partes das funções descritas acima, realizando o processo de teste de forma a garantir a qualidade requerida para o mesmo. Na *Figura A1* [CHA91], é apresentado um diagrama contendo os módulos e os diversos produtos gerados pela ferramenta.

No diagrama apresentado na *Figura A1*, os retângulos representam os módulos, os losangos representam as entradas fornecidas pelos usuários à ferramenta, os círculos representam os produtos gerados, as linhas tracejadas representam o fluxo de controle e as linhas cheias representam o fluxo de informação.

O módulo Poketool (interface) é 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 em teste para a unidade escrita na

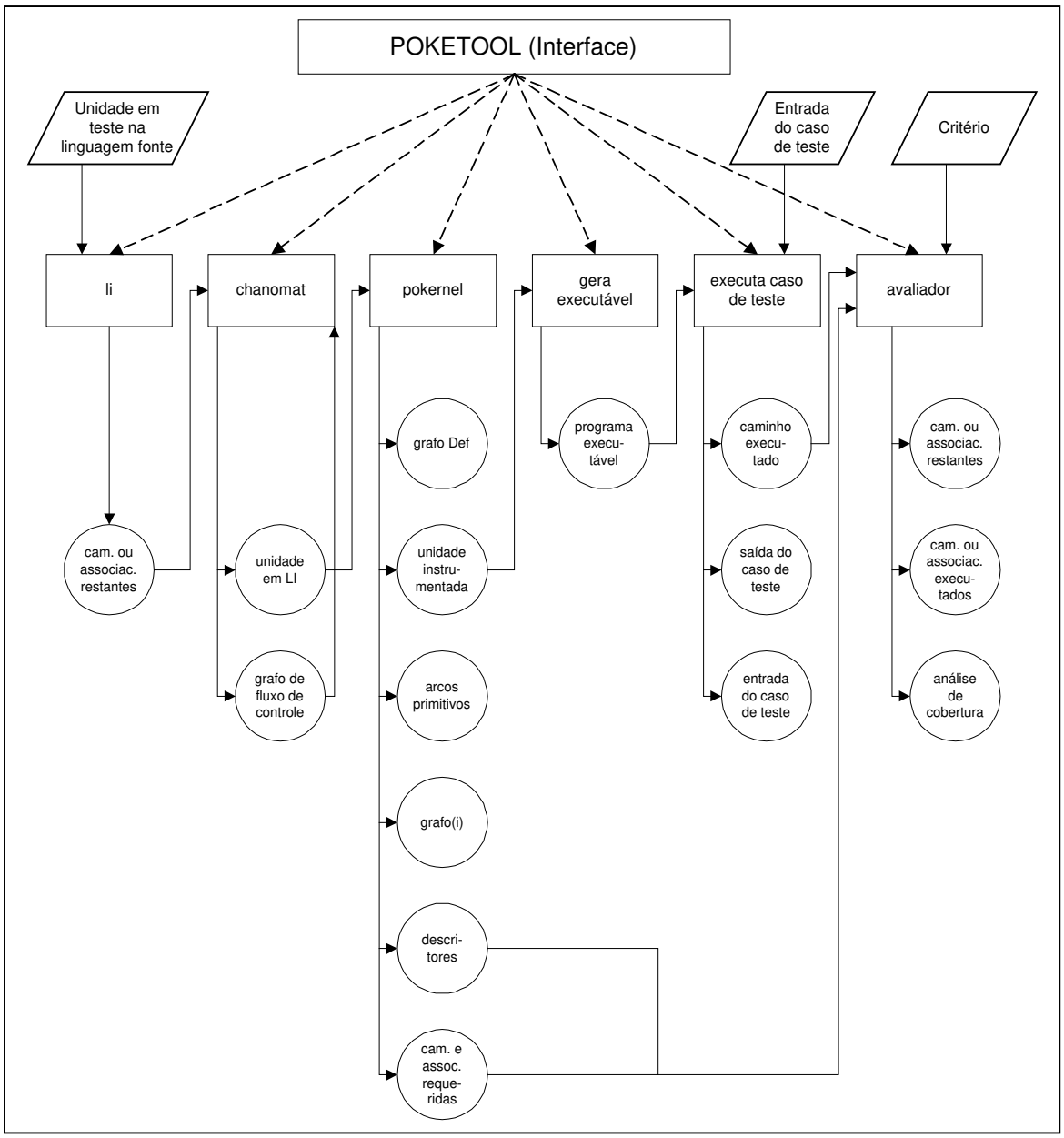



*Figura A1*: Projeto da Ferramenta POKE-TOOL

linguagem intermediária (LI). Possui como entrada a unidade em teste e como saída a unidade em LI [CHA91]. A linguagem intermediária tem como função principal identificar

o fluxo de execução em um programa. Assim, a LI tem dois tipos de comandos principais: comandos seqüenciais e comandos de controle de fluxo. Os comandos seqüenciais da LI indicam os comandos das linguagens procedurais que representam uma declaração ou computação do valor da variável (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.

O módulo Chanomat gera o GFC (Grafo de Fluxo de Controle) da unidade em teste. Possui como entrada a unidade em LI e como saídas à unidade em LI[A1] e o GFC.

O módulo Pokernel é responsável pela análise estática do código-fonte. Possui como entradas a unidade em teste, a unidade em LI numerada (Unidade na linguagem intermediária com os nós onde cada comando está relacionado) e o GFC e como saídas o grafo def (mostra quais nós têm definição e referência a variáveis, e quais são elas), a unidade instrumentada (unidade modificada contendo as pontas de prova), os arcos primitivos (Arcos dentro do GFC que são sempre executados quando um outro arco é executado), e os caminhos e associações requeridas para cada critério de teste.

O módulo Gera Executável fornece ao usuário condições para geração da unidade executável da versão instrumentada e engloba a função seleção de compilador. Possui como saída o código-executável.

O módulo Executa Caso de Teste controla a execução dos casos de testes, salvando as entradas, saídas e o caminho executado para cada caso de teste. Possui como entradas o código-executável e os casos de testes e como saídas os caminhos executados pelos casos de testes, as saídas e a entrada dos casos de testes.

O módulo Avaliador verifica quais caminhos ou associações são executadas pelos casos de testes e fornece uma análise de cobertura do conjunto de casos de testes fornecido. Gera ainda relatórios com as informações de *potenciais-du-caminho* executados e não executados, nós executados e não executados para o critério *todos-nós*, arcos executados e

[A1] Unidade em LI onde cada nó do GFC está relacionado com seus respectivos comandos na unidade em teste.

não executados para o critério *todos-arcos* e associações executadas e associações não executadas para o critério Potenciais Usos.

A POKE-TOOL encontra-se disponível para os ambientes DOS e UNIX. A versão para DOS possui interface simples, baseada em menus, A versão para UNIX possui módulos funcionais cuja utilização se dá através de interface gráfica [NAK97] ou linha de comandos(shell scripts).

As modificações a serem executadas na ferramenta POKE-TOOL passam por duas etapas distintas. A primeira etapa é a inclusão da LI para os comandos da **SQL**. Nesta etapa o módulo LI será modificado para o módulo denominado $LI_{BD}$, onde o novo módulo $LI_{BD}$ irá conter a mesma LI anterior acrescida da LI dos comandos de **SQL**. A modificação do módulo LI para $LI_{BD}$ reflete automaticamente na modificação do módulo Chanomat ocasionando os arquivos LI e grafo de fluxo de controle já incluindo os comandos de **SQL**.

A segunda etapa se refere à modificação do módulo Pokernel que a partir dos arquivos LI e grafo_def gerados pelo módulo Chanomat é gerado o novo programa-fonte instrumentado (já com os comandos de **SQL**) e os demais arquivos que são gerados pelo módulo Pokernel já incluindo as informações dos comandos de **SQL** e *variáveis host* e *variáveis tabela*. Nesta etapa é gerado o arquivo tabvardef.tes que contém todas as variáveis que são definidas em cada nó do grafo. A partir deste arquivo são gerados os descritores para os critérios da FCPU.

Com essas modificações é possível executar a etapa de teste de unidade para os *critérios Potenciais Usos*. Na etapa de teste de integração, será necessária a construção de novos módulos a fim de atender as necessidades dos critérios de teste de integração estudados nesta tese. Essas modificações serão nossos principais temas de estudos futuros, visando a elaboração de uma ferramenta que organize e auxilie o teste de integração para aplicações de Banco de Dados Relacional, aplicando os critérios estudados nesta tese.

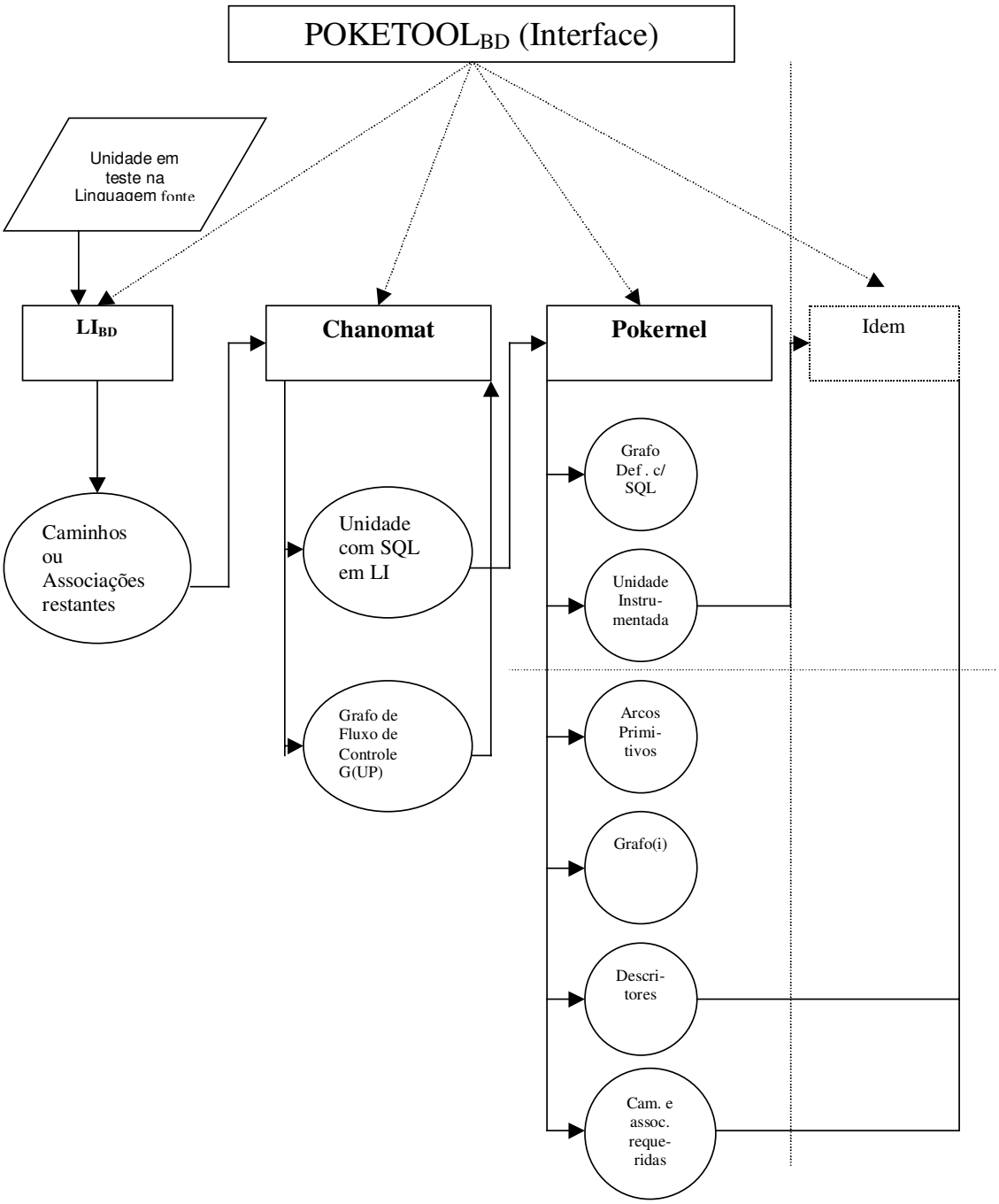



Figura A2: Apresenta uma proposta de modificação do projeto da ferramenta POKETOOL para atender programas de ABDR na etapa de teste de unidade.

# APÊNDICE B

# Comandos de SQL

Nesta seção são apresentados os comandos de **SQL** estudados e aplicados no desenvolvimento desta tese. Apesar da existência de vários Sistemas Gerenciadores de Banco de Dados Relacional, observe que, em grande parte desses sistemas são usados comandos de **SQL** considerados padrão, existindo assim, pouca distinção entre eles.

A Figura B1 apresenta um conjunto de comandos de **SQL** embutida agrupados pelo tipo que cada comando exerce na ABDR. Inicialmente, são apresentados os comandos declarativos e em seguida os comandos executáveis.

Os comandos executáveis são divididos em duas classes de comandos sendo comandos de definição de dados e comandos de manipulação de dados (DDL e DML). Os comandos que compõem a DDL são divididos da seguinte maneira: para definição de dados e para acesso de controle de dados. Os comandos que compõem a DML são divididos em comandos de manipulação de dados, comandos de recuperação de dados, comandos para processo de transação de dados e para uso de **SQL** dinâmica.

Os comandos que possuem um * são apropriados para o modo interativo (SQLPLUS). Eles são comandos próprios para **SQL** embutida em programas hospedeiros (C, Pascal, Ada etc).

Os principais comandos que foram considerados neste trabalho serão apresentados com mais detalhes a seguir. Esses comandos são facilmente encontrados em livros de Banco de Dados. Outros comandos podem ser incluídos no futuro se houver necessidades.

No Sistema Gerenciador de Banco de Dados da Oracle os comandos **SQL** podem ser utilizados tanto nos módulos de PL/SQL como, embutidos nos programas de ABDR. Para o caso de programas de ABDR existem pré-compiladores para transformarem os programas com os comandos de **SQL** para um programa na linguagem hospedeira. No caso da linguagem C o programa é inicialmente escrito com a extensão “pc” e em seguida transformado na extensão “c”.

| | |
|---|---|
| **DECLARATIVE** | |
| BEGIN DECLARE SECTION *<br>END DECLARE SECTION * | Para declarar as *variáveis host* |
| DECLARE * | Para declarar objetos |
| INCLUDE * | Para áreas de comunicacão |
| WHENEVER * | Para tratamento de erros |
| **EXECUTABLE** | |
| Definição de Dados | |
| ALTER<br>CREATE<br>DROP<br>RENAME | Para definicão de dados |
| CONNECT<br>GRANT<br>LOCK TABLE<br>REVOKE | Para controlar acesso aos dados |
| Manipulação de Dados | |
| DELETE<br>INSERT<br>UPDATE | Para manipular dados |
| CLOSE*<br>FETCH*<br>OPEN*<br>SELECT | Para recuperar dados |
| COMMIT<br>ROLLBACK<br>SAVEPOINT<br>SET TRANSACTION | Para processar transacões |
| DESCRIBE*<br>EXECUTE*<br>PREPARE* | Para uso de SOL dinâmico |

*Figura B1*: *Comandos da **SQL** embutida, agrupados por tipos. Os comandos com * não existem no modo interativo.*

As *variáveis host* são as chaves de comunicação entre a linguagem hospedeira e a Base de Dados. Uma *variável host* pode ser declarada da mesma maneira que as demais variáveis existentes no programa e compartilhada com os atributos das tabelas da base de dados. As *variáveis host* são usadas tanto para levar valores do programa para as tabelas,

como trazer os valores das tabelas para o programa, por isso são também conhecidas como *variáveis de ligação*. A única restrição é que as *variáveis host* devem ser pré-fixadas com ":" quando forem referenciadas nos comandos de **SQL**. No programa hospedeiro pode ser usada em qualquer expressão como as *variáveis programa*.

Outras características referentes à *variável host* podem ser encontradas nos SGBDRs existentes no mercado.

**COMANDOS DE SQL EMBUTIDA**

A seguir apresentaremos os principais comandos utilizados em programas de Aplicação. Lembre que os comandos de definição de dados não são apresentados abaixo por considerar que esses comandos são utilizados antes da implementação da Aplicação.


Marca o início do diagrama


Marca o final do diagrama

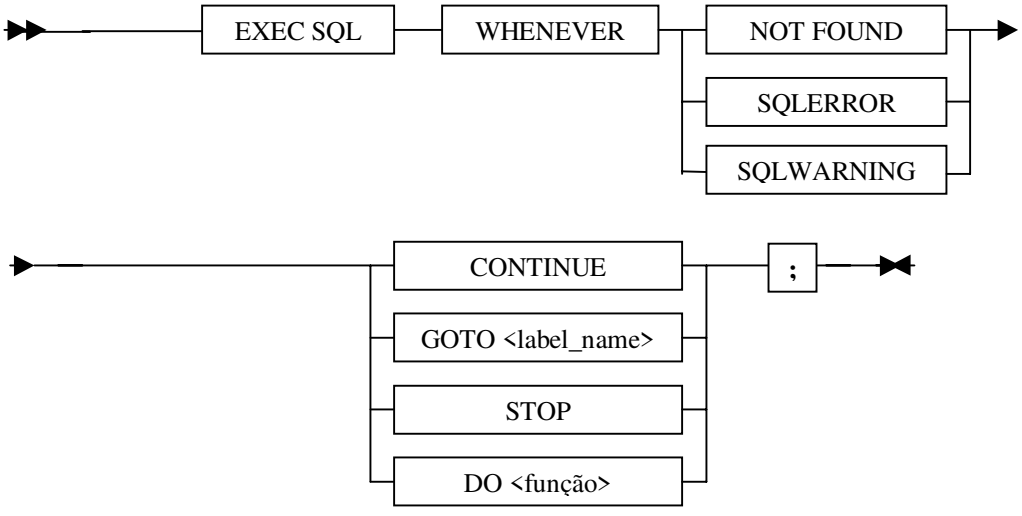

Mostra que o diagrama continua na linha abaixo


Mostra a continuação do diagrama da linha de cima


Representa um laço

**COMANDOS DECLARATIVOS**:

**WHENEVER**

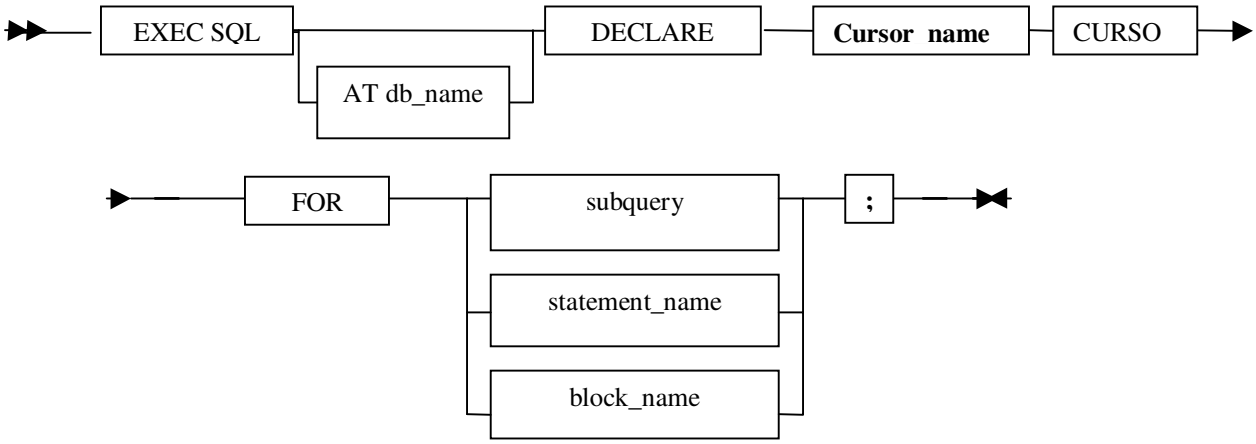



Ex: EXEC SQL WHENEVER NOT FOUND CONTINUE;

- EXEC SQL WHENEVER SQLERROR GOTO parar;

- EXEC SQL WHENEVER SQLWARNING STOP;
- EXEC SQL WHENEVER SQLERROR DO func_parar();

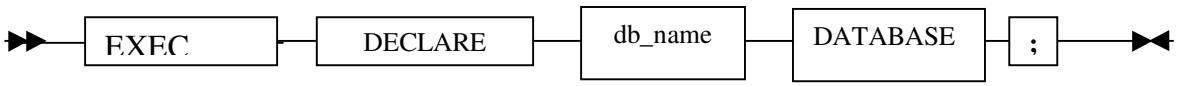
**DECLARE CURSOR**




Ex:

- EXEC SQL DECLARE emp_cursor CURSOR FOR
  SELECT ENAME, EMPNO, JOB, SAL FROM EMP
  WHERE DEPTNO = :deptno
  FOR UPDATE OF SAL;

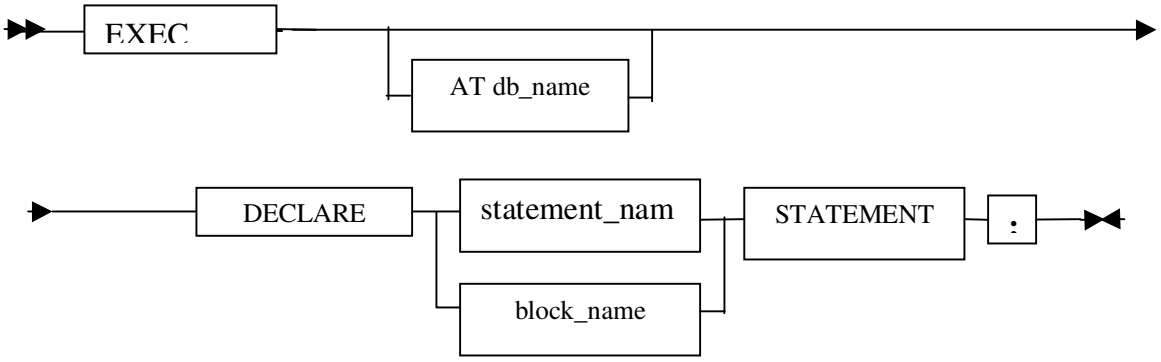
**DECLARE DATABASE**




Ex:

- EXEC SQL DECLARE oracle3 DATABASE;

**DECLARE STATEMENT**



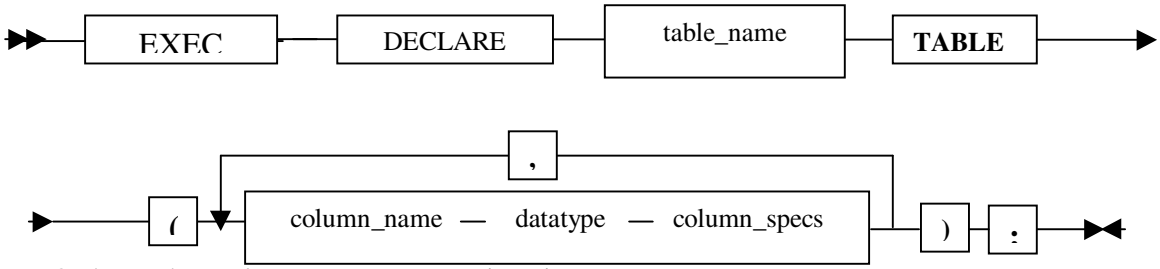
Ex:

EXEC SQL AT remote_db DECLARE sql_stmt STATEMENT;

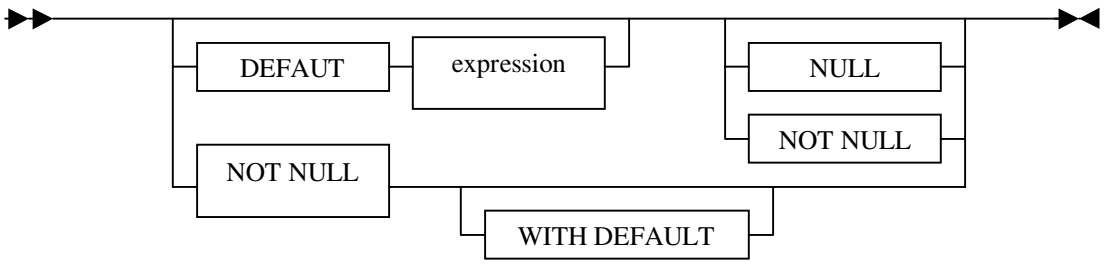
EXEC SQL PREPARE my_statement FROM :my_string;

EXEC SQL EXECUTE :my_statement;

**DECLARE TABLE**




Onde o padrão *column_specs* tem a seguinte sintaxe:




Ex:

- EXEC SQL DECLARE PARTS TABLE

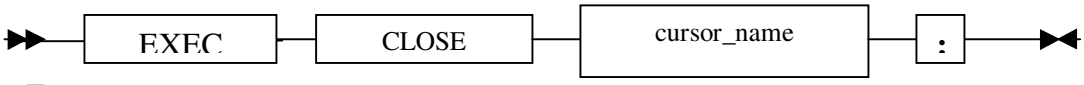
  ( PARTNO NUMBER NOT NULL,

  BIN NUMBER,

  QTY NUMBER);

**COMANDOS EXECUTÁVEIS**

**CLOSE**




Ex:

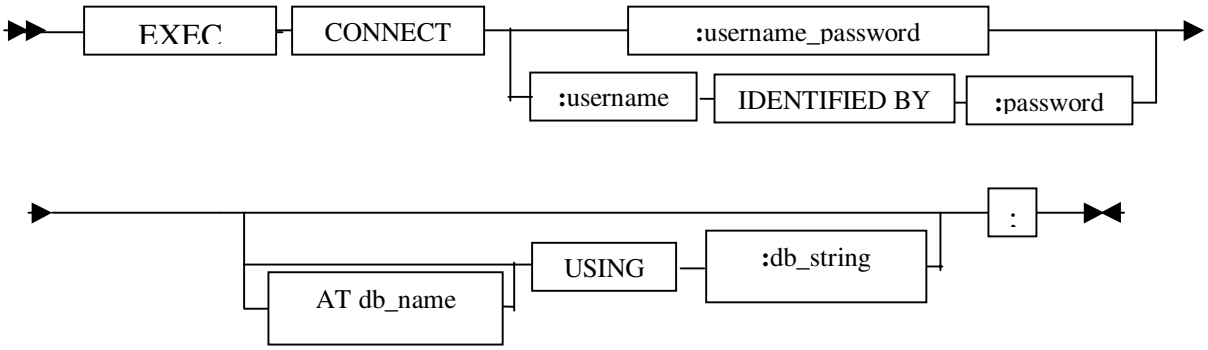
- EXEC SQL CLOSE emp_cursor;

**COMMIT**


Ex:

- EXEC SQL AT sales_db COMMIT RELEASE;
- EXEC SQL COMMIT;

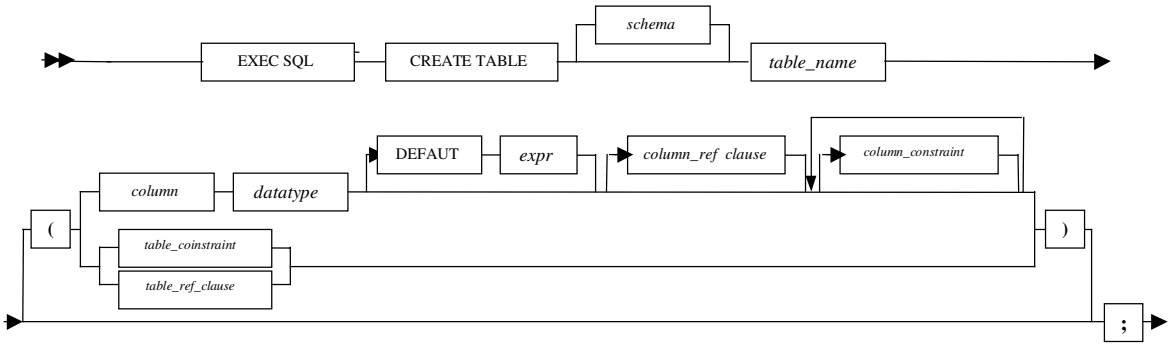
**CONNECT**




Ex:

- EXEC SQL CONNECT :userrid;
- EXEC SQL CONNECT :username IDENTIFIED BY :password;

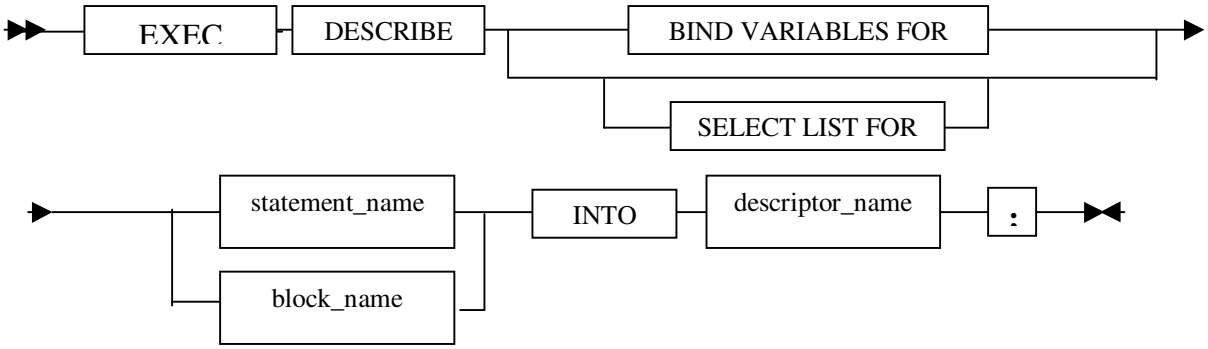
**CREATE TABLE**




Ex:

- EXEC SQL CREATE TABLE *table_name* (empno NUMBER NOT NULL, .., deptno VARCHAR(2) constraint nn_deptno NOT NULL
  CONSTRAINT fk_deptno REFERENCES scott.dep (deptno) );

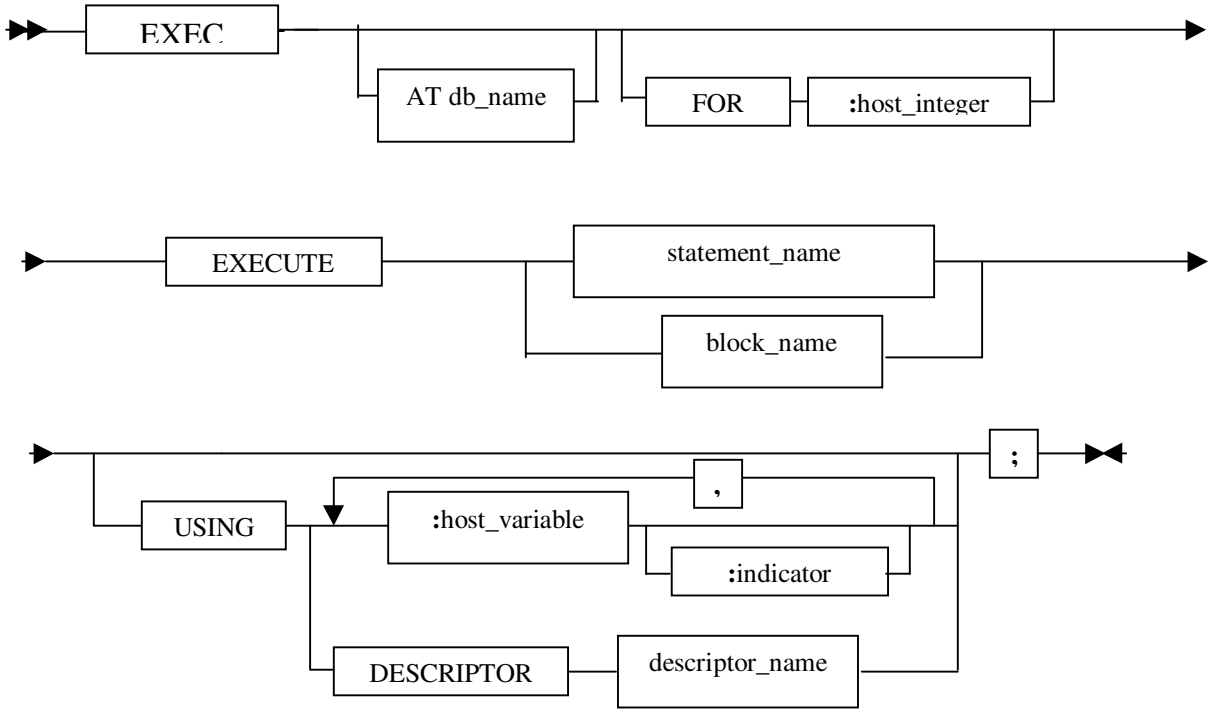
## DESCRIBE




Ex:

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL DECLARE emp_cursor FOR …
EXEC SQL DESCRIBE BIND VARIABLES FOR my_statement
        INTO bind_descriptor;
EXEC SQL OPEN emp_cursor USING …
EXEC SQL DESCRIBE SELECT LIST FOR my_statement
        INTO select_decriptor;
```

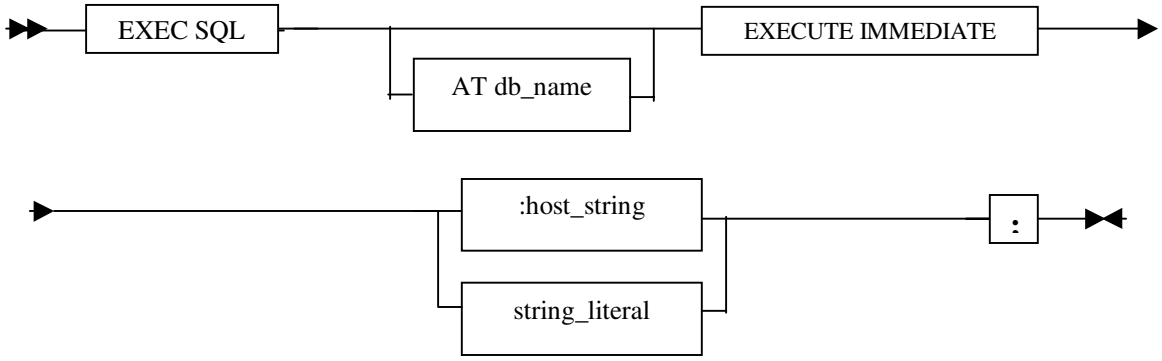
## EXECUTE




Ex:

- EXEC SQL PREPARE my_statement FROM :my_string;
  EXEC SQL EXECUTE my_statement USING :my_var;

**EXECUTE IMMEDIATE**




Ex:

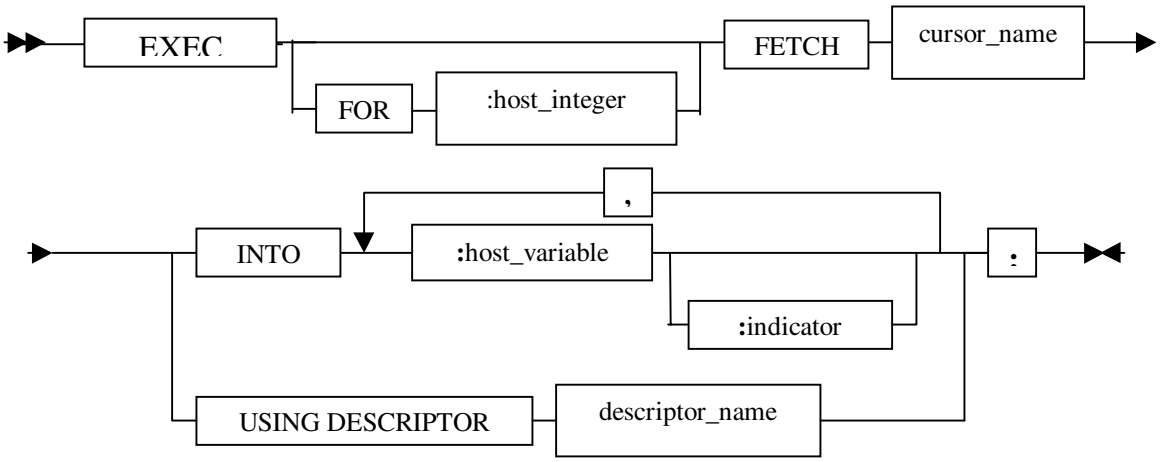
set my_string = “DELETE FROM EMP WHERE EMPNO = 2055”;

EXEC SQL EXECUTE IMMEDIATE :my_string;

ou

EXEC SQL EXECUTE IMMEDIATE “DELETE FROM EMP WHERE EMPNO = 2055”;

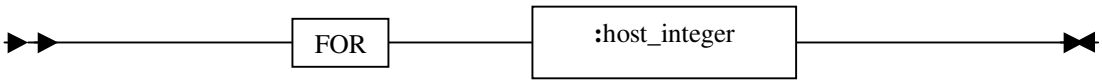
**FETCH**




Ex:

- EXEC SQL FETCH emp_cursor INTO :empno, :ename;

**FOR**



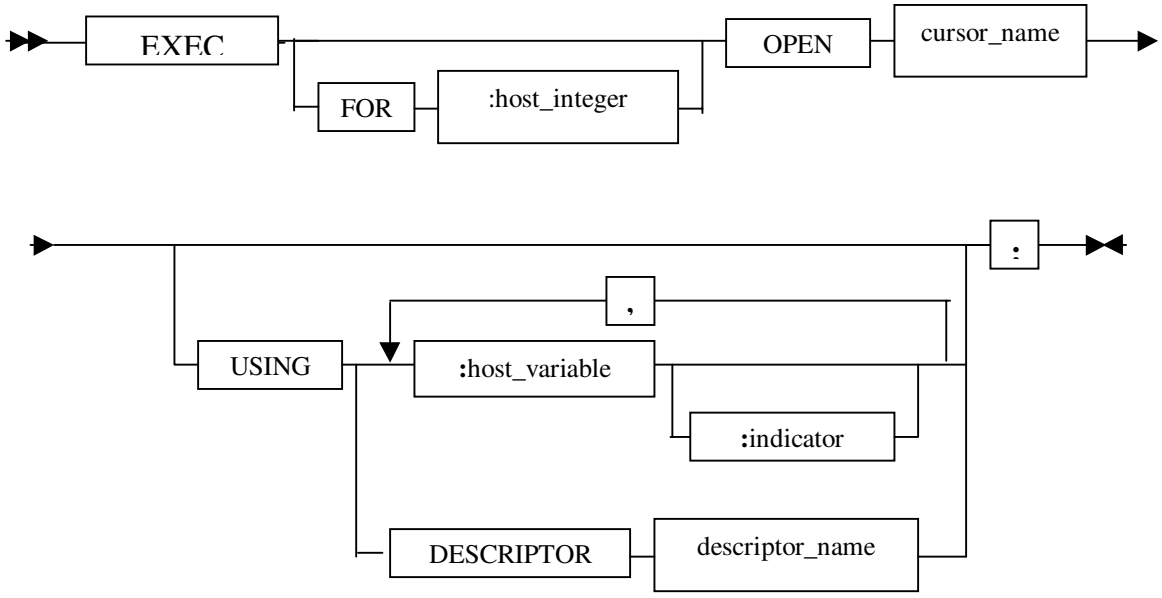

Ex:

- EXEC SQL FOR :limit FETCH emp_cursor INTO …;
- Set limit = 10;

  EXEC SQL FOR :limit DELETE FROM EMP

  WHERE ENAME = :ename_array AND JOB= :job_array;

**OPEN**




O exemplo abaixo ilustra o uso do comando OPEN:

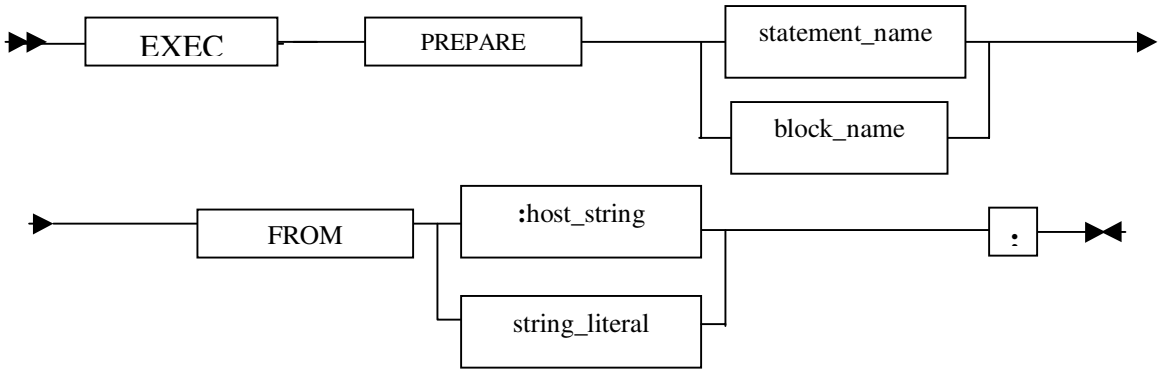
- EXEC SQL DECLARE emp_cursor CURSOR FOR

  SELECT ENAME, EMPNO, JOB, SAL FROM EMP

  WHERE DEPTNO = :deptno;

  EXEC SQL OPEN emp_cursor;

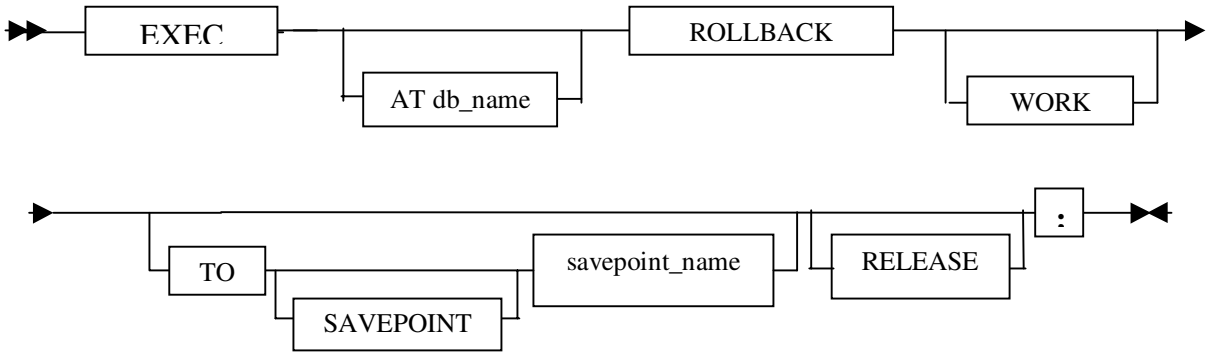
**PREPARE**




Ex:

- EXEC SQL PREPARE my_statement FROM :my_string;
  EXEC SQL EXECUTE my_statement;

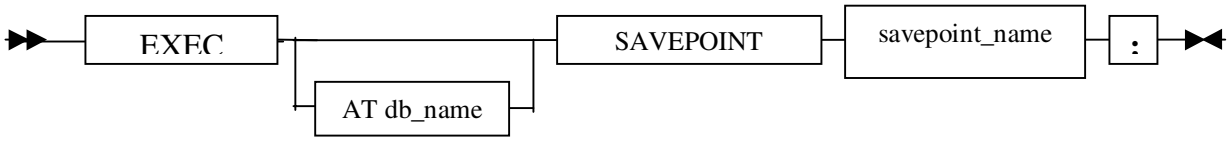
**ROLLBACK**




O exemplo abaixo ilustra o uso do comando ROLLBACK:

- EXEC SQL ROLLBACK TO SAVEPOINT point4;

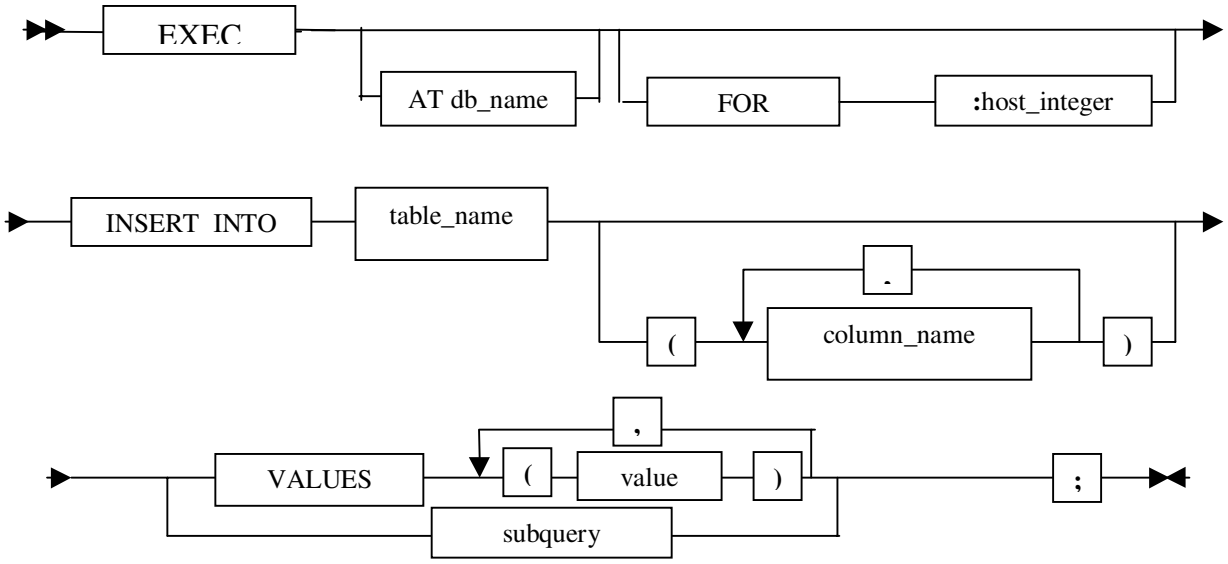
**SAVEPOINT**



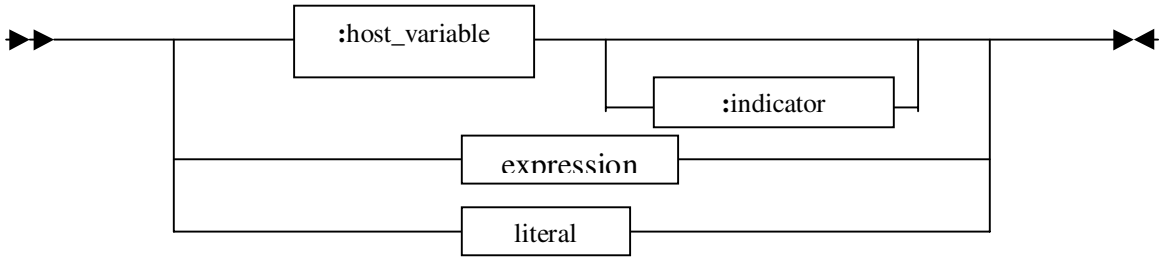

O exemplo abaixo ilustra o uso do comando SAVEPOINT:

- EXEC SQL SAVEPOINT savepoint_name;

**INSERT**



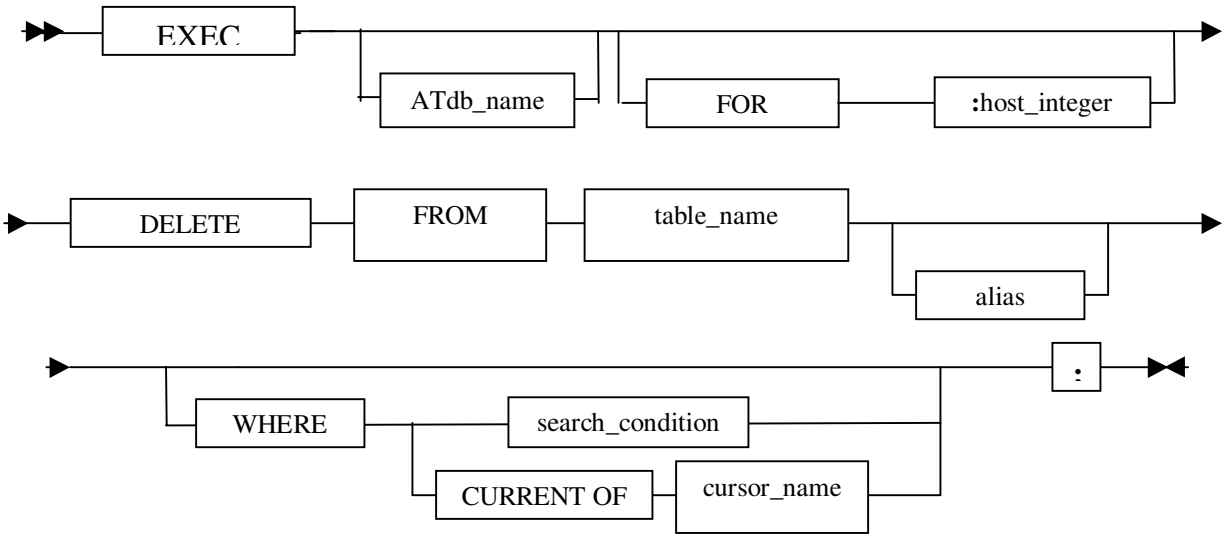

Em caso de utilizar valores (VALUES) através das *variáveis host* a sintaxe seria a seguinte:




Os exemplos abaixo ilustram o uso do INSERT:

- EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
  VALUES (:ename, :empno, :sal);
- EXEC SQL INSERT INTO NEW_EMP (ENAME, EMPNO, SAL)
  SELECT ENAME, EMPNO, SAL FROM EMP
  WHERE DEPTNO= :deptno;

# DELETE




Os exemplos a seguir mostram os usos do comando DELETE:

- EXEC SQL DELETE FROM EMP

  WHERE DEPTNO = :deptno AND JOB = :job;

- EXEC SQL DECLARE emp_cursor CURSOR FOR

  SELECT EMPNO, COMM FROM EMP

  WHERE DEPTNO= :dept_number

  FOR UPDATE OF COMM;

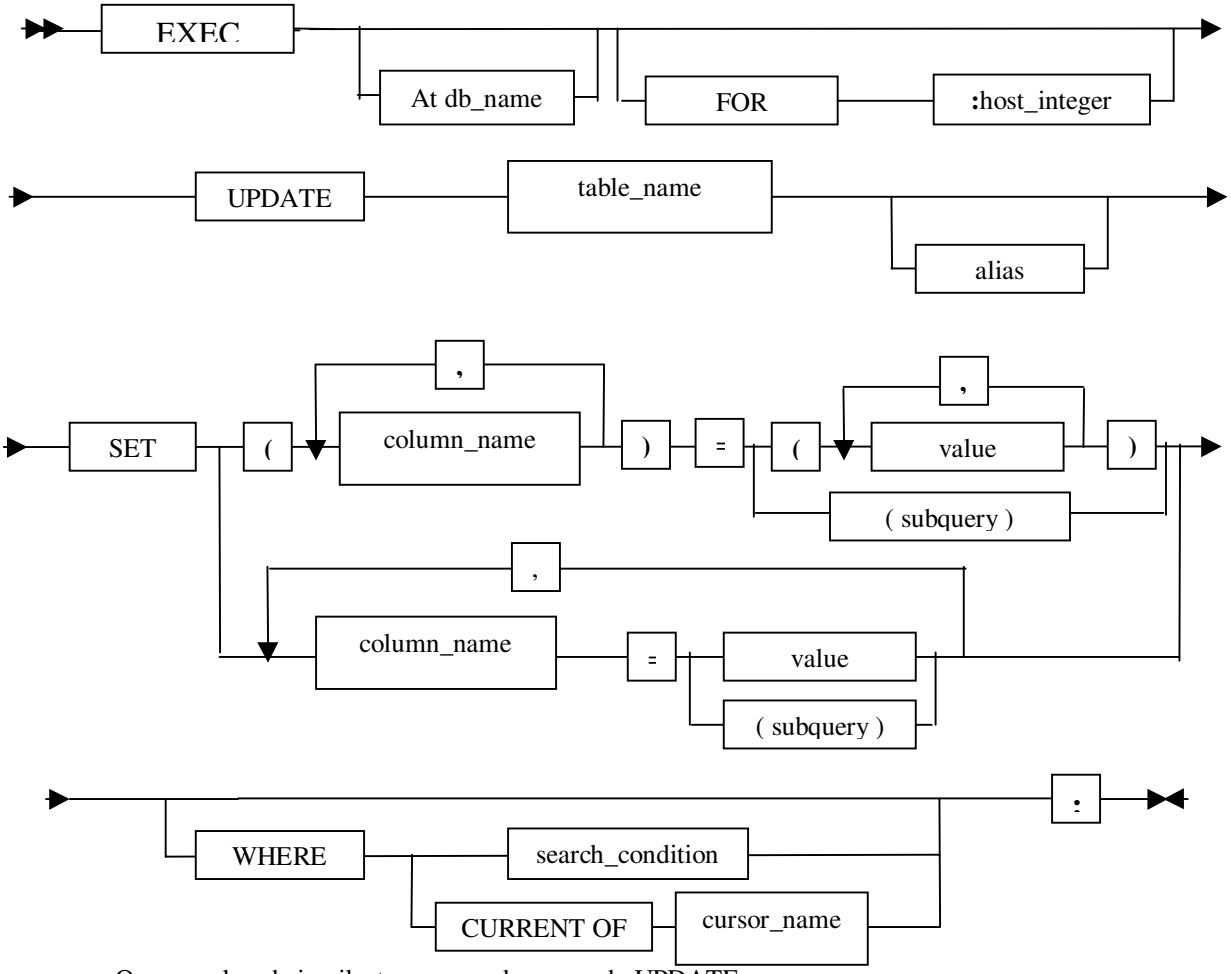
  EXEC SQL OPEN emp_cursor;

  EXEC SQL FETCH emp_cursor INTO :emp_number, :commission;

  EXEC SQL DELETE FROM EMP

  WHERE CURRENT OF emp_cursor;

# UPDATE




Os exemplos abaixo ilustram o uso do comando UPDATE:

- EXEC SQL UPDATE EMP

  SET SAL = :sal, COMM = :comm

  WHERE ENAME = :ename;

- EXEC SQL UPDATE EMP

  SET SAL, COMM = (:sal, :comm.)

  WHERE ENAME = :ename;

- EXEC SQL UPDATE EMP

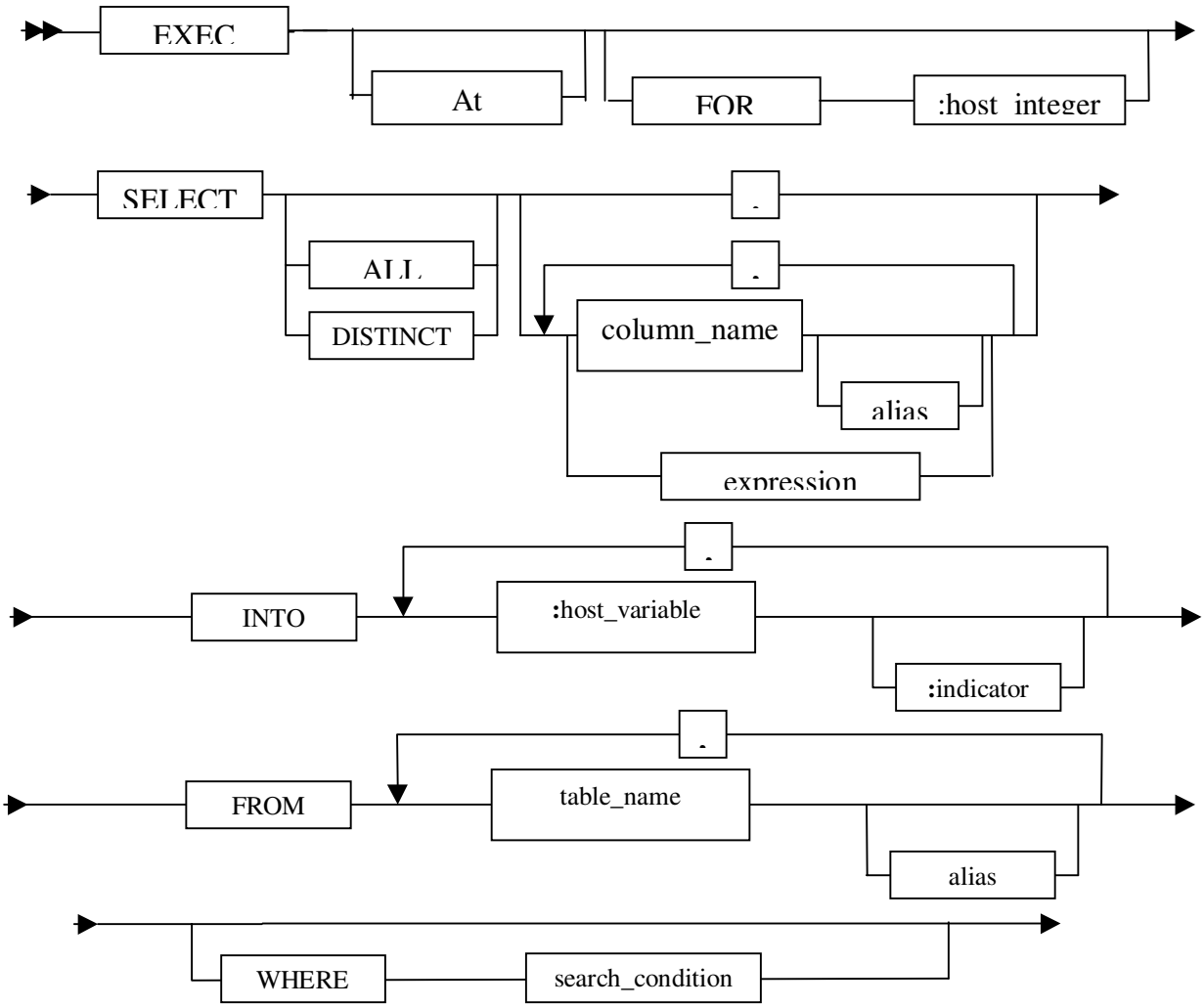
  SET SAL, COMM = (SELECT SAL*1.1, COMM*1.1 FROM EMP);

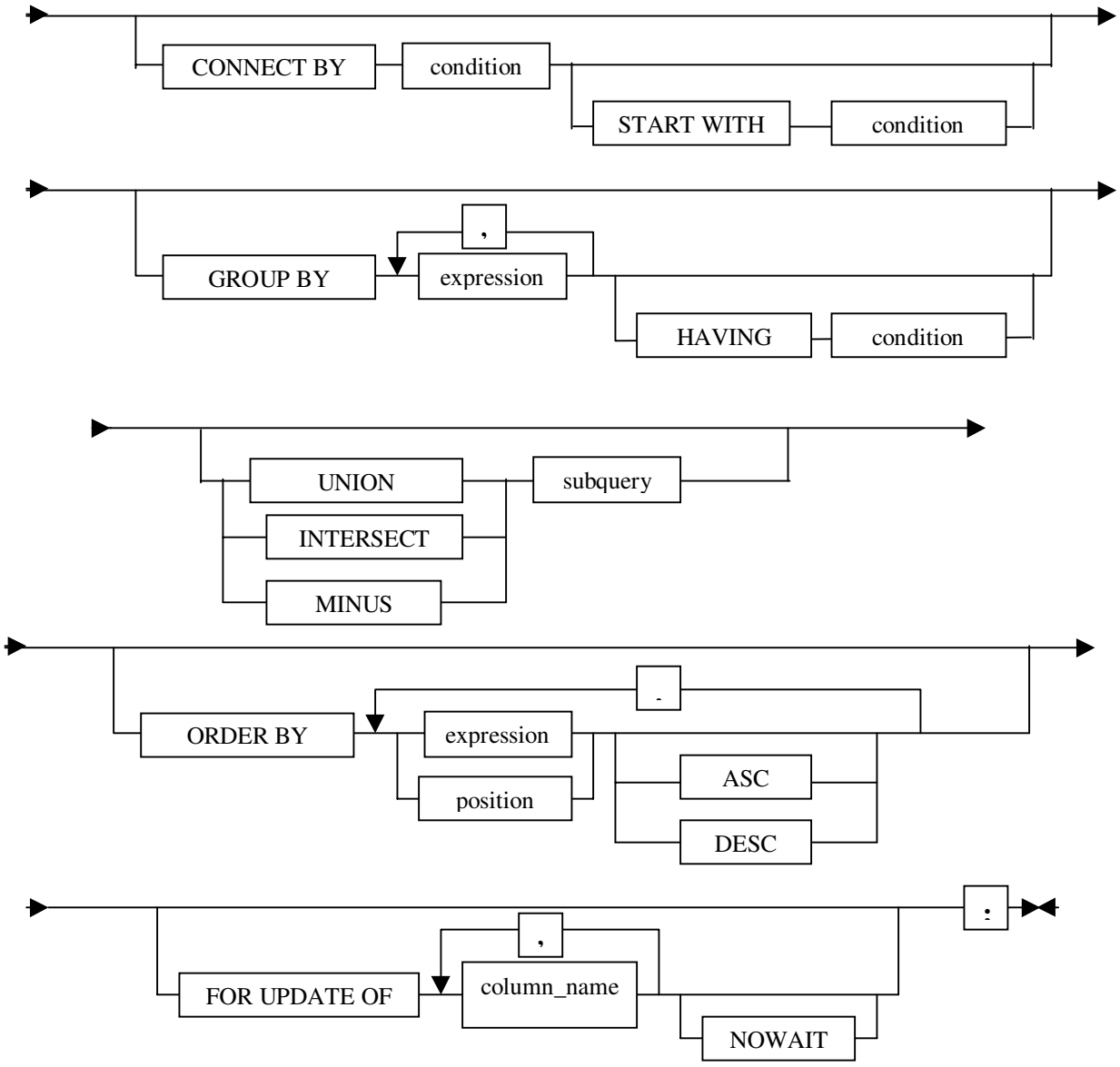
- EXEC SQL UPDATE DEPT

  SET EMPCNT = (SELECT COUNT (*) FROM EMP

  WHERE EMP.DEPTNO = DEPT.DEPTNO);

**SELECT**


EXEC
At
FOR
:host_integer
SELECT
ALL
DISTINCT
column_name
alias
expression
INTO
:host_variable
:indicator
FROM
table_name
alias
WHERE
search_condition




O seguinte exemplo ilustra o uso do SELECT:

```
EXEC SQL ENAME, SAL+100, JOB
        INTO :ename, :sal, :job
        FROM EMP WHERE EMPNO = :empno;
```

# APÊNDICE C

Um Exemplo Completo das etapas de integração *intra-modular* e *inter-modular*.

Os exemplos realizados nesta tese são compostos por quatro programas em C com comandos de **SQL** e com uma base de dados composta por (09) nove tabelas.

A ilustração dos critérios, através de exemplos, resultou em um relatório de 147 páginas. Para mostrar os resultados dos critérios de integração *intra-modular* e *inter-modular*, é apresentado um exemplo completo envolvendo algumas tabelas utilizadas no exemplo da aplicação. Inicialmente apresentou-se partes do programa *Mod$_3$.pc* já instrumentado. A Figura C1 mostra um quadro geral das Unidades existentes neste

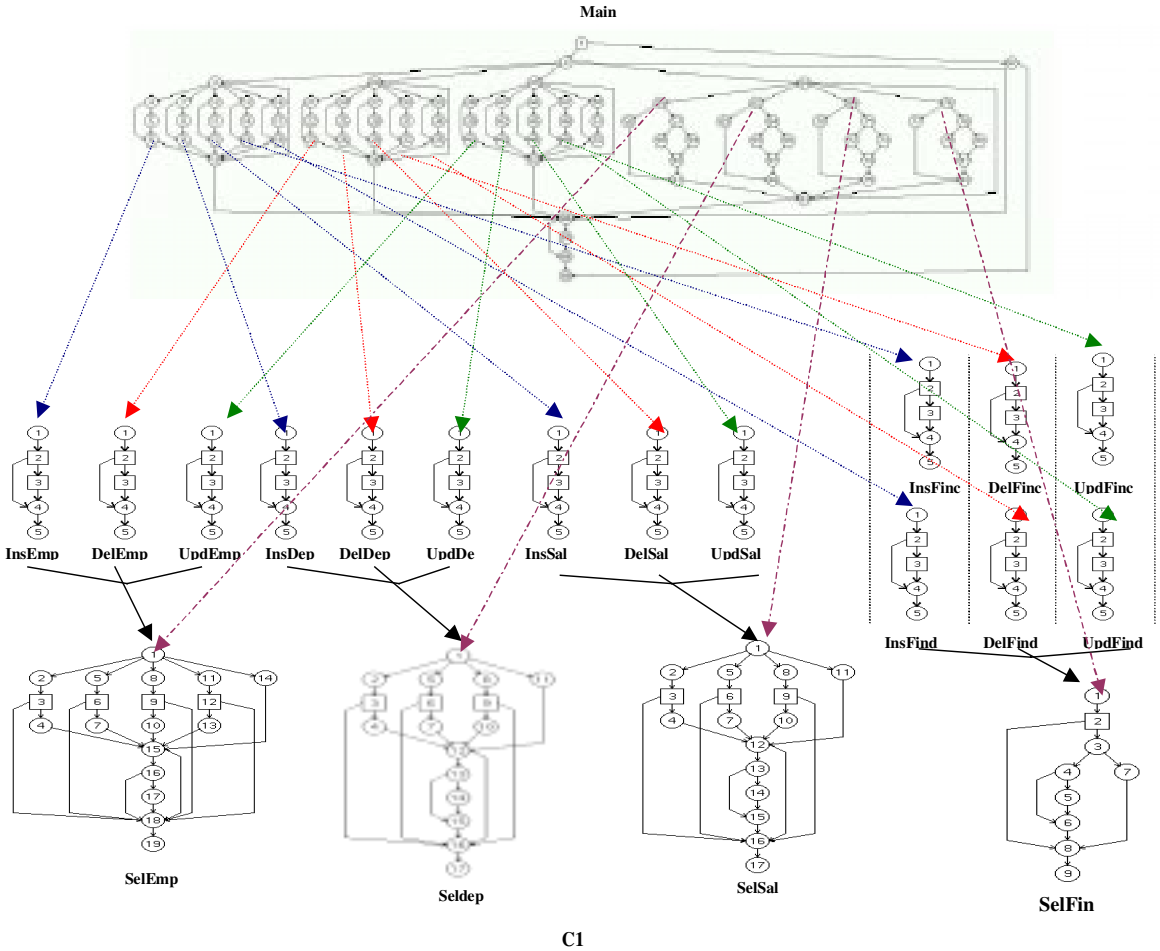



Figura C1: Quadro geral do programa Modulo3.pc utilizado no exemplo

programa.

Para ilustrar o exemplo são apresentadas as principais Unidades que envolvem a tabela DEPARTMENT. Em seguida é mostrado o arquivo que armazena os elementos requeridos pelo critério *todos-t-usos-ciclo1-intra*.

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <curses.h>

/* Declarando estrutura referente a tabela DEPARTMENT  */

typedef struct  {char d_dept_id [11];
                 char d_dept_name[40];
                 char d_manager_id[11]; }RECORD3;

EXEC SQL BEGIN DECLARE SECTION;
int        h1,he1, he2, he5,hso1,hso2,hso6,hd1,hd3,hfd4;
VARCHAR    h2[15],h3[20],h4[35],h5[20],h6[2],he3[20],he4[20],he6[40],
           he7[20],he8[4],he9[10],he10[10],he11[1],he12[11],he17[1],
           he18[1],he19[1],he20[1],hd2[40],hso4[11],hso5[7],hfc1[4],
           hfc2[10],hfc3[50],hfd2[7],hfd3[5];
char       he14[12], he15[12], he16[12],hfd1[5], hso3[12];
float      he13;

EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
  int  mhp1,mhsi1,mhd1,mhe1,mhc1,mhso1;
  char mhfd1[5];
  VARCHAR mhfd2[7], mhfd3[5], mhfc1[4];
EXEC SQL END DECLARE SECTION;
char me_emp_id[11], mso_id[11], md_dept_id[11], mfc_code[7],
     mfd_year[6], mfd_quarter[5], mfd_code[7], mp_prod_id[11],
     msi_id[11], mc_cust_id[11];

. . .

int insdep (line3)
RECORD3 *line3;
/*  1 */  {
             FILE * path = fopen("insdep/path.tes","a");
             FILE * key = fopen("insdep/key.tes","a");
             static int printed_nodes = 0;
        ponta_de_prova(3003);
        ponta_de_prova(1);
/*  1 */     hd1 = atoi (line3->d_dept_id);
                   mhd1=hd1;
/*  1 */     hd3 = atoi (line3->d_manager_id);
/*  1 */     strcpy (hd2.arr, line3->d_dept_name);
/*  1 */     hd2.len = strlen (hd2.arr);

/*  1 */     EXEC SQL WHENEVER SQLERROR GOTO end3;

               ponta_de_prova(2);

               mhd1=hd1; /* INSERT, DELETE, UPDATE */

               itoa(mhd1,md_dept_id); /* int para char */
```

```
              fprintf(key, "DEPARTMENT: %s    \n",md_dept_id);

/*  2 */      EXEC SQL INSERT INTO DEPARTMENT
                           ( dept_id,
                             dept_name,
                             manager_id )

                         VALUES ( :hd1,
                                  :hd2,
                                  :hd3  );


           ponta_de_prova(3);
/*  3 */  EXEC SQL COMMIT;
/*  4 */  end3:
        ponta_de_prova(4);
        ponta_de_prova(5);

        fclose(path);
        fclose(key);
/*  4 */   return (sqlca.sqlcode);
/*  5 */  }

. . .

int deldep (line3)
RECORD3 *line3;
/*  1 */  {
           FILE * path = fopen("deldep/path.tes","a");
           FILE * key = fopen("deldep/key.tes","a");


           static int printed_nodes = 0;
           ponta_de_prova(3008);
           ponta_de_prova(1);
/*  1 */   hd1 = atoi (line3->d_dept_id);
/*  1 */   hd3 = atoi (line3-> d_manager_id);
/*  1 */   EXEC SQL WHENEVER SQLERROR GOTO end8;
           ponta_de_prova(2);
           mhd1=hd1; /* INSERT, DELETE, UPDATE */
           itoa(mhd1,md_dept_id); /* int para char */
           fprintf(key, "DEPARTMENT: %s    \n",md_dept_id);
/*  2 */   EXEC SQL DELETE FROM DEPARTMENT
                 WHERE  dept_id    = :hd1 AND
                        manager_id = :hd3;
           ponta_de_prova(3);
/*  3 */   EXEC SQL COMMIT;
/*  4 */   end8:
              ponta_de_prova(4);
              ponta_de_prova(5);
              fclose(path);
              fclose(key);
/*  4 */    return (sqlca.sqlcode);
/*  5 */  }

. . .

int upddep (line3)
RECORD3 *line3;
/*  1 */  {
            FILE * path = fopen("upddep/path.tes","a");
            FILE * key = fopen("upddep/key.tes","a");
            static int printed_nodes = 0;
            ponta_de_prova(3013);
            ponta_de_prova(1);
/*  1 */    hd1 = atoi (line3-> d_dept_id);
/*  1 */    strcpy (hd2.arr, line3->d_dept_name);
/*  1 */    hd2.len = strlen (hd2.arr);
/*  1 */    hd3 = atoi (line3-> d_manager_id);
/*  1 */    EXEC SQL WHENEVER SQLERROR GOTO end13;
```

```
                ponta_de_prova(2);
                mhd1=hd1; /* INSERT, DELETE, UPDATE */

                itoa(mhd1,md_dept_id); /* int para char */
                fprintf(key, "DEPARTMENT: %s    \n",md_dept_id);
/*  2  */    EXEC SQL UPDATE DEPARTMENT
                SET     dept_name  = :hd2,
                       manager_id  = :hd3
                WHERE     dept_id  = :hd1;
              ponta_de_prova(3);
/*  3  */     EXEC SQL COMMIT;
/*  4  */  end13:
              ponta_de_prova(4);
              ponta_de_prova(5);
              fclose(path);
              fclose(key);
/*  4 */      return (sqlca.sqlcode);
/*  5 */  }

. . .

int seldep  (line3)
RECORD3 *line3;
/*  1 */  {
            FILE * path = fopen("seldep/path.tes","a");
            FILE * key = fopen("seldep/key.tes","a");
            static int printed_nodes = 0;
/*  1 */    char string[50];
/*  1 */    EXEC SQL BEGIN DECLARE SECTION;
                     int     field;
                     int     cod_dep_value,
                             cod_ger_value;
                     VARCHAR nome_value   [40];
            EXEC SQL END DECLARE SECTION;
            ponta_de_prova(3018);
            ponta_de_prova(1);
/*  1 */    EXEC SQL WHENEVER SQLERROR GOTO end18;
/*  1 */    printf(" \n 1-Codigo do DEPARTMENT  2- Codigo_gerente do Dept. 3-
Nome_DEPARTMENT");
/*  1 */     gets (string);
/*  1 */     field = atoi (&string[0]);
/*  1 */     switch(field)
/*  1 */     {
/*  2 */  case 1   :
/*  2 */  {
              ponta_de_prova(2);
/*  2 */    printf ("\n Digite o codigo da DEPARTMENT: ");
/*  2 */    gets (line3->d_dept_id);
/*  2 */    cod_dep_value = atoi (line3->d_dept_id);
            ponta_de_prova(3);
            fprintf(key, "DEPARTMENT (dept_id):  %s  \n ",line3->d_dept_id);

/*  3 */    EXEC SQL SELECT  dept_id,
                     TO_CHAR(dept_id),
                             dept_name,
                     TO_CHAR(manager_id)
                     INTO   :mhd1,
                            :hd1,
                            :hd2,
                            :hd3
                     FROM   DEPARTMENT
                     WHERE  dept_id = :cod_dep_value;

             itoa(mhd1,md_dept_id); /* int para char */
             fprintf(key, "Resultado:  %s    \n",md_dept_id);

             ponta_de_prova(4);

/*  4 */     break;
/*  4 */  }
```

```
/*  5 */  case 2     :
/*  5 */  {
             ponta_de_prova(5);
/*  5 */     printf ("\n Digite Codigo do Gerente do DEPARTMENT:");
/*  5 */     gets (line3->d_manager_id);
/*  5 */     cod_ger_value = atoi (line3->d_manager_id);
             ponta_de_prova(6);


            fprintf(key, "DEPARTMENT (manager_id):  %s   \n ",line3->d_manager_id);

/*  6 */     EXEC SQL SELECT dept_id,
                      TO_CHAR(dept_id),
                              dept_name,
                      TO_CHAR(manager_id)
                      INTO   :mhd1,
                             :hd1,
                             :hd2,
                             :hd3
                      FROM   DEPARTMENT
                      WHERE  manager_id = :cod_ger_value;

             itoa(mhd1,md_dept_id); /* int para char */
             fprintf(key, "Resultado:  %s    \n",md_dept_id);
             ponta_de_prova(7);
/*  7 */     break;
/*  7 */  }
/*  8 */  case 3 :
/*  8 */  {
              ponta_de_prova(8);
/*  8 */     printf ("\n Digite o Nome do DEPARTMENT:");
/*  8 */     gets (line3->d_dept_name);
/*  8 */     strcpy (nome_value.arr, line3->d_dept_name);
/*  8 */     nome_value.len = strlen (nome_value.arr);
             ponta_de_prova(9);

             fprintf(key, "DEPARTMENT (dept_name):  %s  \n ",line3->d_dept_name);


/*  9 */     EXEC SQL SELECT dept_id,
                      TO_CHAR(dept_id),
                              dept_name,
                      TO_CHAR(manager_id)
                      INTO   :mhd1,
                             :hd1,
                             :hd2,
                             :hd3
                      FROM   DEPARTMENT
                      WHERE  dept_name = :nome_value;

             itoa(mhd1,md_dept_id); /* int para char */
             fprintf(key, "Resultado: %s    \n",md_dept_id);

             ponta_de_prova(10);
/* 10 */     break;
/* 10 */  }
/* 11 */      default:
/* 11 */  {
             ponta_de_prova(11);
/* 11 */     printf("\n Codigo errado : ");
/* 11 */     sqlca.sqlcode=-1;
/* 11 */  }
/* 12 */     }
             ponta_de_prova(12);
/* 12 */     if(sqlca.sqlcode == 0)
/* 13 */     {
               ponta_de_prova(13);
/* 13 */      if(sqlca.sqlerrd[2] == 1)
/* 14 */  {
              ponta_de_prova(14);
```

```
/* 14 */      itoa(hd1, line3->d_dept_id);
/* 14 */      hd2.arr [hd2.len] = '\0';
/* 14 */      strcpy (line3->d_dept_name, hd2.arr);
/* 14 */      itoa(hd3, line3->d_manager_id);

/* 14 */  }
             ponta_de_prova(15);
/* 15 */     }

/* 16 */ end18:
             ponta_de_prova(16);
             ponta_de_prova(17);
             fclose(path);
             fclose(key);
/* 16 */     return (sqlca.sqlcode);
/* 17 */  }

. . .

main ()
/*  1 */  {
             FILE * path = fopen("main/path.tes","a");
               static int printed_nodes = 0;
/*  1 */       int    cod_ret, field, transacao, temp;
               char   string[50], strfield[4];

/*  1 */     RECORD  linha;
/*  1 */     RECORD2 linha2;
/*  1 */     RECORD3 linha3;
/*  1 */     RECORD4 linha4;
/*  1 */     RECORD5 linha5;
/*  1 */     RECORD6 linha6;
             ponta_de_prova(3000);
             ponta_de_prova(1);
/*  1 */     cod_ret = dbconn();
/*  1 */     if(cod_ret == 0)
/*  2 */     {
               ponta_de_prova(2);
/*  2 */       printf (" \n digite a transacao: {I, D, U, S}-> ");
/*  2 */       gets(string);
/*  2 */       transacao = string[0];
/*  2 */       transacao=tolower(transacao);
/*  2 */      switch(transacao)
/*  2 */    {
/*  3 */     case 'i' :
/*  3 */     {
               ponta_de_prova(3);
/*  3 */       printf(" \n Inserir 1- Empr 2- Vendas 3-Depart 4-Codigo-fin 5-Data-fin : ");
/*  3 */       gets (strfield);
/*  3 */       field = atoi (&strfield);
/*  3 */       switch(field)
/*  3 */    {
/*  4 */       case 1:
/*  4 */       {
                  ponta_de_prova(4);
/*  4 */          printf (" \n Digite o codigo do Empregado: ");
. . .
/*  4 */          cod_ret = insemp (&linha);
/*  4 */          if( cod_ret == 0)
/*  5 */          {
                     ponta_de_prova(5);
/*  5 */             printf ("Inclusao efetuada EMPLOYEE\n");
/*  5 */          }
                  ponta_de_prova(6);
/*  6 */          break;
/*  6 */       }
/*  7 */       case 2:
/*  7 */       {
                  ponta_de_prova(7);
/*  7 */          printf (" \n Digite o codigo da Venda: ");
```

```
. . .
/*  7 */          cod_ret = inssal (&linha2);
/*  7 */          if( cod_ret == 0)
/*  8 */          {
                     ponta_de_prova(8);
/*  8 */             printf ("Inclusao efetuada SALES_ORDER\n");
/*  8 */          }
                  ponta_de_prova(9);
/*  9 */          break;
/*  9 */       }
/* 10 */       case 3:
/* 10 */       {
                  ponta_de_prova(10);
/* 10 */          printf (" \n Digite o codigo do DEPARTMENT: ");
/* 10 */          gets (linha3.d_dept_id);
/* 10 */          printf (" \n Digite o nome do DEPARTMENT: ");
/* 10 */          gets (linha3.d_dept_name);
/* 10 */          printf (" \n Digite o codigo empregado do DEPARTMENT: ");
/* 10 */          gets (linha3.d_manager_id);
/* 10 */          cod_ret = insdep (&linha3);
/* 10 */          if( cod_ret == 0)
/* 11 */          {
                     ponta_de_prova(11);
/* 11 */             printf ("Inclusao efetuada DEPARTMENT\n");
/* 11 */          }
                  ponta_de_prova(12);
/* 12 */          break;
/* 12 */       }
/* 13 */       case 4:
/* 13 */       {
                  ponta_de_prova(13);
/* 13 */          printf (" \n Digite o codigo do financ. de Venda: ");
. . .
/* 13 */          cod_ret = insfinc(&linha4);
/* 13 */          if( cod_ret == 0)
/* 14 */          {
                     ponta_de_prova(14);
/* 14 */             printf ("Inclusao efetuada FIN_CODE\n");
/* 14 */          }
                  ponta_de_prova(15);
/* 15 */          break;
/* 15 */       }
/* 16 */       case 5:
/* 16 */       {
                  ponta_de_prova(16);
/* 16 */          printf (" \n Digite o Ano da data de Financ: ");
/* 16 */          gets (linha5.fd_year);
. . .
/* 16 */          gets (linha5.fd_amount);
/* 16 */          cod_ret = insfind (&linha5);
/* 16 */          if( cod_ret == 0)
/* 17 */          {
                     ponta_de_prova(17);
/* 17 */             printf ("Inclusao efetuada FIN_DATE\n");
/* 17 */          }
                   ponta_de_prova(18);
/* 18 */       }
/* 19 */    }
            ponta_de_prova(19);
/* 19 */    break;
/* 19 */     }
/* 20 */     case 'u' :
/* 20 */     {
            ponta_de_prova(20);
/* 20 */    printf(" \n Atualizar 1- Empregado  2- Vendas 3-DEPARTMENT 4-Codigo-fin 5-Data-
fin: ");
/* 20 */    gets (strfield);
/* 20 */    field = atoi (&strfield);
/* 20 */    switch(field)
/* 20 */    {
```

```
/* 21 */       case 1:
/* 21 */       {
                  ponta_de_prova(21);
/* 21 */          printf (" \n Digite o codigo do Empregado: ");
/* 21 */          gets (linha.e_emp_id);
. . .
/* 21 */          cod_ret = updemp (&linha);
/* 21 */          if( cod_ret == 0)
/* 22 */          {
                     ponta_de_prova(22);
/* 22 */             printf ("Atualizacao  efetuada EMPLOYEE\n");
/* 22 */          }
                  ponta_de_prova(23);
/* 23 */          break;
/* 23 */       }
/* 24 */       case 2:
/* 24 */       {
                  ponta_de_prova(24);
/* 24 */          printf (" \n Digite o codigo da Venda: ");
/* 24 */          gets (linha2.s_id);
. . .
/* 24 */          cod_ret = updsal (&linha2);
/* 24 */          if( cod_ret == 0)
/* 25 */          {
                     ponta_de_prova(25);
/* 25 */             printf ("Atualizacao  efetuada SALES_ORDER\n");
/* 25 */          }
                  ponta_de_prova(26);
/* 26 */          break;
/* 26 */       }
/* 27 */       case 3:
/* 27 */       {
                  ponta_de_prova(27);
/* 27 */          printf (" \n Digite o codigo do DEPARTMENT: ");
/* 27 */          gets (linha3.d_dept_id);
/* 27 */          printf (" \n Digite o nome do DEPARTMENT: ");
/* 27 */          gets (linha3.d_dept_name);
/* 27 */          printf (" \n Digite o codigo empregado do DEPARTMENT: ");
/* 27 */          gets (linha3.d_manager_id);
/* 27 */          cod_ret = upddep (&linha3);
/* 27 */          if( cod_ret == 0)
/* 28 */          {
                     ponta_de_prova(28);
/* 28 */             printf ("Atualizacao efetuada DEPARTMENT\n");
/* 28 */          }
                  ponta_de_prova(29);
/* 29 */          break;
/* 29 */       }
/* 30 */       case 4:
/* 30 */       {
                  ponta_de_prova(30);
/* 30 */          printf (" \n Digite o codigo do financ. de Venda: ");
/* 30 */          gets (linha4.fc_code);
. . .
/* 30 */          cod_ret = updfinc(&linha4);
/* 30 */          if( cod_ret == 0)
/* 31 */          {
                     ponta_de_prova(31);
/* 31 */             printf ("Atualizacao efetuada FIN_CODE\n");
/* 31 */          }
                  ponta_de_prova(32);
/* 32 */          break;
/* 32 */       }
/* 33 */       case 5:
/* 33 */       {
                  ponta_de_prova(33);
/* 33 */          printf (" \n Digite o Ano da data de Financ: ");
/* 33 */          gets (linha5.fd_year);
. . .
/* 33 */          cod_ret = updfind (&linha5);
```

```
/* 33 */          if( cod_ret == 0)
/* 34 */          {
                     ponta_de_prova(34);
/* 34 */             printf ("Atualizacao efetuada FIN_DATE\n");
/* 34 */          }
                 ponta_de_prova(35);
/* 35 */       }
/* 36 */    }
            ponta_de_prova(36);
/* 36 */    break;
/* 36 */     }
/* 37 */     case 'd' :
/* 37 */     {
            ponta_de_prova(37);
/* 37 */    printf(" \n Remover 1- Empr 2- Vendas 3-Depart 4-Codigo-fin 5-Data-fin :");
/* 37 */    gets(strfield);
/* 37 */    field = atoi (&strfield);
/* 37 */    switch(field)
/* 37 */    {
/* 38 */       case 1:
/* 38 */       {
                  ponta_de_prova(38);
/* 38 */          printf (" \n Digite o codigo do Empregado a ser Removido: ");
/* 38 */          gets (linha.e_emp_id);
/* 38 */          cod_ret = delemp (&linha);
/* 38 */          if((cod_ret == 0) && (sqlca.sqlerrd[2]) > 0 )
/* 39 */          {
                     ponta_de_prova(39);
/* 39 */             printf ("Remocao efetuada EMPLOYEE\n");
/* 39 */          }
                  ponta_de_prova(40);
/* 40 */          break;
/* 40 */       }
/* 41 */       case 2:
/* 41 */       {
                  ponta_de_prova(41);
/* 41 */          printf (" \n Digite o codigo da Venda a ser removida: ");
/* 41 */          gets (linha2.s_id);
/* 41 */          cod_ret = delsal (&linha2);
         printf (" --> "); printf("\n Linhas deletada  =------==> %d", sqlca.sqlerrd[3]);
/* 41 */          if((cod_ret == 0) && (sqlca.sqlerrd[2]) > 0 )
/* 42 */          {
                     ponta_de_prova(42);
/* 42 */             printf ("Remocao efetuada SALES_ORDER\n");
/* 42 */          }
                  ponta_de_prova(43);
/* 43 */          break;
/* 43 */       }
/* 44 */       case 3:
/* 44 */       {
                  ponta_de_prova(44);
/* 44 */          printf (" \n Digite o codigo do DEPARTMENT a ser removido: ");
/* 44 */          gets (linha3.d_dept_id);
/* 44 */          printf (" \n Digite o codigo empregado do DEPARTMENT: ");
/* 44 */          gets (linha3.d_manager_id);
/* 44 */          cod_ret = deldep (&linha3);
/* 44 */          if((cod_ret == 0) && (sqlca.sqlerrd[2]) > 0 )
/* 45 */          {
                     ponta_de_prova(45);
/* 45 */             printf ("Remocao  efetuada DEPARTMENT\n");
/* 45 */          }
                  ponta_de_prova(46);
/* 46 */          break;
/* 46 */       }
/* 47 */       case 4:
/* 47 */       {
                  ponta_de_prova(47);
/* 47 */          printf (" \n Digite o codigo do financ. de Venda: ");
/* 47 */          gets (linha4.fc_code);
/* 47 */          cod_ret = delfinc(&linha4);
```

```
             printf (" --> "); printf("\n Linhas deletada  =-----==> %d", sqlca.sqlerrd[2]);
/* 47 */          if( (cod_ret == 0) && (sqlca.sqlerrd[2]) > 0)
/* 48 */          {
                     ponta_de_prova(48);
/* 48 */             printf ("Remocao efetuada FIN_CODE\n");
/* 48 */          }
                  ponta_de_prova(49);
/* 49 */          break;
/* 49 */       }
/* 50 */       case 5:
/* 50 */       {
                  ponta_de_prova(50);
/* 50 */          printf (" \n Digite o Ano da data de Financ: ");
/* 50 */          gets (linha5.fd_year);
/* 50 */          printf (" \n Digite o Trimestre (Q1/Q2/Q3/Q4) do Financ: ");
/* 50 */          gets (linha5.fd_quarter);
/* 16 */          printf (" \n Digite o codigo do financ. de Venda: ");
/* 16 */          gets (linha5.fd_code);
/* 50 */          cod_ret = delfind (&linha5);
/* 50 */          if((cod_ret == 0) && (sqlca.sqlerrd[2]) > 0 )
/* 51 */          {
                     ponta_de_prova(51);
/* 51 */             printf ("Remocao  efetuada FIN_DATE\n");
/* 51 */          }
                  ponta_de_prova(52);
/* 52 */       }
/* 53 */    }
            ponta_de_prova(53);
/* 53 */    break;
/* 53 */     }
/* 54 */     case 's' :
/* 54 */     {
            ponta_de_prova(54);
/* 54 */    printf(" \n Deseja Selecionar 1:Tabela de Empregados 2:Tabela de ordem_vendas
3:Tabela de DEPARTMENT ou 4:Tabelas de codigo e data de financiamento :");
/* 54 */    gets(strfield);
/* 54 */    field = atoi (&strfield);
/* 54 */    switch(field)
/* 54 */    {
/* 55 */       case 1 :
/* 55 */       {
                  ponta_de_prova(55);
/* 55 */          cod_ret = selemp (&linha);
/* 55 */          if(cod_ret == 0)
/* 56 */          {
                      ponta_de_prova(56);
. . .
/* 56 */          }
/* 57 */          else
/* 57 */          {
                     ponta_de_prova(57);
/* 57 */             if(cod_ret == 1403)
/* 58 */             {
                        ponta_de_prova(58);
/* 58 */                printf ("nao encontrado (employee)\n");
/* 58 */             }
/* 59 */             else
/* 59 */             {
                        ponta_de_prova(59);
/* 59 */                printf ("erro de selecao (employee)\n");
/* 59 */                printf ("sqlcode = %d\n", sqlca.sqlcode);
/* 59 */             }
                    ponta_de_prova(60);
/* 60 */          }
                  ponta_de_prova(61);
/* 61 */          break;
/* 61 */       }
/* 62 */       case 2:
/* 62 */       {
                  ponta_de_prova(62);
```

```
/* 62 */          cod_ret = selsal (&linha2);
/* 62 */          if(cod_ret == 0)
/* 63 */          {
                     ponta_de_prova(63);
. . .
/* 63 */          }
/* 64 */          else
/* 64 */          {
                     ponta_de_prova(64);
/* 64 */             if(cod_ret == 1403)
/* 65 */             {
                        ponta_de_prova(65);
/* 65 */                printf ("nao encontrado (vendas)\n");
/* 65 */             }
/* 66 */             else
/* 66 */             {
                        ponta_de_prova(66);
/* 66 */                printf ("erro de selecao (vendas)\n");
/* 66 */                printf ("sqlcode = %d\n", sqlca.sqlcode);
/* 66 */             }
                    ponta_de_prova(67);
/* 67 */          }
                  ponta_de_prova(68);
/* 68 */          break;
/* 68 */       }
/* 69 */       case 3:
/* 69 */       {
                  ponta_de_prova(69);
/* 69 */          cod_ret = seldep (&linha3);
/* 69 */          if(cod_ret == 0)
/* 70 */          {
                     ponta_de_prova(70);
/* 70 */             printf ("\n Codigo_dept: %s"  , linha3.d_dept_id);
/* 70 */             printf ("\n Nome_dept: %s   Ident: %s ", linha3.d_dept_name,
linha3.d_manager_id);
/* 70 */          }
/* 71 */          else
/* 71 */          {
                     ponta_de_prova(71);
/* 71 */             if(cod_ret == 1403)
/* 72 */             {
                        ponta_de_prova(72);
/* 72 */                printf ("nao encontrado (dept)\n");
/* 72 */             }
/* 73 */             else
/* 73 */             {
                        ponta_de_prova(73);
/* 73 */                printf ("erro de selecao (dept)\n");
/* 73 */                printf ("sqlcode = %d\n", sqlca.sqlcode);
/* 73 */             }
                    ponta_de_prova(74);
/* 74 */          }
                  ponta_de_prova(75);
/* 75 */          break;
/* 75 */       }
/* 76 */       case 4:
/* 76 */       {
                  ponta_de_prova(76);
/* 76 */          cod_ret = selfin (&linha4,&linha5);
/* 76 */          if(cod_ret == 0)
/* 77 */          {
                     ponta_de_prova(77);
/* 77 */             printf ("\n Codigo_fin: %s"  , linha4.fc_code);
. . .
/* 77 */          }
/* 78 */          else
/* 78 */          {
                     ponta_de_prova(78);
/* 78 */             if(cod_ret == 1403)
/* 79 */             {
```

```
                        ponta_de_prova(79);
/* 79 */                printf ("nao encontrado (fin)\n");
/* 79 */              }
/* 80 */              else
/* 80 */              {
                        ponta_de_prova(80);
/* 80 */                printf ("erro de selecao (fin)\n");
/* 80 */                printf ("sqlcode = %d\n", sqlca.sqlcode);
/* 80 */              }
                    ponta_de_prova(81);
/* 81 */           }
                   ponta_de_prova(82);
/* 82 */           break;
/* 82 */        }
/* 83 */     }
             ponta_de_prova(83);
/* 83 */      }
/* 84 */   }
           printf(" \n Codigo de erro %d", cod_ret);
           ponta_de_prova(84);
/* 84 */   cod_ret = dbdisconn(); printf("\n code: = %d",cod_ret);
/* 84 */   if(cod_ret<0)
/* 85 */   {
              ponta_de_prova(85);
/* 85 */      printf ("erro durante a desconexao\n");
/* 85 */   }
              ponta_de_prova(86);
/* 86 */      }
/* 87 */      else
/* 87 */      {
                ponta_de_prova(87);
/* 87 */        printf ("erro de conexao\n");
/* 87 */      }
          ponta_de_prova(88);
/* 88 */   }
```

A seguir são apresentados os elementos requeridos e as seqüências de execução dos casos de testes que exercitam os elementos requeridos para este critério e para a tabela DEPARTMENT.

Associações requeridas pela tabela DEPARTMENT

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS
TODOS T-USOS-INT-INTRA

Associacoes requeridas pelo Grafo(INSDEP -
> 3003)

01) <3003,(2,3),3003,(2,3),{DEPARTMENT}>
n.e
02) <3003,(2,3),3003,(2,4),{DEPARTMENT}> *
03) <3003,(2,3),3008,(2,3),{DEPARTMENT}> *
04) <3003,(2,3),3008,(2,4),{DEPARTMENT}>
n.e
05) <3003,(2,3),3013,(2,3),{DEPARTMENT}> *
06) <3003,(2,3),3013,(2,4),{DEPARTMENT}> *
07) <3003,(2,3),3018,(3,4),{DEPARTMENT}> *
08) <3003,(2,3),3018,(3,16),{DEPARTMENT}>
*
09) <3003,(2,3),3018,(6,7),{DEPARTMENT}> *
10) <3003,(2,3),3018,(6,16),{DEPARTMENT}>
*
11) <3003,(2,3),3018,(9,10),{DEPARTMENT}>
*
12) <3003,(2,3),3018,(9,16),{DEPARTMENT}>

Associacoes requeridas pelo Grafo(DELDEP -
> 3008)

13) <3008,(2,3),3003,(2,3),{DEPARTMENT}> *
14) <3008,(2,3),3003,(2,4),{DEPARTMENT}>
n.e
15) <3008,(2,3),3008,(2,3),{DEPARTMENT}> *
16) <3008,(2,3),3008,(2,4),{DEPARTMENT}>
n.e (mesmo s/chave deleta 0
17) <3008,(2,3),3013,(2,3),{DEPARTMENT}> *
18) <3008,(2,3),3013,(2,4),{DEPARTMENT}> *
19) <3008,(2,3),3018,(3,4),{DEPARTMENT}> *
20) <3008,(2,3),3018,(3,16),{DEPARTMENT}>
n.e
21) <3008,(2,3),3018,(6,7),{DEPARTMENT}> *
22) <3008,(2,3),3018,(6,16),{DEPARTMENT}>
*
23) <3008,(2,3),3018,(9,10),{DEPARTMENT}>
*
24) <3008,(2,3),3018,(9,16),{DEPARTMENT}>
*

Associacoes requeridas pelo Grafo(UPDDEP -
> 3013)
```

```
25) <3013,(2,3),3003,(2,3),{DEPARTMENT}> *
26) <3013,(2,3),3003,(2,4),{DEPARTMENT}> *
27) <3013,(2,3),3008,(2,3),{DEPARTMENT}> *
28) <3013,(2,3),3008,(2,4),{DEPARTMENT}>
n.e
29) <3013,(2,3),3013,(2,3),{DEPARTMENT}> *
30) <3013,(2,3),3013,(2,4),{DEPARTMENT}>
n.e (mesma tupla)
31) <3013,(2,3),3018,(3,4),{DEPARTMENT}> *
32) <3013,(2,3),3018,(3,16),{DEPARTMENT}>
n.e (mesma tupla)
33) <3013,(2,3),3018,(6,7),{DEPARTMENT}> *
34) <3013,(2,3),3018,(6,16),{DEPARTMENT}>
*
35) <3013,(2,3),3018,(9,10),{DEPARTMENT}>
*
36) <3013,(2,3),3018,(9,16),{DEPARTMENT}>
*
```

---

```
Segunda avaliacao: nao permite mesmo
DEPARTMENT (dept e' a chave
primaria).: Tabela DEPARTMENT
==========================================
=========================
::::::::::::::
fint01
::::::::::::::
i
3
600
Geral Manager
1643

 #1

 3003   1   2   3   4   5

DEPARTMENT: 600
=====================(03)============
::::::::::::::
fint02
::::::::::::::
d
3
600
1643

 #2

 3008   1   2   3   4   5

DEPARTMENT: 600
======================(13)==========
::::::::::::::
fint03
::::::::::::::
i
3
600
Geral Manager
1643

 #3

 3003   1   2   3   4   5

DEPARTMENT: 600
======================(05)==========
::::::::::::::
```

Dado de *teste*

*Número do caso de teste*

*Caminho percorrido*

*Tupla usada*

A seqüência fint01 e fint02 satisfaz o elemento requerido número 03

```
fint04
::::::::::::::
u
3
600
Kids & Pats
249

 #4

 3013   1   2   3   4   5

DEPARTMENT: 600
======================(26)==========
::::::::::::::
fint05
::::::::::::::
i
3
600
Geral Manager
1643

 #5

 3003   1   2   4   5

DEPARTMENT: 600
======================n.e==========
::::::::::::::
fint06
::::::::::::::
s
3
1
600

 #6

 3018   1   2   3   4  12  13  14  15  16
17

DEPARTMENT (dept_id):  600
 Resultado:  600
======================n.e==========
::::::::::::::
fint07
::::::::::::::
i
3
700
Drinks
249

 #7

 3003   1   2   3   4   5

DEPARTMENT: 700
======================(02)==========
::::::::::::::
fint08
::::::::::::::
i
3
700
Drinks
249
```

```
 #8

  3003   1   2   4   5

DEPARTMENT: 700
======================n.e===========
:::::::::::::::
fint09
:::::::::::::::
i
3
800
TV & Sound
1576


 #9

 3003   1   2   3   4   5

DEPARTMENT: 800
======================n.e===========
:::::::::::::::
fint10
:::::::::::::::
d
3
600
1643

 #10

 3008   1   2   3   4   5

DEPARTMENT: 600
======================(14)==========
:::::::::::::::
fint11
:::::::::::::::
i
3
600
Geral Manager
102

 #11

 3003   1   2   4   5

DEPARTMENT: 600
======================n.e===========
:::::::::::::::
fint12
:::::::::::::::
s
3
2
102

 #12

 3018   1   5   6   7  12  16  17

DEPARTMENT (manager_id):  102
 Resultado:  0
======================n.e===========
:::::::::::::::
fint13
:::::::::::::::
i
3
650
Controler
247

 #13

 3003   1   2   3   4   5

DEPARTMENT: 650
======================(11)==========
:::::::::::::::
fint14
:::::::::::::::
s
3
3
Controler

 #14

 3018   1   8   9  10  12  13  14  15  16
17

DEPARTMENT (dept_name):  Controler
 Resultado: 650
======================n.e===========
:::::::::::::::
fint15
:::::::::::::::
i
3
660
Controler
247

 #15

 3003   1   2   3   4   5

DEPARTMENT: 660
======================(12)==========
:::::::::::::::
fint16
:::::::::::::::
s
3
3
Controler

 #16

 3018   1   8   9  16  17

DEPARTMENT (dept_name):  Controler

======================n.e===========
:::::::::::::::
fint17
:::::::::::::::
i
3
880
Controler
703

 #17

 3003   1   2   3   4   5
```

```
DEPARTMENT: 880
======================(06)===========
::::::::::::::
fint18
::::::::::::::
u
3
880
Ships & Ships
102

 #18

 3013   1   2   4   5

DEPARTMENT: 880
======================n.e===========
::::::::::::::
fint19
::::::::::::::
i
3
690
Machines
247

 #19

 3003   1   2   3   4   5

DEPARTMENT: 690
======================(10)==========
::::::::::::::
fint20
::::::::::::::
s
3
2
247

 #20

 3018   1   5   6  16  17

DEPARTMENT (manager_id):  247

======================n.e===========
::::::::::::::
fint21
::::::::::::::
i
3
750
Controler
1576

 #21

 3003   1   2   3   4   5

DEPARTMENT: 750
======================n.e===========
::::::::::::::
fint22
::::::::::::::
s
3
3
Controler

 #22

 3018   1   8   9  16  17

DEPARTMENT (dept_name):  Controler

======================n.e===========
::::::::::::::
fint23
::::::::::::::
u
3
750
Birds
1576

 #23

 3013   1   2   3   4   5

DEPARTMENT: 750
======================n.e===========
::::::::::::::
fint24
::::::::::::::
i
3
750
Birds
1576

 #24

 3003   1   2   4   5

DEPARTMENT: 750
======================n.e===========
::::::::::::::
fint25
::::::::::::::
u
3
780
Reg & Dig
102

 #25

 3013   1   2   3   4   5

DEPARTMENT: 780
======================(27)==========
::::::::::::::
fint26
::::::::::::::
d
3
780
102

 #26

 3008   1   2   3   4   5

DEPARTMENT: 780
======================(15)==========
::::::::::::::
fint27
::::::::::::::
d
```

```
3
780
102

 #27

 3008   1   2   3   4   5

DEPARTMENT: 780
======================(17)===========
::::::::::::::
fint28
::::::::::::::
u
3
780
Join & Join
247

 #28

 3013   1   2   3   4   5

DEPARTMENT: 780
======================(29)===========
::::::::::::::
fint29
::::::::::::::
u
3
780
Join & Ships
247

 #29

 3013   1   2   3   4   5

DEPARTMENT: 780
======================n.e===========
::::::::::::::
fint30
::::::::::::::
d
3
600
703

 #30

 3008   1   2   3   4   5

DEPARTMENT: 600
======================(18)===========
::::::::::::::
fint31
::::::::::::::
u
3
600
Shopping
102

 #31

 3013   1   2   4   5

DEPARTMENT: 600
======================n.e===========
::::::::::::::
fint32
::::::::::::::
u
3
600
Join & Join
249

 #32

 3013   1   2   3   4   5

DEPARTMENT: 600
======================(31)===========
::::::::::::::
fint33
::::::::::::::
s
3
1
600

 #33

  3018   1   2   3   4  12  13  14  15  16
17

DEPARTMENT (dept_id):  600
 Resultado:  600
======================n.e===========
::::::::::::::
fint34
::::::::::::::
d
3
660
247

 #34

 3008   1   2   3   4   5

DEPARTMENT: 660
======================(19)===========
::::::::::::::
fint35
::::::::::::::
s
3
1
660

 #35

 3018   1   2   3   4  12  16  17

DEPARTMENT (dept_id):  660
 Resultado:  0
======================n.e===========
::::::::::::::
fint36
::::::::::::::
u
3
670
Shopping
1903

 #36
```

```
 3013   1   2   3   4   5

DEPARTMENT: 670
======================(35)===========
::::::::::::::
fint37
::::::::::::::
s
3
3
Shopping

 #37

  3018   1   8   9  10  12  16  17

DEPARTMENT (dept_name):  Shopping
 Resultado: 0
======================n.e===========
::::::::::::::
fint38
::::::::::::::
d
3
670
1903

 #38

 3008   1   2   3   4   5

DEPARTMENT: 670
======================(21)===========
::::::::::::::
fint39
::::::::::::::
s
3
2
1903

 #39

 3018   1   5   6   7  12  16  17

DEPARTMENT (manager_id):  1903
 Resultado:  0
======================n.e===========
::::::::::::::
fint40
::::::::::::::
u
3
700
Books
1607

 #40

 3013   1   2   3   4   5

DEPARTMENT: 700
======================(33)===========
::::::::::::::
fint41
::::::::::::::
s
3
2
1607

 #41

 3018   1   5   6   7  12  13  14  15  16
17

DEPARTMENT (manager_id):  1607
 Resultado:  700
======================n.e===========
::::::::::::::
fint42
::::::::::::::
d
3
700
1607

 #42

 3008   1   2   3   4   5

DEPARTMENT: 700
=========================n.e========
::::::::::::::
fint43
::::::::::::::
s
3
2
1607

 #43

 3018   1   5   6   7  12  16  17

DEPARTMENT (manager_id):  1607
 Resultado:  0
=========================n.e========
::::::::::::::
fint44
::::::::::::::
i
3
695
Sales
1191

 #44

 3003   1   2   3   4   5

DEPARTMENT: 695
=========================(07)========
::::::::::::::
fint45
::::::::::::::
s
3
1
695

 #45

 3018   1   2   3   4  12  13  14  15  16
17

DEPARTMENT (dept_id):  695
 Resultado:  695
======================n.e===========
::::::::::::::
```

```
fint46
::::::::::::::
u
3
750
Birds
1191

 #46

 3013   1   2   3   4   5

DEPARTMENT: 750
======================(34)===========
::::::::::::::
fint47
::::::::::::::
s
3
1
1191

 #47

 3018   1   5   6  16  17

DEPARTMENT (manager_id):  1191
======================n.e===========
::::::::::::::
fint48
::::::::::::::
d
3
750
1191

 #48

 3008   1   2   3   4   5

DEPARTMENT: 750
======================(24)===========
::::::::::::::
fint49
::::::::::::::
s
3
3
Sales

 #49

 3018   1   8   9  16  17

DEPARTMENT (dept_name):  Sales

======================n.e===========
::::::::::::::
fint50
::::::::::::::
d
3
690
247

 #50

 3008   1   2   3   4   5

DEPARTMENT: 690
======================(23)===========
::::::::::::::
fint51
::::::::::::::
s
3
3
Machines

 #51

 3018   1   8   9  10  12  16  17

DEPARTMENT (dept_name):  Machines
 Resultado: 0
======================n.e===========
::::::::::::::
fint52
::::::::::::::
i
3
760
Machines
1576

 #52

 3003   1   2   3   4   5

DEPARTMENT: 760
======================n.e===========
::::::::::::::
fint53
::::::::::::::
d
3
760
1576

 #53

 3008   1   2   3   4   5

DEPARTMENT: 760
======================(22)===========
::::::::::::::
fint54
::::::::::::::
s
3
2
1576

 #54

  3018   1   5   6  16  17

DEPARTMENT (manager_id):  1576

======================n.e===========
::::::::::::::
fint55
::::::::::::::
u
3
650
Controler
902

 #55
```

```
 3013   1   2   3   4   5

DEPARTMENT: 650
======================n.e===========
::::::::::::::
fint56
::::::::::::::
s
3
2
902

 #56

 3018   1   5   6  16  17

DEPARTMENT (manager_id):  902

======================n.e===========
::::::::::::::
fint57
::::::::::::::
u
3
650
Sales
902

 #57

 3013   1   2   3   4   5

DEPARTMENT: 650
======================(36)==========
::::::::::::::
fint58
::::::::::::::
s
3
3
Sales

 #58

 3018   1   8   9  16  17

DEPARTMENT (dept_name):  Sales

======================n.e===========
::::::::::::::
fint59
::::::::::::::
i
3
910
Controler
1740

 #59

 3003   1   2   3   4   5

DEPARTMENT: 910
======================(09)==========
::::::::::::::
fint60
::::::::::::::
s
3
2
1740

 #60

 3018   1   5   6   7  12  13  14  15  16
17

DEPARTMENT (manager_id):  1740
 Resultado:  910
```

Esta etapa encerra o teste de Integração para a variável DEPARTMENT para o Módulo $Mod_3$ com associações de *ciclo1-intra*.

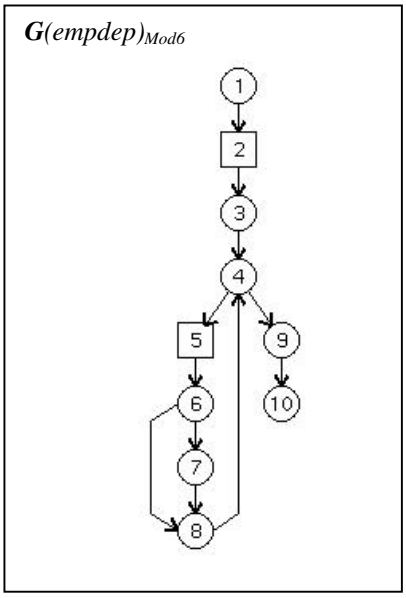
Existem Unidades que para serem exercitadas ainda no teste *intra-modular* necessitam ser associadas a partir dos critérios de ciclo2. Como exemplo iremos apresentar casos que recaem em associações de ciclo2 aplicadas ao teste *inter-modular* cujas características são semelhantes aos critérios do teste *intra-modular*.

Inicialmente, é apresentada uma parte do Módulo $Mod_6$ considerando apenas a Unidade (6002) que utiliza as variáveis DEPARTMENT e EMPLOYEE.

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\ else
fprintf(path," %2d\n",num);

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

. . .
/* Declarando estrutura referente a tabela EMPLOYEE */
```

```
typedef struct {char e_emp_id [11];
                char e_manager_id[11];
                char e_emp_fname[20];
                char e_emp_lname [20];
                char e_dept_id[11];
                char e_street[40];
                char e_city[20];
                char e_state[5];
                char e_zip_code[12];
                char e_phone [15];
                char e_status[11];
                char e_ss_number[11];
                float e_salary;
                char e_start_date[16];
                char e_termination_date[16];
                char e_birth_date[16];
                char e_bene_health_ins[15];
                char e_bene_life_ins[13];
                char e_bene_day_care[13];
                char e_sex[3]; }RECORD1;

/* Declarando estrutura referente a tabela SALES_ORDER  */

typedef struct {char s_id [11];
                char s_cust_id[11];
                char s_order_date[16];
                char s_code[11];
                char s_region[10];
                char s_emp_id[11]; }RECORD2;

/* Declarando estrutura referente a tabela DEPARTMENT  */

typedef struct {char d_dept_id [11];
                char d_dept_name[40];
                char d_manager_id[11]; }RECORD3;
. . .

EXEC SQL BEGIN DECLARE SECTION;
int        h1, he1, he2, he5, hso1,hso2,hso6,hd1,hd3,hfd4;

VARCHAR    h2[15], h3[20], h4[35],h5[20],h6[2],h7[10],h8[12],h9[35],
           he3[20], he4[20],he6[40],he7[20],he8[4],he9[9],he10[10],
           he11[1],he12[11],he17[1],he18[1],he19[1],he20[1],hd2[40],
           hso4[2],hso5[7],hfc1[4],hfc2[10],hfc3[50],hfd2[7],
           hfd3[5],hp1[11],hp2[20],hp3[20],hp4[30],hp5[20],hsi1[11],hsi2[7],
           hsi3[11],hsi5[10];
char       he14[12], he15[12], he16[12],hso3[12], hfd1[5];
float      he13,hsi4,hp6,hp7;
EXEC SQL END DECLARE SECTION;

EXEC SQL BEGIN DECLARE SECTION;
  int  mhp1, mhsi1, mhd1, mhe1,mhc1,mhso1;
  char mhfd1[5];
  VARCHAR mhfd2[7], mhfd3[5], mhfc1[4];
EXEC SQL END DECLARE SECTION;

char me_emp_id[11], mso_id[11],md_dept_id[11],mfc_code[7],
     mfd_year[6],mfd_quarter[5],mfd_code[7], mp_prod_id[11],
     msi_id[11],mc_cust_id[11];

EXEC SQL INCLUDE SQLCA;
. . .
int empdep()

/*  1 */   {
             FILE * path = fopen("empdep/path.tes","a");
             FILE * key = fopen("empdep/key.tes","a");
             static int printed_nodes = 0;
/*  1 */    EXEC SQL BEGIN DECLARE SECTION;
               int    cod_dept;
```

```
            EXEC SQL END DECLARE SECTION;
            char e_emp_fname[20];
            char e_emp_lname [20];
            char e_emp_nome_aux[40];
            char e_city[20];
            char e_state[5];
            char e_phone [15];
            char d_dept_id [11];
            char d_dept_name[40];
            ponta_de_prova(6002);
            ponta_de_prova(1);
/*  1 */    printf ("\n Digite o codigo do DEPARTMENT (100..1000): ");
/*  1 */    gets (d_dept_id);
/*  1 */    cod_dept = atoi (d_dept_id);
            ponta_de_prova(2);
/*  2 */    EXEC SQL DECLARE emp_dep_cursor CURSOR
                     FOR SELECT  d.dept_id,
                                 e.emp_id,
                                 d.dept_name,
                                 e.emp_fname,
                                 e.emp_lname,
                                 e.city,
                                 e.state,
                                 e.phone
                         FROM   DEPARTMENT d, EMPLOYEE e
                         WHERE  d.dept_id  = :cod_dept  and
                                d.dept_id  = e.dept_id;
                ponta_de_prova(3);
/*  3 */        EXEC SQL OPEN emp_dep_cursor;
/*  3 */        printf("\n Relacao de empregados:\n");
/*  3 */        printf("DEPARTMENT: %s \n", d_dept_id);
/*  3 */        printf("|    DEPARTMENT =  |             NOME DO EMPREGADO           |
CIDADE         |ESTADO|  TELEFONE    | \n");
/*  3 */       printf("|====================");
                                          /* 3 */
printf("|=========================================|==================
                    ===|======|=============| \n");
/*  3 */       sqlca.sqlcode=0;
/*  4 */       while (sqlca.sqlcode != 1403)
                 {
                    ponta_de_prova(4);
                    ponta_de_prova(5);
/*  5 */              EXEC SQL FETCH emp_dep_cursor
                             INTO   :mhd1,
                                    :mhe1,
                                    :hd2,
                                    :he3,
                                    :he4,
                                    :he7,
                                    :he8,
                                    :he10;
                    ponta_de_prova(6);
/*  6 */              if (sqlca.sqlcode != 1403)
                       {
                         ponta_de_prova(7);
/*  7 */                   hd2.arr [hd2.len] = '\0';
/*  7 */                   strcpy (d_dept_name, hd2.arr);
/*  7 */                   he3.arr [he3.len] = '\0';
/*  7 */                   strcpy (e_emp_fname, he3.arr);
/*  7 */                   he4.arr [he4.len] = '\0';
/*  7 */                   strcpy (e_emp_lname, he4.arr);
/*  7 */                   he7.arr [he7.len] = '\0';
/*  7 */                   strcpy (e_city, he7.arr);
/*  7 */                   he8.arr [he8.len] = '\0';
/*  7 */                   strcpy (e_state, he8.arr);
/*  7 */                   he10.arr [he10.len] = '\0';
/*  7 */                   strcpy (e_phone, he10.arr);
                          strcpy (e_emp_nome_aux,e_emp_fname);
                          strcat (e_emp_nome_aux, " ");
```



```
                      strcat (e_emp_nome_aux,e_emp_lname);
                      itoa(mhd1,md_dept_id); /* int para char */
                      fprintf(key, "DEPARTMENT: %s    \n",md_dept_id);
                      itoa(mhe1,me_emp_id); /* int para char */
                      fprintf(key, "EMPLOYEE: %s    \n",me_emp_id);
/*  7 */              printf("| %18s | %40s | %19s | %4s | %11s |\n",d_dept_name,
e_emp_nome_aux, e_city, e_state, e_phone);

                    }
                  ponta_de_prova(8);
/*  8 */        }
              ponta_de_prova(4);
              ponta_de_prova(9);
/*  9 */
printf("|=================================================================================
======================|\n");
/*  9 */
              ponta_de_prova(10);
              EXEC SQL WHENEVER NOT FOUND CONTINUE;
              EXEC SQL CLOSE emp_dep_cursor;
              fclose(path);
              fclose(key);
/*  9 */      return (sqlca.sqlcode);
/* 10 */    }

. . .
main( )
. . .
```

Os Módulos $Mod_3$ são associados com o Módulo $Mod_6$ cujas associações envolvem as *variáveis tabela* DEPARTMENT, EMPLOYEE e SALES_ORDER.

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS DE INTEGRACAO POT_S_USO INTER_MODULAR: MOD3 X MOD6 --
CICLO2 - MESMA TUPLA - EMPLOYEE, SALES_ORDER E DEPARTMENT

Associacoes originadas de Insemp(3001),Inssal(3002) e Insdep(3003)

01- <3003,(3,4),3001,(3,4),6002,(4,9),{DEPARTMENT,EMPLOYEE}> * 1 2 3
02- <3003,(3,4),3001,(3,4),6002,(6,7),{DEPARTMENT,EMPLOYEE}> * 1 2 3
03- <3003,(3,4),3001,(3,4),6002,(6,8),{DEPARTMENT,EMPLOYEE}> * 1 2 3
04- <3001,(3,4),3003,(3,4),6003,(4,9),{DEPARTMENT,EMPLOYEE}> * 4 5 6
05- <3001,(3,4),3003,(3.4),6003,(6,7),{DEPARTMENT,EMPLOYEE}> * 4 5 6
06- <3001,(3,4),3003,(3,4),6003,(6,8),{DEPARTMENT,EMPLOYEE}> * 4 5 6
07- <3001,(3,4),3002,(3,4),6004,(4,9),{SALES_ORDER,EMPLOYEE}> * 7 8 9
08- <3001,(3,4),3002,(3,4),6004,(6,7),{SALES_ORDER,EMPLOYEE}> * 7 8 9
09- <3001,(3,4),3002,(3,4),6004,(6,8),{SALES_ORDER,EMPLOYEE}> * 7 8 9

Associacoes originadas de Delemp(3006),Delsal(3007) e Deldep(3008)

10- <3006,(3,4),3008,(3,4),6002,(4,9),{DEPARTMENT,EMPLOYEE}> * 10 11 12
11- <3006,(3,4),3008,(3,4),6002,(6,7),{DEPARTMENT,EMPLOYEE}> * 10 11 12
12- <3006,(3,4),3008,(3,4),6002,(6,8),{DEPARTMENT,EMPLOYEE}> * 10 11 12
13- <3008,(3,4),3006,(3,4),6003,(4,9),{DEPARTMENT,EMPLOYEE}> * 13 14 15
14- <3008,(3,4),3006,(3,4),6003,(6,7),{DEPARTMENT,EMPLOYEE}> * 13 14 15
15- <3008,(3,4),3006,(3,4),6003,(6,8),{DEPARTMENT,EMPLOYEE}> * 13 14 15
16- <3007,(3,4),3006,(3,4),6004,(4,9),{SALES_ORDER,EMPLOYEE}> * 16 17 18
17- <3007,(3,4),3006,(3,4),6004,(6,7),{SALES_ORDER,EMPLOYEE}> * 16 17 18
18- <3007,(3,4),3006,(3,4),6004,(6,8),{SALES_ORDER,EMPLOYEE}> * 16 17 18



Associacoes originadas de Updemp(3011), Updsal(3012) e Upddep(3013)

19- <3013,(3,4),3011,(3,4),6002,(4,9),{DEPARTMENT,EMPLOYEE}> n.e 19 20 21
20- <3013,(3,4),3011,(3,4),6002,(6,7),{DEPARTMENT,EMPLOYEE}> n.e 19 20 21
21- <3013,(3,4),3011,(3,4),6002,(6,8),{DEPARTMENT,EMPLOYEE}> n.e 19 20 21
22- <3011,(3,4),3013,(3,4),6003,(4,9),{DEPARTMENT,EMPLOYEE}> n.e 22 23 24
23- <3011,(3,4),3013,(3,4),6003,(6,7),{DEPARTMENT,EMPLOYEE}> n.e 22 23 24
24- <3011,(3,4),3013,(3,4),6003,(6,8),{DEPARTMENT,EMPLOYEE}> n.e 22 23 24
```

```
25- <3011,(3,4),3012,(3,4),6004,(4,9),{SALES_ORDER,EMPLOYEE}> n.e 25 26 27
26- <3011,(3,4),3012,(3,4),6004,(6,7),{SALES_ORDER,EMPLOYEE}> n.e 25 26 27
27- <3011,(3,4),3012,(3,4),6004,(6,8),{SALES_ORDER,EMPLOYEE}> n.e 25 26 27

=============================================================
```

As seqüências de execução, mostradas a seguir, são utilizadas para exercitar as associações de ciclo2, mostradas acima, que envolvendo dois Módulos: $Mod_3$ e $Mod_6$ (*inter-modular*) (no exemplo são mostrados apenas os casos de testes que exercitam os elementos requeridos em negrito).

```
::::::::::::::
fddep01
::::::::::::::
i
3
700
Drinks
249

 #1

 3003   1   2   3   4   5

DEPARTMENT: 700
========================
::::::::::::::
fdemp01
::::::::::::::
i
1
1920
249
Edmundo
Spoto
700
520 Nelson de Souza Barbara Street
Campinas
SP
13080-260
0192080705
A
118998234
34992.000
12-FEB-87
(NULL)
11-MAR-57
Y
Y
N
M

 #2

 3001   1   2   3   4   5

EMPLOYEE: 1920
========================
::::::::::::::
finu01
::::::::::::::
e
700
t

 #3

 6002   1   2   3   4   5   6   7   8   4
  5   6   8   4   9  10

DEPARTMENT: 700
EMPLOYEE: 1920
================*1,2,3==============
::::::::::::::
fddep04
::::::::::::::
u
3
700
Encadernacao
501

 #10

 3013   1   2   3   4   5

DEPARTMENT: 700
========================
::::::::::::::
fdemp04
::::::::::::::
u
1
1920
501
700
114 Great Plain Avenue
Winchester
MA
01890
6175557835
A
45000.00

 #11

 3011   1   2   3   4   5

EMPLOYEE: 1920
========================
::::::::::::::
finu04
::::::::::::::
e
700
t
```

```
 #12

 6002  1  2  3  4  5  6  7  8  4
  5  6  8  4  9  10

DEPARTMENT: 700
EMPLOYEE: 1920
==================*10,11,12=======
::::::::::::::
fdemp07
::::::::::::::
d
1
1920

 #19

 3006  1  2  3  4  5

EMPLOYEE: 1920
========================
::::::::::::::
fddep07
::::::::::::::
d
3
700
501

 #20

 3008  1  2  3  4  5

DEPARTMENT: 700
========================
::::::::::::::
finu07
::::::::::::::
e
700
t

 #21

 6002  1  2  3  4  5  6  8  4  9
 10

============n.e:19,20,21=======
```

❖ **COBERTURA**

A cobertura é mostrada da seguinte forma:

Os dados de teste *fddep01*, *fdemp01* e *finu01* quando executados seqüencialmente geram os caminhos 3003 1 2 3 4 5 – 3001 1 2 3 4 5 – 6002 1 2 3 4 5 6 7 8 4 5 6 8 4 9 10. Como as tuplas são respectivamente:
3003 DEPARTMENT: 700 –
3001 EMPLOYEE: 1920 – e
6002 DEPARTMENT: 700
EMPLOYEE: 1920
Neste exemplo, os casos de teste 1, 2 e 3 cobrem as associações 1, 2 e 3 da página *194*. Assim, cada seqüência de execução de 3 dados de teste gera outros três casos de teste para satisfazer uma ou mais associações de ciclo2. Os números dos casos de teste anotados na frente das associações representam as seqüências de casos de testes necessárias para cobrir a associação. Neste exemplo foram apresentados apenas alguns casos de teste para ilustrar a cobertura dos elementos requeridos (*associações def-t-uso de ciclo2*).

# ÍNDICE REMISSIVO

A
ABDR, 3, 4, 5, 20, 23, 26, 28, 30, 32, 35, 37-39, 43, 45, 48, 50, 89, 90, 92, 105, 109-112, 115, 128-129, 133, 136-142, 157
Análise de inclusão, 76
Análise de Propriedades, 61, 75
análise estática, 8, 116, 151, 154
*A*nomalia, 39, 67
*A*plicabilidade, 13
Aplicação de Banco de Dados Relacional, 23, 35
arcos, 7-9, 14, 16, 19, 36-37, 41-43, 54-55, 67, 69, 91, 111-112, 122, 137-138, 152, 154
*A*ssociação, 10, 15, 17, 45-47, 65-66, 70-74, 77-78, 81, 105, 111-112, 124, 126, 130, 136
*A*ssociação definição-c-uso, 9-10, 45
*A*ssociação definição-p-uso, 10
*A*ssociação definição-s-uso, 45
*A*ssociação definição-t-uso, 45, 51, 68, 72, 75
*A*ssociação definição-t-uso interprocedimental de chamada, 52
*A*ssociação definição-t-uso-inter, 64
Associação definição-uso, 15, 40, 54
Associação inter-unidade, 52
*A*ssociação potencial-uso, 16
*A*ssociação-definição-s- interprocedimental de chamada, 50
*A*ssociação-definição-s-uso interprocedimental de retorno, 50
*A*ssociação-definição-t-uso interprocedimental de chamada, 50
*A*ssociação-definição-t-uso interprocedimental de retorno, 50
*A*ssociação-p-uso, 9, 45
*A*ssociações definição-t-uso, 61
*Associações* definição-t-uso de integração, 72
*Associações* definição-t-usos, 83
*Associações* definição-uso, 19, 64
*A*tributo, 21
*Avaliação*, 2, 97, 99, 112, 116, 128, 129, 132, 151
B
Banco de Dados Relacional, 3, 7, 20, 23, 32, 35, 39, 52, 57, 112, 119, 139, 141, 143, 149, 155, 157
big-bang, 12
bloco de comandos, 7, 8, 19, 98
Bottom-Up, 12
C
C–adequado, 13

caminho, 8-9, 37, 40-41, 43, 45-46, 50, 66, 69-70, 72-74, 76-78, 81, 111, 123-124, 126, 151, 154
caminho completo, 8, 14-15, 47, 51, 66, 72, 76
caminho de integração, 19, 51
caminho de integração interprocedimental, 51
caminho executável, 47
*caminho* livre de definição, 74
*caminho* livre de definição persistente, 43, 45, 52, 66
*caminho* livre de laço, 45
caminho simples, 8-9, 45
caso de teste, 2, 7, 11, 32, 98, 111-112, 123-124, 127, 154
classes de defeitos, 4, 10, 11, 57, 138, 142
classes distintas de erros, 7
clusters, 12
cobertura, 47, 131,136,198
coleta e análise de requisitos, 23
comando de manipulação da SQL, 44, 166-171
comandos da LI, 99
comandos de linguagem, 3
comandos de SQL, 27,159-173
comando COMMIT, 40
comandos executáveis da linguagem SQL, 25, 28, 36, 90,163-173
complexidade de um critério, 4, 13,76, 83-85, 87
conceitos básicos, 7, 20, 23, 35, 39, 43, 45, 142
conjunto factível:, 47
Critérios:
- contexto elementar de dados, 14
- estrutural de teste de integração, 51
- todos os arcos, 16
- todos os caminhos, 13
- todos os ramos, 13
- todos-arcos, 8, 154
- todos-caminhos, 8, 17
- todos-nós, 8, 154
- *ver (todos os...)*
- baseados em fluxo de dados, 13
- de integração inter-modular, 59
- de teste, 4, 7, 19, 32, 35, 39, 45, 47, 57,61, 65, 68-69, 71, 76, 78, 83, 87, 89, 98, 101, 105, 110-112, 117-119, 124, 128, 134-136, 138-142, 155
- de teste de integração, 18
- de teste estrutural, 17
c-uso, 9, 15, 43, 47
c-uso global, 43
c-uso local, 43
D
declarativos, 25

defeito, 1, 11, 46, 67
defg(i), 16
*definição*, 8, 9, 13, 16-17, 30, 39-43, 46, 53, 55, 56, 57, 65, 66, 68-75, 77-78, 83, 87, 89, 98, 101, 106, 109, 111-112, 118, 124, 126, 130-131, 134, 137, 139-140, 142, 154, 157, 159
definição global, 9-10, 17, 43, 45-46, 69
definição persistente, 40, 43-44, 51-52, 62
definição por referência, 41
definição por valor, 41

definições globais, 9, 50
defT<ℓ,i>, 43
Dependência de dados, 52, 55, 68, 72
Dependência externa de dados, 53, 55
Dependência interna, 53
*Desenvolvimento*, 1-5, 27, 30-31, 139, 141, 157
domínio, 21
domínio de entrada, 7
drivers, 11
dsu(v,i), 43
dtu-caminho, 45
dtu-caminhos, 68
du-caminho, 9, 45
E
*elementos requeridos*, 9, 10, 65, 67, 74-75, 83-85, 87, 89, 97, 109-112, 116, 124-131, 133, 135-136, 139, 174, 184, 195
error, 1, 90
*especificação*, 2, 7, 10, 24, 31
esquema da base de dados relacional, 22
estrutural, 4, 5, 7, 11, 19, 23, 35, 38, 39, 47, 57, 61, 65-66, 69, 75, 89, 97, 110, 115, 133, 139-140
executável ou factível, 15, 25
exemplo de aplicação, 37, 47, 74, 115, 130, 136, 140
exemplos de utilização, 103, 115, 117-118, 137, 197
F
falha, 1, 4, 72, 99
Família de Critérios, 9, 10, 15-18, 20, 66, 147 - Potenciais Usos, 10
fases de teste, 35, 141
ferramenta de suporte, 87, 141
ferramenta de visualização, 47
ferramentas, 2, 17, 30, 47, 136, 139-140
Fluxo de Controle, 89, 90, 148, 154
fluxo de dados inter-modular, 64
fluxo de dados intra-modular, 62
fpdcu(x,i), 16
fpdpu(x,i), 16
funcional, 3, 11
G
grafo, 9, 14, 18, 39, 41, 43, 47, 48, 66, 68, 69, 76, 83, 87, 90, 105, 106, 110-112, 121-123, 154-155
grafo (i), 46
grafo de chamada, 18, 19, 39, 47, 48, 52, 65, 68-69
grafo de dependência de dados, 54, 62
grafo de dependência Interna, 55
grafo de fluxo de controle, 7, 83
grafo de fluxo de dados, 46
grafo de programa, 7, 9, 14, 17, 19, 36, 37, 46, 54, 62, 66, 87, 90, 105, 111-112, 121-123
Grafo geral, 55
grafo(i), 46
grafo-def, 46
grau de uma relação, 21
I
implementação, 2, 3, 4, 7, 11, 23, 26, 28, 31, 44, 61, 89, 98, 111-112, 129, 133, 137, 139, 141, 159
inclui, 14, 76
inclui parcialmente, 77
incomparáveis, 14, 76, 81
indefinição, 8, 39, 106
instância, 22
integração baseada no grafo de chamada, 35
inter-modular, 35, 37, 38, 39, 47, 53, 57, 61, 74, 75, 76, 97-98, 105, 106, 110, 112, 115, 118, 127, 130, 131, 133-137, 139-142, 173, 192, 195
interprocedimental, 19
intra-modular, 35, 37, 39, 47, 53, 55, 57,61, 68, 72, 74, 76, 97-98, 105-106, 112, 115, 118-119, 127-128, 131-137, 139-142, 173, 192
L
linguagem de definição de dados, 32
linguagem de manipulação de dados, 32
linguagem hospedeira, 24
linguagens hospedeiras, 4, 25
M
mesma tupla, 65-74, 77, 81-85, 116, 129, 134-135, 137, 139-140, 142, 185
mistake, 1
Modelo de Dados Relacional, 21
modelo relacional formal, 21, 25
modelos de fluxo de dados, 89
modelos de instrumentação, 89
módulo, 2, 12, 36, 62, 72-75, 98, 115, 142, 153-155
Módulos de programa, 35
multi-grafo, 19, 55
N
nós, 7-9, 18-19, 36-37, 41, 43, 48-49, 64, 67-72, 75-76, 88, 90, 96, 101, 107, 109-110, 120, 135, 150, 152
n-tuplas, 22
O
ocorrência, 40, 42, 57
ordem de complexidade, 85
ordem exponencial, 81
ordem parcial, 14
P
pairwise, 12, 20, 67
par de nós, 40, 45, 51, 72, 79, 109
pares de nós, 65
pdcu(v,i), 16
pdpu(v,i), 16
persistência dos dados, 52
planejamento, 1, 2, 31, 139
POKE-TOOL, 46, 47, 87, 96, 97, 104, 109-110, 113, 114, 118, 120, 129, 134, 139, 141, 143, 145, 149, 150, 153
potencial associação, 10, 17, 46, 83
potencial-du-caminho, 16

programas convencionais, 35, 47, 48, 52, 74, 126
programas de aplicação, 4, 25, 35, 39, 43, 45-49, 57, 59, 66-67, 75, 95-96, 109-110, 115, 131, 135
programas de Aplicações de Banco de Dados Relacional, 3
projeto, 1, 2, 23, 31, 59, 87, 113, 126, 135
projeto de banco de dados conceitual, 23
projeto de banco de dados físico, 24
projeto de banco de dados lógico, 23
projeto lógico, 27
propriedades, 13, 14, 23, 30, 59, 69, 73, 138-139
p-uso, 9-10, 15, 45, 47
Q
qualidade de, 3, 13
R
redefinição, 9, 43, 52
relação, 22, 23
requisitos de teste, 2, 13, 20, 67
Restrições de Integridade Referencial, 32
S
sintaticamente alcançável, 43
Sistema Gerenciador, 3, 27, 36, 155

SQL, 3, 4, 23-27, 29-30, 32, 35-37, 40-42, 49, 57, 59, 60, 62, 65-67, 75, 76, 87-90, 96-99, 104, 106-107, 109-110, 113, 116-118, 120, 122-123, 131, 135, 137-144, 153, 155, 157-175, 190-192
SQL embutida, 27
SQLBench, 30
stubs, 11, 12
sub-caminho, 65
sub-caminho livre de definição, 50
s-uso, 42
s-uso (i), 43
T
Tabelas de Visão, 26
técnica de teste, 3, 57
técnica estrutural, 2, 7
Todos os t-usos-ciclo2-inter, 75
Todos os t-usos-ciclo2-intra, 74
*T*odos-c-usos, 15
*T*odos-c-usos/algum-p-uso, 15
*T*odos-du-caminhos, 15
*T*odos-potenciais-du-caminhos, 17
*T*odos-potenciais-usos, 17
*T*odos-potenciais-usos/DU, 17
*T*odos-p-usos, 15
*T*odos-p-usos/algum-c-uso, 15
*T*odos-usos, 15
*t*op-down, 12
tratamento de erro, 41, 100
U
unidade chamada, 48
unidade chamadora, 48
técnica funcional, 2, 7
terminologia, 7, 21, 35, 39, 137
teste de integração, 2, 4, 11-12, 14, 17-20, 35, 37, 39, 47-48, 55-57, 60, 63, 66-67, 69, 74, 85, 96, 104, 108-110, 114, 117, 122-123, 125-126, 128-130, 132-135, 137-140, 153
teste de integração baseado na dependência de dados, 47, 56, 59, 122, 126, 137
teste de integração baseado no grafo, 59
teste de integração intra-modular, 59, 60
teste de sistema, 2
teste de software, 2-3, 103, 139
Teste de Software, 1, 2, 141, 144, 146
teste de unidade, 2, 4, 10-12, 14, 17, 18, 20, 35, 37, 39, 44-45, 47-48, 57, 59-60, 63-65, 67, 69, 80, 96, 104, 107, 109-110, 117-118, 122, 132, 134, 136-140, 149, 153
teste estrutural, 11, 13, 20, 37
teste inter-métodos:, 20
teste intra-classe, 20
teste intra-método, 20
tipos de dados, 21
todas as relações, 18
Todas os t-usos-ciclo1-intra, 70
todas relações múltiplas, 18
todas seqüências de chamadas, 18
todas seqüências simples descendentes, 18
todas seqüências simples descendentes sem laço, 18
todas-definições, 15
Todos os dtu-caminho interprocedimentais de retorno, 69
Todos os dtu-caminhos interprocedimentais de chamada, 68
Todos os dtu-caminhos-intra, 71
todos os módulos, 18
Todos os t-usos interprocedimentais de chamada, 68
Todos os t-usos interprocedimentais de retorno, 68
Todos os t-usos-ciclo1-inter, 75

Unidades de Programa, 35
uso, 8, 10, 17, 23, 39-43, 46-47, 55, 65-66, 68-75, 77, 81, 87, 89, 98, 106, 109, 111-112, 116, 118, 124, 126, 130, 133, 137, 139-140, 142, 157, 165-167, 169, 171
uso de t, 67
uso persistente, 42, 62
V
valores atômicos, 21
variáveis de programa, 13, 39-40, 49, 61, 69, 106
variáveis host, 29, 35, 39-40, 42, 45, 49, 51, 61, 69, 101, 106, 118, 155, 158, 167
variáveis persistentes, 30, 57, 61, 65-66, 68-69, 72, 77-78, 89, 101, 108, 112, 124, 139, 140-142

variáveis tabela, 30, 35, 38, 39, 42, 44-46, 49, 57, 61-62, 64-65, 69, 71, 74, 78, 101, 106, 109, 118, 124, 130-131, 133, 135, 137, 139-141, 155, 194
variáveis tabela de visão, 39
ViewGraph, 47

***Informações***:

O relatório dos exemplos de utilização dos critérios propostos nesta tese pode ser obtido com o autor *Prof. Edmundo Sérgio Spoto (dino@din.uem.br)* – Universidade Estadual de Maringá –PR – Departamento de Informática, ou com o *Prof. Dr. Mario Jino* na Universidade Estadual de Campinas – Faculdade de Engenharia Elétrica e de Computação (FEEC).