Teste Estrutural de Software: Uma abordagem para Aplicações de Banco de Dados Relacional

Edmundo Sérgio Spoto
DIN/CTC/UEM
Av. Colombo, 5790
Cep: 87020-900
Maringá - PR, Brasil
dino@din.uem.br

Mario Jino
DCA/FEEC/UNICAMP
Av. Albert Einstein, 400,
Cep: 13083-970,
Cx. Postal 6101,
Campinas - SP, Brasil
jino@dca.fee.unicamp.br

José Carlos Maldonado
ICMC-USP
Av. Dr. Carlos Botelho, 1465,
Cep: 13560-970,
Cx. Postal 668,
São Carlos - SP, Brasil
jcmaldon@icmc.sc.usp.br

Resumo

Uma nova abordagem é apresentada para o Teste Estrutural de programas de Aplicação de Banco de Dados Relacional (ABDR) com **SQL** embutida. São apresentados os conceitos básicos de teste estrutural baseados em fluxo de controle e fluxo de dados utilizados nesta abordagem. Devido à natureza de uma ABDR, a estratégia de teste proposta neste artigo considera dois modelos de fluxo de dados: *intra-modular* e *inter-modular*. O Modelo *intra-modular* acomoda as etapas de teste de unidade e teste de integração de um programa da ABDR. O Modelo *inter-modular* faz a integração dos programas que compõem a ABDR. Os critérios da etapa de teste de unidade são apresentados e discutidos. Os demais critérios são apenas citados. Para finalizar apresentamos resultados extraídos do experimento realizado.

Key Words: Banco de dados relacional, SQL, Teste estrutural, Teste de fluxo de dados, critério de teste.

Abstract

An approach is proposed for structural testing of programs concerning Relational Database Applications (RDA). The basic concepts of data flow and control-flow based on structural testing are presented. Due to the nature of RDA the proposed testing strategy considers two data-flow models: intra-modular and inter-modular data-flow. The two models aim to accommodate the usual stages of an RDA testing strategy. The criteria to be used in the unit testing stage (*intra-modular data flow model*) are presented and discussed with examples of their application; results from an experiment are also presented.

Key Words: Relational Database, SQL, Structural Testing, Data Flow testing, testing criteria.

1 Introdução

As etapas do ciclo de vida de um software dependem de técnicas apropriadas e ferramentas para conduzir o processo de desenvolvimento com mais segurança e torná-lo menos oneroso. Uma das etapas mais caras e que muitas vezes é sacrificada pela falta de ferramentas e técnicas disponíveis, é a de teste estrutural de software, que é o enfoque deste trabalho. Apesar da existência de vários métodos e técnicas, estes não cobrem todas as áreas de desenvolvimento de software. A atividade de teste é hoje considerada uma das mais caras, atingindo aproximadamente 50% do custo de todo o desenvolvimento do software [MYE79, PRE97]. Um dos problemas básicos da atividade de teste é assegurar a qualidade dos dados de teste; essa qualidade é determinada com medidas de cobertura de critérios de teste obtidas através de ferramentas de apoio.

Nos últimos 20 anos, pesquisadores da área de teste têm contribuído para a sistematização da atividade de teste tornando-a mais rigorosa através do uso de critérios de Teste Estrutural Baseado em Análise de Fluxo de Dados e em Fluxo de Controle. Herman [HER76] apresentou critérios de teste baseados em fluxo de dados que 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. Frankl e Weyuker [FRA85, FRA88] desenvolveram uma Família de Critérios baseados na análise de Fluxo de Dados (FCFD) com relação às variáveis. Laski e Korel [LAS83] introduziram uma família de critérios de teste que usam informações de fluxo de dados denominados ambiente de dados, contexto elementar de dados e contexto ordenado de dados. Ntafos [NTA84] introduziu uma família de critérios denominada *K-tuplas requeridas*; a família de critérios requer que todas as seqüências de (k-1) interações *definição-uso* sejam exercitadas. Maldonado [MAL91] desenvolveu uma Família de Critérios baseados no conceito *potencial uso* (FCPU). Em cada caso, deve-se especificar um conjunto de elementos requeridos para o programa cuja cobertura deve ser obtida através dos *casos de teste*[1], ou seja, elementos que devem ser exercitados pelos casos de teste.

Para implementar os critérios da FCPU, Chaim [CHA97] desenvolveu uma ferramenta denominada POKE-TOOL que dá apoio ao teste de unidade. A ferramenta VIEWGRAPH [VIL97, CRU99] permite visualizar as etapas parciais do teste de unidade a partir das informações geradas pela POKE-TOOL. Essas duas ferramentas foram adaptadas para a realização do experimento deste trabalho.

Aranha [ARA00] desenvolveu vários critérios para testar o esquema da base de dados utilizado na Aplicação de Banco de Dados Relacional (ABDR). Os elementos da base de dados a serem testados são os atributos e as restrições de integridade que deverão ser exercitados através de operações da linguagem **SQL**. Na prática do teste de uma ABDR, a aplicação desses critérios deve anteceder a dos critérios apresentados neste trabalho, neste caso, a base de dados é considerada correta e o que testamos são os programas da ABDR com **SQL** embutida no código.

Nosso trabalho constituiu em investigar a adequação do uso de critérios de teste estrutural existentes na literatura e adaptá-los para o seu uso em programas de ABDR com **SQL** embutida. São apresentadas as técnicas de teste estrutural que possibilitaram a realização deste tipo de teste em programas de ABDR. Apresentamos apenas os resultados obtidos referentes à etapa de teste de unidade [SPO99].

Na Seção 2 apresentamos as definições básicas e a terminologia adotada. Na Seção 3 apresentamos os modelos de fluxo de dados e as estratégias para o teste de ABDR. Na Seção 4 apresentamos as definições dos critérios de teste de unidade e discutimos as demais etapas de teste. Na Seção 5 apresentamos e discutimos os resultados obtidos. Finalmente, na Seção 6 apresentamos as conclusões e os trabalhos futuros.

2 Definições Básicas e Terminologia

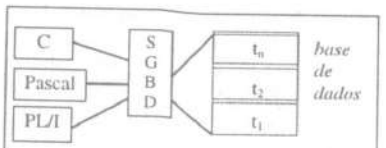

Figura 1: Exemplo de uma ABDR

A literatura investigada não traz nenhuma abordagem efetiva de teste estrutural aplicado a sistemas de Banco de Dados Relacional (BDR). A abordagem apresentada em [MAN89] descreve uma técnica que gera uma base de dados mínima a partir de uma base mais completa que é suficiente para ser usada como massa de teste para exercitar as dependências entre as possíveis classes de

[1]Um *caso de teste* é o conjunto formado por: *i)* dados de entrada para uma execução do programa e *ii)* a saída esperada para esses dados de entrada.

consultas (do tipo *select-project-join*) de um programa em teste. Foi desenvolvida na Universidade de *Cornell - EUA* a ferramenta denominada ***SQLBench*** que utiliza um conjunto de testes *Benckmark* AS3AP (ANSI **SQL** *Standard Scalable and Portable*) para sistemas de BDR com **SQL**, com o objetivo de testar o desempenho do banco de dados [BUT93].

Uma Aplicação de Banco de Dados Relacional é um conjunto de *Módulos de Programas*: ABDR = $\{Mod_1, Mod_2, ..., Mod_m\}$, $m \geq 1$, onde cada Módulo de Programa Mod_i pode ser escrito em linguagem C, Pascal, Fortran, Ada ou PL/I, dependendo do Sistema Gerenciador de Banco de Dados (SGBD) que hospeda a linguagem de manipulação da base de dados **SQL** [ELM94].

Cada *Módulo de Programa* é composto por vários procedimentos que denominamos de *Unidades de Programas* - $Mod = \{UP_1, .., UP_n\}$, para $n \geq 1$. Esses programas interagem com as tabelas (t_1, t_2,..) da Base de Dados que compõem a Aplicação (*Figura1*).

Para satisfazer todas as etapas de teste de uma ABDR, foram criados dois modelos de Fluxo de Dados, baseados em Harrold e Rothermel [HAR94]: ***modelo intra-modular*** - fluxo de dados dentro de um programa; e ***modelo inter-modular*** - fluxo de dados entre programas distintos da ABDR [SPO97].

2.1 Fluxo de Controle de Programa de ABDR

O grafo de fluxo de controle que representa uma *UP* de um programa da ABDR é denotado por $\mathbf{G}(UP) = (N^{BD}, E, n_{in}, n_{out})$, $N^{BD} = N_h \cup N_S$, onde N_h é o conjunto de nós da linguagem hospedeira (C, Pascal, etc), representados no grafo por nós arredondados, e N_S é o conjunto de nós tal que cada nó corresponde a um comando executável da **SQL**, representados por nós retangulares. E é o conjunto de arcos, $E \subseteq N^{BD} \times N^{BD}$. Os nós $n_{in} \in N_h$ são os nós de entrada e $n_{out} \in N^{BD}$ são os nós de saída do grafo de programa. O fluxo de controle dos nós da **SQL** é gerado pelos comandos de tratamento de erros existentes dos SGBDRs, representados pelo comando declarativo:

"EXEC SQL WHENEVER <condição> <ação>"

Um caminho é 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 $(n_i, n_{i+1}) \in E$ que vai de n_i para n_{i+1}. O grafo de programa estabelece uma correspondência entre os nós de N_h e N_S indicando os possíveis fluxos de controle entre os nós através dos arcos [SPO97]. Para cada *UP* existe uma seqüência finita de comandos da linguagem hospedeira e comandos da linguagem **SQL**, onde os nós arredondados representam blocos de comandos e os nós retangulares representam comandos isolados da **SQL**. Um *bloco de comando* é uma seqüência de um ou mais comandos tendo a propriedade de que, sempre que o primeiro comando do bloco for executado, todos os demais comandos do bloco também o serão [HUA75].

Apenas os comandos executáveis da **SQL** como: *a)* INSERT, DELETE, UPDATE e SELECT - de manipulação de dados; *b)* COMMIT e ROLLBACK - de validação das tarefas; *c)* CONNECT e DISCONNECT - de conexão e desconexão com a Base de Dados; e *d)* OPEN, CLOSE e FETCH - de tratamento de arquivos e cursores; terão representação gráfica com nós retangulares. Os demais comandos como: INCLUDE - de inclusão de funções; e os comandos declarativos: DECLARE, WHENEVER e outros, não têm representações exclusivas e são acomodados em blocos de comandos com os demais comandos da linguagem hospedeira.

Na *Figura 2*, os nós 3, 6, 9 e 12 representam comandos SELECT da **SQL**. Os nós 1 e 19 são, respectivamente, os nós de entrada e de saída do grafo. Os arcos (3,18), (6,18), (9,18) e (12,18) representam o fluxo do programa decorrente de um erro ocorrido no comando SELECT da **SQL**, proveniente do comando de tratamento de erro "EXEC SQL WHENEVER SQLERROR GOTO *label*;" colocado no nó 1, com o *label* no nó 18. A seqüência de nós (1, 2, 3, 4, 15, 16, 17, 18, 19) é um exemplo de caminho completo. Esse grafo foi criado pela ferramenta POKE-TOOL e desenhado pela ferramenta VIEWGRAPH a partir do arquivo

"Selemp.gfc" que representa o grafo da Unidade de Programa *Selemp* pertencente ao Módulo de Programa Mod_3 utilizado no experimento.

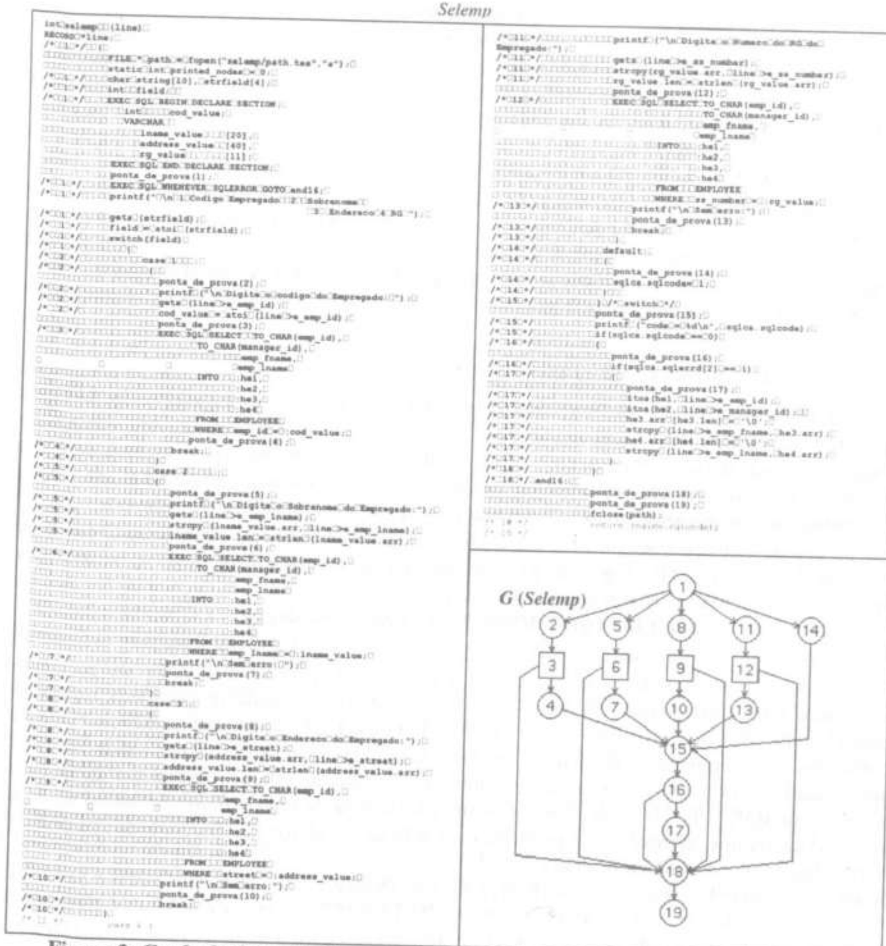

Figura 2: Grafo de Fluxo de Controle da Unidade de Programa *Selemp* do Módulo de Programa Mod_3.

A Figura 3 mostra os tipos de Fluxos gerados pelas condições dos comandos de tratamento de erros da **SQL**.

2.2 Modelos de Fluxo de Dados

O modelo de Fluxo de Dados é usado para estender o grafo de Fluxo de Controle pela associação de tipos de ocorrências de variáveis aos elementos do grafo para a determinação dos caminhos que satisfazem associações requeridas.

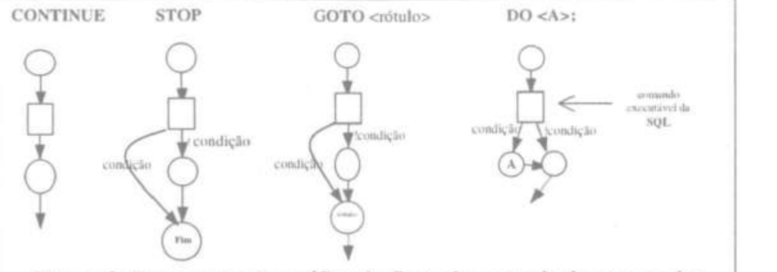

Figura 3: Representação gráfica do fluxo de controle dos comandos executáveis da SQL gerados pelas ações do comando: *"WHENEVER <condição> <ação>"*

Em geral, uma variável pode sofrer as seguintes ações no programa: *definição* (*d*), *indefinição* (*i*); ou *uso* (*u*). As variáveis utilizadas em uma Aplicação de Banco de Dados Relacional (ABDR) são classificadas em três tipos: *i) Variáveis de programa* (***P***) são as variáveis definidas e usadas apenas na *linguagem hospedeira*; *ii) Variáveis hosts* (***H***), também conhecidas como variáveis de *ligação*, são definidas e usadas em qualquer parte do *Módulo de Programa* (dentro e fora da **SQL**) e são fundamentais para estabelecer a comunicação entre as duas linguagens; *iii) Variáveis de tabela* (***T***) são definidas e usadas apenas nos comandos executáveis da **SQL.** A *definição* de uma variável ocorre sempre que um valor é armazenado em uma posição de memória. Tais definições são tratadas apenas no momento da execução do programa que as gerou. Seja $\boldsymbol{v} = \boldsymbol{P} \cup \boldsymbol{H}$ o conjunto de variáveis presentes em programas de ABDR com **SQL** embutida e seja **t** a *variável de tabela*.

Definição-1 Uma variável é *persistente* quando se mantém viva após a execução do programa que a gerou. Uma *definição* de uma *variável persistente* ocorre quando um valor é armazenado em uma posição de memória secundária (disco, fita, etc.).

Vale observar que toda variável em um programa convencional possui, algumas vezes, a exigência sintática de ser declarada antes que ocorra uma definição ou ser definida antes que ocorra um uso; caso contrário, poderá ocorrer um erro de compilação ou uma *"anomalia"*, respectivamente. Isso não ocorre para as *variáveis de tabela*, definidas antes da execução de qualquer *Módulo de Programa* da ABDR [ELM94].

Os comandos de manipulação INSERT, DELETE e UPDATE da **SQL** caracterizam a ocorrência de *definição* da *variável de tabela*. A *variável de tabela* pode ser tratada como uma *variável persistente* e, neste caso, a *definição persistente* é efetivada quando um comando de manipulação for executado com o comando COMMIT que valida a transação e estabelece o armazenamento em memória secundária. Consideraremos que existe uma definição por referência nos comandos INSERT, DELETE e UPDATE e uma definição por valor no comando COMMIT; a concatenação das duas definições estabelece a ocorrência da *definição persistente*. Na ausência do comando COMMIT, é exibida uma mensagem de *WARNING* (indicando sua ausência) e é adotada, como *default*, sua alocação no nó n_{out} (de saída) do grafo *G(UP)* da unidade em teste.

A *Figura 4* mostra o exemplo de um grafo da *UP(Cadcus)* do Módulo de Programa Mod_1, com as quatro operações de manipulação da **SQL**. A ocorrência de uma *definição persistente* é concretizada se e somente se o caminho percorrido contém um dos pares de nós <4,9> ou <6,9> ou <8,9>. Apesar do comando DELETE caracterizar uma indefinição da *tupla*, considera-se que esse comando ocasiona uma definição *com respeito a* (*c.r.a.*) ***t***, tendo em vista que ele altera o estado da *variável de tabela* como uma *variável persistente*.

O *uso* de uma variável ocorre sempre que existir recuperação do valor em posição de memória associada à variável [NTA84]. Rapps e Weyuker [RAP82] definem referência a uma variável *v* como sendo:

(i) c-uso (uso computacional) quando *v* afetar diretamente uma computação que está sendo realizada no programa ou permitir que o valor de *v*, definido anteriormente, seja observado (neste caso o uso está associado ao nó); e,

(ii) p-uso (uso predicativo) quando *v* afetar diretamente o fluxo de controle do programa (este uso está associado ao arco).

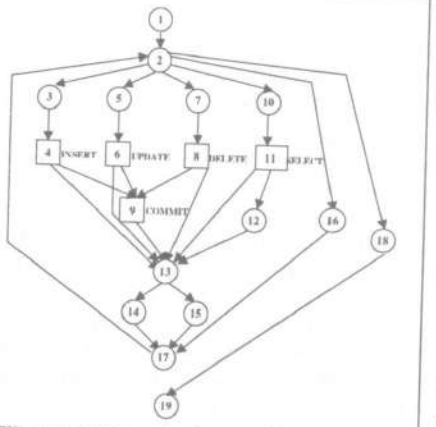

Figura 4: Exemplo de uma função com todos comandos de manipulação da SQL para uma tabela *t* (CUSTOMER).

Nos nós N_S da **SQL**, as *variáveis host* ***H*** e *variáveis de tabela* ***T*** podem estar tanto nos predicados das consultas (*queries*) quanto nas cláusulas dos comandos executáveis da **SQL**. Neste caso, uma referência a uma variável *v* foi definida como:

- *s-uso* (uso na **SQL**) quando *v* afetar o resultado de uma consulta ou de uma computação em um comando executável da **SQL** (uso associado ao nó da **SQL**).

Um *s-uso* ocorre nos nós de N_S e pode afetar o fluxo nos arcos de saída dos nós de N_S quando houver ações de desvios para controle de erros (como *SQLERROR*). Essas ações afetam somente as variáveis ***H*** e ***T***.

Em particular, o *uso* de uma *variável de tabela* ocorre somente no nó *j* tal que $j \in N_S$ e *j* contiver um dos comandos da **SQL** responsáveis pela manipulação da base de dados (SELECT, INSERT, UPDATE, DELETE). Como é comum a presença de comandos de tratamento de erros (que ocasionam desvios incondicionais a partir dos comandos executáveis da **SQL**) em ABDR, foi considerado que o *uso* da *variável de tabela* está nos arcos de saída dos nós onde ocorre um *s-uso [(j, k)]*, aplicando a idéia de *t-uso* (uso persistente de ***t***). No exemplo da *Figura 4*, está associada a ocorrência de *uso* da variável ***t*** aos arcos *(4,9), (4,13), (6,9), (6,13), (8,9), (8,13), (11,12) e (11,13)*, em vez de associarmos o *uso* da variável ***t*** aos nós 4, 6, 8 e 11. Essa estratégia somente é aplicada quando a variável ***t*** é tratada no escopo da definição de *variável persistente*.

Uma variável é *indefinida* quando não existir nenhum valor associado a ela naquele ponto do programa (Por exemplo: ***x*** *local* em procedimento **A** que chama procedimento **B**; em **B** a variável ***x*** é *indefinida*).

Rapps e Weyuker [RAP85] definem o grafo *def-uso* como uma extensão do grafo de controle, com a incorporação de informações semânticas do programa; a partir do grafo *def-uso* são definidos vários conceitos básicos utilizados na definição da Família de Critérios baseados em Fluxo de Dados (FCFD). Alguns desses conceitos são utilizados neste trabalho.

Um caminho $(i, n_1, n_2, ..., n_k, j)$, $k \geq 0$, que contém uma *definição* da variável *v* no nó *i* e que não contenha nenhuma redefinição de *v* nos nós $n_1, n_2, ...n_k$ é chamado de caminho *livre de definição c.r.a. v* do nó *i* ao arco (n_k, j). Essa definição é válida para todos os nós N^{BD}, todos os arcos *E* e todas as variáveis de uma ABDR. Como exemplo, na Figura 4 os caminhos

(4, 9, 13, 14, 17, 2, 10, 11, 12) e (4, 9, 13, 14, 17, 2, 3, 4, 9) são caminhos *livres de definição c.r.a. t*. Existindo vários outros caminhos livres de definição *c.r.a. t* nesse exemplo.

Um caminho $(n_1, n_2, ..., n_k)$ é um caminho *simples* se todos os nós contidos no caminho, exceto possivelmente o primeiro e o último, são distintos. No exemplo da *Figura 4*, o caminho (2, 3, 4, 9, 13, 14, 17, 2) é um caminho simples.

Um caminho $(n_1, n_2, .., n_k,)$ é um caminho *livre de laço*, se todos os nós pertencentes a ele são distintos. No exemplo da *Figura 4*, o caminho (3, 4, 9, 13, 14, 17, 2, 5, 6) é um caminho livre de laço.

Um caminho $(n_1, n_2,..., n_j, n_k)$ é um *du-caminho* c.r.a *v* se n_1 contiver 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 simples livre de definição c.r.a *v* e $n_1, n_2, ...,n_j$ é um caminho livre de laço.

Um nó $i \in N^{BD}$ possui uma definição *global* de uma variável *v* se ocorre uma definição de *v* no nó *i* e existe um caminho *livre de definição* de *i* para algum nó ou para algum arco que contém um *c-uso/s-uso* ou um *p*-uso, respectivamente, da variável *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*. No caso dos nós N_S da **SQL**, considera-se que todo *s-uso* é um *s-uso global*, devido às características dos comandos executáveis da **SQL**.

Maldonado [MAL91] define o grafo de fluxo de dados como *grafo-def*, tendo em vista que seus critérios não requerem o uso explícito das variáveis. A partir do conceito de *grafo-def*, Maldonado definiu os seguintes conjuntos: *defg(i)* = {variável ***x*** | ***x*** é definida no nó *i*}; *pdcu(x,i)* = {nós $j \in N$ | existe um caminho livre de definição *c.r.a.* ***x*** do nó *i* para o nó *j* e ***x*** $\in$ *defg(i)*}; *pdpu(x,i)* = {arcos *(j,k)* | existe um caminho livre de definição c.r.a ***x*** do nó *i* para o arco *(j,k)* e ***x*** $\in$ *defg(i)*}; *potencial-du-caminho c.r.a.* ***x*** é um caminho livre de definição $(n_1, n_2, ...,n_j, n_k)$ *c.r.a.* ***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 ***x***. Para completar as definições dadas em Maldonado [MAL91], ele definiu como conjunto factível: *fpdcu(x,i)=[j* $\in$ *pdcu(x,i)* | a potencial-associação *[i, j, x]* é executável}; e *fpdpu(x,i)* = *[(j, k)* $\in$ *pdpu(x,i)* | a potencial-associação *[i, (j, k), x]* é executável}. Uma associação é dita ser *executável* se existem dados de entrada que a exercitem.

No teste de um programa de ABDR, a etapa de teste de unidade pode ser realizada sob qualquer critério de teste estrutural (por exemplo, FCFD ou FCPU) com respeito às variáveis que são definidas na memória interna (caso das *variáveis de programa*, *variáveis hosts* e *variáveis de tabela de visão*).

Para considerar as *variáveis de tabela* como *variáveis persistentes*, definimos os seguintes conjuntos:

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

(ii) *defT* $<t,i>$ = {Variáveis ***t*** tal que cada ***t*** possui uma *definição persistente* pela *concatenação* da execução dos nós *t* e *i* da **SQL**, denotada por $<t, i>$ - no nó *t* $\in N_S$ existe um comando de manipulação (INSERT, UPDATE e DELETE) de ***t*** e no nó $i \in N_S$ existe um comando COMMIT - e os dois nós são sempre executados conjuntamente}.

(iii) *dsu(v,i)*={nós $j \in N_S$ tal que *v* $\in$ *s-uso(j)* e existe um caminho livre de definição *c.r.a. v* do nó *i* para o nó *j*}.

(iv) *fdsu(v, i)* = *[j* $\in$ *dsu(v,i)* tal que a associação *[i, j, v]* , $j \in N_S$, é executável};

(v) Associação *definição-s-uso* (*dsu*) é a tripla [*i, j, v*] onde *v* $\in$ *defg(i)* e *j* $\in$ *dsu(v,i)*, válida para as variáveis ***H*** e ***T***, onde *defg(i)* = {variável *v* tal que *v* é definida no nó *i*}.

No exemplo da *Figura 4*, existem várias associações *definição-s-uso* com relação às variáveis ***H*** e ***T***. A variável ***t*** é *definida* no nó 9 do grafo da *Figura 4*. Tratando-a como uma variável comum, a ferramenta POKE-TOOL gera vários elementos requeridos extraídos das *associações dsu* c.r.a ***t***, restringindo somente os elementos requeridos que contêm *os nós* N_S no uso, tendo em vista que as *variáveis de tabela* só podem ser *definidas* e *usadas* em nós N_S. Foram geradas as seguintes associações:

[9, (4,9), t]	[9, (4,13), t]	[9, (6,9), t]	[9,(6,13), t]
[9,(8,9), t]	[9,(8,13), t]	[9, (11,12), t]	[9, (11,13), t]

Tendo em vista que a *definição persistente* de uma *variável de tabela* ocorre com a concatenação de execução do par de nós $< \ell, i > \in N_S$ e que a definição só é concretizada quando é executado o comando COMMIT supostamente localizado no nó *i*, são dadas duas definições para considerar a *variável de tabela* como *variável persistente*.

Definição-*2 Associação definição-t-uso* é uma tripla $[< \ell,i>, (j, k), t]$, onde: $t \in defT(\ell, i)$; $j \in dsu(t, i)$; o arco $(j, k) \in Arc_{out}(j)$ (*arco de saída de j*); os nós ℓ, *i* e $j \in N_S$; $< \ell,i>$ é a concatenação que estabelece uma definição *persistente* de ***t*** no nó *i* vinda do nó ℓ que alcança um uso de ***t*** no arco *(j,k)*; e existe um caminho *livre de definição* c.r.a ***t*** de *i* ao arco *(j, k)*.

Examinando o exemplo da *Figura 4*, pode-se gerar as seguintes *associações definição-t-uso* (denotada por *associações persistentes*):

[<4,9>, (4,9), t]	[<4,9>, (4,13), t]	[<4,9>, (6,9), t]	[<4,9>, (6,13), t]
[<4,9>, (8,9), t]	[<4,9>, (8,13), t]	[<4,9>, (11,12), t]	[<4,9>, (11,13), t]
[<6,9>, (4,9), t]	[<6,9>, (6,13), t]	[<6,9>, (6,9), t]	[<6,9>, (6,13), t]
[<6,9>, (8,9), t]	[<6,9>, (8,13), t]	[<6,9>, (11,12), t]	[<6,9>, (11,13), t]
[<8,9>, (4,9), t]	[<8,9>, (4,13), t]	[<8,9>, (6,9), t]	[<8,9>, (6,13), t]
[<8,9>, (8,9), t]	[<8,9>, (8,13), t]	[<8,9>, (11,12), t]	[<8,9>, (11,13), t]

Definição-*3* Um *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 $(n_\ell, ..., n_i, ..., n_j, n_k)$ é um caminho livre de laço e nos nós n_ℓ e n_i ocorre uma *definição persistente* de ***t***.

O nó n_ℓ possui um comando INSERT, DELETE ou UPDATE, que mais proximamente antecede o comando COMMIT no *dtu-caminho c.r.a.* ***t***. No grafo da Figura 4 existem vários *dtu-caminhos*, por exemplo: (**4, 9**, 13, 14, 17, 2, 5, **6, 13**), (**4, 9**, 13, 15, 17, 2, 5, **6, 13**), etc.

O maior efeito da associação estabelecida na *Definição-2* é a sua utilização no teste de integração, dando um tratamento exclusivo às variáveis *persistentes* (*variáveis de tabela*). O teste de integração *intra-modular* foi dividido em duas etapas: a primeira relacionada ao grafo de chamada entre as *UPs*; e a segunda relacionada à integração baseada na dependência de dados entre as variáveis *persistentes*, representadas pelas *variáveis de tabela*. Na etapa do teste de integração *inter-modular* foi aplicada apenas a integração baseada nas dependências de dados ocasionadas pelas variáveis *persistentes* [SPO99].

3 Etapas de Teste

Este trabalho aborda a técnica de teste "*caixa branca*", a qual utiliza-se de informações da estrutura interna do programa (fluxo de controle e/ou estrutura de dados) com o intuito de caracterizar um conjunto de componentes elementares de uma aplicação que devam ser exercitados pelos dados de teste. Sob o aspecto da seleção dos dados de teste, os critérios de

teste a serem satisfeitos requerem um conjunto de dados que exercitem todos os componentes elementares associados (os elementos requeridos de cada critério de teste [NTA84]). Do ponto de vista da adequação dos dados de teste, o conjunto é considerado adequado se todos os elementos requeridos são exercitados. Nosso estudo de fluxo de dados para programas de ABDR considera dois modelos: *o modelo intra-modular* e *o modelo inter-modular*.

3.1 Modelo de Fluxo de Dados Intra-Modular

O Modelo de Fluxo de Dados Intra-Modular é aplicado nas etapas de teste de cada *Módulo de Programa* da ABDR, iniciando na etapa do *Teste de Unidades* e estendendo-se para a etapa de *Teste de Integração* entre as *UPs* do Módulo de Programa. Cada Unidade de Programa pertencente ao Módulo de Programa é testado isoladamente observando os fluxos de dados das variáveis utilizadas no programa. A Figura 5 ilustra os passos de execução do teste de unidade do procedimento *Inssal* do Módulo de Programa Mod_3, mostrando os arquivos produzidos pela POKE-TOOL e o grafo gerado pela VIEWGRAPH. No caso de programas de ABDR com **SQL** embutida, o programa fonte "modulo*n*.pc" é instrumentado gerando o programa "mod*n*.pc"; em seguida, o programa é pré-compilado e compilado gerando o programa executável "mod*n**" para o qual são gerados os casos de teste.

A outra etapa de teste em que se aplica o modelo *intra-modular* é a do teste de

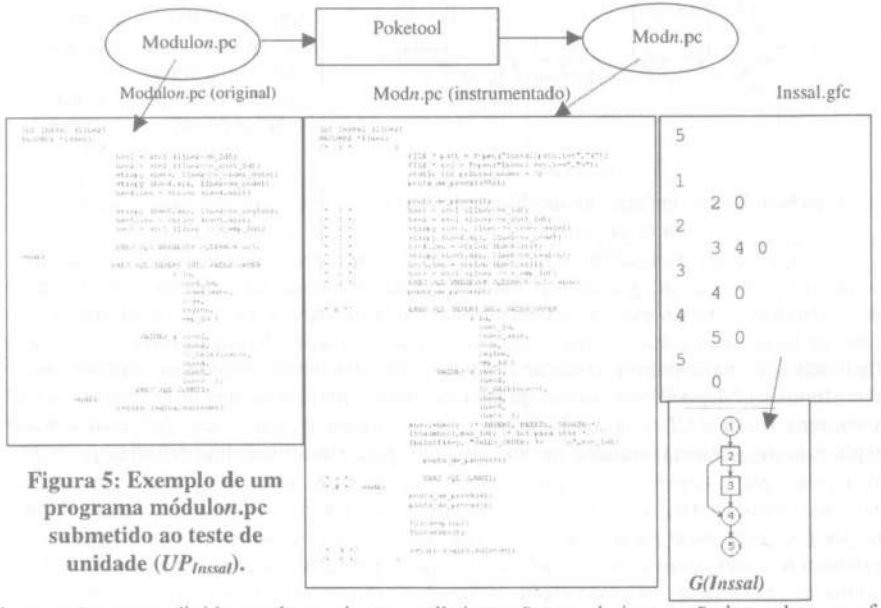

Figura 5: Exemplo de um programa módulo*n*.pc submetido ao teste de unidade (UP_{Inssal}).

integração que se divide em duas sub-etapas distintas: *i*) teste de integração baseado no grafo de chamadas; e, *ii*) teste de integração baseado na dependência de dados persistentes.

O teste de integração baseado na dependência de dados persistentes tem como objetivo exercitar o fluxo de dados das *variáveis de tabela* entre as Unidades de Programa de um mesmo Módulo de Programa. Neste caso, quando existir pelo menos um dos comandos INSERT, DELETE ou UPDATE em uma *UP*, dizemos que esta *UP* contém uma *definição persistente* da variável ***t***; para as *UPs* que tiverem pelo menos um dos comandos INSERT, DELETE, UPDATE ou SELECT, dizemos que esta *UP* contém um uso da variável ***t***.

3.2 Modelo de Fluxo de Dados Inter-Modular

Uma aplicação é composta por vários módulos de programas que manipulam e consultam dados existentes nas tabelas da base de dados do sistema. As operações especificadas nos Módulos da ABDR estabelecem fluxos de dados das tabelas para os Módulos e dos Módulos para as tabelas que, indiretamente, estabelecem fluxos de dados entre as diferentes tabelas da aplicação. Dizemos que ocorre fluxo de dados *inter-modular* quando um dado armazenado em uma tabela t_i por um módulo Mod_k é usado por outro módulo Mod_j, em outro tempo de execução, podendo esse dado ser utilizado para interferir no armazenamento de dados de outras tabelas; a mesma coisa pode ocorrer com os novos dados armazenados e, assim sucessivamente, gerando um fluxo de dados entre os Módulos através das variáveis persistentes (*variáveis de tabela*).

O fluxo de dados é representado por uma seta que vai do módulo para a tabela, quando caracterizar uma *definição persistente* da tabela pelo módulo, e da tabela para o módulo, quando caracterizar um *uso* (*persistente*) da tabela pelo módulo. Pode-se dizer que o fluxo de dados é baseado nas variáveis *persistentes*.

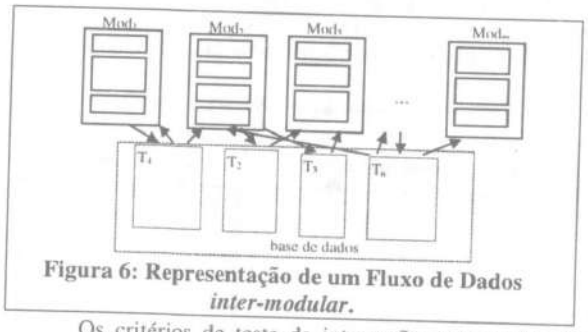

Figura 6: Representação de um Fluxo de Dados *inter-modular*.

Representamos um grafo de fluxo de dados *inter-modular* como um conjunto de nós com traços contínuos representando os módulos da aplicação de Banco de Dados, um conjunto de nós com traços pontilhados representando as tabelas da base de dados, e os arcos direcionados indicando os fluxos de dados entre os nós (vide Figura 6).

Os critérios de teste de integração para o teste *intra-modular* e para o teste *inter-modular* (critérios de integração inter-unidades) determinam que seus elementos requeridos só são satisfeitos se forem exercitados pela mesma tupla da variável ***t*** envolvida na execução do caso de teste. Considera-se, neste caso, que uma associação ***definição-t-uso*** inter-unidade (aplicada tanto no teste *intra-modular* como no teste *inter-modular*) pode ser caracterizada de duas formas: *a) dependência direta:* quando a *definição persistente* de ***t*** ocasionada pela tupla τ em uma unidade UP_i é utilizada por um uso *persistente* de ***t*** por outra UP_j com a mesma tupla τ; *b) dependência múltipla*: quando uma UP_i, para estabelecer uma definição *persistente* de ***t*** pela tupla τ, depende da *definição persistente* de ***t'*** por outra tupla τ' em uma segunda UP_k (tal como ocorre quando a *chave primária* da tabela ***t'*** é uma chave estrangeira de outra tabela ***t*** e, para gerar uma tupla em ***t***, necessita-se da criação prévia em ***t'***) para poder estabelecer o uso *persistente* de ***t*** pela mesma tupla τ em uma terceira UP_j. Quando todas as Unidades envolvidas na associação *definição-t-uso* pertencem ao mesmo Módulo de Programa, a associação é tratada pelo modelo *intra-modular*; caso contrário, é tratada pelo modelo *inter-modular*.

3.3 Estratégias

Uma estratégia de teste de Software pode consistir na aplicação sistemática de diferentes critérios, de natureza distinta, e que revelam classes diversas de defeitos no programa. No processo de teste de um sistema caracterizam-se três níveis de teste: o teste de unidade, o teste de integração e o teste de sistema. O teste de unidade tem como função detectar defeitos na menor unidade do Software, aqui representada como a Unidade do

Programa *UP*. O teste de integração objetiva testar as relações e interfaces entre as *UPs*, sendo conduzido após o teste de unidade para todas *UPs* do mesmo Módulo de Programa envolvido na integração. O teste de sistema, realizado após o teste de integração, visa a identificar erros de função e/ou características de desempenho [MAL91]. Entretanto, para ABDRs, além dos critérios de teste de Unidade e de teste de Integração da literatura, são propostos critérios de teste de integração baseados nas variáveis *persistentes* (aqui representadas pelas *variáveis de tabela*), que estabeleceram a realização de novos testes para a integração dos Módulos de Programas em uma ABDR [SPO99]. A seguir apresentamos os critérios para o teste de Unidade, adaptados para aplicação em programas de ABDR.

4 Definição dos Critérios de Teste para Programas de ABDR

Seja *G(UP)* o grafo de fluxo de controle de uma Unidade de Programa de um Módulo de Programa da ABDR, Π um conjunto de caminhos completos de *G(UP)*, e Γ um conjunto de tuplas τ usadas para exercitar as *associações definição-t-uso*. Os critérios de teste são classificados como ***Intra-modulares*** e ***Inter-modulares***.

4.1 Critérios Intra-modulares

São critérios aplicados durante o teste de Módulo de Programa, desde o teste de unidade até o teste de integração das unidades, abrangendo todo o Módulo de Programa.

4.1.1 Critérios de teste intra-unidade (teste de unidade)

O teste de unidade visa a testar cada Unidade de Programa do Módulo de Programa. Vários critérios baseados em fluxo de dados poderiam ser utilizados para considerar as variáveis definidas em memória interna para programas de ABDR. Neste trabalho, os critérios da FCPU, suportados pela POKE-TOOL, constituem a base dos critérios propostos para o teste de unidade. Os critérios a seguir são os critérios da FCPU adaptados para programas de ABDR.

- **Todos-Nós**: Π satisfaz o critério (*todos-nós*) se Π incluir todos os nós executáveis $i \in N^{BD}$, de *G(UP)*.
- **Todos-Arcos**: Π satisfaz o critério (*todos-arcos*) se Π incluir todos os arcos executáveis $(i, j) \in G(UP)$.
- **Todos-Potenciais-Usos**: Π satisfaz o critério, (*todos-potenciais-usos*) se, para todo nó $i \in G(UP) \mid defg(i) \neq \emptyset$ e para toda variável $v \mid v \in defg(i)$, Π incluir todas potenciais associações $[i, j, v] \mid j \in fpdcu(v,i)$ executáveis, todas as potenciais associações $[i,(j,k),v] \mid (j,k) \in fpdpu(v,i)$ executáveis e todas as potenciais associações $[i, j, v] \mid j \in fdsu(v,i)$ executáveis.
- **Todos-Potenciais-Usos/DU**: Π satisfaz o critério, (*todos-potenciais-usos/du*) se, para todo nó $i \in G(UP) \mid defg(i) \neq \emptyset$ e para toda variável $v \mid v \in defg(i)$, Π incluir um potencial-du-caminho executável para todas as potenciais associações $[i, j, v] \mid j \in fpdcu(v,i)$, um potencial-du-caminho executável para todas as potenciais associações $[i,(j,k),v] \mid (j,k) \in fpdpu(v,i)$ e um potencial-du-caminho executável para todas as potenciais associações $[i,j,v] \mid j \in fdsu(v,i)$.
- **Todos-Potenciais-DU-Caminhos**: Π satisfaz o critério, (*todos-potenciais-du-caminhos*) se, para todo nó $i \in G(UP) \mid defg(i) \neq \emptyset$, Π incluir todos os potenciais-du-caminhos executáveis de i para j, c.r.a cada variável $v \in defg(i)$ para todas as potenciais associações executáveis $[i,j,v] \mid j \in fpdcu(v,i)$, todos os potenciais-du-caminhos executáveis de i para (j,k) c.r.a cada variável $v \in defg(i)$ para todas as potenciais associações executáveis $[i,(j,k),v] \mid (j,k) \in fpdpu(v,i)$ e todos os

potenciais-du-caminhos executáveis de i para j, c.r.a $v \in defg(i)$ para todas as potenciais associações executáveis $[i,j,v] \mid j \in fdsu(v,i)$.

Os critérios a seguir consideram as *variáveis persistentes* de programas de ABDR.

- **Todos-t-Usos:** Π e Γ satisfazem o critério (*todos-t-usos*) se, para todo par $<\ell, i>$ de nós $\in G(UP)$ tal que $defT(\ell,i) \neq \emptyset$ para toda variável t onde $t \in defT(\ell,i)$, Π incluir todas associações $[<\ell,i>, (j, k), t]$ executáveis e existe um caminho livre de definição persistente de $<\ell,i>$ até (j, k), c.r.a. t. A associação será satisfeita se e somente se for exercitada com a mesma *tupla* $\tau \in \Gamma$.
- **Todos dtu-caminhos:** Π e Γ satisfazem o critério (*todos-dtu-caminhos*) se, para todo par $<\ell, i>$ de nós $\in G(UP)$ tal que $defT(\ell,i) \neq \emptyset$, Π incluir todos *dtu-caminhos* executáveis de $<\ell,i>$ até (j, k), *c. r. a.* t onde $t \in defT(\ell,i)$ para todas as associações executáveis $[<\ell,i>, (j, k), t]$ e existe um caminho livre de *definição persistente* de $<\ell,i>$ até (j, k), *c.r.a.* t. Cada associação será satisfeita se e somente se for exercitada com a mesma *tupla* $\tau \in \Gamma$.

Os dois critérios acima foram criados para considerar as *variáveis de tabela* tratadas como *variáveis persistentes*. Os elementos requeridos pelos critérios *todos-t-usos* e *todos-dtu-caminhos* só são satisfeitos se forem executados para a mesma tupla. A exigência de mesma tupla é fundamental para o teste de *variáveis persistentes*.

4.1.2 Critérios de teste de integração inter-unidades

Os critérios de integração inter-unidades são classificados em: critérios baseados no grafo de chamadas (a integração das unidades é realizada a partir do grafo de chamada das unidades envolvidas) [CLA89, VIL98] (Figura 7); e critérios baseados nas dependências de dados *persistentes* (Figura 8) . Neste último caso, o fluxo de dados das variáveis de tabela não é definido pelos pontos de chamada entre as unidades. São definidos sete critérios específicos para ABDR, quatro critérios para o teste de integração baseado no grafo de chamadas e três critérios para o teste de integração baseado na dependência de dados definida por variáveis persistentes [SPO99].

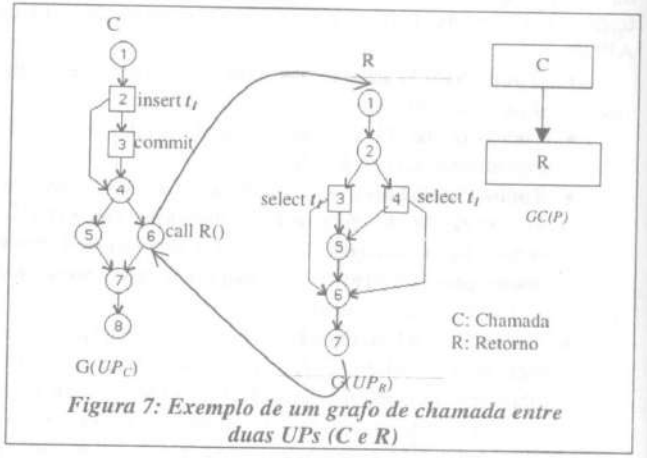

Figura 7: Exemplo de um grafo de chamada entre duas UPs (C e R)

A Figura 7 mostra as unidades UP_C (unidade de chamada) e UP_R (unidade de retorno) pertencentes ao mesmo Módulo de Programa. Os seguintes critérios estão baseados no grafo de chamada: *a)* ***Todas as associações-definição-t-uso interprocedimentais de chamada***; *b)* ***Todas as associações-definição-t-uso interprocedimentais de retorno***; *c)* ***Todos os dtu-caminho interprocedimentais de chamada***; e *d)* ***Todos os dtu-caminho interprocedimentais de retorno***.

A Figura 8 mostra as unidades UP_A e UP_B pertencentes ao Módulo Mod_3 e sua integração com respeito à dependência de dados estabelecida pela variável DEPARTMENT.

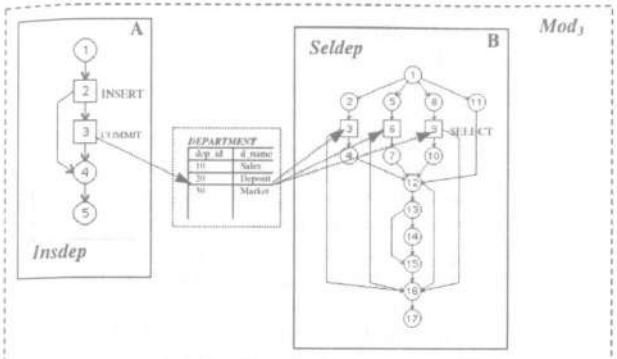

Figura 8: Grafo de dependência Interna de dados c.r.a DEPARTMENT. $GD_{PA}(Insdep_{Mod3} \times Seldep_{Mod3})_{DEPARTMENT}$

Em UP_A existe um comando que faz uma definição persistente *c.r.a. DEPARTMENT* e em UP_B existe um uso desta mesma variável, estabelecendo uma *associação definição-t-uso* persistente *c.r.a. DEPARTMENT*.

Os seguintes critérios estão baseados na dependência de dados: *a)* ***todos-t-usos-ciclo1-intra***; *b)* ***todos-dtu-caminhos-intra***; e *c)* ***todos-t-usos-ciclo2-intra***. ***Critérios de teste de integração inter-modular***

Os critérios para o teste de integração *inter-modular* são idênticos aos critérios de teste de integração baseados na dependência de dados *intra-modular*, distinguindo-se destes apenas pela exigência de que as Unidades associadas devem pertencer a Módulos de Programas distintos. Esses critérios foram elaborados exclusivamente para exercitar as associações definição-uso c.r.a t estabelecidas pelos comandos de **SQL** em programas de uma ABDR. Os elementos requeridos pelos critérios são satisfeitos se forem exercitados com as mesmas tuplas (observadas pelas chaves primárias em cada execução).

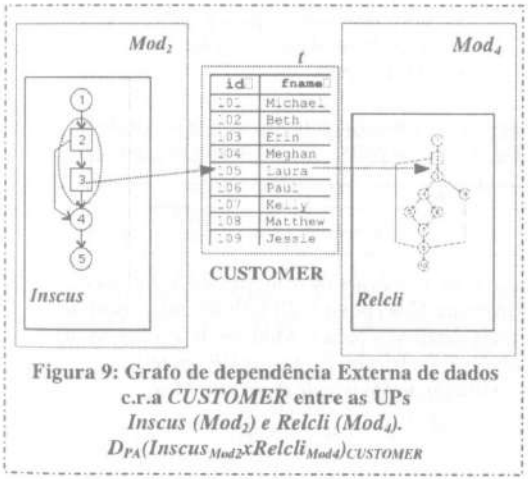

Figura 9: Grafo de dependência Externa de dados c.r.a *CUSTOMER* entre as UPs *Inscus* (Mod_2) e *Relcli* (Mod_4). $D_{PA}(Inscus_{Mod2} x Relcli_{Mod4})_{CUSTOMER}$

A *Figura 9* mostra duas Unidades de Programa e a sua integração relativa à dependência de dados c.r.a variável *persistente* **CUSTOMER**. As unidades envolvidas pertencem a Módulos distintos (Mod_2 e Mod_4) e, por isso, são tratadas pelos critérios de *integração inter-modular*. Os seguintes critérios estão definidos para o de teste de integração *inter-modular*: *a)* ***Todos-t-usos-ciclo1-inter***; *b)* ***Todos-dtu-caminhos-inter***; e *c)* ***Todos-t-usos-ciclo2-inter***.

Os critérios de integração apresentados nas Sub-seções 4.1.2 e 4.2 estão sendo analisados com relação à sua complexidade; os resultados dessa análise e uma discussão mais aprofundada serão apresentados em trabalhos futuros.

5 Resultados Obtidos

Nesta seção são apresentados os resultados obtidos no experimento de aplicação dos critérios de teste de Unidade em 4 Módulos de Programa escritos em linguagem C com **SQL** embutida. Foi utilizado o sistema de Banco de Dados da Oracle para ambiente Solaris (SUN) e o pré-compilador para linguagem C (ProC). O experimento de teste inicia-se com a instrumentação do programa-fonte <modulo_prog>.pc, que representa um Módulo de Programa da ABDR. O programa instrumentado é pré-compilado e compilado gerando um programa executável. Com o executável, inicia-se o teste para cada unidade isoladamente através da geração e execução de vários casos de testes acompanhados pela ferramenta VIEWGRAPH que possibilita ao testador um controle mais seguro das tarefas de teste. A Figura 10 mostra a interface da VIEWGRAPH v.2.0 (aplicada no teste da unidade *delsal* do módulo Mod_2).

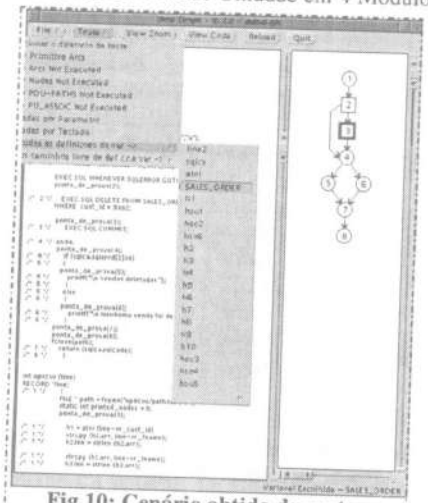

Fig.10: Cenário obtido durante o teste de unidade. Interface da ferramenta VIEWGRAPH v.2.0

A Tabela 1 apresenta o número de casos de teste necessários para satisfazer os critérios do teste de unidade para cada Módulo utilizado no experimento.

Tabela1: Quadro geral dos Módulos de Programas Executados no teste de Unidade.

Módulo de Programa	Total de Casos de testes	N° de UPs em teste
Mod_1	48	02
Mod_2	168	07
Mod_3	98	20
Mod_4	56	06

O Módulo de Programa Mod_2 precisou de 168 casos de teste devido à presença de laço na *UP main*(). Apesar de o Módulo de Programa Mod_3 possuir 20 *UPs* em teste, nenhuma *UP* possui *laços*. Para mostrar o resultado mais detalhado para o Módulo de Programa Mod_2, é apresentada a Tabela 2, que separa o resultado da cobertura para cada UP em teste.

***Tabela2*: Teste de Unidade realizado no *Módulo de Programa* Mod_2.**

Funções	Número de Casos de Teste	Critérios Potenciais Usos (*E r/ Ne*) Unidade					
		Nós %	Arcos %	PU %	PDU %	PUDU %	TtU %
Main	168	36/1	19/1	264/29	418/91	264/41	-
		97,22	94,74	89,23	78,23	84,47	-
Inscus	4	5/0	3/0	4/0	4/0	4/0	-
		100,00	100,00	100,00	100,00	100,00	-
Delsal	4	8/0	4/1	8/2	8/4	8/2	-
		100,00	75,00	75,00	50,00	75,00	-
Updcus	4	5/0	3/0	4/0	4/0	4/0	-
		100,00	100,00	100,00	100,00	100,00	-
Delcus	2	5/0	3/0	4/0	4/0	4/0	-
		100,00	100,00	100,00	100,00	100,00	-
Selcus	22	21/0	13/0	57/19	49/12	57/12	-
		100,00	100,00	66,67	75,51	78,95	-

Na Tabela 2, a coluna dos critérios possui dois resultados para cada UP colocada na coluna à esquerda: a parte superior referente a cada resultado contém os valores *Er/Ne*, onde *Er* é o número total de elementos requeridos de cada Critério de teste e *Ne* é o número de elementos requeridos não executáveis (não existem casos de testes que executam tais elementos); a parte inferior contém a porcentagem de cobertura obtida para cada critério de teste.

O critério *todos-t-usos* não teve nenhum elemento requerido no teste de unidade por não existirem funções com laços que associam comandos de manipulação da **SQL** em nenhuma *UP* do módulo Mod_2. No caso do módulo Mod_1 a unidade *UP*(*Cadcus*), foi possível observar que satisfazer o critério *todos-t-usos* implicou satisfazer o critério *todos-potenciais-usos* quanto à parte do grafo que envolve os nós dos comandos da **SQL**; contudo, os casos de testes utilizados para satisfazer os critérios da FCPU não cobriram os elementos requeridos pelo critério *todos-t-usos*, devido à exigência da mesma *tupla* para satisfazer a *associação definição-t-uso*. Outra observação obtida na *UP*(*Cadcus*) do Mod_1 é que os critérios que exercitam as variáveis de tabela (como variáveis persistentes) demonstraram ser mais eficazes na detecção de erros relacionados com as "*queries*" usadas nos comandos da **SQL**.

Como sugestão, sempre que existirem elementos requeridos pelo critério *todos-t-usos* no teste de unidade pode-se iniciar o teste com esse critério. Depois de executados todos os casos de testes que satisfazem o critério *todos-t-usos*, determinam-se os elementos requeridos pelos demais critérios ainda não satisfeitos e continua-se o teste para exercitar elementos requeridos ainda não cobertos. O uso da mesma *tupla* para satisfazer as associações *def-uso* aumenta a probabilidade de o testador detectar defeitos relacionados aos comandos da **SQL**, melhorando a eficácia do teste.

6 Conclusões

A abordagem aqui apresentada propõe o teste de *variáveis persistentes* e o teste baseado em dependência de dados para o teste de aplicações de banco de dados relacionais.

Modificações efetuadas nas ferramentas POKE-TOOL e VIEWGRAPH possibilitaram a realização do teste de Unidade em programas de ABDR com **SQL** embutida, aproveitando os critérios da FCPU.

Comprovou-se a importância de critérios que incluam as *variáveis de tabela* de forma a requerer o uso de cada variável definida, além da exigência de se utilizar a mesma *tupla* para satisfazer uma associação. Tais critérios demonstraram ser mais eficazes, para a detecção de classes de erros relacionados aos atributos das variáveis de tabela, do que os critérios que exigiam simplesmente as associações sem requerer a mesma tupla. Assim, esses critérios são complementares aos critérios baseados em fluxo de dados (FCPU e FCFD), para ABDR.

Para cada caso de teste foi preciso acompanhar o valor inicial de cada variável de tabela envolvida, sem perder de vista os resultados obtidos e esperados. Como trabalho futuro, pretende-se incorporar recursos para ações interativas na base de dados diretamente na ferramenta para um melhor controle do estado inicial das tabelas. Pretende-se também tornar automática a instrumentação de programas com **SQL**, incluindo as informações das *tuplas* utilizadas para a cobertura dos critérios que envolvem *associações definição-t-uso* (*persistentes*).

Agradecimentos: Aos Órgãos de Fomento à Pesquisa: Capes, CNPq e Fundação Araucária.

7 Bibliografia

[ARA00] Aranha, C. L. F. M, Mendes, N. C., Jino, M. e Toledo, C. M. T. "RDBTool: Uma Ferramenta de Apoio ao Teste de Bases de Dados Relacionais", XI CITS, Curitiba, PR, Brasil, Junho, 2000, pp. 31-43.

[BUT93] Butler, B. and Canter, S., "Testes de Performance: Bancos de Dados SQL", PC Magazine Labs, Dezembro de 1993, pp.5-9.

[CHA97] Chaim, M.L., Madonado, J. C. e Jino, M., "Ferramentas para Teste Estrutural de Software baseado em Análise de Fluxo de Dados: o caso POKE-TOOL", Workshop do Projeto Validação e Teste de Sistemas de Operação, Águas de Lindóia , SP, Jan. 1997.

[CLA89] Clarke, L. A. et al., "A Formal Evaluation of Data Flow Path Selection Criteria", IEEE TSE, 15(11), November 1989, pp. 1318-1332.

[CRU99] Cruzes, D., *Geração e Visualização de Informações para Suporte à Depuração e Teste de Programas*, Dissertação de Mestrado, DCA/FEEC/ÚNICAMP, Campinas, SP, Brasil, Julho de 1999.

[ELM94] Elmasri, R. and Navathe, S. B., *Fundamentals of Database Systems* , 2nd Edition, Addison Wesley, 1994.

[FRA85] Frankl, F. G. and Weyuker, E. J., "Selection of Software Test Data Using Data Flow Information," IEEE TSE, Vol. 11, April, 1985, pp. 367-375.

[FRA88] Frankl, F. G. and Weyuker, E. J., "An Applicable Family of Data Flow Testing Criteria", IEEE TSE, Vol. 14, No. 10, October, 1988, pp. 1483-1498.

[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, December, 1994, pp. 154-163.

[HER76] Herman, P. M., "A Data Flow Analysis Approach to Program Testing", Australian Computer Journal, Vol.8, N. 3, November, 1976.

[HUA75] Huang, J. C., "An Approach to Program Testing", Computing Surveys, Vol. 7, No. 3, September, 1975, pp. 113-128.

[LAS83] Laski, J. W. and Korel, B., "A Data Flow Oriented Program Testing Strategy", IEEE TSE, Vol. SE- 9, No. 3, May, 1983, pp. 347-354.

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

[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.

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

[NTA84] Ntafos, S. C., "On Required Element Testing", IEEE TSE, Vol. SE-10, Nov., 1984, pp. 795-803.

[PRE97] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 4th Ed. - New York, 1997.

[RAP82] Rapps, S. and Weyuker, E.J., "Data Flow Analysis Techniques for Test Data Selection", *in Internat. Conf. on Soft. Eng.*, pp. 272-278, Tokio, Japan, Sept., 1982.

[RAP85] Rapps, S.and Weyuker, E.J., "Selection of Software Test Data Using Data Flow Information", IEEE TSE, SE-11(4), April, 1985.

[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, Janeiro de 1997.

[SPO99] Spoto, E. S., "Experimento de Teste Estrutural em Programas de Aplicação de Banco de Dados Relacional", Relatório Técnico, DCA/FEEC/UNICAMP, Campinas, SP, 1999.

[VIL97] Vilela, P. R. S.; Maldonado, J. C. and Jino, M., "Program Graph Visualization", Software-Practice and Experience, 27 (11), Nov. 1997.

[VIL98] Vilela, P. R. S., *Critérios Potenciais Usos de Integração: Definição e Análise* , Tese de doutorado, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, Abril de 1998.