Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Uma Ferramenta para Teste Estrutural de Regras Ativas

Virgínia Mara Cardoso
Orientador: Prof. Dr. Mario Jino

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.

Banca Examinadora: Prof. Dr. Mario Jino
(FEEC/UNICAMP)

Dr. Marcos Lordello Chaim
(EMBRAPA, Campinas - SP)

Prof. Dr.Ivan Luiz Marques Ricarte
(FEEC/UNICAMP)

Dissertação de Mestrado
Campinas – SP – Brasil
Fevereiro – 2004

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

C179f

Cardoso, Virgínia Mara
Uma ferramenta para teste estrutural de regras ativas / Virgínia Mara Cardoso. --Campinas, SP: [s.n.], 2004.

Orientador: Mario Jino.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Banco de Dado relacionais. 2. Engenharia de Software. 3. SQL (Linguagem de programação de Computador). 4. ORACLE (Sistema de Computador). 5. Teste de Software. I. Jino, Mario. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Regras ativas definem ações sobre um banco de dados ativo, acionadas sem a intervenção do usuário; são utilizadas para a manutenção de bancos de dados bem como para controlar diversas outras atividades. Existem poucos trabalhos enfocando o teste de regras e os que o fazem não exploram a linguagem SQL, uma das linguagens mais utilizadas em bancos de dados relacionais. Propõe-se neste trabalho uma ferramenta de suporte ao teste de unidade de regras ativas – a ferramenta ART-TOOL (*Active Rule Testing Tool*). Técnicas de teste estrutural de software foram adaptadas para apoiar a aplicação de critérios baseados em análise de fluxo de dados ao teste de regras ativas. O suporte da ART-TOOL dá-se por meio de análises estática e dinâmica de regras ativas envolvendo: determinação dos elementos requeridos pelos critérios; a instrumentação do código fonte da regra; a monitoração da execução da regra instrumentada; e a análise de cobertura dos critérios. Um exemplo mostra a aplicação da ferramenta no teste de uma regra ativa. Mostra-se também, no teste de 15 regras ativas, que a abordagem contribui para a revelação de defeitos.

Agradecimentos

Em primeiro lugar agradeço ao meu Deus de amor, que me iluminou, abençoou e proporcionou tudo para a realização deste trabalho.

Ao meu orientador Prof. Dr. Mario Jino pela oportunidade para desenvolver este trabalho, pelo profissionalismo e pela orientação segura. Obrigado pela amizade e dedicação.

Em especial a meus pais pelo grande incentivo, carinho e compreensão. As minhas irmãs, Carmem e Giselle pela amizade e carinho. Ao meu irmão Paulo que sempre me incentivou e nunca mediu esforços para me acompanhar. A minha cunhada e sobrinhos pelo apoio. Aos meus tios e primos que acreditaram em mim, em especial a minha prima Eliana pela amizade e ajuda na conclusão deste trabalho.

Agradeço a todos os membros do grupo de teste; Paulinho, pelo incentivo e pela amizade sincera; Plínio, pelo apoio e prestatividade constante durante o desenvolvimento deste trabalho. As minhas grandes amigas, Andréia e Adriana, pelo incentivo e companheirismo.

Ao pessoal do LCA e dos outros laboratórios da FEEC pela amizade, apoio e companhia diária. Em especial ao Cecílio, Ivana, Naur, Bicho, Cláudio e Vinícius pela amizade e contribuições; Marina, pela pessoa prestativa e querida; Nicola, pela grande amizade, carinho e pelo apoio nos momentos difíceis.

Aos membros da banca, Dr. Marcos Chaim, Prof. Dr. Ivan Ricarte e Prof. Dr. Léo Pini agradeço pela participação e contribuição.

Ao órgão de apoio à pesquisa, CNPQ, pelo apoio financeiro.

A todos amigos, que embora não estejam citados, estiveram ao meu lado durante esta caminhada, muito obrigada.

"Foi pela sabedoria que o Senhor criou a terra,
foi com inteligência que ele formou os céus.
Foi pela ciência que se fenderam os abismos,
por ela as nuvens destilam o orvalho."
(Prov 3, 19-20)

A Deus, a Virgem Maria e
aos meus pais

Índice

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

INTRODUÇÃO

1.1 Contexto

Teste de software é uma das atividades de garantia de qualidade de software e constitui um dos elementos para aprimorar a produtividade e a confiabilidade do software (Maldonado, 1991).

A atividade de teste é composta de: planejamento do teste; projeto de casos de teste; execução do teste e avaliação dos resultados. Para testar o software, basicamente utilizam-se técnicas funcionais e estruturais. A técnica funcional, também chamada de teste caixa-preta, por não considerar os detalhes da implementação, baseia-se na especificação para derivar os requisitos de teste. Já a técnica estrutural baseia-se em informações da implementação, do texto do programa, e por isso recebe o nome de teste caixa-branca. Outra técnica de teste – teste baseado em defeitos (DeMillo, 1978) - estabelece os requisitos de teste explorando os defeitos mais comuns introduzidos pelos desenvolvedores de software. Além dessas existe a técnica de teste baseada em máquinas de estados finitos (Maldonado e Fabbri, 2001) que utiliza a estrutura de máquinas de estado finito e o conhecimento subjacente para derivar requisitos de teste.

Como no caso geral não é possível testar um software por meio de teste exaustivo, critérios de teste devem ser estabelecidos para auxiliar a seleção de subconjuntos do conjunto total de casos de teste. Os primeiros critérios estruturais propostos foram os baseados no fluxo de controle dos programas; utilizam informações relacionadas ao controle de execução do programa para definir os elementos a serem exercitados.

Posteriormente surgiram critérios baseados na análise do fluxo de dados do programa, que requerem o exercício de associações estabelecidas pela ocorrência das variáveis.

Em decorrência do crescimento do tamanho e da complexidade de sistemas de software tornou-se necessário o uso de ferramentas automáticas para auxiliar no teste e na análise dos resultados do teste. Existem muitas ferramentas automatizadas que auxiliam todo ou parte do processo de teste. Por exemplo, a ferramenta POKE-TOOL (Chaim, 1991) permite a análise estática a partir do código fonte, auxilia na instrumentação e possibilita a avaliação do teste, fornecendo a cobertura de acordo com o critério aplicado.

As várias técnicas de teste são aplicadas a diversos tipos de programas, incluindo bancos de dados e suas aplicações. Há propostas de teste envolvendo diversos aspectos de bancos de dados; propostas que exploram o teste de programas de aplicação de banco de dados (Chays *et al.*, 2000 ; Spoto, 2000) são nosso foco de interesse.

Não são somente programas de aplicação que permitem a manipulação de uma base de dados. Regras ativas ou *triggers* são recursos de banco de dados ativos que realizam tarefas automaticamente sem a intervenção do usuário. Tais regras são responsáveis pelas tarefas executadas em resposta a eventos; podem ser qualquer seqüência de operações executadas sobre o banco de dados. Dentre as aplicações de regras ativas incluem-se: manutenção de integridade; manutenção de dados e aplicações externas, freqüentemente chamadas de regras de negócio. Estas são executadas como parte do serviço da computação que normalmente estaria contido no código da aplicação. Exemplos dessas regras são gerenciamento de inventário e alertas ao usuário sem mudanças no estado do banco de dados.

Programas de aplicação de banco de dados são usualmente escritos em linguagem semi-declarativa, tal como SQL (*Structured Query Language*), ou uma combinação de uma linguagem imperativa e uma linguagem declarativa. Muitas técnicas de teste de software baseadas em programa são projetadas para linguagem imperativas, não sendo diretamente aplicáveis a programas de aplicação de banco de dados por não considerar as manipulações de dados (Chays *et. al*, 2000).

As regras ativas diferem de programas procedimentais no modo de como são ativadas, pois essas dependem da ocorrência de eventos. Para a aplicação de técnicas de

teste em regras devem ser consideradas as três partes constituintes da regra, o evento, a condição e a ação.

Existem poucos trabalhos que exploram o teste de regras ativas. Chan *et. al* (1997) desenvolveram uma ferramenta que enfatiza o teste de regras isoladas para uma linguagem declarativa e orientada a objetos. Vaduva (1999) propõe o teste do conjunto de regras ativas. As propostas de teste de regras ativas não exploram a linguagem SQL, uma das linguagens mais usadas atualmente em ambientes de banco de dados relacional. Não foram encontrados trabalhos que apliquem análise do fluxo de controle e do fluxo de dados a regras ativas escritas em SQL. A SQL é uma linguagem amplamente usada no desenvolvimento de sistemas comerciais. Desde 1986 padrões estão sendo desenvolvidos para esta linguagem; SQL-3 (Fortier, 1999) é o padrão atualmente adotado.

1.2 Objetivos

A percepção da necessidade de testar regras ativas escritas em SQL e a carência de suporte automatizado para auxiliar este tipo de teste foram as motivações para o desenvolvimento deste trabalho que tem como objetivos principais:

- Proposição de modelos de fluxo de controle, de fluxo de dados e de instrumentação para suporte à automatização de teste estrutural de regras ativas isoladas.
- Construção de uma ferramenta, ART-TOOL (*Active Rule Testing Tool*), para apoiar o teste estrutural de regras ativas.
- Validação da ferramenta desenvolvida por meio de sua aplicação.
- Demonstração do uso da ferramenta no teste de uma regra ativa.
- Aplicação da ferramenta no teste de um conjunto de regras ativas.

1.3 Organização da Dissertação

Este trabalho se divide em seis capítulos. Neste capítulo apresentam-se o contexto e as metas a serem alcançadas.

No Capítulo 2 são apresentados os conceitos básicos de teste de software, especificando o teste de banco de dados, com uma referência especial a regras ativas em banco de dados relacionais ativos.

A fundamentação dos modelos de fluxo de controle, fluxo de dados e instrumentação para as regras ativas é descrita no Capítulo 3, baseada no relatório técnico "Aspectos de Fluxo de Controle e de Dados no Contexto de Teste de Regras Ativas" (Leitão *et. al*, 2002).

A ferramenta ART-TOOL é descrita juntamente com aspectos de implementação no Capítulo 4. Um exemplo de aplicação da ART-TOOL é apresentado no Capítulo 5. No Capítulo 6 são apresentadas as conclusões e as sugestões de trabalhos futuros.

O Apêndice A apresenta a gramática da linguagem SQL e o Apêndice B contém a descrição do banco de dados utilizado nos exemplos.

CAPÍTULO 2

TESTE EM BANCO DE DADOS

2.1 Introdução

Uma das metas da Engenharia de Software é produzir software de alta qualidade. Por meio do teste aplicado de maneira rigorosa tem-se a chance de identificar a presença de defeitos no software e, desta forma, contribuir decisivamente para a melhoria da sua qualidade (Vilela, 1998).

No teste de banco de dados vários aspectos têm sido explorados; em relação ao acesso a base de dados, não só programas de aplicações são considerados, mas todos os procedimentos que permitem um acesso aos dados da base de dados. Dentro desse conjunto de procedimentos são destacadas as regras ativas, componentes de banco de dados ativos, que permitem a manipulação de dados sem a intervenção do usuário, mas como efeito da ocorrência de eventos.

Neste capítulo são apresentados os conceitos básicos de teste de software com ênfase na técnica estrutural. Como o objetivo desse trabalho é o teste estrutural de regras ativas também são apresentados conceitos de banco de dados ativos e de regras ativas. No final é feita uma discussão dos trabalhos de teste de banco de dados existentes e das propostas que auxiliaram o desenvolvimento desta dissertação.

2.2 *Conceitos Básicos de Teste de Software*

"O desenvolvimento de sistemas de software envolve uma série de atividades de produção em que as oportunidades de inserir defeitos em decorrência de ações humanas são enormes. Defeitos podem ser introduzidos logo no começo do processo, onde os objetivos podem estar errôneos ou imperfeitamente especificados, além de erros que venham a ocorrer em fases posteriores de projeto e desenvolvimento. Por causa da incapacidade que os seres humanos têm de agir e comunicar com perfeição, o desenvolvimento de software deve ser acompanhado por uma atividade de garantia de qualidade" (Deutsch, 1979).

A atividade de teste é a chave para aprimorar não só a produtividade como também a confiabilidade do software (Chusho, 1987). O objetivo do teste é encontrar defeitos mas, quando detectados, o que ocorre de imediato é diminuir a confiança no software. O propósito da sua realização é justamente o contrário, é tornar o software mais confiável. A tentativa de "destruição" do software deve, ao invés, aumentar sua confiança e sua credibilidade (Graham, 1994). O teste também pode ser usado para avaliar o quanto o sistema atende às necessidades especificadas (Myers, 1979).

Em todas as etapas do processo de desenvolvimento de software, pode-se incluir uma atividade de teste, sendo esta a última atividade na qual é possível revelar defeitos na especificação, no projeto e na codificação do software (Pressman, 1997).

A condução sistemática do teste requer planejamento, projeto de casos de teste, execução do teste e avaliação dos resultados. O planejamento do teste deve estar incluído no planejamento global do sistema com um plano de teste. Nesse documento denominado plano de teste são estimados os recursos e são definidas as estratégias de teste a serem utilizadas, incluindo as técnicas e ferramentas de teste (Rocha *et al.*,2001). O projeto de caso de teste deve ser muito bem elaborado para que tenha boa probabilidade de encontrar defeitos, com o mínimo de tempo e esforço. Assim, durante o projeto de caso de teste é necessário selecionar um subconjunto de dados de teste; denomina-se caso de teste o par ordenado composto por um dado de entrada e pelo respectivo resultado esperado. Executados os casos de teste, os resultados – obtido e esperado – devem ser

comparados; para isto é necessário um mecanismo ou mesmo uma pessoa para identificar se houve ou não a ocorrência de falhas. Este mecanismo é o oráculo, que determina se os resultados encontrados estão corretos ou incorretos.

A atividade de teste pode ser dividida em fases com o objetivo de minimizar a complexidade do teste e possibilitar abordar diferentes tipos de defeitos e aspectos do software. De acordo com Pressman (1997) a atividade de teste deve ser dividida em quatro fases: o teste de unidade, o teste de integração, o teste de validação e o teste de sistema. Essas fases são executadas de forma incremental e complementar, em função das atividades do processo global de engenharia de software (Rocha *et al.*, 2001).

O teste de unidade concentra-se na implementação do código-fonte e tem como objetivo identificar defeitos de lógica e de implementação. O teste de integração visa a detectar defeitos de integração entre as unidades que compõem o programa. Após o teste de unidade e o teste de integração, um conjunto de testes de alto nível é realizado. Em seqüência deve ser conduzido o teste de validação, que tem como objetivo testar o programa como um todo e oferecer uma garantia final de que o software atende a todas as exigências funcionais, comportamentais e de desempenho. A última etapa, o teste de sistema, visa a identificar defeitos de função e/ou características de desempenho do programa implantado no ambiente de operação.

Um programa pode ser testado de maneira *ad hoc* ou sistemática. Na primeira abordagem, os casos de teste são derivados a partir da intuição do testador. Já o teste sistemático implica utilizar uma técnica de teste para projetar os casos de teste. Um dos objetivos das técnicas sistemáticas de teste é garantir que aspectos considerados importantes da especificação e da implementação do programa sejam exercitados pelo teste; elas podem ainda ter como objetivo detectar os defeitos mais comuns ou identificar máquinas de estados finitos errôneas. Por isso, as técnicas de teste são classificadas em funcionais, estruturais, baseadas em defeitos e baseadas em máquinas de estados finitos (Chaim, 2001).

A técnica funcional deriva requisitos de teste a partir das especificações do software e não leva em consideração a estrutura do código do sistema; é também chamada de técnica caixa-preta e os casos de teste são baseados nas funcionalidades do

programa. Exemplos de técnicas funcionais são: Particionamento de Equivalência, Análise de Valores Limites, Grafos de Causa e Efeito (Myers, 1979), Categorização-Particionamento (Ostrand e Balcer, 1988), etc. Já a técnica baseada em defeitos estabelece os requisitos de teste explorando os erros típicos e comuns cometidos durante o desenvolvimento de software (Rocha *et al.*, 2001). Os critérios Análise de Mutantes (DeMillo *et al.*, 1978) e Semeadura de Defeitos (Budd, 1981) são exemplos da técnica baseada em defeitos. A técnica de teste baseada em máquinas de estados finitos, por sua vez, utiliza a estrutura de máquinas de estado finito e o conhecimento subjacente para derivar requisitos de teste (Maldonado e Fabbri, 2001).

O teste estrutural ou teste caixa-branca tem como foco a implementação do software (estrutura de controle e dados do programa). A técnica estrutural de teste é discutida mais detalhadamente na próxima seção.

2.3 Técnica Estrutural de Teste

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 (Spoto, 2000). Esta técnica não é uma alternativa para o teste funcional e sim um complemento ao mesmo pois detecta classes de defeitos diferentes dos revelados pelas técnicas funcionais (Pressman, 1997).

Casos de teste derivados diretamente da implementação possibilitam o exercício das instruções bem como das condições lógicas do programa durante o teste. Abordagens de teste caixa-branca, tal como caminhos básicos, fazem uso de grafos do programa para derivar o conjunto de casos de teste.

A utilização desta técnica requer que casos de teste selecionados executem determinados caminhos do programa considerados relevantes para que o testador aumente a sua confiança em relação à corretude do programa. A execução desses caminhos exercita elementos do programa – requisitos de teste – definidos por um critério de teste, utilizado para avaliar a adequação do conjunto de casos de teste. Um

conjunto de casos de teste *T* é considerado adequado do ponto de vista do teste estrutural e de acordo com um critério *C*, se o conjunto de requisitos de teste estabelecido por *C* é exercitado pelos casos de teste de *T*. Para completar a discussão sobre esta técnica, conceitos são apresentados na Seção 2.3.1 e critérios de teste estruturais são descritos na Seção 2.3.2.

2.3.1 Conceitos de Fluxo de Controle e de Fluxo de Dados

Conceitos básicos de fluxo de controle e de fluxo de dados (Rapps e Weyuker, 1982 e 1985; Weyuker, 1984; Frankl e Weyuker, 1985 e 1988; Maldonado, 1991; Maldonado et. al, 1992) são necessários para a definição de critérios estruturais.

Considere *P* um programa representado por um grafo de fluxo de controle *G(N, R, s, e)* onde *N* é o conjunto de nós, *e* é o nó de entrada, *s* é o nó de saída e *R* é o conjunto de arcos, pares ordenados de nós *(n,m)* de *N* que representam a possível transferência de fluxo de controle entre dois nós.

Um *caminho* é uma seqüência finita de nós $(n_i, ..., n_k)$, $k \geq 2$, tal que existe um arco de n_i para n_{i+1}, $i = 1, 2, ..., k-1$. Um *caminho simples* é um caminho onde todos os nós, exceto possivelmente o primeiro e o último, são distintos; um *caminho livre de laço* é aquele que possui todos os nós distintos. Um *caminho completo* começa no nó de entrada e termina no nó de saída.

Pela análise do fluxo de dados é possível definir os tipos de ocorrências de variáveis; uma *definição de variável* (*def*) ocorre sempre que um valor é armazenado em uma posição de memória. A ocorrência de uma variável é um *uso* quando ela não estiver sendo definida; um uso pode ser um *c-uso* ou um *p-uso*. Um *c-uso*, ou *uso computacional*, afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado. Um *p-uso*, ou *uso predicativo*, afeta diretamente o fluxo de controle do programa; é o uso de variáveis em predicados de comandos condicionais. A ocorrência de uma variável é *indefinida* quando o seu valor se torna inacessível ou sua localização deixa de estar vinculada a uma posição de memória.

Um caminho $(i, n_1, \ldots, n_m, j)$, $m \geq 0$, que não contenha definição de uma variável x nos nós $n_1, \ldots, n_m$ é chamado de *caminho livre de definição* com respeito a (c.r.a.) x do nó i ao nó j e do nó i ao arco (n_m, j).

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

São importantes também os conceitos de *du-caminhos*, *potencial-du-caminho*, *associação c-uso*, *associação-p-uso* e *associação.*

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

Um caminho livre de definição $(n_1, n_2, \ldots, n_j, n_k)$ c.r.a. a uma variável x, onde o caminho $(n_1, n_2, \ldots, n_j)$ é um caminho livre de laço e n_1 tem uma definição de x, é denominado *potencial-du-caminho* c.r.a. x.

Uma *associação definição-c-uso* é uma tripla (i, j, x) onde i é um nó que contém uma definição global de x e $j \in dcu(x,i)$[1]. Uma *associação definição-p-uso* é uma tripla $(i, (j,k), x)$ onde i é um nó que contém uma definição global de x e $(j, k) \in dpu(x, i)$[2]. Uma *associação* é uma *associação definição-c-uso*, uma *associação definição-p-uso* ou um *du-caminho*.

---

[1] $dcu(x,i)$ = {nós j tal que $x \in$ *c-uso*(j) e existe um caminho livre de definição c.r.a. x do nó i para o nó j}.

[2] $dpu(x,i)$ = {arcos (j,k) tal que $x \in$ *p-uso*(j,k) e existe um caminho livre de definição c.r.a. x do nó i para o arco (j,k)}.

2.3.2 Critérios de Teste Estrutural

Critérios de teste estabelecem requisitos a serem satisfeitos e podem orientar a seleção de dados de teste, a avaliação da qualidade do teste e a definição de requisitos de suficiência a serem atingidos para o encerramento da atividade de teste (Bueno, 1999).

Os critérios que consideram apenas a especificação do programa para derivar os requisitos de teste são chamados critérios funcionais; os critérios estruturais são aqueles que consideram a implementação do programa (Howden, 1975; Woodward *et. al*, 1980; Rapps e Weyuker , 1982; Laski e Korel, 1983; Ntafos, 1984; Rapps e Weyuker, 1985; Ural e Yang, 1988; Maldonado, 1991, Taylor *et. al*, 1992; Parrish *et. al*, 1993; Harrold e Rothermel, 1994; Marx e Frankl, 1996).

Os primeiros critérios de teste estrutural propostos baseavam-se exclusivamente no fluxo de controle do programa. O fundamento desses critérios é de que não se pode confiar em uma parte do programa que não tenha sido exercitada. Os principais critérios dessa técnica são: Todos os Nós, Todos os Arcos e Todos os Caminhos.

Os critérios Todos os Nós e Todos os Arcos requerem o exercício de cada nó e de cada arco do programa, respectivamente. São critérios aplicáveis devido ao fato de exigirem uma quantidade finita de dados de teste para serem satisfeitos. Já o critério Todos os Caminhos exige que cada caminho do grafo seja exercitado; é denominado *ideal* ou *exaustivo* do ponto de vista estrutural. O teste exaustivo de programas é, na maioria das vezes, impraticável visto que o número de caminhos pode ser muito grande ou até mesmo infinito (quando laços estão presentes), tornando-se inviável devido ao alto custo.

Vários trabalhos de otimização de compiladores e teste de programas foram desenvolvidos utilizando a análise de fluxo de dados de um programa (Hecht, 1977; Herman, 1976; Laski e Korel, 1983; Ntafos, 1984; Rapps e Weyuker 1985; Maldonado *et. al* 1988 a; Maldonado *et. al* 1988 b; Ural e Yang, 1988). O teste baseado em análise de fluxo de dados estabelece requisitos de teste que exigem a execução de caminhos entre a definição e o (potencial) uso de uma variável; por isso, os seus requisitos de teste são chamados de associações definição-(potencial)-uso, derivadas da idéia de que não se

pode confiar em um resultado de uma computação se o ponto onde ela foi realizada e o subseqüente ponto onde o valor é usado, não foram testados conjuntamente. A definição de critérios baseados em análise de fluxo de dados visa construir uma ponte de ligação entre o Critério Todos os Arcos e o Critério Todos os Caminhos, disponibilizando critérios mais eficazes que Todos os Arcos e menos exigentes que Todos os Caminhos.

Os critérios Potenciais Usos constituem uma família de critérios estruturais, baseada em análise de fluxo de dados para o teste de unidade, definida por Maldonado (1991). Pode-se dizer que a abordagem Potencial Uso alivia uma das fraquezas do teste estrutural; não se está apenas tentando identificar defeitos devidos ao uso incorreto de variáveis, mas inclui a possibilidade de se identificar defeitos devido à ausência de uso, que também constitui um defeito (Vilela, 1998).

São denominados Potenciais Usos devido ao fato de serem baseados no conceito de potencial uso e são variações da família de critérios apresentada por Rapps e Weyuker (1982 e 1985). Esses critérios requerem associações que não exigem a ocorrência explícita de um *uso* de variável diferindo dos conceitos apresentados por Rapps e Weyuker, em que a associação é requerida se um *uso* existir. Os critérios Potenciais Usos requerem associações que implicam o exercício de caminhos entre a definição de uma variável e um possível (potencial) uso dessa variável.

A seguir, são descritos alguns critérios de fluxo de dados das famílias Fluxo de Dados (Rapps e Weyuker,1985) e Potenciais Usos (Maldonado *et al.*, 1992).

- **Critério Todos os P-usos**: requer que todas as associações definição-uso (*adu*) do tipo $(i, (j,k), x)$ do programa em teste sejam exercitadas pelos casos de teste de um conjunto T. Uma associação $(i, (j,k), x)$ é exercitada quando pelo menos um caminho livre de definição c.r.a. x do nó i até o ramo (j,k) é executado por um caso de teste $t \in T$.
- **Critério Todos os Usos**: requer que todas as associações definição-uso dos tipos (i, j, x) e $(i, (j,k), x)$ sejam exercitadas pelos casos de teste de um conjunto T. Uma associação $(i, (j,k) x)$ ou $(i, (j,k), x)$ é exercitada quando pelo menos um caminho livre de definição c.r.a. x do nó i até o nó j ou ramo (j,k) é executado por um caso de teste $t \in T$.

- **Critério Todos Potenciais Usos**: requer que todas as associações definição-potencial-uso (*adpu*) dos tipos (i, j, x) e $(i, (j,k), x)$ sejam exercitadas pelos casos de teste de um conjunto T. Note-se que para caracterizar uma associação definição-potencial-uso não é necessário um uso explícito de x em j ou (i,j), apenas que o nó j ou ramo (i,j) seja alcançável por um caminho livre de definição c.r.a. x a partir de i.

- **Critério 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 laço que cubram essas associações; ou seja, requer que todos du-caminhos sejam exercitados.

- **Critério Todos Potenciais Usos/Du**: requer que todas as associações definição-potencial-uso dos tipos (i, j, x) e $(i, (j,k), x)$ sejam exercitadas pelos casos de teste de um conjunto T, porém, por potenciais du-caminhos.

- **Critério Todos Potenciais Du-Caminhos**: requer todas as *adpus* do tipo $(i, (j,k), D)$, onde D é um conjunto de variáveis definidas no nó i, sejam exercitadas pelos casos de teste de um conjunto T. Para satisfazer a *adpu* $(i, (j,k), D)$, os casos de teste devem executar caminhos que alcancem (j,k) e que são livres de definição c.r.a. todas as variáveis x; ou seja, requer a execução de todos os potenciais du-caminhos.

É possível avaliar a qualidade do teste por meio de uma análise de cobertura. Essa análise determina o percentual de elementos requeridos pelo critério efetivamente exercitados pelo conjunto de casos de teste aplicados. Deste modo, quanto maior este valor, mais completo terá sido o teste. Com essas informações, o conjunto de casos de teste pode ser melhorado para que os elementos ainda não abordados sejam testados com adição de novos casos de teste (Rocha *et al.*, 2001).

2.4 Conceitos de Sistema de Banco de Dados Ativos

Regras ativas são procedimentos que pertencem exclusivamente a banco de dados ativo e que também manipulam dados de uma base de dados. A proposta desta pesquisa é adaptar técnicas de teste estrutural, utilizadas em teste de unidade, para testar regras ativas.

Segundo Elmasri e Navathe (2000) banco de dados é uma coleção de dados relacionados com um significado implícito, ou seja, os dados possuem uma coerência lógica com significados inerentes. Um banco de dados com programas específicos para sua manipulação forma um conjunto denominado sistema de banco de dados. Um sistema de banco de dados inclui também o sistema gerenciador de banco de dados (SGBD), que é uma coleção de programas que capacita usuários a criar e manter um banco de dados. O SGBD é portanto um sistema de software que facilita o processo de definição, construção e manutenção de banco de dados para várias aplicações. Todas as solicitações dos usuários de acesso ao banco de dados são manipuladas pelo SGBD: criação de tabelas, inserção de dados, recuperação de dados, etc.

Sistemas de banco de dados tradicionais são passivos - a manipulação de dados só ocorre quando comandos são executados pelo banco de dados por uma requisição feita pelo usuário ou por programa de aplicação. Sistemas de bancos de dados ativos são uma extensão a um modelo de banco de dados (relacional, relacional estendido, funcional ou orientado a objeto) com algumas variantes de um paradigma comum de computação, as regras (Campin *et. al*, 1994). A comparação entre sistemas de bancos de dados tradicionais e ativos está representada na Figura 2.1.

Bancos de dados ativos suportam mecanismos que possibilitam ao sistema responder automaticamente a eventos, que ocorrem dentro ou fora do sistema de banco de dados, tais como, operações de manipulações de dados, eventos temporais, etc. O comportamento reativo a eventos, segundo Montesi e Torlone (1995), é baseado em um sistema de regras que conectam uma causa a um efeito esperado. Essas regras, denominadas regras ativas, servem para monitorar e reagir a circunstâncias específicas relevantes.

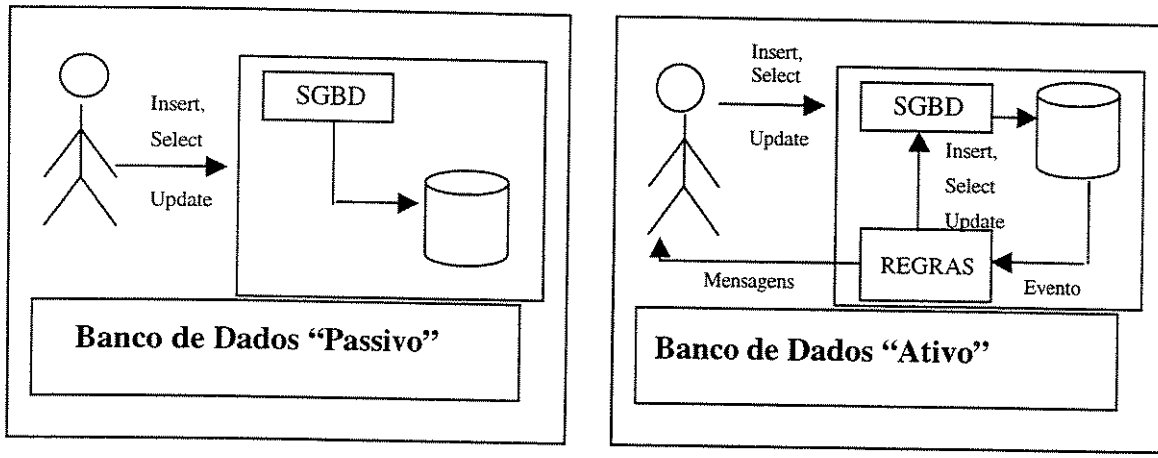


Figura 2.1: Comparação entre BD passivo e BD ativo (Paduan, 2000).

Segundo Montesi e Torlone (1995), um sistema de regras é baseado em regras que conectam uma causa a um efeito esperado. Na estrutura de um sistema de banco de dados, causa e efeito são manipulações de dados. Tais regras permitem expressar, de uma maneira muito natural, importantes conceitos de banco de dados como cenários, restrições e métodos. Uma regra ativa é uma generalização deste simples esquema de linguagem.

Como definido por Fraternali *et. al* (1997), regras ativas são uma vasta e difundida extensão tecnológica de bancos de dados modernos, que são largamente aceitáveis como poderosos mecanismos para implementar aplicações ou comportamento de sistemas. A definição de uma regra ativa é discutida na Seção 2.4.1.

2.4.1 Definição de Regras Ativas

Um sistema de banco de dados ativos dispara regras automaticamente em resposta a ocorrências de eventos. Para melhor definir pode-se dizer que a regra ativa é uma associação de um evento a uma condição que, se verdadeira, dispara uma ação.

Segundo Zaniolo *et. al* (1997) as aplicações de regras ativas apóiam características de gerenciamento de banco de dados clássicos, tais como manutenção de integridade, manutenção de dados e gerenciamento de replicações, que são aplicações internas. Já as

aplicações externas, freqüentemente chamadas de regras de negócio, são executadas como parte do serviço da computação que normalmente estaria contido no código da aplicação. Exemplos dessas regras são gerenciamento de inventário e alertas ao usuário sem mudanças no estado do banco de dados. Pode-se dizer que as regras são capazes de monitorar e reagir a circunstâncias específicas relevantes para a aplicação. Como mencionado anteriormente o termo *triggers* é usado para referir-se a regras; o nome é dado a regras de sistemas comerciais de bancos de dados; neste trabalho os dois termos são usados indistintamente.

A maioria dos sistemas são baseados no conceito de regras ativas na forma de Evento-Condição-Ação (ECA), onde o processamento de regras é mantido em um ambiente específico como uma aplicação de banco de dados ativos incluindo transações (Coupaye e Collet, 1998). As regras ativas estão associadas a tabelas em banco de dados relacional e, em banco de dados orientado a objetos, podem estar associadas a uma classe. Na Figura 2.2 são mostrados os componentes estruturais de uma regra ativa (ECA) e a codificação da regra no padrão SQL-3 (BNF SQL3, 1998). A associação da regra à tabela é expressa na cláusula ON.

O paradigma evento-condição-ação pode ser entendido melhor como definido em Campin *et al.* (1994): um evento representa uma ocorrência no banco de dados, que pode ser um acesso ou uma modificação para um item de dado específico ou um acontecimento externo tal como o estado de um relógio. Este evento dispara a regra propriamente dita, com o primeiro passo sendo a verificação de uma condição, um valor *booleano* de aplicação sobre o banco de dados que não tem efeitos colaterais. Condições são geralmente predicados sobre o estado do banco de dados expressos em uma linguagem de aplicação. Se esta aplicação retornar valor verdadeiro, uma ação será executada; pode ser qualquer operação expressa no banco de dados.

Sintetizando os componentes de uma regra: o evento é a parte da regra que descreve uma ocorrência a qual a regra deve ser capaz de responder; a condição é o componente que examina o contexto no qual o evento ocorreu; e a ação descreve a tarefa a ser cumprida pela regra. A maioria dos sistemas de banco de dados ativos suporta

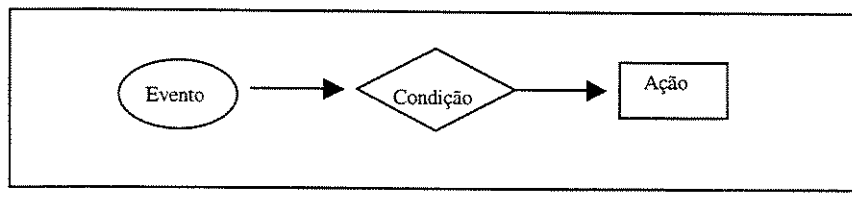
regras com os três componentes; em algumas propostas o evento ou a condição podem ser omitidos ou estar implícitos.

Para melhor entendimento do comportamento reativo do sistema de banco de dados ativos, todo o mecanismo deve ser descrito por meio de um modelo de conhecimento e de um modelo de execução, que são discutidos nas seções seguintes.

2.4.2 Modelo de Conhecimento de Regras Ativas

Segundo Paton e Díaz (1999), o modelo de conhecimento descreve as funcionalidades das regras, suportadas pelo SGBD. Paton e Díaz (1999) definiram um modelo de conhecimento que determina os possíveis estados de banco de dados em tempo do disparo e execução da regra. Este modelo de conhecimento de uma regra ativa está ilustrado na Figura 2.2 com os três componentes: um evento, uma condição e uma ação.

O evento é um acontecimento em um ponto no tempo, produzindo uma descrição de um fato monitorado, sendo gerado para o sistema e detectado pelo mesmo. Para iniciar uma regra, um evento deve ser gerado. A natureza da descrição e a maneira pela qual o evento pode ser detectado dependem do gerador do evento, pois este pode ter origem interna ou externa. Eventos internos podem ser conseqüência de uma operação sobre dados do próprio banco de dados; eventos externos são ocorrências no ambiente que envolve o banco de dados, tal como uma leitura da temperatura em determinado momento. Embora todos estes eventos sejam citados teoricamente há restrições de acordo com o SGBD. Outro aspecto do evento é a sua granulosidade, que indica se um evento é definido para todos os objetos de um conjunto ou para um subconjunto específico. O tipo de evento pode ser primitivo ou composto; primitivo quando se referir a ocorrências elementares como atualizações do banco de dados; composto quando criado por uma combinação de eventos primitivos usando conectores lógicos.



CREATE TRIGGER <nome da regra > (AFTER|BEFORE) <operação[OR <operação>]> OF <atributos> **ON** <nome da tabela> FOR EACH (ROW|STATEMENT)	**EVENTO**
[WHEN <predicado>]	**CONDIÇÃO**
BEGIN :: :: END	**AÇÃO**

Figura 2.2: Componentes estruturais da regra ativa e codificação no padrão SQL-3.

Após a ocorrência do evento é considerada a condição, que tipicamente é um predicado sobre o estado do banco de dados expresso em uma linguagem de aplicação. O papel de uma condição é indicar se a ação vai ser executada. O evento combinado com a condição descreve a situação que uma regra ECA está monitorando. A informação sobre a ocorrência do evento deve ser passada do evento para a condição, quando esta existir, pois a condição é opcional.

As ações podem ser qualquer seqüência de operações dentro do banco de dados, operações de modificações e/ou recuperação de dados e procedimentos de aplicações.

2.4.3 Modelo de Execução de Regra Ativa

A semântica básica de uma regra ECA resume-se na seguinte frase: quando um evento do tipo E ocorre, se a condição C é verdadeira então a ação A é executada. A semântica formal segundo Coupaye e Collet (1998) expressa na verdade o modelo de execução, isto é, o mecanismo da execução da regra.

O modelo de execução de regra ativa define o comportamento de um conjunto de regras em tempo de execução (Paton e Díaz, 1999), ou seja, monitoramento e reação a determinadas circunstâncias. Por meio deste modelo é possível conhecer um outro aspecto do sistema: como um conjunto de regras, ou apenas uma regra, é tratado em tempo de execução e as principais etapas desta fase.

Um aspecto importante a ser considerado durante a execução de regras ativas é o tratamento de exceções. A maioria dos sistemas simplesmente aborta a transação corrente quando ocorre algum erro no processamento da regra; é o comportamento padrão em banco de dados. No entanto, outras alternativas podem ser convenientes, tais como: ignorar a regra que criou o erro e continuar processando outras regras; voltar para o estado de início do processamento da regra e recomeçar o processamento; ou continuar com a transação. Alguns planos de contingência devem ser adotados para recuperar do estado de erro, possivelmente usando o mecanismo de exceção de sistema de banco de dados.

O modelo de execução de regra proposto por Paton (1999) é mostrado na Figura 2.3. Com a geração de um evento, responsável pelo disparo da regra, ocorre uma mudança no estado da regra, de inativo para ativo. A partir do disparo, a condição da regra é avaliada e a ação da regra é executada. No fim de sua execução, a regra volta para o estado inicial, como se observa na Figura 2.3; a regra fica inativa, pronta para ser disparada por outro evento. Uma regra disparada não é necessariamente avaliada imediatamente; do mesmo modo, uma regra avaliada não é necessariamente executada imediatamente. O passo da avaliação corresponde à parte da condição da regra, se a condição é satisfeita a regra vai para um estado avaliado; caso contrário, ela volta para o estado disparável. Quando se tratar de um conjunto de regras, estas são avaliadas e depois devem obedecer a uma prioridade para serem selecionadas para execução. Estas fases mostradas na Figura 2.3 não são necessariamente executadas continuamente; dependem do modo de acoplamento entre evento-condição e condição-ação. As opções para o modo de acoplamento freqüentemente suportadas são imediata e adiada, segundo Berndtsson e Calestam (2000). No primeiro modo a condição é avaliada imediatamente depois do evento. No modo adiado, a condição é avaliada dentro da mesma transação como o evento da regra, mas não necessariamente na primeira oportunidade.

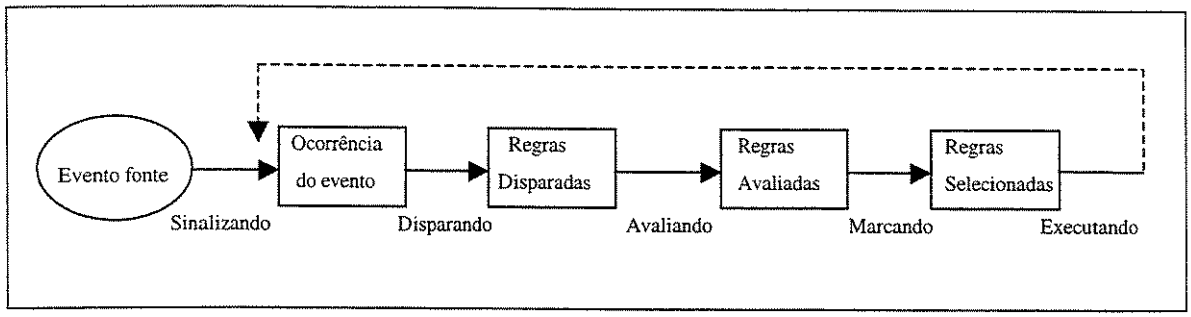


Figura 2.3: Principais passos durante a execução de uma regra.

A granulosidade da regra indica se a execução de uma regra é orientada a instância (granulosidade de linha) ou orientada a conjunto (granulosidade de comando). Uma regra orientada a instância é executada uma vez a cada instância da operação do banco de dados que dispara a regra. Por exemplo, considere uma regra R_i que é disparada quando ocorre a inclusão de uma *tupla* na tabela T_1; a condição da regra será avaliada e sua ação será executada uma vez para cada *tupla* inserida. Quando a regra é orientada a conjunto, esta é executada uma vez para todas as instâncias da operação do banco de dados que dispara a regra. Neste caso, a condição da regra R_i será avaliada e sua ação será executada uma vez para o conjunto de *tuplas* inseridas. Para regras escritas no padrão SQL-3, esta granulosidade é indicada na cláusula do evento FOR EACH que pode ser executada para cada linha ou *tupla* usando o ROW ou por conjunto usando o STATEMENT, como ilustrado na codificação do evento da Figura 2.2.

Considerando a execução de um conjunto de regras que podem ser disparadas pelo mesmo evento, o projetista de sistema de regras deve especificar a ordem na qual as regras serão executadas. Em geral, um mecanismo de resolução de conflitos pode ser usado para determinar se estas serão executadas seqüencialmente ou concorrentemente; também são adotadas políticas para dar prioridade a algumas regras.

2.5 *Revisão Bibliográfica*

Os trabalhos que abordam teste de banco de dados enfatizam diferentes aspectos. Tais aspectos são:

- geração automática de dados de teste (Lyons, 1977; Noble, 1983; Davies *et al.*, 2000);
- teste de consultas (Mannila e Rih ,1989; Yan, 1991);
- teste de sistema de banco de dados (Roper e Rahim, 1993);
- teste de programa de aplicação de banco de dados (Chays *et al.*, 2000; Spoto, 2000);
- teste do esquema da base de dados (Aranha *et al.*, 2001).

O conjunto de regras ativas de um banco de dados é parte integrante da aplicação e, portanto, requer esforços na direção de abordagens de teste específicos para este contexto. Somente no final da década de 90 surgiu o primeiro trabalho que explora regras ativas; Chan *et al.* (1997) adaptam técnicas de teste de programas convencionais para o teste de banco de dados ativos, procurando explorar uma regra ativa. Também neste enfoque Vaduva (1999) propõe o teste de regras ativas, explorando a combinação de regras.

2.5.1 Trabalhos Relacionados ao Teste de Banco de Dados

Uma das primeiras publicações em teste de banco de dados, por Lyons (1977), propõe uma ferramenta de geração automática de dados para simulação e teste de uma base de dados. A ferramenta destina-se à geração de dados de teste para sistemas que tipicamente manipulam grande volume de dados. O objetivo é produzir grande variedade de dados de teste muito rapidamente para cobrir uma larga variedade de situações possíveis de encontrar no mundo real.

Noble (1983) apresenta um trabalho para também gerar dados de teste automaticamente para banco de dados relacional. O referencial deste trabalho é a estrutura do banco de dados, onde é estabelecida uma hierarquia de relações visando

especificar e gerar dados de teste que não são somente válidos com respeito a atributos e domínios mas também na preservação das dependências entre as relações.

A abordagem apresentada por Manila e Rih (1989) é um teste de consultas, que gera uma base de dados mínima a partir de uma base real. São apresentados dois algoritmos para a geração de base de teste com o objetivo de testar consultas feitas à base de dados relacional. A base de teste é gerada a partir das consultas do programa, usando as operações de SELECT, PROJECT e JOIN. Esta base de dados é suficiente para ser usada como dados de teste para exercitar as dependências entre as possíveis classes de consultas. É uma técnica que depende do conhecimento de toda a base de dados necessária para cada programa de aplicação e também das consultas envolvidas no programa.

Em 1991, Yan desenvolveu um trabalho que apresenta um processo de teste declarativo para bancos de dados lógicos. O seu objetivo foi testar as consultas da base de dados, ou seja, verificar separadamente a corretude dos predicados usados na consulta e da base de dados. Esta abordagem parte do princípio de que tanto a consulta como a base de dados podem estar incorretas. Com os resultados obtidos e esperados é feita uma análise comparativa. Esse enfoque torna possível verificar se a base de dados ou se a consulta está inconsistente, incorreta ou incompleta.

Uma abordagem diferente é proposta no trabalho de Roper e Rahim (1993). Voltada especificamente para o esquema da base de dados, concentra-se na modelagem dos dados do sistema; é um teste de modelo lógico de dados. São estabelecidas situações para verificar a conectividade de entidades (mandatórias ou não; 1:1; 1:N ; N:N), onde são geradas combinações de entidades que devem ser suportadas pelo sistema. Neste estudo também é feita uma análise dos resultados esperados com os resultados computacionais. Usa-se o diagrama entidade-relacionamento para uma análise de valores limites e para examinar se as entidades e relacionamentos podem realmente ser suportadas pelo sistema. É desejável testar se o sistema suporta relacionamentos ilegais, que constituem fontes de falhas no sistema. É uma técnica fácil para aplicar em diagramas com várias entidades, podendo converter esses requisitos de teste em casos de teste.

A produção automática de dados de teste para popular um banco de dados, estabelecendo a integridade dos dados armazenados, é proposta por Davies *et. al* (2000). Este trabalho parte da análise da estrutura das tabelas da base de dados, ajusta os limites da entrada dos campos analisando gramaticalmente a validação das regras, gera dados de teste válidos e inválidos, manipula e recupera erros. Neste trabalho é possível observar que o teste de restrição de integridade dos dados definido em um esquema de banco de dados pode ser incluído em uma geração automática de dados, ou seja, no processo de popular tabelas.

Chays *et al.* (2000) apresentam uma proposta para testar sistemas de banco de dados e programas de aplicação de banco de dados relacional. O principal foco para estes testes é o estado do banco de dados e entradas e saídas do sistema. A ferramenta gera dados que satisfaçam as restrições prováveis de ocorrer e que possam apresentar falhas na aplicação. O trabalho explora a corretude do programa de aplicação do banco de dados, pelo teste de uma variedade de diferentes situações que as aplicações podem enfrentar.

Aranha *et al.* (2001) desenvolveram uma ferramenta de apoio ao teste de banco de dados relacional, sendo seu objetivo o teste do esquema de uma base de dados. Para tal propósito a ferramenta executa as operações sobre a base de dados, exercitando atributos específicos do esquema segundo os critérios de teste. São usados critérios de teste de relação (unidade) e de relacionamento (integração), que requerem o exercício de elementos por meio de operações da linguagem SQL. A idéia de aplicação desses critérios tem como objetivo a percepção de defeitos nas declarações de chaves primárias, chave estrangeira não definida ou definida incorretamente, restrições de domínio definidas incorretamente ou de forma incompleta, e atributos que deveriam ter sido definidos com restrições de domínio e não o foram. Os elementos da base de dados testados são os atributos e as restrições de integridade. Mais especificamente, este trabalho enfoca os atributos, relações e relacionamentos e também exige a ativação das restrições. A ferramenta torna possível a detecção de defeitos introduzidos na implementação e no projeto da base de dados.

2.5.2 Trabalhos Relacionados ao Teste de Regras Ativas

A primeira proposta efetiva de teste de bancos de dados ativos foi introduzida por Chan *et al.* (1997). O trabalho faz uso de um ambiente que apóia técnicas de teste estrutural existentes aplicadas a programas de aplicação de bancos de dados ativo orientado a objetos, por meio de uma ferramenta de teste. A ferramenta fornece a visualização de um grafo de fluxo de controle de cada regra ativa, para permitir testes com dados que as ativem e com dados que não as ativem. É uma proposta de teste que se concentra em fluxo de controle, incluindo um exame intra-regra e inter-regra. A idéia é selecionar casos de teste para revelar defeitos na especificação de cada regra. A análise do fluxo de controle intra-regra depende de particularidades da especificação da linguagem da regra, que neste caso é a CDOL, uma linguagem orientada a objeto. A ferramenta dispõe de recursos para gerar relatórios sobre as atividades de teste que podem ser considerados como um guia para auxílio ao testador.

Vaduva (1999) explora o teste de conjuntos de regras. O objetivo do teste é determinar a existência de defeitos causados por conflitos e por dependências entre regras. A proposta deste trabalho é o teste de regras baseado em fluxo de controle inter-regra e em interações entre regras para tornar possível a detecção de comportamento errôneo de regras. A idéia básica é a cobertura de tantas execuções de cenários quanto possível, em particular a execução de diferentes seqüências de regras. Para complementar o teste é aplicada a técnica de teste aleatório, usada em testes estatísticos, por meio do qual é possível extrair informações adicionais necessárias para entender o comportamento das regras. Com esta proposta é possível revelar interações inadequadas entre regras e falsas seqüências de execuções das regras.

São poucos os trabalhos específicos em teste de regras ativas; são encontradas mais propostas relacionadas com a depuração de regras ativas. As primeiras propostas de depuração de regras ativas são encontradas nos trabalhos de Díaz *et al.* (1993) e de Behrends (1994), que propõem ferramentas para mostrar ciclos de eventos e eventos complexos. Outros trabalhos desenvolvidos incluem ferramentas, propostas por Chakravarthy *et al.* (1995), Benazet *et al.* (1995), e Jähne *et al.* (1996), focadas em observar diferentes partes das regras em execução.

A ferramenta de depuração desenvolvida por Jähne *et al.* (1996) simula a execução de regras em bancos de dados ativos e está focada em observar diferentes partes da regra em execução de acordo com o modo de acoplamento. O modo de acoplamento determina quando a ação será executada em relação à condição da regra. A ação pode ser executada imediatamente após a condição ser satisfeita ou ser adiada; neste caso, pode ser executada outra regra no intervalo para depois sua ação ser executada. Usuários podem fazer experimentos com diferentes modos de acoplamento, modificar condições da regra e ações, e modificar o estado do banco de dados como parte do processo de depuração.As funcionalidades específicas incluem gerenciamento flexível da regra, *breakpoint* de depuração, visualização do disparo da regra, inspeção e mudança do estado das variáveis, detecção de ciclos da regra e controle do usuário sobre a seqüência de disparo da regra.

Esforços mais recentes em depuração, incluindo os trabalhos de Coupaye *et al.* (1999) e Kappel *et al.* (2000), estão mais voltados para a visualização do comportamento das regras ativas em várias etapas do processamento.

2.5.3 Trabalhos Utilizados

Resultados de dois trabalhos foram diretamente utilizados no desenvolvimento desta pesquisa. O primeiro é uma ferramenta de teste estrutural (Chaim, 1991); alguns de seus módulos foram acoplados à ferramenta ART-TOOL. O segundo inspirou a definição de modelos de implementação da ART-TOOL (Spoto, 2000). Ambos os trabalhos são descritos a seguir.

Chaim (1991) desenvolveu uma ferramenta de suporte ao teste estrutural de programas baseado em análise de fluxo de dados, denominada POKE-TOOL (Potencial Uses Criteria Tool for Program Testing). A ferramenta apóia a aplicação dos critérios Potenciais Usos bem como alguns critérios estruturais como: Todos os Nós, Todos os Arcos e Todos os Usos. Trata-se de uma ferramenta de teste de unidade de programas, multi-linguagem, ou seja, não está atrelada a uma linguagem de programação específica. É uma ferramenta interativa, que disponibiliza ao usuário todas as tarefas básicas de um teste: a análise do código fonte e a criação de uma base de dados; a geração de relatórios baseados na análise estática do código fonte; a instrumentação do código fonte; a análise

dos resultados, gerando relatórios; e a geração de relatórios para auxiliar a organização das atividades de teste e a derivação de conjunto de casos de teste específicos.

A sessão de trabalho da POKE-TOOL está dividida em duas partes: a fase estática e a fase dinâmica. Na primeira fase é feita uma análise estática da unidade em teste, o programa em teste é instrumentado, gerando uma nova versão para o teste propriamente dito. A partir de informações do código fonte, obtém-se o grafo de fluxo de controle e o grafo de fluxo de dados, sendo então gerados os elementos requeridos para os critérios determinados. Resumidamente, nesta fase há a geração de informações estáticas do programa para a execução do teste, com a versão instrumentada.

A fase dinâmica consiste no processo de execução e avaliação dos casos de teste aplicados. É executado o programa instrumentado e a ferramenta armazena em arquivos as informações obtidas em tempo de execução tais como: os dados de entrada, os parâmetros fornecidos, as saídas obtidas e os caminhos executados para cada caso de teste. Após a execução da unidade em teste, o módulo avaliador analisa as informações recebidas. O módulo avaliador determina a cobertura atingida com o conjunto de casos de teste aplicados para o critério escolhido e fornece os elementos requeridos executados e os não executados.

Como a ação de uma regra ativa pode ser tratada de modo análogo a uma unidade de programa e são usadas técnicas de teste estrutural para testá-las com a aplicação dos mesmos critérios da POKE-TOOL, foi feito um acoplamento de dois módulos com a ferramenta ART-TOOL para completar o processo de teste. O uso desses módulos da POKE-TOOL só é possível por serem módulos independentes. Mais informações são encontradas nas Seções 4.3.5 e 4.3.7, onde são apresentadas as especificações desses módulos.

A proposta de Spoto (2000) trata do teste estrutural de programas de aplicação de banco de dados relacional (ABDR). Como o foco principal é o programa de aplicação, este é visto como um conjunto de programas e cada programa como um conjunto de unidades. Devido a este referencial o trabalho é proposto em duas partes, uma voltada somente para o teste de unidade, e outra para o teste de integração, ou seja, o teste está baseado em dois tipos de fluxo de dados. O primeiro aplica-se ao teste isolado de

programas de ABDR com SQL embutida, o fluxo intra-modular. É utilizada a técnica de teste de caixa-branca com o intuito de caracterizar um conjunto de componentes elementares de uma aplicação que devem ser exercitados pelos dados de teste. O segundo tipo de fluxo de dados aplica-se à integração de módulos, o fluxo inter-modular, requerendo a execução de comandos SQL que manipulam as variáveis tabela de uma ABDR. São propostos critérios de teste estrutural baseados nos dois tipos de fluxo.

Os conceitos básicos desenvolvidos neste trabalho para exercitar variáveis tabela, teste de unidade envolvendo comandos da linguagem SQL e todo o estudo feito com respeito à interação entre tabelas servem de apoio à proposta apresentada nesta pesquisa, uma vez que as regras ativas se enquadram como uma unidade de programa envolvendo variáveis tabela e também são escritas em linguagem SQL.

2.6 Considerações Finais

Neste capítulo foram apresentados conceitos básicos de teste de software, ressaltando a técnica estrutural, uma das bases deste trabalho. A técnica caixa-branca, como é comumente chamada, parte do princípio de que os requisitos de teste devem ser derivados da implementação do software. Os critérios de teste mais comumente aplicados são critérios que visam o exercício de caminhos, nós e arcos; os critérios Potenciais Usos requerem associações que demandam a execução de caminhos entre a definição de uma variável e um potencial uso da definição da variável.

Bancos de dados ativos, objeto deste trabalho, possuem regras ativas, componentes que desempenham parte da funcionalidade de uma aplicação. Regras ativas fazem parte de um sistema reativo, são acionadas por eventos e atuam interna ou externamente ao banco de dados.

Regras ou *triggers* são disparadas em resposta a ocorrência de eventos, internos ou externos. São compostas de três componentes: o evento, a condição e a ação. Quando um evento ocorre a condição da regra é avaliada; se esta é verdadeira a ação é executada.

As regras ativas foram definidas por meio de um modelo de conhecimento que descreve as suas funcionalidades; e por um modelo de execução que mostra o comportamento da regra em tempo de execução.

Foram discutidos os trabalhos existentes de teste banco de dados, que exploram diferentes aspectos. Trabalhos específicos de teste de regras ativas foram apresentados, um com ênfase em teste de regras isoladas escritas em uma linguagem orientada a objetos e o outro que propõe um teste do conjunto de regras.

Para finalizar foram apresentados os trabalhos que apóiam a definição e implementação da ferramenta desse trabalho (ART-TOOL). São duas pesquisas: o trabalho de Chaim (1991) que apresenta uma ferramenta de teste estrutural, e o trabalho de Spoto (2000), que propõe um teste para programas de ABDR envolvendo banco de dados e teste estrutural de unidade.

CAPÍTULO 3

MODELOS PARA A AUTOMAÇÃO DO TESTE DE REGRAS ATIVAS

3.1 Introdução

Neste capítulo são propostos os modelos de fluxo de controle e de fluxo de dados a partir dos quais são abstraídos os elementos requeridos para o teste de regras ativas. Nos modelos são considerados os comandos de manipulação de dados, os comandos associados à estrutura de controle como seleção e laço e as rotinas de tratamento de exceção.

O modelo de instrumentação de uma regra ativa define a estrutura de coleta de informações dinâmicas que retratam o modelo proposto de fluxo de controle e de dados do programa durante a sua execução (Carniello, 2003). Para a obtenção de informações sobre os dados manipulados é proposto um modelo de instrumentação cujo objetivo é permitir a identificação não somente das tabelas afetadas durante a execução, mas também das *tuplas*.

As propostas dos modelos de fluxo de controle, de fluxo de dados, de instrumentação e de instrumentação de *tuplas* apresentadas neste capítulo podem ser vistas em detalhes em Leitão *et. al* (2002).

O suporte da linguagem SQL a regras ativas ocorre pela extensão dessa linguagem, onde construções são adicionadas para atender às demandas de conduta ativa. Basicamente comandos foram adicionados para definição de *triggers* e são estabelecidas diretrizes para a sua conduta em tempo de execução.

Embora as análises deste trabalho sejam conduzidas considerando o padrão SQL-3 (Fortier, 1999; Eisenberg e Melton, 2000; Standards SQL3, 1994; BNF-SQL3, 1998; BNF-SQL3, syntax, 1992), são explorados também alguns comandos da linguagem utilizada no ORACLE, a PL/SQL (Urman, 1997).

3.2 Modelo de Fluxo de Controle

O modelo de fluxo de controle de uma regra ativa foi inicialmente estudado por Chan *et al.* (1997), que mostram sua utilização na geração de dados de teste usando uma linguagem declarativa orientada a objetos para a definição de regras.

Neste trabalho as regras estão escritas no padrão SQL-3; as extensões da linguagem relacionadas com construções estruturadas são similares àquelas usadas em linguagem do tipo Algol. Portanto, além dos comandos de manipulação de dados persistentes, encontram-se comandos associados a estruturas de controle como seleção e laço.

Nós distintos são usados para representar o evento e a condição da regra no grafo de fluxo de controle. A ação da regra é representada por um subgrafo. Em regras ativas, um bloco B é definido como B = (D, S, E), sendo D o conjunto de variáveis declaradas de escopo local ao bloco; S representa os comandos que definem a tarefa do bloco; e E consiste nas rotinas de tratamento de exceções originadas pela execução do conjunto S de comandos. Essa definição de bloco é similar à definição de Haraty *et al.* (2002) para comando composto: um conjunto de comandos caracterizado por um único tratador de exceção, implícito ou explícito. Os autores exploram a manipulação de tabelas por diversos módulos, a qual cria dependências de fluxo de controle e relações de fluxo de dados entre os módulos; os dados gerados por um comando são usados por outros comandos no mesmo módulo ou em outros módulos.

Para representar uma regra ativa por meio de grafo de fluxo de controle considere R uma regra ativa e o grafo de fluxo de controle G = (N, E, e, c, x), onde: N indica o conjunto de nós; E o conjunto de arcos; e o nó de entrada que está associado ao evento da regra; c a condição (opcional) da regra e x o nó de saída. A Figura 3.1 mostra a estrutura

de controle de duas regras ativas, onde são ressaltados o disparo da regra ativa e o seu controle interno. Seja P um processo que possui uma operação θ capaz de provocar eventos de disparo de regras, onde P pode ser uma aplicação do usuário ou uma rotina gravada no banco de dados ou, mais especificamente, uma regra ativa da base de regras. Considere R_i e R_j, respectivamente, regras sem a presença de condição e com a presença de condição, cujos eventos de disparo podem ser provocados pela execução da operação θ. Particularmente, quando o evento de R_i é provocado, o controle é transferido do processo P para o escopo da regra R_i e retornado a P após a execução de R_i; o mesmo raciocínio é aplicado à regra R_j. Na estrutura de controle de R_j o nó e_j é o nó de entrada da regra, que representa o evento associado ao disparo da regra; o nó c_j, a condição da regra, significa uma seleção entre o subgrafo a_j e o nó x_j. O subgrafo a_j está associado à ação da regra e o nó x_j é o nó de saída da regra R_j.

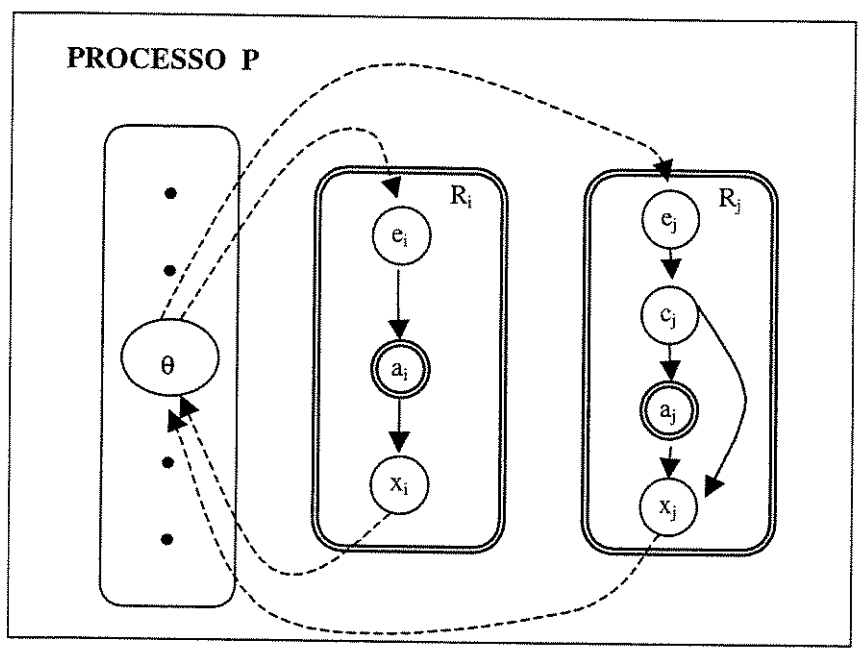


Figura 3.1: Grafo de fluxo de controle para regras ativas.

Toda a discussão sobre o fluxo de controle nas subseções seguintes refere-se aos recursos da linguagem SQL, para a construção do subgrafo associado à ação da regra, ressaltando que este é obtido pela concatenação das construções básicas apresentadas.

3.2.1 Fluxo de Controle para os Comandos de Manipulação de Dados

Cada ocorrência de comando de manipulação de dados (INSERT, UPDATE, DELETE e SELECT) é representada por um nó no grafo de fluxo de controle. Abordagem similar foi usada por Spoto (2000) e Haraty *et al.* (2002); no primeiro trabalho é alterada a definição de grafo de programas escritos em linguagem C, para acomodar os comandos executáveis de SQL em nós especiais, um comando em cada nó. Neste trabalho adotou-se esse modelo para melhor explorar as relações de fluxo de dados entre os comandos de manipulação em uma mesma regra ativa e em regras distintas. Subconsultas acopladas a estes comandos de manipulação de dados ficam embutidas no nó do comando em que estão contidas.

No conjunto dos comandos de manipulações de dados está também incluído o uso de cursores, uma seqüência de comandos pré-determinada que torna possível tratar um conjunto de *tuplas* resultante de uma consulta. Pode-se dizer que um cursor é um ponteiro que aponta para uma única *tupla* do resultado de um comando SELECT. O cursor é declarado como uma consulta na área de declarações, como mostrado na cláusula DECLARE da Figura 3.2; dentro da ação da regra ele é manipulado com o uso de alguns comandos. Sua declaração é sempre seguida de um comando de consulta, que só será executado no comando OPEN, onde todos os registros estarão disponíveis em memória. Para processar as *tuplas* referenciadas pelo cursor é necessário que o conteúdo seja transferido para variáveis; isto é feito pelo comando FETCH que também posiciona o ponteiro para a próxima *tupla*. Após sua manipulação o cursor deve ser fechado com o comando CLOSE para que a área de memória ocupada com esse processo seja liberada. Sintetizando, o comando OPEN refere-se ao processamento da consulta, o comando FETCH está ligado à recuperação de uma linha do conjunto resultante da consulta, e o comando CLOSE indica o encerramento do cursor (Figura 3.2). Os cursores podem ser explícitos ou implícitos; são denominados explícitos quando usam os comandos OPEN, FETCH e CLOSE, como mostrado na Figura 3.2. Já os implícitos não aplicam esses comandos e manipulam o conjunto resultante da consulta por meio do comando FOR no padrão SQL-3, que pode ser visto no exemplo da Figura 3.3.

Os comandos OPEN, FETCH e CLOSE, implícitos ou não, são também mapeados para nós distintos pois estão ligados à consulta atribuída ao cursor. Na Figura 3.2 está ilustrado um grafo de fluxo de controle de um cursor explícito específico da linguagem PL/SQL, sendo o nó **o** o comando OPEN; o nó **f** do subgrafo S_1, representando o comando FETCH; e o nó **x,** o nó final da consulta, o comando CLOSE. Os nós **l** e **e** são representação do laço, comando LOOP, e da saída do laço, respectivamente.

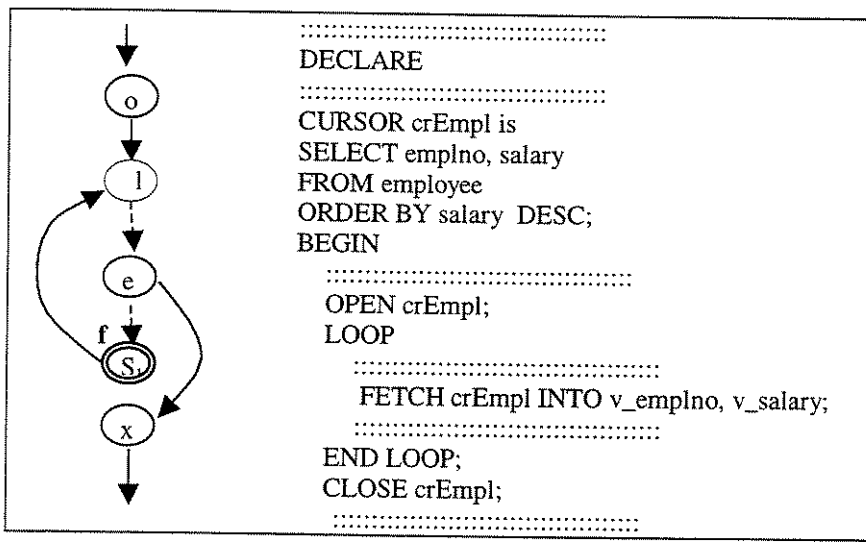


Figura 3.2: Representação do fluxo de controle do cursor explícito.

Na abstração de fluxo de controle do cursor implícito é possível observar a inclusão de nós para representar os comandos identificados nos cursores explícitos. Um exemplo desse cursor está na Figura 3.3, onde os nós **o**, **f**, $\mathbf{f_1}$ e **x** representam os comandos OPEN, FETCH, FETCH e CLOSE, respectivamente. Pode-se notar no código apresentado ao lado do fluxo de controle que os nós **o**, **f**, e **x** são nós adicionados para representar os comandos de cursores. O comando FETCH é representado pelo nó **o**, para indicar a *tupla* de iniciação do laço, e **f**, para indicar a próxima *tupla* dentro do laço.

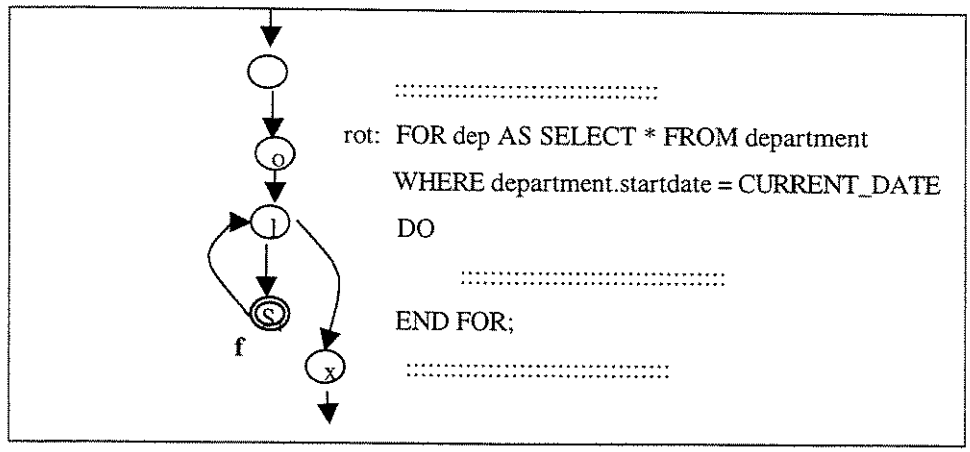


Figura 3.3: Representação do fluxo de controle do cursor implícito.

3.2.2 Fluxo de Controle para os Comandos Estruturados

O comando de seleção IF é representado no grafo de fluxo de controle da Figura 3.4, onde estão ilustradas as várias possibilidades condicionais. O comando de seleção múltipla, o CASE, apresenta dois casos distintos: no primeiro a condição está anexada à cláusula CASE, podendo a condição possuir uma subconsulta, como mostra a Figura 3.5; no segundo, a condição está anexada à cláusula WHEN, representado na Figura 3.6.

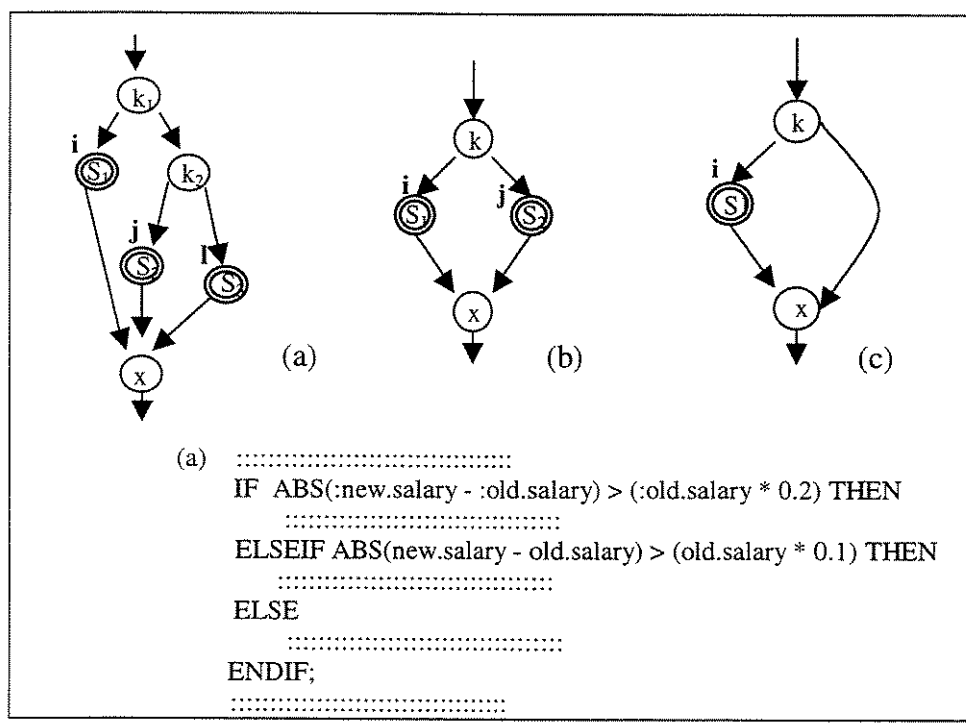


Figura 3.4: Representação do fluxo de controle do comando IF.

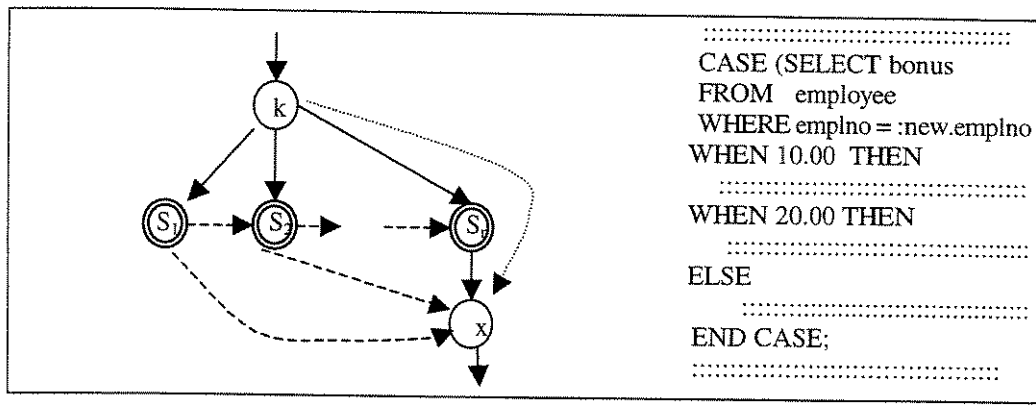


Figura 3.5: Representação do fluxo de controle do comando CASE (caso1).

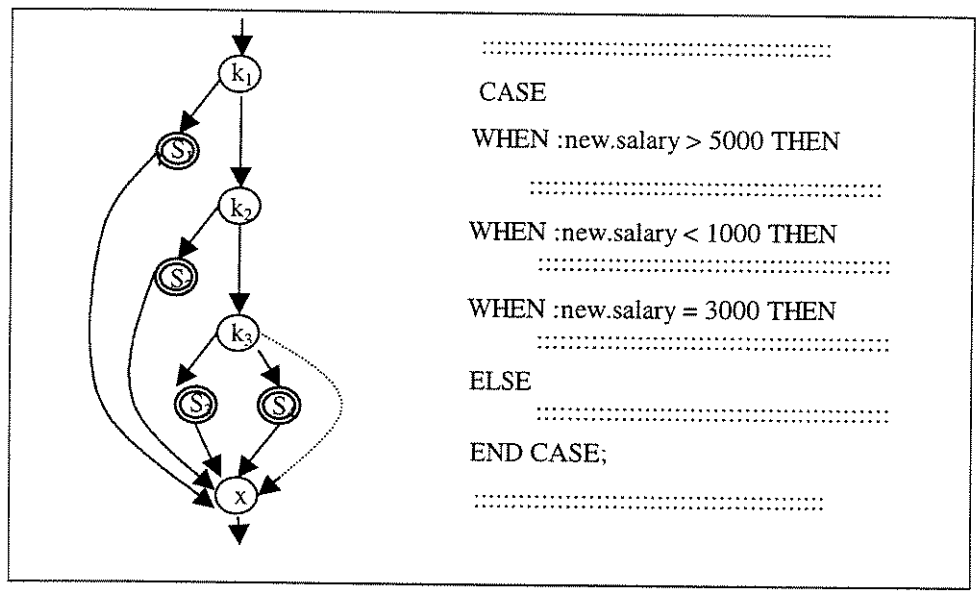


Figura 3.6: Representação do fluxo de controle do comando CASE (caso2).

Os comandos de iteração WHILE, LOOP e REPEAT são mostrados nas Figuras 3.7, 3.8 e 3.9, respectivamente. Vale ressaltar que o comando LOOP requer um comando de desvio para abandonar o laço, pois não possui predicado para encerrar a iteração. Uma construção do comando FOR, específica da PL/SQL e diferente daquela dedicada a cursores implícitos, é ilustrada na Figura 3.10.

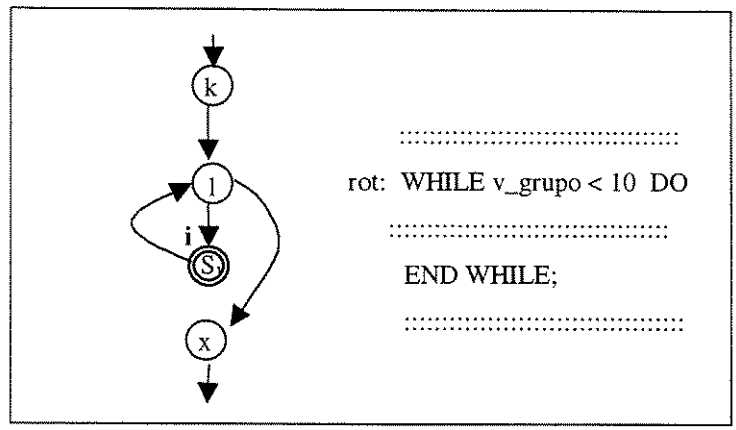


Figura 3.7: Representação do fluxo de controle do comando WHILE.

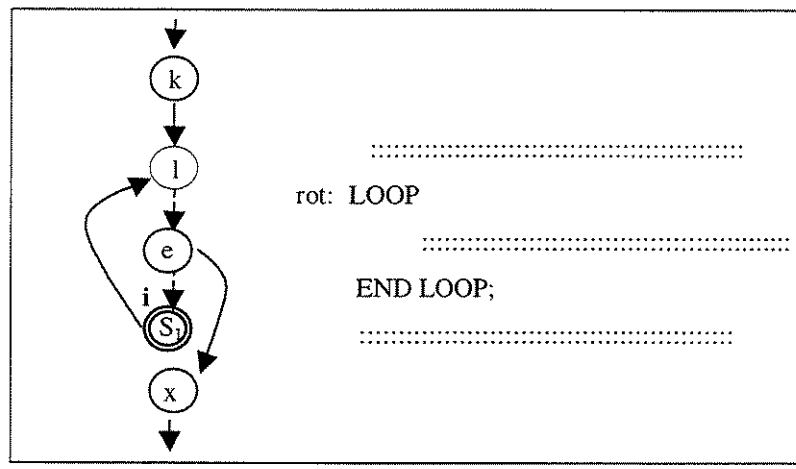


Figura 3.8: Representação do fluxo de controle do comando LOOP.

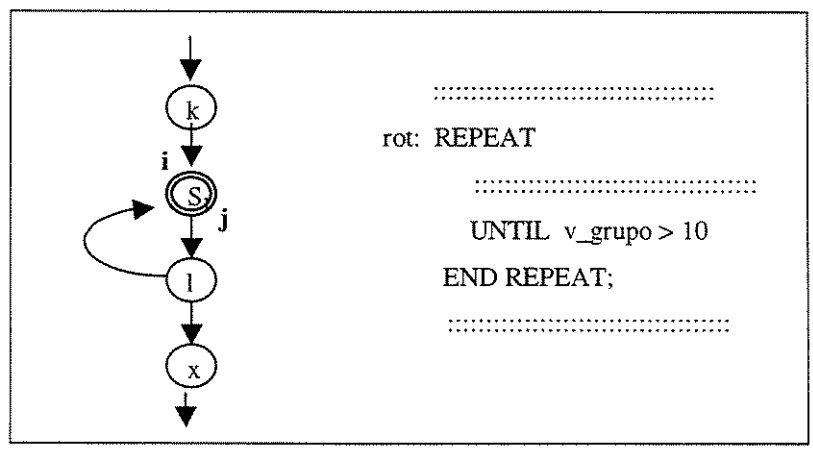


Figura 3.9: Representação do fluxo de controle do comando REPEAT.

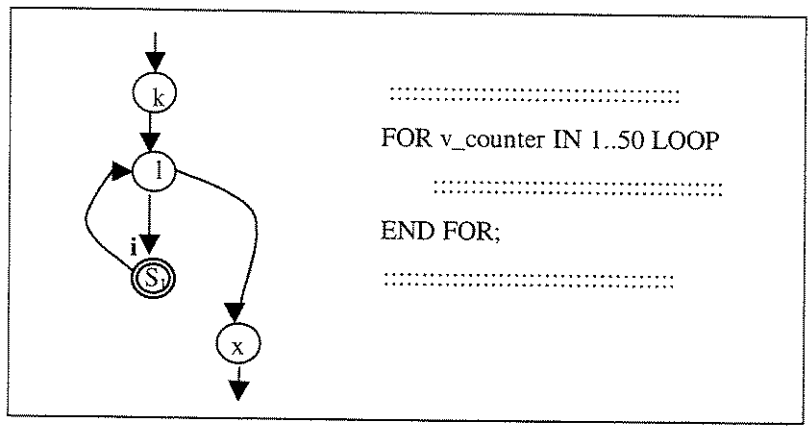


Figura 3.10: Representação do fluxo de controle do comando FOR (PL/SQL).

Os comandos de desvio incondicional provocam a mudança do fluxo de execução em um programa. Os comandos SIGN e RAISE desviam o controle para a área de exceção, que será detalhada mais adiante nesta seção. O SIGN é específico do padrão SQL-3 e o RAISE da linguagem PL/SQL. Os comandos LEAVE e EXIT têm a função de interromper a execução de um comando de repetição; o LEAVE é exclusivo do padrão SQL-3 e o EXIT da linguagem PL/SQL; o EXIT pode vir associado a uma condição, que permite interromper o laço somente quando for satisfeita. O comando RETURN desloca para o comando END, final da ação da regra, e o comando GOTO é utilizado para desviar o fluxo normal dos comandos para determinado ponto endereçado por um rótulo; esses comandos têm o objetivo de encerrar a execução da estrutura ao qual estão incluídos, definida por um comando condicional ou por um laço. A transferência de controle atribuída aos comandos de desvio depende basicamente do local em que eles ocorrem; por exemplo, encerrar um bloco, ir para uma rotina especificada, deslocar-se para o comando subseqüente ou, ainda, sair de um laço mais interno ou mais externo.

3.2.3 Fluxo de Controle para o Tratamento de Exceções

Em regras ativas as exceções podem ser implícitas, quando definidas pelo sistema, ou explícitas, quando definidas pelo usuário. No primeiro caso, o sistema pode detectar uma exceção em qualquer comando de manipulação de dados, incluindo cursores, e, no

último caso, um tipo de exceção é definido pelo usuário e deverá ser explicitamente provocado pelo programador.

Cada bloco pode ter uma única área de tratamento de exceção, a qual deverá situar-se no final do bloco. Os blocos podem ser aninhados dentro da ação de uma regra. Uma vez que o controle é desviado para a área de tratamento de exceção, ele não retorna para o bloco em que está inserido, pois o tratamento de exceção é processado e o controle prossegue no bloco imediatamente mais externo. Várias exceções podem ser tratadas em um único bloco, onde a cláusula OTHERS habilita o tratamento de exceções que não foram explicitamente tratadas. Um exemplo de bloco de tratamento de exceção no corpo da ação da regra ativa é apresentado na Figura 3.11.

```
DECLARE USER_EXC EXCEPTION;
        ::::::::::::::::::::::::
BEGIN
        ::::::::::::::::::::::::
        ::::::::::::::::::::::::
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        ::::::::::::::::::::::
    WHEN USER_EXC THEN
        ::::::::::::::::::::::
    WHEN OTHERS THEN
        ::::::::::::::::::::::
END;
```

Figura 3.11: Exemplo de bloco para tratamento de exceção.

Sempre que uma exceção ocorre o seu tratamento é realizado dentro do bloco em que foi provocada mas, se não houver tratamento específico neste bloco, o controle é desviado para a área de tratamento em blocos mais externos, a não ser que esteja incluída a cláusula OTHERS, pois esta evita que o desvio ocorra. Resumidamente, quando ocorre uma exceção, o sistema prioritariamente busca um tratamento de exceção dentro do próprio bloco onde ocorreu a exceção; se não encontrar, vai buscar em blocos mais externos. Qualquer exceção provocada nas rotinas de tratamento de exceção, independentemente do nível do bloco, resultará em um erro na execução da regra; a primeira exceção tratada na regra será retornada ao processo que possui a operação que

provocou o evento de disparo da regra. Um exemplo de tratamento de exceção aninhado é mostrado na Figura 3.12, onde pode-se observar que uma ocorrência de exceção no bloco mais interno que necessite ser tratada no bloco externo evita que o controle alcance a área (*) localizada na figura.

```
DECLARE USER_EXC EXCEPTION;
        ::::::::::::::::::::
BEGIN
        ::::::::::::::::::::
        DECLARE
            ::::::::::::::::::::
        BEGIN
            ::::::::::::::::::::
        EXCEPTION
            WHEN TOO_MANY_ROWS THEN
                ::::::::::::::::::::
        END;
        ::::::::::::::::::::          (*)
        ::::::::::::::::::::
EXCEPTION
        WHEN USER_EXC THEN
            ::::::::::::::::::::
        WHEN OTHERS THEN
            ::::::::::::::::::::
END;
```

Figura 3.12: Exceção em blocos aninhados.

Exceções definidas pelo usuário requerem a declaração explícita de variáveis do tipo *exception* e são provocadas pelo comando RAISE. A estrutura de controle para este tipo de exceção é diferente daquela atribuída a exceções definidas pelo sistema. Esta estrutura é observada na Figura 3.13, onde os nós **s**, **u**, **i** e **z** representam os comandos que definem a ação de uma regra ativa; **x** é o nó de saída; o nó **m**, juntamente com os subgrafos $S_1...S_n$ e S_u, definem a área de exceção; e os subgrafos $S_1...S_n$ e S_u são as rotinas do tratamento de exceção. Se a execução do comando atribuído ao nó **s** resultar em exceção, ocorre desvio de controle para a área de exceção, ou seja, para o nó **m**. O fluxo de controle, após chegar ao nó **m**, é dependente do tipo de exceção que ocorreu para a escolha de um subgrafo que trata a exceção. Se nenhum subgrafo associado a tratamento específico de exceção for escolhido, o controle é transferido para o subgrafo definido pela cláusula OTHERS, caso exista. Em outra parte, sendo o nó **u** um comando do tipo RAISE, o controle é desviado para um subgrafo específico S_u, pois é uma rotina

de tratamento de exceção definida pelo usuário. Após a execução das rotinas de tratamento de exceção, independentemente de serem ou não provocadas pelo usuário, o controle é transferido para o nó **x**. Quando o processamento do bloco ocorrer sem que nenhuma exceção tenha sido provocada, ao alcançar o nó **z** o controle é desviado para o nó de saída do bloco, não sendo executado o mecanismo de tratamento de exceção.

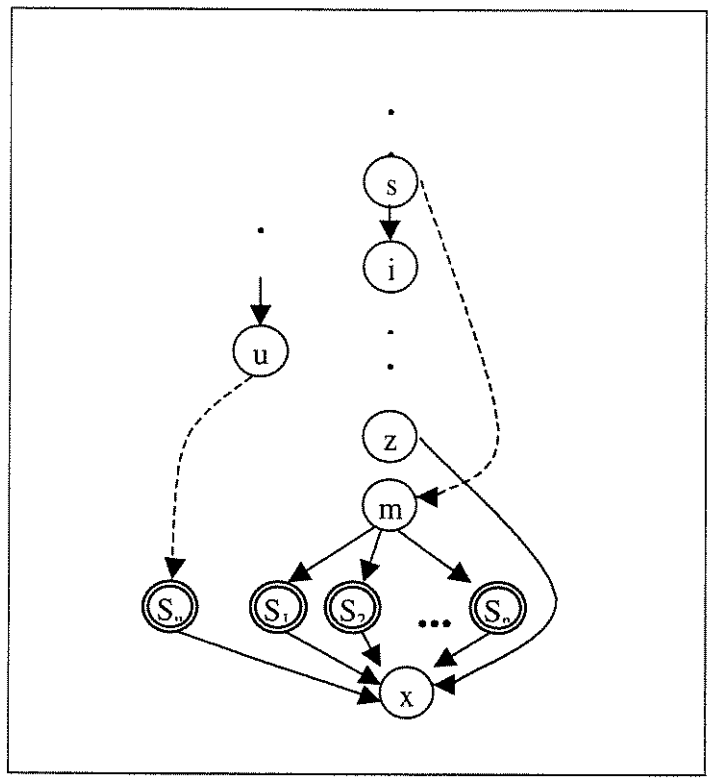


Figura 3.13: Estrutura de controle para tratamento de exceções.

Segundo Haraty *et. al* (2002), programação de exceção complica as dependências de fluxo de controle, resultando na construção de grafos de fluxo de controle de módulos de banco de dados diferentes dos de programas convencionais. Em regras ativas a natureza seqüencial de comandos de manipulação é alterada quando existe uma área para tratamento de exceção. Um comando de manipulação ou cursor pode resultar em exceção que, por sua vez, desvia o controle para as rotinas de exceção.

3.3 Modelo de Instrumentação

No teste estrutural é necessário fazer o monitoramento da execução do programa ou da regra em teste para rastrear a execução do caso de teste ("*trace*"); o "*trace*" é uma informação imprescindível para a avaliação do teste. Nesta análise dinâmica devem ser tomadas as devidas providências para obter informações sobre o percurso de nós da unidade em teste para os casos de teste executados. Para acompanhar a execução do programa é necessário que o código da unidade em teste seja alterado por meio de comandos inseridos. Esta versão instrumentada do programa é útil também para a depuração, manutenção e geração de dados de teste. Os comandos inseridos no programa são denominados pontas-de-prova e são colocados em pontos apropriados; possuem a função de registrar os caminhos percorridos em um arquivo de saída.

A inserção de pontas de prova considera os nós conforme o modelo de fluxo de controle proposto. Cada comando de escrita indica um único nó executado. Com estas informações é possível saber quais comandos foram executados, em que ordem e o número de vezes. Essas informações são importantes na avaliação de cobertura de critérios baseados em análise de fluxo de dados e na análise de adequação de um conjunto de casos de teste.

A instrumentação de regras ativas segue os padrões de instrumentação de um programa procedimental. Considerando uma regra completa, sua instrumentação deve abranger as três partes: evento, condição e ação.

Como o nó do evento não permite a inclusão de comandos de escrita, a instrumentação só é possível na ação da regra; a ponta de prova que indica a execução do evento é inserida no início da ação. Considere a regra $\mathbf{R_1}$, ilustrada na Figura 3.14, onde: **e** é o nó que representa o evento; **c** o nó que representa a condição; **x** o nó de saída; e o corpo ou ação da regra é representado pelo subgrafo $\mathbf{A_1}$. A ponta de prova ***e*** é inserida como primeiro comando da ação e a sua ativação indica a ocorrência do evento.

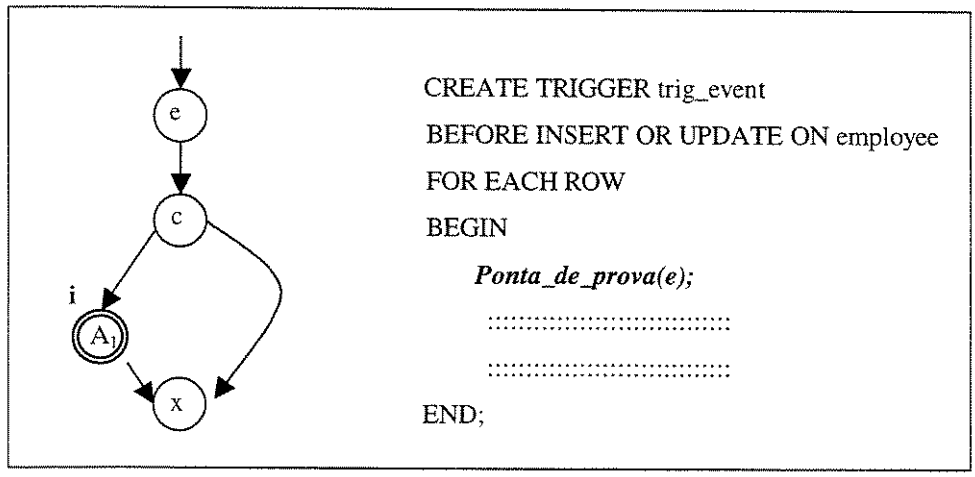
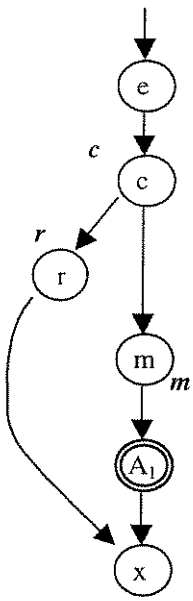


Figura 3.14: Representação do fluxo de controle e instrumentação do evento.

A condição de regra, representada em um nó posterior ao evento, também não permite a inclusão de comandos de escrita; vale ressaltar que se a condição não for satisfeita, a ação da regra não será executada. Similarmente à instrumentação do evento, a instrumentação da condição de regra ocorre no início da ação de regra, expressa em um comando IF com a negação da condição. Dentro do corpo deste IF é posto um comando RETURN para que o fato de a condição não ser satisfeita seja indicada. Na Figura 3.15 o nó **e** representa o evento; o nó **c** é associado à condição do comando IF; **r** é o nó do comando RETURN; e **m** é o nó final da condição, o END IF. A ação da regra é representada pelo subgrafo $\mathbf{A_1}$ e **x** é o nó de saída da regra. A instrumentação insere as pontas de prova ***c*** e ***r*** no início dos respectivos nós e a ponta de prova ***m*** no final do nó **m**, para informar a execução da condição da regra.

A instrumentação da ação da regra segue o modelo de instrumentação apresentado por Chaim (1991) para comandos de seleção, laço e desvios condicional e incondicional. No caso de comandos de manipulação de dados a instrumentação se restringe à inserção do comando de escrita antes de cada nó. As Figuras 3.9 a 3.19 ilustram a instrumentação de alguns comandos, onde as pontas de prova que estão em negrito são a especificação de cada comando.




```
CREATE TRIGGER trig_condition
BEFORE INSERT OR UPDATE ON employee
    FOR EACH ROW
    WHEN (new.salary > 3000)
    BEGIN
        Ponta_de_prova(evento);
        Ponta_de_prova(c);
        IF not (new.salary > 2000) THEN
                Ponta_de_prova(r);
                Ponta_de_prova(end);
                RETURN;
        END IF;
    Ponta_de_prova(m);
    ::::::::::::::::::::::::::::::::::::
    ::::::::::::::::::::::::::::::::::::
    END;
```

Figura 3.15: Representação do fluxo de controle e instrumentação da condição.

3.3.1. Instrumentação dos Comandos Estruturados

Para instrumentar o comando IF, representado na Figura 3.16(a), considere **i**, **j** e **l** os nós de entrada dos subgrafos S_1, S_2 e S_3, respectivamente, e **x** o nó de saída. No início dos nós $\mathbf{k_1}$, **i**, **j** e **l** são inseridas as pontas de prova k_1, i, j e l, que indicam a execução desses nós. Também no início dos blocos **j** e **l** é inserida a ponta-de-prova k_2 indicando a execução do nó $\mathbf{k_2}$. Para encerrar a instrumentação desta estrutura, no final do nó **x** é inserida a ponta de prova x. O código instrumentado da Figura 3.16, refere-se à Figura 3.16(a). A Figura 3.16(b) mostra o comando IF com as cláusulas ELSE e THEN e a Figura 3.16 (c) o comando IF somente com a cláusula THEN.

O comando CASE tem dois modos de representação que devem ser instrumentados de maneiras distintas. No primeiro caso, como mostra a Figura 3.17, a cláusula CASE está associada a uma condição. Considere $\mathbf{i_1}$, $\mathbf{i_2}$ e $\mathbf{i_n}$ os nós de entrada dos subgrafos S_1, S_2 e S_n, respectivamente. No início dos nós **k**, $\mathbf{i_1}$, $\mathbf{i_2}$ e $\mathbf{i_n}$ são inseridas as pontas de prova k, i_1, i_2 e i_n que indicam a execução desses nós; a ponta de prova x é incluída no início do nó **x**.

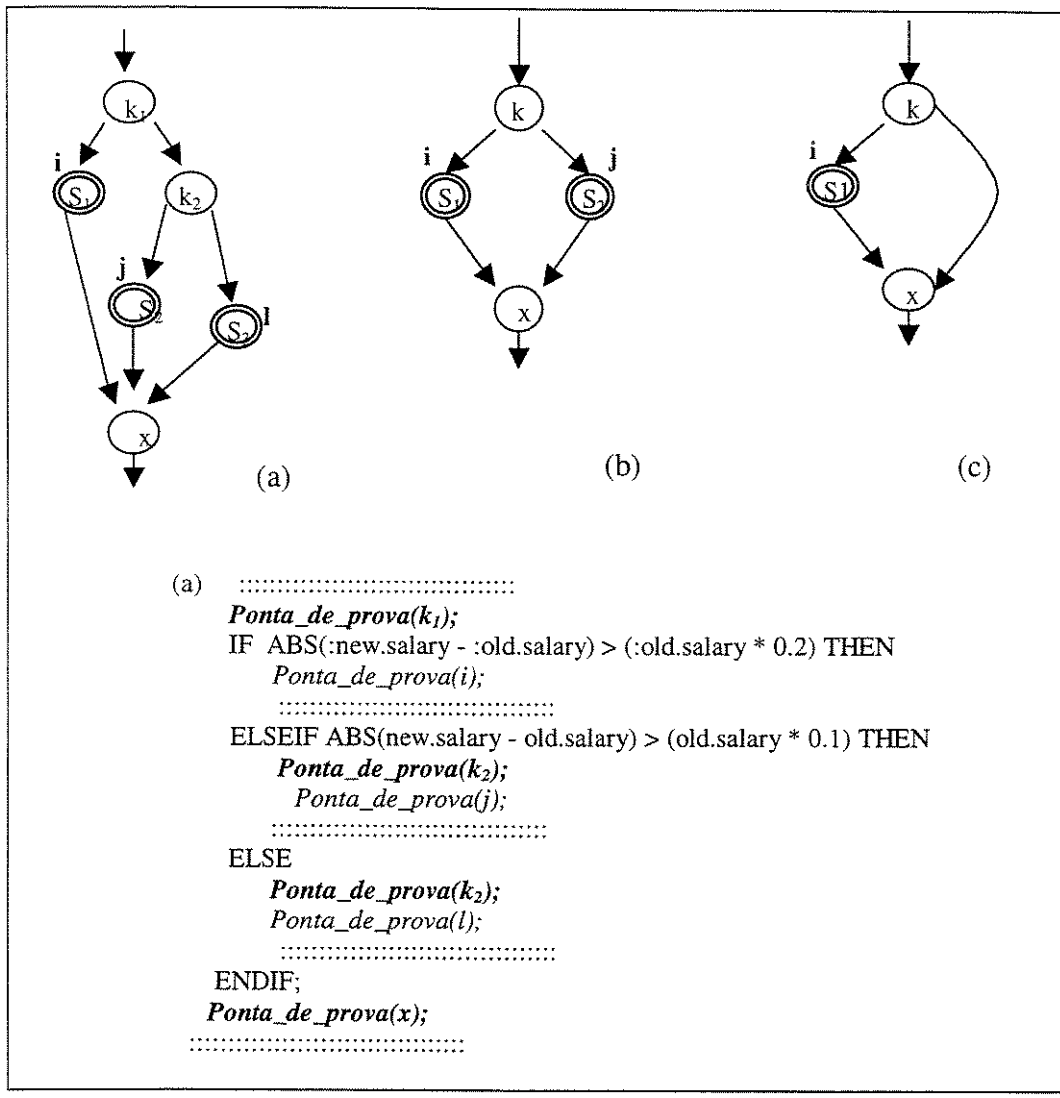


Figura 3.16: Comando IF: fluxo de controle e instrumentação.

No segundo caso, a condição do comando não está associada à cláusula CASE e sim à cláusula WHEN. Na Figura 3.18, $\mathbf{k_1}$, $\mathbf{k_2}$ e $\mathbf{k_3}$ são as cláusulas de decisão; **i**, **j**, **k** e **l** são os nós de entrada dos subgrafos S_1, S_2, S_3 e S_4, respectivamente. No início dos nós $\mathbf{k_1}$, **i**, **j**, **k** e **l** são inseridas as pontas de prova k_1,*i*, *j*, *k* e *l* que indicam a execução desses nós, a ponta de prova do nó **x** é incluída no final desse nó.

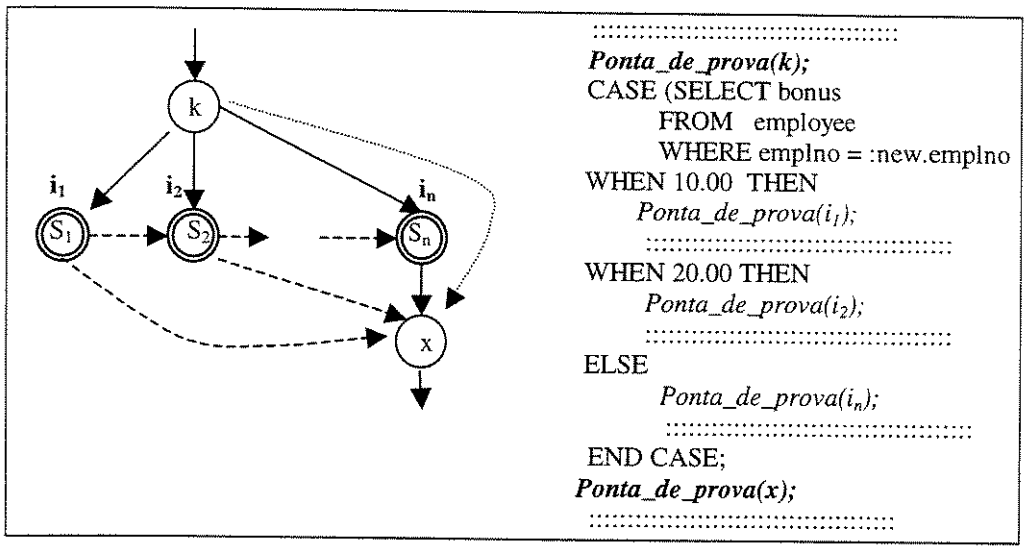


Figura 3.17: Comando CASE (caso1): fluxo de controle e instrumentação.

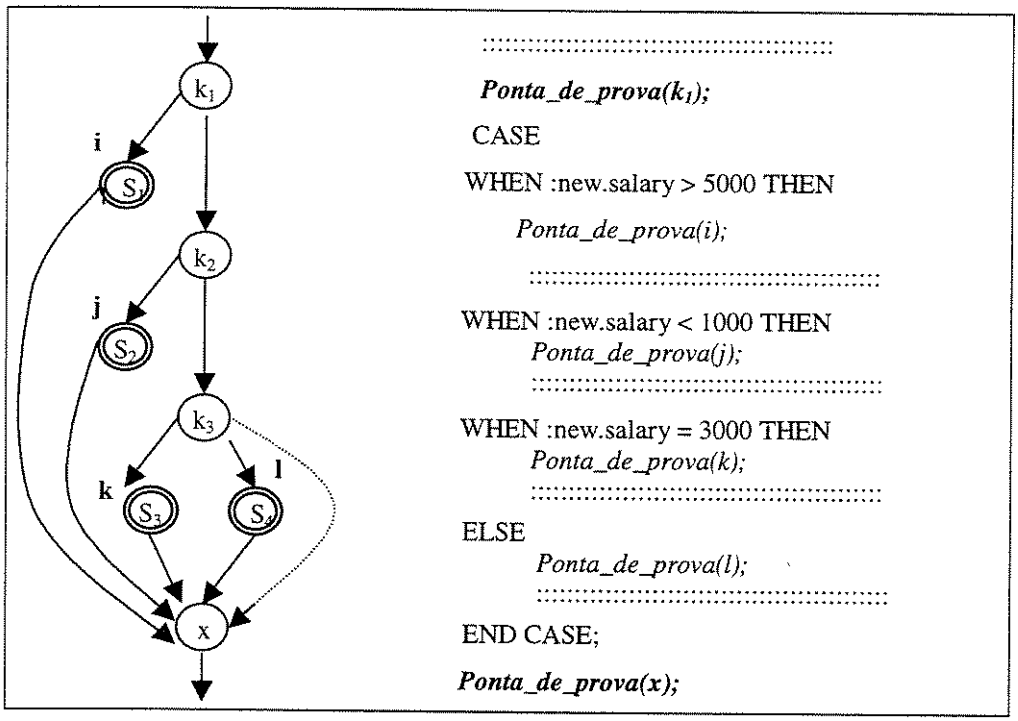


Figura 3.18: Comando CASE (caso2): fluxo de controle e instrumentação.

A instrumentação do comando WHILE é realizada de acordo com a Figura 3.19. No início do nó **k** é inserida a ponta de prova ***k***, que indica a execução do nó que possivelmente define ou atribui valores às variáveis da condição do laço. No corpo do

laço, considere **i** o nó de entrada do subgrafo S_1; neste nó são inseridas as pontas de prova ***l*** e ***i***, onde o nó **l** representa a condição de controle do laço; após o comando do final do laço, o nó **x**, devem ser inseridas as pontas de prova ***l*** e ***x*** que, indicam, respectivamente, a condição de controle e o final do laço.

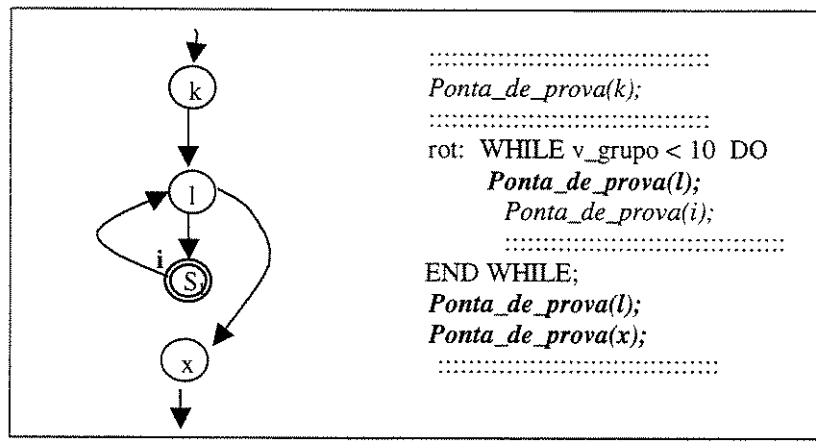


Figura 3.19: Comando WHILE: fluxo de controle e instrumentação.

O grafo da Figura 3.20 representa o comando REPEAT. Sendo **i** e **j** os nós de entrada e saída do subgrafo $\mathbf{S_1}$, as pontas de prova ***i***, ***j*** e ***x*** são inseridas no início dos respectivos nós **i**, **j** e **x**, e no final do nó **j** é também inserida a ponta de prova ***l***, para marcar a condição de controle do laço.

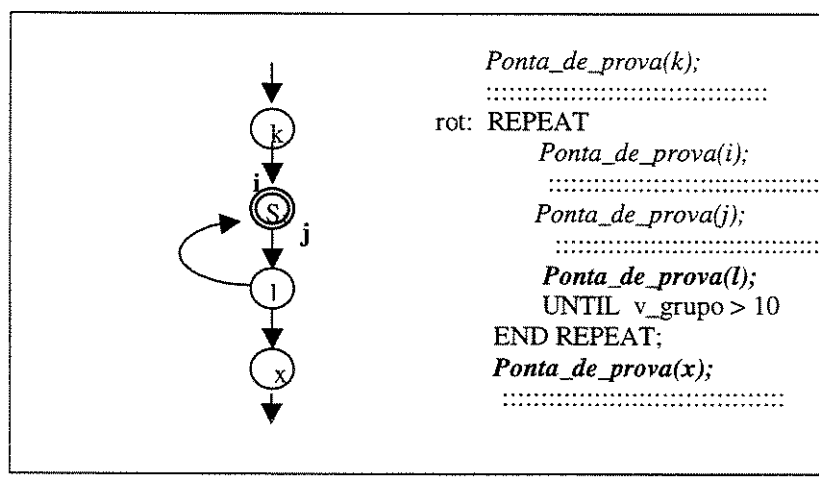


Figura 3.20: Comando REPEAT: fluxo de controle e instrumentação.

O comando FOR do padrão SQL-3, que indica um cursor implícito, é instrumentado considerando os comandos do cursor explícito, de acordo com o fluxo de controle proposto na Seção 3.2.1. Na Figura 3.21, considere o nó **o** comando OPEN do cursor (nó adicionado); o nó **f** o comando FETCH (nó adicionado) para a primeira *tupla*; e o nó $\mathbf{f_1}$ que indica a próxima *tupla* selecionada. No início do nó **o** são inseridas as pontas de prova ***o*** e ***f***. No início do nó **i**, o primeiro nó do subgrafo S_1, são inseridas as pontas de prova ***i*** e ***l***, que indica o início do laço. No final do corpo do laço é inserida uma ponta de prova para o nó $\mathbf{f_1}$. Após o comando do final do laço, o comando END FOR, são inseridas as pontas de prova ***l*** e ***x***, representando o final deste comando.

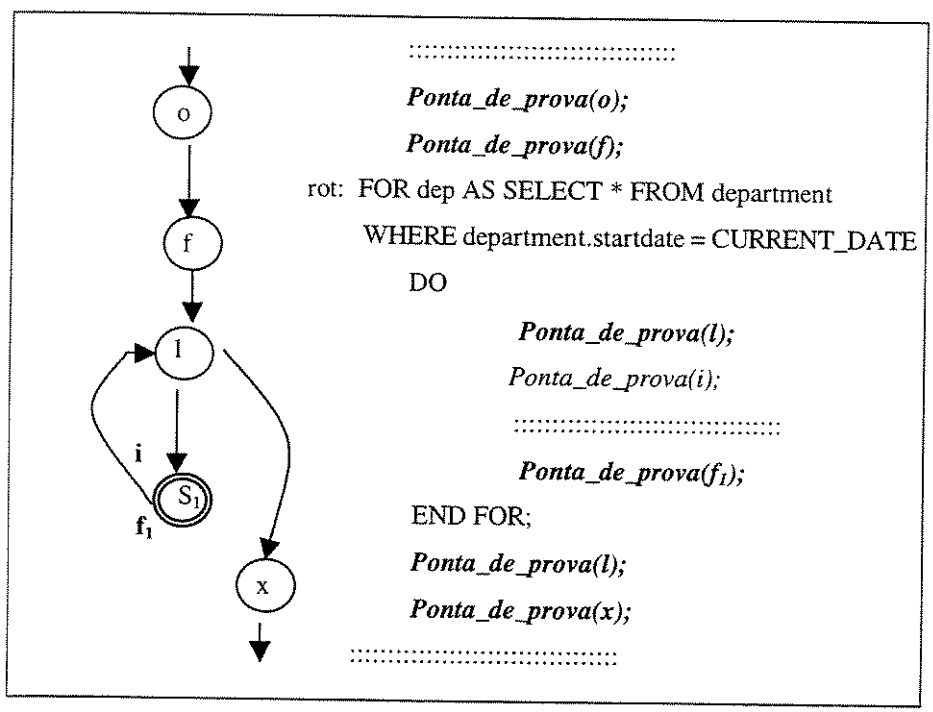


Figura 3.21: Comando FOR: fluxo de controle e instrumentação.

A instrumentação do comando FOR, utilizado na linguagem PL/SQL, está representado na Figura 3.22. Considere que o nó **k** refere-se ao nó de iniciação da variável de controle do laço, o nó **l** à condição de controle do laço e o nó **j** ao incremento da variável de controle. Os nós **i** e **j** pertencem ao subgrafo S_1, nós de entrada e saída respectivamente. No início do nó **k**, é inserida a ponta de prova ***k***; e no início do nó **i** são

inseridas as pontas de prova *l* e *i*. No final do nó **j** é inserida a ponta de prova *j*; e no nó **x** são inseridas as pontas de prova *l* e *x*.

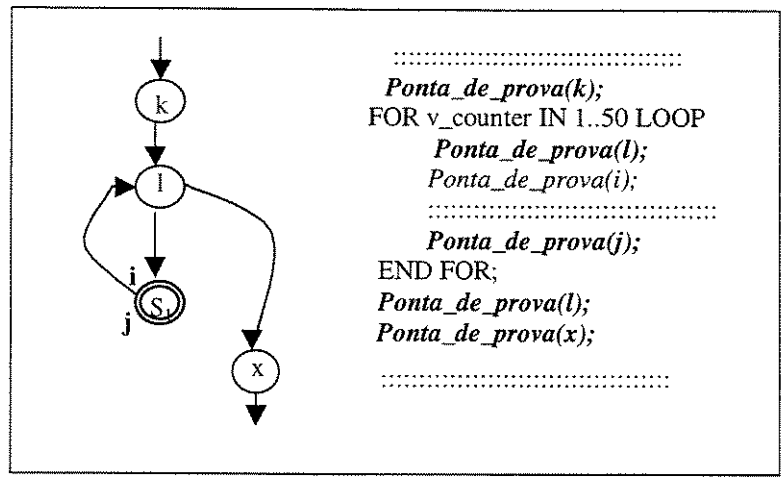


Figura 3.22: Comando FOR (PL/SQL): fluxo de controle e instrumentação.

A Figura 3.23 apresenta as pontas de prova que devem ser inseridas dentro e fora do laço para o comando LOOP. No início do nó **k,** antes do início do laço, é inserida a ponta de prova *k*. O nó **l**, que indica o início do laço, deve ser finalizado com a ponta de prova *l*. Considere **i** e **j** os nós de entrada e saída do subgrafo S_1, no início desses nós são inseridas as pontas de prova *i* e *j*, respectivamente. No início do nó **e**, que desvia o laço, deve ser inserida uma ponta de prova *e*. No final do nó **x**, que finaliza o laço, são inseridas as pontas de prova *l* e *x*.

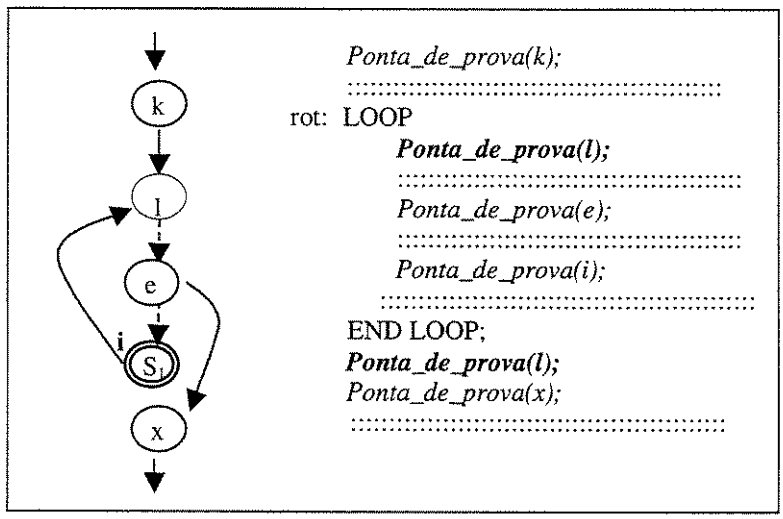


Figura 3.23: Comando LOOP: fluxo de controle e instrumentação.

Na Figura 3.24 está ilustrada a instrumentação de alguns comandos de desvio incondicional que podem ser encontrados na ação de uma regra ativa. Na Figura 3.24(1) é apresentado o comando GOTO; para caracterizar a execução deste comando de desvio a ponta de prova deve ser incluída antes da linha do comando. O mesmo procedimento pode ser usado para os comandos RAISE e EXIT, mostrados na Figura 3.24 (2) e 3.24 (3), respectivamente. Já para o comando RETURN, por este provocar desvio para o fim da ação da regra, deve ser inserida adicionalmente a ponta de prova que caracteriza o final da regra, como mostra a Figura 3.24 (4).

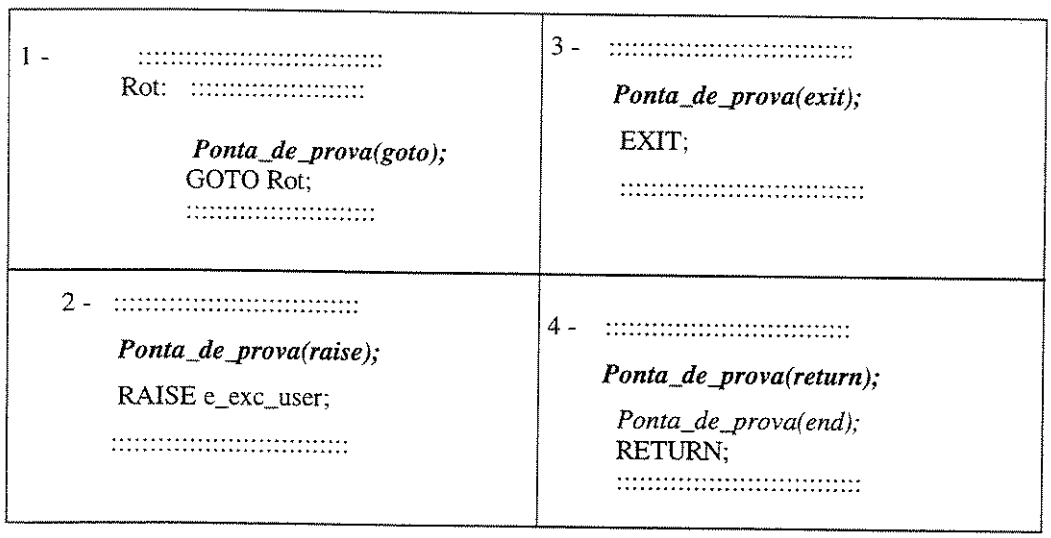
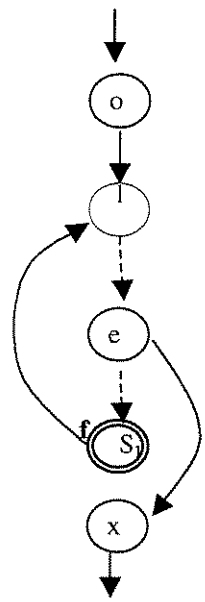


Figura 3.24: Instrumentação dos comandos de desvios incondicionais.

3.3.2 Instrumentação do Mecanismo de Cursor e do Bloco de Exceção

A instrumentação do mecanismo de cursor explícito segue o exemplo ilustrado na Figura 3.25 que neste caso inclui um laço determinado pelo comando LOOP. No início do nó **o**, que representa o comando OPEN, é inserida a ponta de prova ***o***. Considere **f** o nó de entrada do subgrafo S1, que representa o comando FETCH, neste nó é inserida a ponta de prova ***f***. No início do nó **x**, que indica o comando CLOSE do cursor, é inserida a ponta de prova ***x***.




```
::::::::::::::::::::::::::::::::::::::::
DECLARE
::::::::::::::::::::::::::::::::::::::::
CURSOR crEmpl is
SELECT emplno, salary
FROM employee
ORDER BY salary  DESC;

BEGIN
    ::::::::::::::::::::::::::::::::::::::::
    Ponta_de_prova (o);
    OPEN crEmpl;

LOOP
        Ponta_de_prova (l);
        ::::::::::::::::::::::::::::::::::::::::
        Ponta_de_prova (f);
        FETCH crEmpl INTO v_emplno, v_salary;

        ::::::::::::::::::::::::::::::::::::::::
END LOOP;
Ponta_de_prova (l)
Ponta_de_prova (x)
CLOSE crEmpl;

::::::::::::::::::::::::::::::::::::::::
```

Figura 3.25: Representação do fluxo de controle e instrumentação do cursor explícito.

O grafo de controle para o mecanismo de tratamento de exceção, localizado a partir da cláusula EXCEPTION, é representado na Figura 3.26. O nó **m** mapeia a condição de seleção de exceção e S_1, S_2, ...S_n e S_u são os subgrafos correspondentes às rotinas de tratamento de exceção. Considere $\mathbf{i_1}$, $\mathbf{i_2}$,...$\mathbf{i_n}$ e $\mathbf{i_u}$ os nós iniciais dos subgrafos S_1, S_2, ...S_n e S_u, respectivamente; no início desses nós devem ser inseridas pontas de prova i_1, i_2,..., i_n e i_u, que indicam o percurso da rotina de tratamento de exceção selecionada. Após executada a rotina selecionada, ocorre o desvio para o nó final; este fato é registrado pela inserção de ponta de prova no final de cada rotina de exceção do sistema.

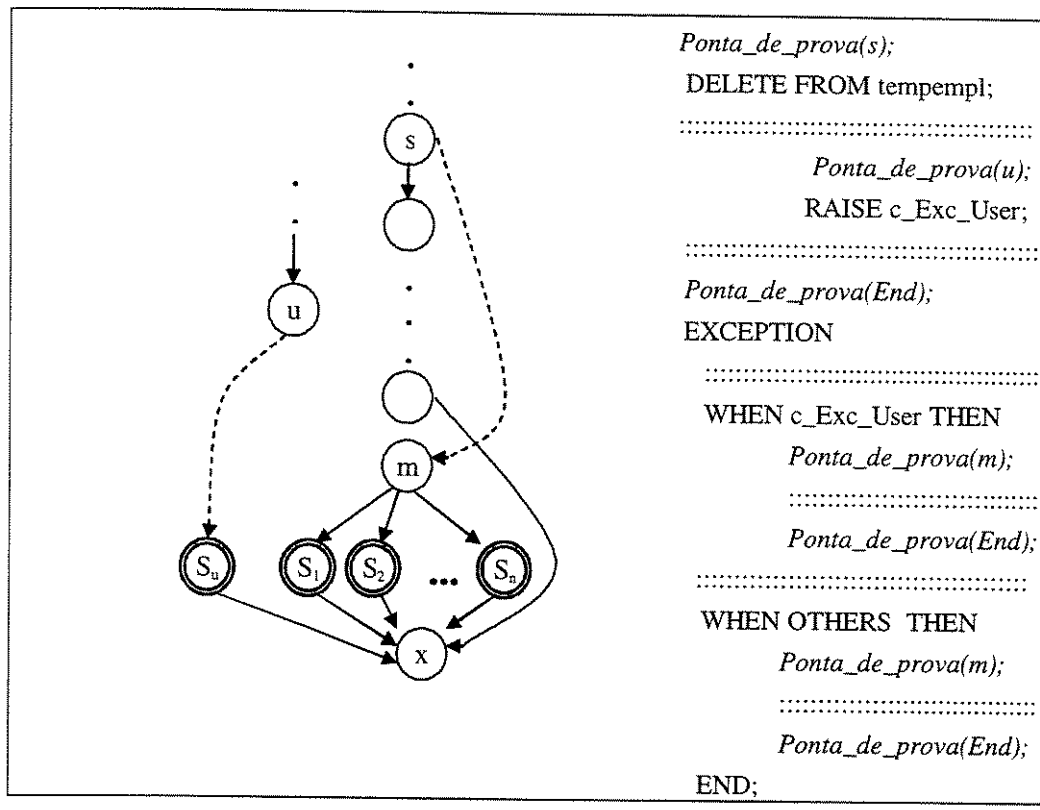


Figura 3.26: Instrumentação do mecanismo de tratamento de exceção.

3.3.3 Instrumentação Incluindo o Bloco de Exceção

Exceções na manipulação de dados podem ocorrer nos comandos INSERT, DELETE, UPDATE, SELECT e nos comandos de cursor (OPEN, FETCH e CLOSE). Quando isto ocorre o gerenciador do banco de dados desvia o controle para a área de tratamento de exceção. Exemplos de exceções são: um comando SELECT deveria resultar em uma única *tupla* mas produz um resultado com várias *tuplas*; um comando INSERT tenta armazenar um valor duplicado em uma coluna de uma tabela que possui chave única ou primária e comandos do cursor tentam executar operações ilegais, como fechar um cursor que não esteja aberto. Existem casos em que a execução de regra é abortada quando exceções são provocadas; por exemplo, a regra não possui área de tratamento de exceção; ou a regra possui área de tratamento de exceção mas nenhuma rotina de exceção é prevista para o tipo de exceção. Se a regra abortar, a instrumentação

da regra é prejudicada pois as atualizações aplicadas ao banco de dados serão desfeitas, incluindo as aplicadas na tabela que grava informações sobre a instrumentação. Para a prevenção desses casos, impedindo que a regra seja abortada e para tornar possível a instrumentação até o ponto em que ocorra um desvio para o final da regra, é incluída uma rotina de exceção capaz de tratar qualquer exceção não prevista.

A inclusão desta rotina de exceção é realizada em regras que não possuam rotina de exceção do tipo OTHERS. Na Figura 3.27 está ilustrada a instrumentação de um comando DELETE, com a inserção da ponta de prova *C* e a inclusão do bloco de exceção. Neste caso, como não havia bloco de exceção, foi inserida a cláusula EXCEPTION apenas com a rotina de exceção do tipo OTHERS.

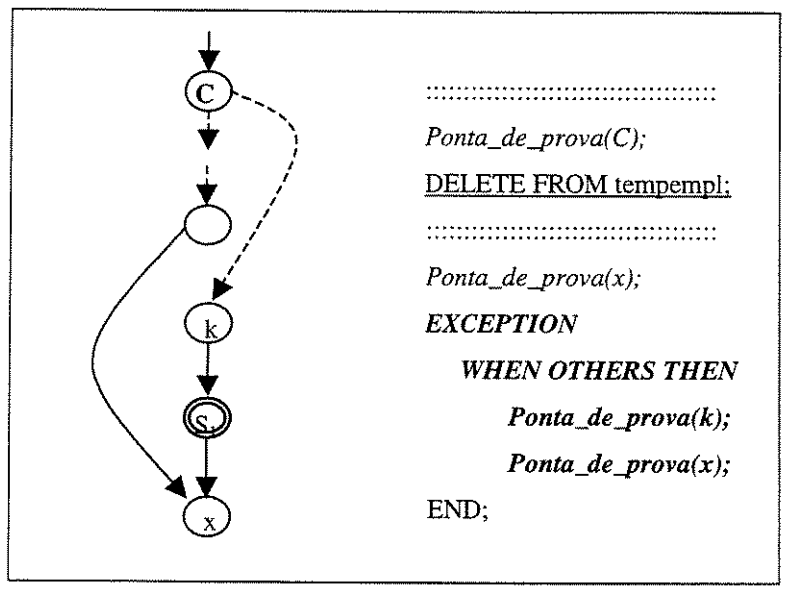


Figura 3.27: Instrumentação do comando seqüencial com inserção do bloco de exceção

3.3.4 Modelo de Identificação das *Tuplas*

Apesar de os comandos de manipulação e consulta serem instrumentados com referência ao nó, um refinamento dessa instrumentação é importante para obter uma análise de dados mais precisa. Neste sentido é proposto identificar as *tuplas* afetadas em cada ocorrência de dados persistentes, ou seja, um modelo de instrumentação para a determinação dinâmica dos dados manipulados na granulosidade de *tupla*. Isto torna

possível ampliar o conjunto das associações entre ocorrências que devem ser exercitadas cobrindo situações que não podem ser cobertas considerando apenas tabelas. Uma associação válida para tabela pode não ser válida para *tupla*, pois mesmo que uma tabela seja referenciada em dois comandos de manipulação, estes podem não estar afetando a mesma *tupla*.

Segundo Vaduva (1999), em geral é praticamente impossível estabelecer um mapeamento para lógica de primeira ordem para os comandos SQL de manipulação de dados, a dificuldade situa-se, por exemplo, no aninhamento de construções, uso de funções agregadas e recursos de agrupamento de dados. Assim, estabelecer o uso de expressões obtidas estaticamente para caracterizar o conjunto de dados afetados por uma determinada operação requer muitas restrições para a aplicação dessa idéia. Já em Spoto (2000), para cada comando que atribui valores a uma tabela existe um comando específico para indicar a *tupla* utilizada.

A instrumentação visando determinar os dados manipulados em cada operação é realizada pela introdução de comandos de escrita que possuem subconsulta para recuperar a identificação das *tuplas* afetadas pelo comando instrumentado. Essas informações serão úteis na avaliação da aplicação de critérios baseados em análise de fluxo de dados.

Um aspecto pertinente é saber as colunas associadas à chave primária de cada tabela, essa informação pode ser obtida a partir da base de meta-dados onde se encontra a regra ativa em teste.

Do ponto de vista semântico, a introdução de comandos de escrita de instrumentação não alteraria o comportamento de uma regra. Ressalta-se que esta observação é feita levando em consideração apenas uma regra em execução, para o teste de conjuntos de regras requer-se um estudo mais aprofundado.

A instrumentação de *tuplas* é independente da instrumentação de nós; assim, os comandos de escrita inseridos visando à identificação dos nós devem permanecer inalterados, ou seja, a instrumentação de *tuplas* requer comandos adicionais. A proposta feita nesta seção visa somente a instrumentação de *tuplas*.

Esta instrumentação é realizada pela escrita em uma tabela específica, usando-se um comando INSERT para cada comando instrumentado; o comando INSERT é construído com uma subconsulta que recupera a identificação das *tuplas* manipuladas pelo comando que está sendo instrumentado. Esta tabela onde são armazenadas as informações obtidas a partir da instrumentação segue uma estrutura como a apresentada na Tabela 3.1. Para cada comando de manipulação devem ser seguidas regras específicas para a inserção desses comandos, que vão possibilitar obter informações das *tuplas* afetadas.

Tabela 3.1: Estrutura da tabela de instrumentação.

Atributo	**Tipo de Dado**	**Restrições**	**Observações**
Código	Data/hora	Not null, PK	Data e hora da execução do comando
Regra	String	Not null	Identificação da regra
NumNó	String	Not null	Identificação do nó
Tabela	String	Null	Identificação da tabela
Obs	String	Null	Observações gerais
CasoTeste	Integer/String	Not null, FK	Identificação do caso de teste
Operação	String	Null	Tipo de operação (S, I, U, D)
Chave	String	Null	Valor de chave primária da tabela

O comando SELECT pode ser instrumentado considerando as várias tabelas que são afetadas e também os casos de seleção simples, seleção com função agregada, junção e subconsultas correlatas; este raciocínio se aplica também ao uso de cursor. Um exemplo simples de instrumentação de comando SELECT é apresentado na Figura 3.28. O comando INSERT tem como alvo a tabela INSTRUM, criada para armazenar informações da instrumentação. O conjunto de dados a serem inseridos pelo comando INSERT é obtido por uma subconsulta cujo predicado é idêntico ao do comando SELECT que está sendo instrumentado, essa subconsulta recupera a identificação das *tuplas* afetadas pelo comando sendo instrumentado. Este exemplo é um caso simples do comando SELECT, onde é manipulada apenas uma tabela e o predicado da consulta não

é complexo. O mesmo raciocínio pode ser aplicado a comandos SELECT que implementam consultas mais elaboradas, incluindo funções agregadas, como no exemplo da Figura 3.29.

```
Ponta_de_Prova (1)
INSERT INTO INSTRUM
    (Código,Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave )
SELECT DISTINCT Num, 'R1', 11, 1222, Null, 'EMPLOYEE', 'S',
                    EMPLOYEE.emplno
FROM EMPLOYEE
WHERE SALARY > 500
SELECT emplno, fname
INTO vEmplno, vName
FROM EMPLOYEE
WHERE SALARY > 500
```

Figura 3.28: Instrumentação de *tuplas* para uma seleção simples.

```
Ponta_de_Prova (1)
INSERT INTO INSTRUM
 (Código,Regra,NumNó, CasoTeste, Obs, Tabela, Operação, Chave)
SELECT DISTINCT Num, 'R1', 11, 1122, Null,
        'EMPLOYEE', 'S', EMPLOYEE.emplno
 FROM EMPLOYEE
 WHERE SALARY > 500
 GROUP BY deptworksfor
 HAVING COUNT(*) > 10

SELECT deptworksfor, COUNT(*)
        INTO vDeptworksfor, vQtde
        FROM EMPLOYEE
WHERE SALARY > 500
GROUP BY deptworksfor
HAVING COUNT(*) > 10
```

Figura 3.29: Instrumentação de *tuplas* para uma seleção com função agregada.

Uma junção simples no comando SELECT deve seguir o mesmo procedimento das Figuras 3.30 e 3.31. A instrumentação de *tuplas* deve ser realizada para cada tabela manipulada. No exemplo da Figura 3.30, são manipuladas três tabelas; em cada INSERT de instrumentação de *tuplas* a chave primária de cada tabela é referenciada separadamente.

Ponta_de_Prova (1)

```
INSERT INTO INSTRUM
   (Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave)
SELECT DISTINCT Num, 'R1', 11, 1122, Null,
    'EMPLOYEE', 'S', EMPLOYEE.emplno
FROM EMPLOYEE, DEPARTMENT, DEPENDENT
WHERE EMPLOYEE.deptworksfor = DEPARTMENT.deptno
AND DEPENDENT.empldependsof = EMPLOYEE.emplno
AND EMPLOYEE.salary > 500
AND DEPENDENT.relationship = 'FILHO'
```

Tabela EMPLOYEE

```
INSERT INTO INSTRUM
  (Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave)
SELECT DISTINCT Num, 'R1', 11, 1122, Null,
    'DEPARTMENT', 'S', DEPARTMENT.deptno
FROM EMPLOYEE, DEPARTMENT, DEPENDENT
 WHERE EMPLOYEE.deptworksfor = DEPARTMENT.deptno
AND DEPENDENT.empldependsof = EMPLOYEE.emplno
AND EMPLOYEE.salary > 500
AND DEPENDENT.relationship = 'FILHO'
```

Tabela DEPARTMENT

```
INSERT INTO INSTRUM
  (Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave)
SELECT DISTINCT Num, 'R1', 11, 1122, Null,
'DEPARTMENT', 'S',
   DEPENDENT.empldependsof CONCAT DEPENDENT.dependSeq
FROM EMPLOYEE, DEPARTMENT, DEPENDENT
WHERE EMPLOYEE.deptworksfor = DEPARTMENT.deptworksfor
AND DEPENDENT.empldependsfor = EMPLOYEE.empldependsof
AND EMPLOYEE.salary > 500
AND DEPENDENT.relationship = 'FILHO'
```

Tabela DEPENDENT

```
SELECT EMPLOYEE.lname, DEPARTMENT.name, DEPENDENT.name
INTO vEmployeeName, vDeptName, vDependName
FROM EMPLOYEE, DEPARTMENT, DEPENDENT
WHERE EMPLOYEE.deptworksfor = DEPARTMENT.deptno
AND DEPENDENT.empldependsof = EMPLOYEE.emplno
AND EMPLOYEE.salary > 500   AND DEPENDENT.relationship = 'FILHO'
```

Figura 3.30: Instrumentação de *tuplas* para uma junção simples com várias tabelas.

Para a instrumentação de subconsultas dentro de comandos SELECT, a instrumentação só é feita para a consulta mais externa pois as consultas mais internas apenas extraem informações complementares para a consulta mais externa. No exemplo da Figura 3.31, a instrumentação de *tuplas* envolve apenas a tabela DEPARTMENT, sem considerar a tabela manipulada pela subconsulta.

```
Ponta_de_Prova (1)
INSERT INTO INSTRUM
        ( Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave )
SELECT DISTINCT Num,'R1',11,1122, Null,'EMPLOYEE', 'S',
                        DEPARTMENT.Deptno
FROM DEPARTMENT D
WHERE ( SELECT AVG (Salario)
            FROM EMPLOYEE E
            WHERE E.deptworksfor = D.deptno ) > 500 )

SELECT deptno, name
INTO vDeptno, vDeptName
FROM DEPARTMENT D
WHERE (SELECT AVG(Salario)
            FROM EMPLOYEE E
            WHERE E.deptworksfor = D.deptno ) > 500
```

Figura 3.31: Instrumentação de *tuplas* para uma sub-consulta correlata.

No uso de cursores as *tuplas* também devem ser identificadas, seguindo o procedimento da instrumentação de consultas. Na Figura 3.32 está exemplificado o uso de cursor definido para um comando SELECT que possui junção e subconsulta correlata. Os comandos de instrumentação devem ser inseridos antes do comando OPEN do cursor; esta é a única restrição para cursores.

```
DECLARE CURSOR crCursor IS
        SELECT E.lname, D.name
        FROM DEPARTMENT D, EMPLOYEE E
        WHERE E.deptworksfor = D.deptno
              AND  ( SELECT Count(*)
                     FROM DEPENDENT DE
         WHERE DE.empldependsof = E.emplno ) > 2
::::::::::::::::::::::::::
::::::::::::::::::::::::::
Ponta_de_Prova (Open)
INSERT INTO INSTRUM
      (Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave )
SELECT DISTINCT Num, 'R1', 11, 1122, Null,'EMPLOYEE', 'S', EMPLOYEE.emplno
FROM DEPARTMENT D, EMPLOYEE E
WHERE E.deptworksof = D.deptno
      AND (SELECT Count(*)
           FROM DEPENDENT DE
           WHERE DE.empldependsof = E.emplno ) > 2

INSERT INTO INSTRUM
      ( Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave )
SELECT DISTINCT SYSDATE, 'R1', 11, 1122, Null, 'EMPLOYEE', 'S',
       DEPARTMENT.deptno
FROM DEPARTMENT D, EMPLOYEE E
WHERE E.deptworksfor = D.deptono
          AND (  SELECT Count(*)
                 FROM DEPENDENT DE
                 WHERE DE.empldependsof = E.emplno ) > 2
OPEN crCursor
      ::::::::::::::::::::::::
```

Figura 3.32: Instrumentação de *tuplas* para um cursor.

Para a instrumentação do comando INSERT, é necessário o conhecimento dos atributos da chave primária; essa informação é obtida a partir dos meta-dados da base de dados. O comando INSERT pode ou não explicitar o conjunto de colunas da tabela; em

ambos os casos, é preciso saber a posição relativa dos valores de atributos da chave primária na *tupla* sendo inserida. O procedimento para a instrumentação é similar ao do comando SELECT. Quando o conjunto de colunas não estiver explícito no comando INSERT, a seqüência de valores após a cláusula VALUES é a mesma definida nos metadados. No comando da Figura 3.33, sua instrumentação considera que o valor da chave primária refere-se ao primeiro atributo, cujo valor é 'DVK'.

```
::::::::::::::::::::::::::::::::::
Ponta_de_Prova (1)
INSERT INTO INSTRUM
  (Código, Regra, NumNó, CasoTeste,Obs, Tabela,Operação, Chave)
VALUES (Num,'R1', 11, 1122, Null, 'DEPARTMENT', 'I', 'DVK' )
INSERT INTO DEPARTMENT
VALUES ('DVK' , 'DEPTO DE VENDAS ', '2233' )
::::::::::::::::::::::::::::::::::
```

Figura 3.33: Instrumentação de *tuplas* para o INSERT sem especificação de colunas.

Nos casos em que o comando INSERT explicita o elenco de colunas é imprescindível saber a identificação dos atributos da chave primária, a partir dos metadados, para determinar a posição relativa desses atributos na seqüência de colunas mencionadas no comando sendo instrumentado. No comando da Figura 3.34, a coluna *deptno* é a primeira coluna mencionada no comando.

```
::::::::::::::::::::::::::::::::::
Ponta_de_Prova (1)
INSERT INTO INSTRUM
  (Código,Regra, NumNó,CasoTeste,Obs,Tabela,Operação,Chave)
VALUES
    ( Num, 'R1', 11, 1122, Null, 'DEPARTMENT', 'I', 'DVK' )
INSERT INTO DEPARTMENT (deptno, name, emplmanages)
VALUES ('DVK' , 'DEPTO DE VENDAS ', '2223333' )
::::::::::::::::::::::::::::::::::
```

Figura 3.34: Instrumentação de *tuplas* para o INSERT com especificação de colunas.

O comando INSERT possibilita a inserção de linhas a partir de dados de outras tabelas, utilizando subconsulta. Tais dados podem ser derivados de agrupamentos e seleção de dados em uma ou mais tabelas. Para estes casos, é necessário conhecer a

definição das tabelas envolvidas na subconsulta, para saber o conteúdo da chave primária das *tuplas* sendo inseridas. Um exemplo deste caso é ilustrado na Figura 3.35.

```
::::::::::::::::::::::::::::::::::
Ponta_de_Prova (1)
INSERT INTO INSTRUM
   (Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave )
SELECT DISTINCT Num, 'R1', 11, 1122, Null, 'DEPARTMENT', 'I',
                    DEPARTMENT1.name
FROM DEPARTMENT1
INSERT INTO DEPARTMENT
SELECT *
FROM DEPARTMENT1
::::::::::::::::::::::::::::::::::
```

Figura 3.35: Instrumentação de *tuplas* para o INSERT a partir de outras tabelas.

A instrumentação do comando DELETE segue os padrões do comando SELECT, pois a identificação das *tuplas* afetadas é obtida a partir de uma consulta de dados na tabela do comando que está sendo instrumentado. Na Figura 3.36, a expressão **empldependsof CONCAT dependSeq** representa o valor da chave primária da tabela DEPENDENT.

```
::::::::::::::::::::::::::::::::::
Ponta_de_Prova (1)
INSERT INTO INSTRUM
      (Código, Regra, NumNó, CasoTeste, Obs, Tabela, Operação, Chave )
SELECT DISTINCT Num,R1',11,1122,Null,'DEPENDENT','D',empldependsof CONCAT dependSeq
FROM DEPENDENT
WHERE NOME LIKE 'JOSE%'
   DELETE FROM DEPENDENT
 WHERE NOME LIKE 'JOSE%';
  ::::::::::::::::::::::::::::::::::
```

Figura 3.36: Instrumentação de *tuplas* para o comando DELETE.

A instrumentação do comando UPDATE para identificar *tuplas* segue o modelo do comando SELECT, como apresentado no exemplo da Figura 3.37.

```
::::::::::::::::::::::::::::::::
Ponta_de_Prova (1)
INSERT INTO INSTRUM
        (Código, Regra, NumNo, CasoTeste, Obs, Tabela, Operação, Chave )
SELECT DISTINCT Num, 'R1', 11, 1122, Null, 'EMPLOYEE', 'D', emplno)
FROM EMPLOYEE
WHERE salary > 500
UPDATE EMPLOYEE
        SET salary = salary * 1.1
WHERE salary > 500
::::::::::::::::::::::::::::::::
```

Figura 3.37: Instrumentação de *tuplas* para o comando UPDATE.

3.4 Modelo de Fluxo de Dados

A análise de fluxo de dados produz informações que podem ser utilizadas para derivar requisitos de teste (Hecht, 1977); vários critérios de teste são baseados na análise de fluxo de dados (Herman, 1976; Rapps e Weyuker, 1985; Frankl e Weyuker, 1988; Maldonado *et. al*, 1988 b; Maldonado, 1991). Em programas convencionais, esta análise busca a determinação de ocorrências de definição e de uso de variáveis, para, no contexto de teste de software, estabelecer associações entre tais ocorrências que deveriam ser exercitadas por casos de teste.

O teste de regras ativas é similar ao teste de aplicações de banco de dados; neste, intercalam-se instruções da linguagem hospedeira e instruções SQL (Spoto, 2000). O estado do banco de dados é incluído como entrada e saída da rotina em teste. As variáveis persistentes existentes em regras ativas fazem parte do estado do banco de dados.

Além das relações de fluxo de dados envolvendo variáveis locais, a análise de fluxo de dados explora também a interação entre comandos devido à manipulação de dados persistentes e visa a descoberta de defeitos focados nesta interação. Cada ocorrência de comando por si requer uma apreciação para a determinação do conjunto de

dados afetados. Essa manipulação pode impactar o estado do banco de dados resultante e afeta a execução de outros comandos, ou seja, em geral existe uma dependência de fluxo de dados entre comandos de manipulação de uma regra ativa. Adicionalmente, essa dependência pode ir além da fronteira de uma única regra ativa, caracterizando uma associação entre regras, que não é explorada neste trabalho.

No contexto de análise de fluxo de dados, a granulosidade determina o grau de precisão da análise e define as relações de fluxo de dados das variáveis persistentes. Os graus de precisão podem ser tabela ou relação, coluna ou atributo e linha ou *tupla*.

A precisão de tabela mapeia as variáveis persistentes em tabelas ou relações do banco de dados, não importando quais atributos ou *tuplas* são afetadas por operações de manipulação. Esse enfoque simplifica a análise e reduz o número de variáveis pois considera cada tabela como uma única variável. Contudo, estabelece uma abordagem conservadora, já que toda definição de variável seguida de uso da variável é uma associação de fluxo de dados, independentemente de estar manipulando dados distintos na tabela.

Explorando um pouco mais a tabela, a precisão de coluna enfoca as ocorrências de definição e de uso para os atributos de tabelas, sem consideração das *tuplas* que são manipuladas. Comparando com a precisão de tabela, representa uma maior precisão pois distingue as colunas de cada tabela. As associações de fluxo de dados são caracterizadas quando a interseção dos dois conjuntos de colunas manipuladas atribuídos respectivamente a uma ocorrência de definição seguida de uma ocorrência de uso, é não vazia. Uma vantagem dessa abordagem é que, como o conjunto de colunas de uma relação é fixo, suas ocorrências podem ser determinadas estaticamente. Esta abordagem é usada por Aranha (2003) para definir requisitos de teste de esquema de base de dados relacionais, onde os casos de teste buscam exercitar atributos e restrições associadas a atributos. Haraty *et. al* (2002) propõem uma estratégia para o teste de regressão de aplicações de banco de dados e usam a granulosidade de coluna para determinar os módulos de banco de dados afetados por modificações na definição de componentes de banco de dados.

A precisão de linha é adotada na análise de fluxo de dados envolvendo *tuplas* afetadas em cada operação de manipulação de dados. Este tema é abordado na Seção 3.3.4, que explora a instrumentação de *tuplas*. Em geral, não se pode determinar estaticamente que *tuplas* serão alvo de uma dada operação devido ao grau de complexidade possível nos predicados de seleção de linhas (Vaduva, 1999), e devido à dependência do estado de banco de dados corrente. No contexto de bancos de dados relacionais uma tupla representa uma entidade do mundo real, uma associação entre entidades ou algum aspecto particular de uma entidade. Utilizar a granulosidade de linha significa considerar as entidades manipuladas por operações de banco de dados.

A análise de fluxo de dados em regras ativas inicia-se com a determinação dos tipos de variáveis que considera as ocorrências de definição e uso de variáveis locais, de variáveis persistentes e de variáveis de transição.

A ação de uma regra ativa é definida como um bloco de comandos, que pode possuir outros blocos internos. As variáveis locais são declaradas em um bloco e têm seu escopo de vida dentro desse mesmo bloco e em quaisquer blocos internos a esse.

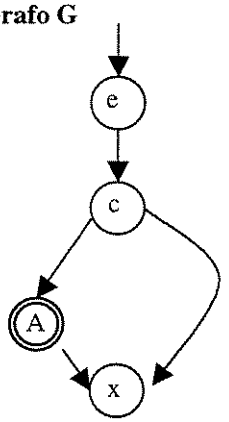
Em banco de dados devem ser consideradas as variáveis persistentes, associadas ao estado do banco de dados, e que são referenciadas como tabelas no contexto de bancos de dados relacionais. A definição e o uso de variáveis persistentes em SQL são realizados por comandos de manipulação, de cursores e de tratamento de exceção; essa análise pode ser feita em granulosidade tabela, coluna ou linha.

Outro tipo de variável que está incluída para esta análise do fluxo de dados são as variáveis de transição, de escopo local à regra, que referem-se aos dados afetados durante uma operação de manipulação, a qual provocou o evento da regra.

Para a implementação da ferramenta ART-TOOL foi adotada a granulosidade de tabela, como passo inicial para uma análise dos dados persistentes; procura-se primeiro analisar os dados de um modo mais geral para depois avaliar a necessidade de um refinamento, para granulosidade de *tupla*, por exemplo.

A Figura 3.38 ilustra a análise de fluxo de dados de uma regra ativa. A regra **R** é representada pelo grafo **G**: **e** é o nó de evento; **c** é a condição; e **A** é o subgrafo da ação

da regra. $\mathbf{V_P}$ é o conjunto das variáveis persistentes de escopo global ao banco de dados, $\mathbf{V_T}$ é o conjunto das variáveis de transição da regra **R** e $\mathbf{V_L}$ é o conjunto das variáveis locais declaradas em **R**. No evento ocorre uma *definição* das variáveis locais, variáveis persistentes e variáveis de transição. A variável persistente referenciada na cláusula ON caracateriza um *c-uso*. Na condição ocorre um *p-uso* nos arcos que envolvem as variáveis deste nó, pois pode haver neste caso um desvio de fluxo. Na ação da regra as variáveis devem ser analisadas separadamente.



Grafo G	Nó, arco e subgrafo	Definição	C-uso	P-uso
	Nó e	Variáveis persistentes (V_P) e variáveis de transição (V_T).	V_P	
	Arcos (c,A) e (c, x)			V_P e V_T
	Subgrafo A	V_P, V_T e variáveis locais (V_L)	V_P, V_T e V_L	V_P, V_T e V_L
	Nó x		V_P e V_T	

Figura 3.38: Representação de ocorrências de variáveis em uma regra ativa.

Na análise de fluxo de dados para variáveis persistentes deve-se considerar a granulosidade adotada. A Tabela 3.2 mostra uma análise de cada comando de manipulação para as granulosidades de tabela, de coluna e de linha. Em uma coluna adicional estão as ocorrências das variáveis locais referenciadas nos comandos de manipulação.

Os comandos de cursor, OPEN, FETCH e CLOSE, que afetam o fluxo de dados, estão representados na Tabela 3.2.

Tabela 3.2: Representação da ocorrência de variáveis em comandos de manipulação.

Comando	Granulosidade tabela	Granulosidade coluna	Granulosidade linha	Variáveis locais
SELECT	C-uso das tabelas referenciadas na cláusula FROM.	C-uso dos atributos referenciados na cláusula SELECT ou na cláusula WHERE.	C-uso das *tuplas* consultadas das tabelas referenciadas na cláusula FROM.	Definição das variáveis locais referenciadas na cláusula INTO. C-uso das variáveis locais referenciadas na cláusula SELECT ou na cláusula WHERE.
INSERT	Definição da tabela.	Definição de todos os atributos da tabela.	Definição das *tuplas* inseridas.	C-uso das variáveis locais referenciadas na cláusula INTO.
UPDATE	Definição da tabela. C-uso da tabela.	Definição dos atributos referenciados no lado esquerdo de "=" na cláusula SET. C-uso dos atributos referenciados no lado direito de "=" na cláusula SET ou na cláusula WHERE.	C-uso seguido de definição das *tuplas* modificadas.	C-uso das variáveis locais referenciadas na cláusula SET ou na cláusula WHERE.
DELETE	Definição da tabela.	Definição de todos atributos da tabela.	C-uso das *tuplas* excluídas (em seguida tais *tuplas* tornam-se indefinidas).	C-uso das variáveis locais referenciadas na cláusula WHERE.
OPEN (cursor)	C-uso das tabelas referenciadas na cláusula FROM.	C-uso dos atributos referenciados na cláusula SELECT ou na cláusula WHERE.	C-uso das *tuplas* consultadas das tabelas referenciadas na cláusula FROM.	C-uso das variáveis locais referenciadas na cláusula SELECT ou na cláusula WHERE. Definição da **variável cursor.**
FETCH (cursor)				Definição das variáveis locais referenciadas na cláusula INTO. C-uso da **variável cursor.**
CLOSE (cursor)				Definição da **variável cursor.**

Os comandos de manipulação são exemplificados pelos códigos ilustrados na Tabela 3.3, mostrando as ocorrências das variáveis em cada comando. Nessa análise algumas conclusões são ressaltadas, tais como, as ocorrências de subconsultas seguem o mesmo raciocínio aplicado ao comando SELECT, visto que representa uma consulta

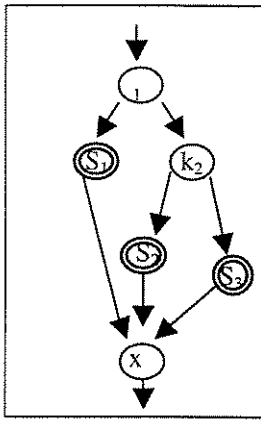
embutida em outro comando de manipulação. As variáveis de transição, dados afetados em cada linha definida, seguem o mesmo raciocínio aplicado a variáveis locais. Já as tabelas de transição, conjunto de todos os dados afetados pela definição de dados, seguem o mesmo raciocínio aplicado a variáveis persistentes.

Tabela 3.3: Representação de ocorrência de variáveis em comandos de manipulação codificados.

Comandos de manipulação codificados:	Definição	**C-Uso**
SELECT emplno, fname		emplno, fname (atributo)
	vEmplno, vName (v. locais)	
INTO vEmplno, vName		EMPLOYEE (tabela)
FROM EMPLOYEE		salary (atributo)
WHERE salary > 500		
::::::::::::::::::::::::::::::::::		
INSERT INTO DEPARTMENT	DEPARTMENT (tabela)	
(deptno, name,emplqty, emplmanages)		
VALUES ('DVK' ,'DEPTO DE VENDAS ',		vQtde (variável local)
vQtde, '2223333')		
::::::::::::::::::::::::::::::::::		
UPDATE EMPLOYEE	EMPLOYEE (tabela)	EMPLOYEE (tabela)
SET salary = salary * vBonus	salary (atributo)	salary (atributo) vBonus (v. local)
		salary (atributo)
WHERE salary > 500		
::::::::::::::::::::::::::::::::::		
DELETE FROM DEPENDENT	DEPENDENT (tabela)	
WHERE empldependsof = new.emplno;		new.emplno(v.Transição)
::::::::::::::::::::::::::::::::::		empldependsof (atributo)

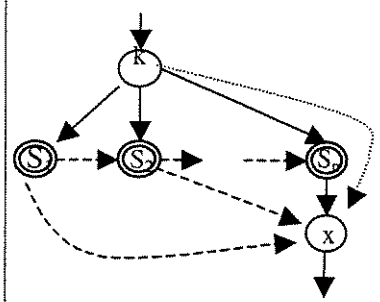
A análise de fluxo de dados para comandos estruturados é similar à definida por Chaim (1991). Uma observação relevante é que podem existir usos de variáveis

persistentes nos comandos estruturados; nesses casos serão seguidas as convenções para definição e uso estabelecidas para as variáveis locais. As Figuras 3.39 a 3.45 ilustram a análise de fluxo de dados para os comandos IF, CASE, WHILE, REPEAT e FOR, respectivamente.



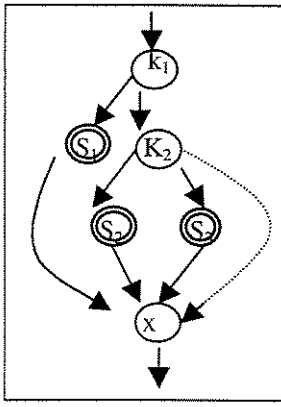
Nó, arco ou subgrafo	Comando **IF**
(K_1, S_1), (K_1,K_2) e (K_2, S_2),(K_2,S_3)	P-uso das variáveis referenciadas na cláusula IF.
S_1, S_2 e S_3	A extensão dos subgrafos depende dos comandos associados à cláusula THEN.

Figura 3.39: Análise de fluxo de dados para o comando IF.



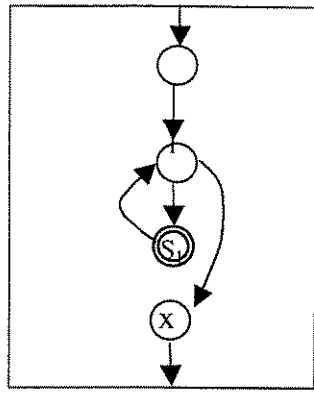
Nó, arco ou subgrafo	Comando **CASE** (caso1)
(k, S_1), (k, S_2), ...(k, S_n)	P-uso das variáveis referenciadas na cláusula CASE.
S_1, S_2 e S_n	A extensão dos subgrafos depende dos comandos associados à cláusula THEN.

Figura 3.40: Análise de fluxo de dados para o comando CASE (caso1).



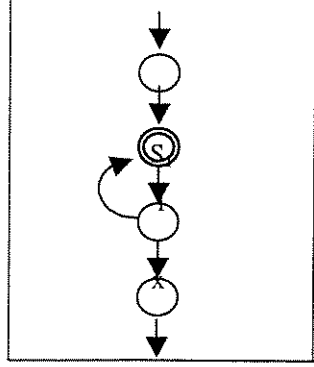
Nó, arco ou subgrafo	Comando **CASE** (caso2)
(K_1, S_1), (K_1,K_2), (K_2, S_2), (K_2,S_3)	P-uso das variáveis referenciadas na cláusula WHEN.
S_1, S_2 e S_3	A extensão dos subgrafos depende dos comandos associados à cláusula THEN.

Figura 3.41: Análise de fluxo de dados para o comando CASE (caso2).



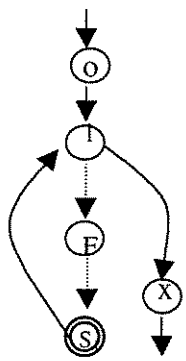
Nó, arco ou subgrafo	Comando **WHILE**
(I,S_1), (I,x)	P-uso das variáveis referenciadas na cláusula WHILE.
S_1	A extensão do subgrafo depende dos comandos associados aos nós de S_1.

Figura 3.42: Análise de fluxo de dados para o comando WHILE.



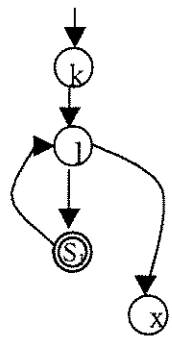
Nó, arco ou subgrafo	Comando **REPEAT**
S_1	A extensão dos subgrafos depende dos comandos associados aos nós de S_1.
(I, S_1), (I, x)	P-uso da variável referenciada na cláusula UNTIL.

Figura 3.43: Análise de fluxo de dados para o comando REPEAT.



Nó, arco ou subgrafo	Comando **FOR** (cursor implícito)
O	Definição da variável contadora do comando FOR.
(I, F), (I, x)	P-uso da variável contadora referenciada na cláusula FOR.
F	Definição e c-uso da variável contadora do comando FOR.
S_1	A extensão do subgrafo depende dos comandos associados aos nós de S_1.

Figura 3.44: Análise de fluxo de dados para o comando FOR (cursor implícito).



Nó, arco ou subgrafo	Comando **FOR**
K	Definição da variável contadora do comando FOR.
(l, S_1), (l, x)	P-uso das variável referenciadas na cláusula FOR.
S_1	A extensão do subgrafo depende dos comandos associados aos nós de S_1. Definição e c-uso da variável contadora do comando FOR.

Figura 3.45: Análise de fluxo de dados para o comando FOR.

A análise de fluxo de dados para as rotinas de tratamento de exceção depende da natureza da exceção, definida pelo sistema ou definida pelo usuário. Para as exceções definidas pelo sistema, o fluxo de dados específico é estabelecido pelas ocorrências de definição e uso de uma pseudo-variável dedicada ao armazenamento do tipo de exceção ocorrida, a variável *exception*. As seguintes convenções foram estabelecidas:

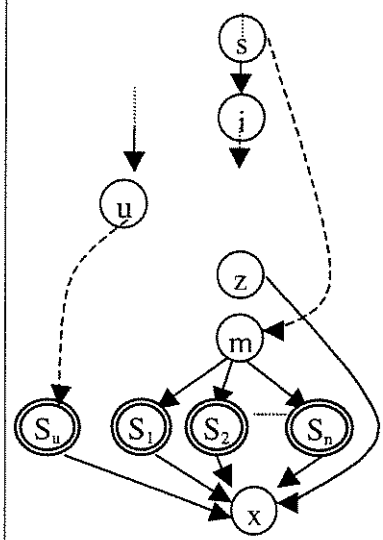
- em todo nó associado a comandos de manipulação – INSERT, UPDATE, DELETE, SELECT, OPEN (cursor), FETCH (cursor) e CLOSE (cursor) – ocorre uma definição da variável *exception*; esse enfoque é devido à possível ocorrência de exceção em comandos dessa natureza;
- nos arcos de saída dos nós associados a comandos de manipulação ocorre um *p-uso* de *exception*; essa abordagem estabelece associações *def-uso* que, se exigidas, requerem execuções normal e com exceção de cada comando de manipulação;
- nos arcos de saída do nó associado à seleção dependente do tipo de exceção (na Figura 3.46, os arcos (s,i), (s,m), (m,S_1), (m,S_2) e (m,S_n)) ocorre um *p-uso* de *exception*; esse enfoque estabelece associações *def-uso* que, se exigidas, requerem, para todo comando de manipulação, o exercício de cada tipo de exceção tratada; neste caso, poderão existir elementos requeridos não executáveis, pois determinados tipos de exceção não ocorrem em certos comandos de manipulação.

Para ilustrar as observações acima, no grafo da Figura 3.46 existe definição de *exception* no nó s; nos arcos (s,i), (s,m), (m,S_1), (m,S_2) e (m,S_n) existe *p-uso* de *exception*.

Nas exceções definidas pelo usuário, o foco não é a variável *exception*, mas sim as variáveis locais V_u declaradas com o tipo *exception*; por exemplo, considere a variável *USER_EXC* na Figura 3.11. Para essas variáveis, valem as seguintes considerações:

- nos nós em que alguma exceção do usuário é explicitamente provocada (por exemplo, via comando RAISE), ocorre uma definição de V_u;
- no arco de saída dos nós em que alguma exceção do usuário é explicitamente provocada, ocorre um *p-uso* de V_u.

Para exemplificar, no grafo da Figura 3.46 existe definição de V_u no nó u, e *p-uso* de V_u no arco (u,S_u).



Nó, arco ou subgrafo	Tratamento de **EXCEÇÃO**
Nó S	Definição da variável *exception*.
Arcos (s,m),(s,i), (m,S_1), (m,S_2), ...(m,S_n)	P-uso da variável *exception*.
Arco (u, S_u)	P-uso da variável V_u
Subgrafos S_1, S_2,..S_n	A extensão dos subgrafos depende dos comandos associados aos nós de S_1, S_2,...S_n.
Nó U	Definição das variáveis locais declaradas com o tipo exception.

Figura 3.46: Análise de fluxo de dados para o tratamento de exceção.

3.5 Considerações Finais

Foram explorados os fluxos de controle, de dados e a instrumentação em regras ativas visando o teste estrutural de regras isoladas no contexto de banco de dados relacionais.

Do ponto de vista de fluxo de controle, foi estabelecida uma estrutura geral de controle, onde se isolam as partes constituintes de uma regra: evento, condição e ação. Para os comandos de manipulação, focados no acesso a dados persistentes, foi estabelecido um nó exclusivo para cada comando. Subconsultas ficam embutidas no nó do comando de manipulação em que estão contidas. No caso dos comandos de cursores – OPEN, FETCH e CLOSE, implícitos ou não, são mapeados para nós exclusivos.

Ênfase foi dada ao tratamento de exceção, que incorpora complexidade ao grafo de fluxo de controle resultante de uma regra. Em cada ocorrência de comando de manipulação pode ocorrer um desvio para a o bloco de tratamento de exceção, que incrementa a complexidade ciclomática e eleva o esforço para a aplicação de critérios baseados na estrutura de controle.

No modelo de instrumentação, para evitar que a regra seja abortada durante o teste, foi ressaltada a importância da inclusão do bloco de exceção para desvios dos comandos de manipulação e consultas quando detectados resultados não previsíveis.

O modelo de instrumentação foi refinado em granulosidade de *tupla* para os comandos de manipulação de banco de dados, com o objetivo de determinar as *tuplas* afetadas em cada definição e uso de dados persistentes. Observou-se que, adotando esta filosofia, o número de linhas do código fonte aumenta consideravelmente, podendo elevar o tempo de execução, sendo imprescindível uma análise para verificar se é necessário este refinamento.

CAPÍTULO 4

A FERRAMENTA ART-TOOL

4.1 Introdução

A ferramenta ART-TOOL (*Active Rule Testing Tool*) foi desenvolvida para auxiliar o teste estrutural de regras ativas; ela auxilia o testador no teste de unidade de regras ativas escritas na linguagem SQL e ativadas em bancos de dados ORACLE versão 7 ou superior. A ART-TOOL dá suporte à aplicação de critérios baseados em fluxo de controle e dos critérios Potenciais Usos. Possui uma interface simples (sem recursos gráficos) mas fornece ao usuário vários arquivos para o acompanhamento da sessão de trabalho. Esta ferramenta foi implementada na linguagem Java versão 1.4.

A utilização da ART-TOOL no processo de teste ocorre em três etapas: a análise estática, o teste e a avaliação do teste. Na primeira fase, a partir da análise do código fonte da regra, são obtidas informações necessárias para a elaboração dos casos de teste que serão executados na próxima fase. Também nesta etapa a regra ativa é instrumentada para o monitoramento da execução dos casos de teste. Para finalizar é feita uma avaliação dos casos de teste. Detalhes de implementação são descritos com exemplos dos arquivos de entrada e saída dessas etapas. São utilizados módulos de outra ferramenta de teste estrutural - POKE-TOOL (Chaim, 1991) – para gerar os elementos requeridos e para a avaliação do processo de teste.

4.2 O Sistema Gerenciador de Banco de Dados (SGBD)

A ferramenta ART-TOOL analisa regras ativas escritas na linguagem SQL (*Structured Query Language*) segundo o padrão SQL-3.

Os sistemas gerenciadores de banco de dados comerciais não implementam todas as características das regras ativas das propostas teóricas; não são implementadas todas as operações com regras nem todas as ferramentas de suporte a regras. É notável que as propostas são muito mais complexas do que as implementações que têm sido feitas (Baralis e Widom, 1994). Esta discrepância pode ser vista nos tipos de eventos detectados; por exemplo, os eventos que podem ser detectados por uma regra são restritos aos aplicados sobre os dados de uma tabela como comandos de manipulação de dados.

O sistema gerenciador de banco de dados adotado é o ORACLE, dada a sua grande difusão, a preferência da comunidade acadêmica e a proximidade ao estado da arte quanto ao suporte a regras ativas.

Paduan (2000) apresenta um quadro comparativo das implementações das regras de três sistemas gerenciadores de banco de dados disponíveis comercialmente, mostrado na Tabela 4.1. A teoria propõe regras complexas e abrangentes, então buscou-se um produto que as aceitasse com um grau maior de complexidade. Na Tabela 4.1 os critérios de comparação são:

- **Regras ativas disparadas** refere-se ao momento de execução da regra, ou seja, se esta é disparada antes ou depois do evento ser concluído.
- **Granulosidade** está associada ao conjunto de dados afetados por uma operação de manipulação de dados. Neste caso a execução da regra pode ser de duas maneiras: o disparo da regra uma única vez por evento ou uma vez a cada registro que o evento afeta. Observa-se no ORACLE que são suportadas quatro combinações. A ocorrência do evento pode ser antes ou depois de cada registro ou comando. No padrão SQL-3, estas combinações são escritas das seguintes formas: AFTER STATEMENT; AFTER ROW; BEFORE STATEMENT; BEFORE ROW.

- **Regras ativas por tabela**, ou seja, o número de regras que podem estar associadas a cada tabela. Na maioria dos sistemas gerenciadores de bancos de dados ativos são suportadas três regras por tabela pois, apesar de uma mesma regra poder monitorar várias operações de manipulação de dados (INSERT, DELETE, UPDATE), cada operação é monitorada por no máximo uma regra ativa. O ORACLE permite associar até doze regras por tabela, sendo uma regra para cada tipo de evento; este evento pode ocorrer antes ou depois de cada registro ou comando, como descrito acima. Se cada combinação for para cada tipo de evento, o produto será de doze combinações.

- **Ações** diz respeito aos comandos que não podem ser escritos nas ações de uma regra (controle de transação, definição de novos objetos, etc).

- **Cascateamento**, ou seja, na ação de uma regra pode ocorrer um evento que dispare outra regra e o item de comparação pode referir quantas regras ativas serão disparadas em cascata e por um só evento.

- **Recursividade** refere-se à possibilidade de uma regra disparar a si mesma em resposta a sua própria ação.

Tabela 4.1: Comparação entre os Banco de Dados (Paduan, 2000).

	ORACLE	**SYBASE**	**SQL Server**
Regras ativas Disparadas	Antes ou depois do Evento	Após o evento	Após o evento
Granulosidade	Registro ou comando	Comando	Comando
Regras ativas por tabela	12	3	3
Ações	Não suporta controle de transações	Não suporta definições	Não suporta definição nem controle de concorrência
Cascateamento	32	8	16
Recursividade	Permitida	Pode ser permitida	Não é permitida

4.3 Descrição da Ferramenta ART-TOOL

A ferramenta ART-TOOL auxilia na execução das atividades de teste, como mostrado na Figura 4.1. Na etapa de análise estática é feita toda a preparação para o teste e inclui: as análises léxica e sintática, a geração do fluxo de controle, a geração do fluxo de dados, a geração dos elementos requeridos e a instrumentação da regra. Na segunda etapa são executados os casos de teste e gerados os caminhos exercitados. Na etapa de avaliação é feita a análise de cobertura.

Módulo de Análises Léxica e Sintática – Efetua a análise léxica e sintática da regra ativa. Este módulo foi desenvolvido utilizando as ferramentas *JavaCC* e *JJTree*, detalhadas na Seção 4.3.1, e gera informações referentes aos comandos analisados.

Módulo de Geração do Fluxo de Controle – Com base nas informações obtidas a partir da árvore sintática, o módulo produz o grafo de fluxo de controle do código fonte da regra ativa a ser testada, baseado no modelo proposto no Capítulo 3, Seção 3.2.

Módulo de Geração do Fluxo de Dados – A partir do fluxo de controle construído, as variáveis associadas a cada nó são analisadas de acordo com a sua ocorrência, gerando o grafo de fluxo de dados, de acordo com o modelo proposto no Capítulo 3, Seção 3.4.

Módulo de Instrumentação – O módulo gera a nova versão da unidade a ser testada, a regra ativa com comandos de escrita inseridos. Pontas de prova, os comandos de escrita, são incluídas em cada nó, como detalhado no modelo especificado no Capítulo 3, Seção 3.3, para tornar possível o acompanhamento da execução dos casos de teste.

Módulo de Geração dos Elementos Requeridos - Este módulo, da ferramenta POKE-TOOL, é usado para a geração dos elementos requeridos. Com base no fluxo de controle e no fluxo de dados, provê um conjunto de caminhos e associações requeridos para satisfazer os critérios Potenciais Usos.

Módulo de Geração dos Caminhos Exercitados – Este módulo rastreia a execução dos casos de teste. O caminho percorrido é obtido a partir dos números dos nós do grafo de fluxo de controle do programa fonte, armazenados em uma tabela específica durante o disparo da regra.

Módulo de Avaliação – Este módulo, da ferramenta POKE-TOOL, realiza a avaliação da cobertura atingida pelo conjunto de casos de teste executados para os critérios escolhidos. Produz uma relação de elementos (associações) requeridos pelo critério e não executados, e uma medida percentual da cobertura provida pelo conjunto de casos de teste, ou seja, uma relação entre o número de elementos (associações) exercitados e o número de elementos (associações) requeridos (Chaim, 1991).

Resumidamente, o testador fornece à ferramenta o código fonte da regra ativa que vai ser testada; a ART-TOOL lê o arquivo texto, a regra ativa e constrói a árvore sintática com o objetivo de disponibilizar informações da regra para os próximos módulos. A partir da árvore sintática é determinado o grafo fluxo de controle. Este é estendido incorporando informações das ocorrências das variáveis para obter o grafo de fluxo de dados. Como parte da análise estática, o módulo de instrumentação gera uma nova versão da regra em teste, a versão instrumentada, para auxiliar na determinação dos caminhos efetivamente executados pelo conjunto de casos de teste. Para concluir a fase de análise, a partir das informações dos fluxos de controle e de dados o módulo *Newdescr* da POKE-TOOL abstrai elementos requeridos pelos critérios Potenciais Usos. Na etapa de teste propriamente dita, a versão instrumentada da regra ativa é executada com os casos de teste. Durante a execução dos casos de teste os nós exercitados são armazenados na tabela INSTRUM. Tendo-se obtido os elementos requeridos e os caminhos exercitados, inicia-se a fase de avaliação do teste. Esta atividade é realizada pelo módulo avaliador da POKE-TOOL para verificar a adequação de um conjunto de casos de teste de acordo com o critério aplicado. São determinados os caminhos (associações) efetivamente exercitados e verificado se o critério foi satisfeito; é fornecido ao usuário uma lista de caminhos requeridos pelo critério e não exercitados pelo conjunto de caso de teste.

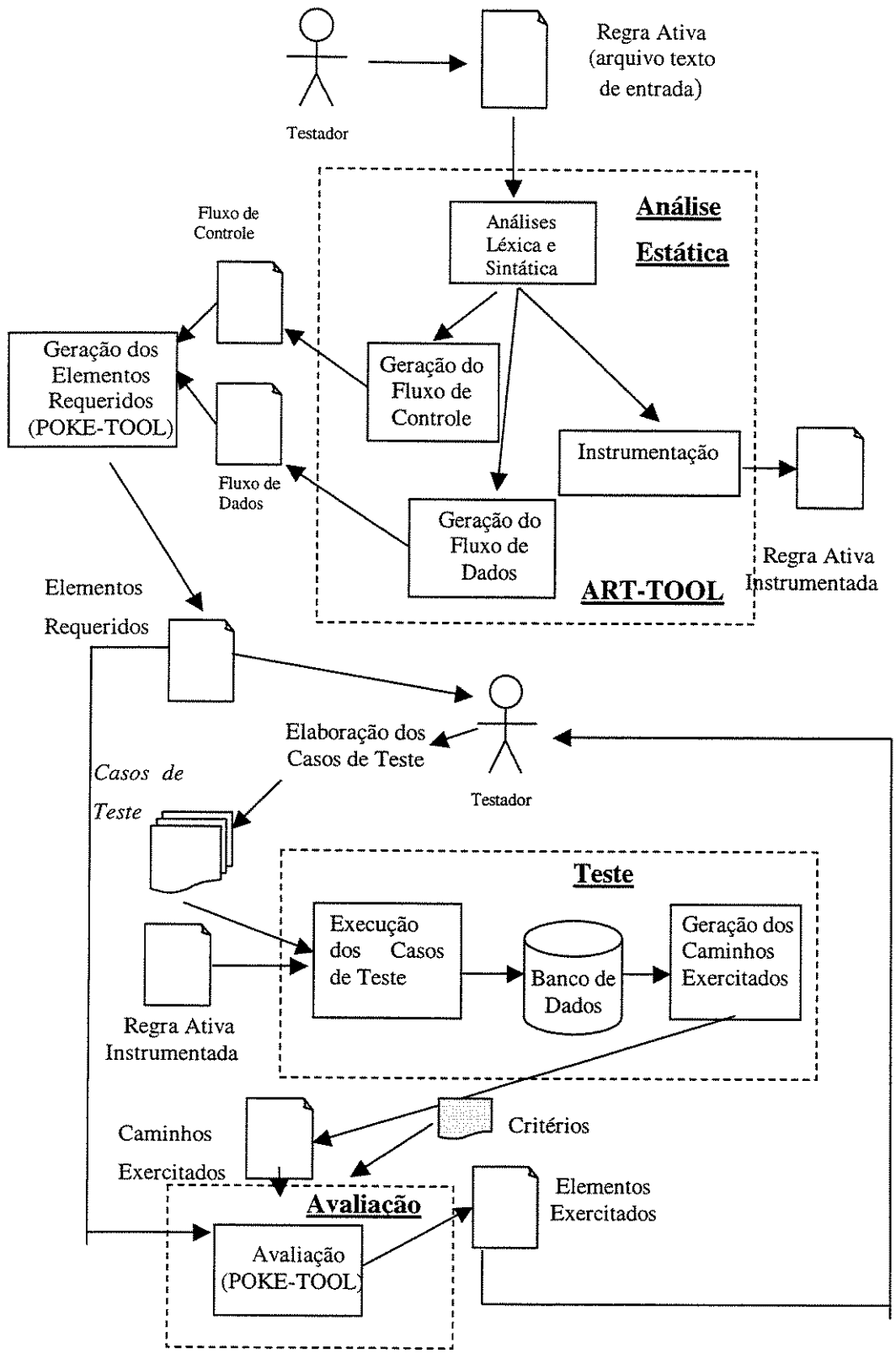


Figura 4.1: A ferramenta ART-TOOL.

4.3.1 Análises Léxica e Sintática

Este módulo inicia a fase estática da ferramenta, realiza análises léxica e sintática do código fonte da regra ativa e gera informações necessárias para as fases de teste e avaliação.

Os analisadores léxico e sintático foram construídos usando as ferramentas *JavaCC*, *Java Compiler Compiler*, um gerador de analisadores léxico e sintático, e *JJTree*, um construtor de árvores sintáticas, ambas específicas para aplicações escritas na linguagem Java (Appel e Ginsburg, 1997; Aho *et. al*, 1995; JavaCC e JJTree, a, b, c, d).

Para o reconhecimento da gramática é preciso ler sua especificação (Figura 4.2) e convertê-la para um programa. A gramática da linguagem é expressa usando um conjunto de regras (ou produções) na notação BNF (*Backus Naur Form*), conforme descrito no Apêndice A.

A ferramenta JavaCC agrega as funcionalidades de gerador de analisador léxico e gerador de analisador sintático; dessa forma, o arquivo de especificação da gramática contém tanto as especificações léxicas quanto as sintáticas. Com a especificação da gramática para o JavaCC, é obtido um parser; entretanto, tal software serve somente para determinar se um dado programa atende ou não às regras da gramática. Para que o processo de parsing gere algum resultado (interpretação do programa) é necessário programação adicional. O JavaCC permite a inserção de trechos de código Java para geração de uma estrutura de dados representando o programa fruto do parsing, a árvore de Sintaxe Abstrata. O JJTree faz isto automaticamente, insere ações semânticas para criação da AST (*Abstract Syntax Tree*) e gera classes que modelam a estrutura da mesma.

JJTree é um pré-processador para o JavaCC, que insere ações para criação de uma árvore de sintaxe abstrata (AST) em vários lugares, em uma gramática JavaCC. A saída do JJTree é então processada pelo JavaCC para criação do parser, como mostrado na Figura 4.3.

```
TOKEN:
{      <K_DELETE:"DELETE">
     | <K_FROM:"FROM">
     | <K_WHERE:"WHERE">
     | <K_CURRENT:"CURRENT">
     | <K_OF:"OF">
     | <S_IDENTIFIER:(<LETTER>|<SPECIAL_CHARS>)* >
     | < #LETTER: ["a"-"z", "A"-"Z"] >
     | < #SPECIAL_CHARS: "$" | "_">
  }
void nDeleteStatement() #nDeleteStatement :
{}
{  tDELETE() [tFROM()] tIDENTIFIER()
   [tWHERE()(  vnDeleteWhereVar()
            | tCURRENT()tOF()tIDENTIFIER())]  tSEMICOLON()       }
void vnDeleteWhereVar()#vnDeleteWhereVar:
{}
{     nSQLExpression()        }
```

Figura 4.2 : Exemplo do código de entrada para JavaCC.

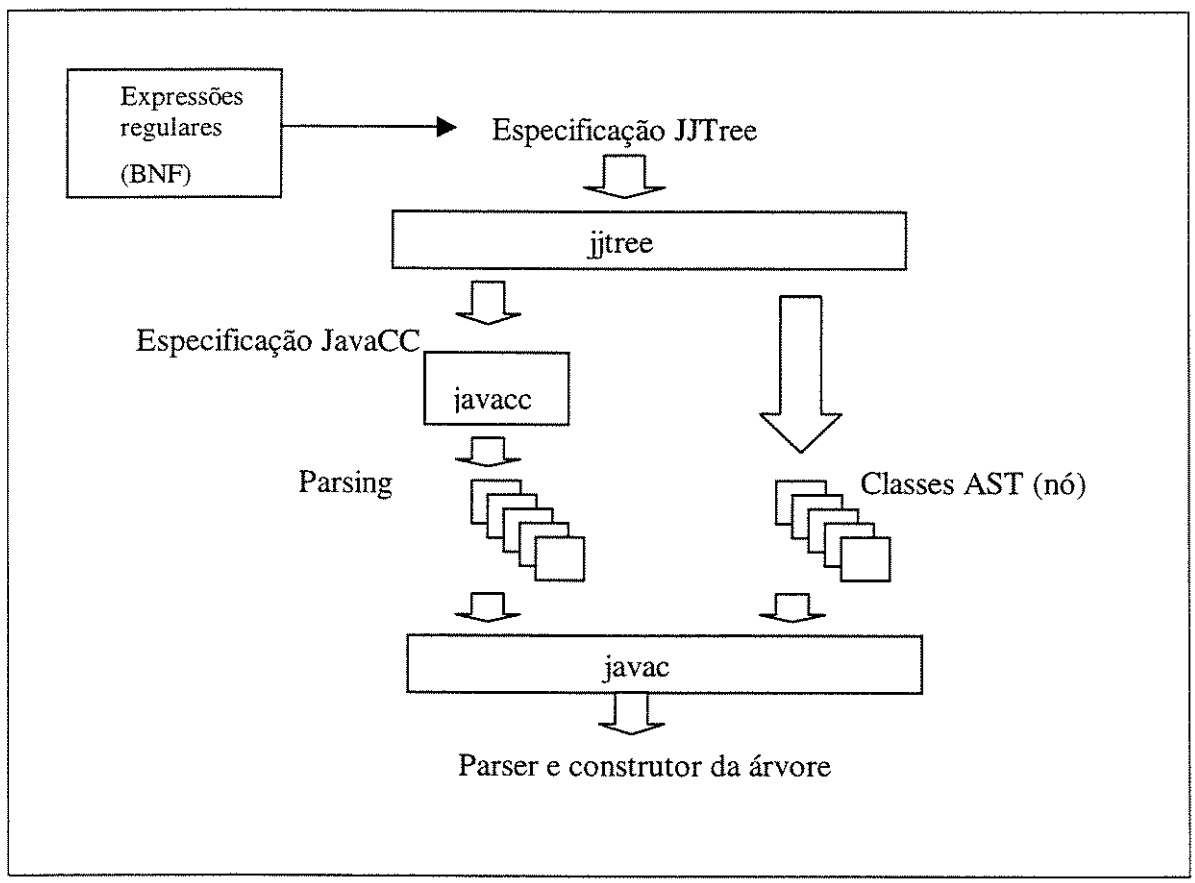


Figura 4.3: Geração dos analisadores léxico e sintático usando o JavaCC e do JJTree

4.3.2 Geração do Fluxo de Controle

Na fase de análise estática, para obter o Grafo de Fluxo de Controle (GFC), de acordo com o modelo proposto na Seção 3.2, são necessárias as informações obtidas a partir da árvore sintática. O GFC resultante, a saída principal deste módulo, é um arquivo contendo os números dos nós e as suas seqüências. Este arquivo tem o nome da regra em teste com extensão GFC (<nome da regra>.GFC) e está ilustrado na Figura 4.4. A sua organização deve obedecer à seguinte regra: o primeiro número deve indicar o número total de nós do grafo, os demais devem estar divididos em listas finalizadas pelo número 0; o primeiro número da lista indica o nó do GFC e os demais números, até o zero, são sucessores do primeiro, sendo que o número 0 é a representação de um finalizador de lista. A Figura 4.4 mostra o GFC correspondente ao fluxo de controle da regra ativa R1 (Figura 5.2).

```
13

1
   2 0
2
   8 3 0
3
   8 4 0
4
   8 5 0
5
   6 7 0
6
   11 0
7
   13 0
8
   9 10 12 0
9
   13 0
10
   13 0
11
   13 0
12
   13 0
13
   0
```

Figura 4.4: Arquivo com representação do GFC (regra ativa R1).

4.3.3 Geração do Fluxo de Dados

O grafo de fluxo de dados (GFD), gerado na análise estática, é construído a partir de informações referentes a cada nó, ou seja, informações das análises sintática e semântica dos comandos que compõem cada nó do GFC, com o intuito de identificar as ocorrências das variáveis. É necessário identificar o tipo de ocorrência nestes nós; mais especificamente, saber quais variáveis estão sendo definidas ou usadas e se o uso é computacional ou predicativo. A saída deste módulo é um arquivo, com a representação do GFD, denominado TABVARDEF.TES, nome usado para arquivo de entrada do módulo avaliador. A Figura 4.5 mostra o arquivo TABVARDEF.TES do GFD para a regra ativa R1 (Tabela 5.1).

```
#Tabela de Variaveis Definidas


$ 7
@ 0 e_salaryerror
@ 1 v_qtde
@ 2 v_empIno
@ 3 v_salary
@ 4 Employee
@ 5 log
@ 6 EXCEPTION
@@

# Grafo Def Sintetico


@@ 1
Defs: 5 4 @
C-Uses: 4 @
P-Uses: @
Refs: @
Undefs: @
@@ 2
Defs: 2 3 6 @
C-Uses: 4 @
P-Uses: @
Refs: @
Undefs: @
@@ 3
Defs: 5 6 @
C-Uses: 2 3 @
P-Uses: @
Refs: @
Undefs: @
@@ 4
Defs: 1 6 @
C-Uses: 4 3 @
P-Uses: @
Refs: @
Undefs: @
@@ 5
Defs: @
C-Uses: @
P-Uses: 1 @
Refs: @
Undefs: @
@@ 6
Defs: 0 @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
@@ 7
Defs: @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
@@ 8
Defs: @
C-Uses: @
P-Uses: 6 @
Refs: @
Undefs: @
@@ 9
Defs: 5 @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
@@ 10
Defs: 5 @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
@@ 11
Defs: 5 @
C-Uses: 0 @
P-Uses: @
Refs: @
Undefs: @
@@ 12
Defs: 5 @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
@@ 13
Defs: @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
```

Figura 4.5: Representação do arquivo que contém o GFD (Regra Ativa R1).

4.3.4 A Instrumentação

Na fase de análise estática também é gerada a versão instrumentada da regra. Para o monitoramento da execução dos casos de teste, é necessária a inserção de comandos de escrita no código fonte da regra em teste. A regra instrumentada é um arquivo com o nome do arquivo de entrada seguido do número 1 (<nome da regra>1.txt).

Este módulo tem como entrada principal o código fonte da regra ativa, que é reescrito baseado no modelo proposto na Seção 3.3 do Capítulo 3. Para a execução desta função também são necessárias informações do GFC para que se possa determinar o local de inserção dos comandos de escrita ou pontas de prova.

No banco de dados onde vai ser realizado o teste deve ser criada a tabela para definição do fluxo de execução - INSTRUM, de acordo com as especificações da Tabela 3.1 da Seção 3.3.4, Capítulo 3. Esta tabela armazenará dados referentes às indicações dos comandos de escrita, como mostra a Tabela 4.2.

Tabela 4.2: Representação dos atributos da tabela INSTRUM.

Código	**Regra**	**NumNó**	**Tabela**	**Obs**	**Caso Teste**
26/06/03	TgEmpl_Error Handling	8	tabela	EXCEPTION	1

Além da tabela deve ser criado um procedimento (*stored procedure*) para incluir dados na tabela INSTRUM. Os parâmetros deste procedimento são o nome da regra, o número do nó, o número do caso de teste e a tabela que está sendo afetada pelo comando. A *stored procedure* recebe os parâmetros e insere-os na tabela INSTRUM, como pode ser observado na Figura 4.6. Nesta figura, no item *a* está ilustrado um comando de escrita da instrumentação e no item *b* a codificação da *stored procedure* denominada *ponta-de-prova*. A instrumentação da regra R1 é mostrada nas Figuras 4.7 e 4.8, onde os comandos de escrita inseridos estão em negrito.

```
a ) Ponta_de_prova ('R1', 1, 'tabela', 'EVENTO', 1)

b ) PROCEDURE   PONTA_DE_PROVA
    (NOME INSTRUM.REGRA%TYPE,  NNO  INSTRUM.NO%TYPE,
     NOMTAB INSTRUM.TABELA%TYPE,  NOBS   INSTRUM.OBS%TYPE,
     NCASOT  INSTRUM.CASOTESTE%TYPE)
   IS
  BEGIN
     INSERT INTO TESTE (CODIGO, REGRA, NUMNO, TABELA,OBS, CASOTESTE)
     VALUES (SYSDATE, NOME, NNO, NOMTAB, NOBS, NCASOT);
  END;
```

Figura 4.6: Código da *stored procedure* ***Ponta-de-Prova*** ((a) comando de escrita, (b) codificação da *stored procedure*).

Para instrumentar é necessário manter a regra original e inserir comandos de escritas nos locais apropriados. A partir dos *tokens* nas posições analisadas e definidas para instrumentação são incluídas chamadas à *stored procedure*. A instrumentação do código, de acordo com o GFC, é controlada com funções específicas que determinam o início e fim de cada nó. Além de caracterizar os nós a instrumentação verifica a presença do bloco de tratamento de exceção, pois quando este não está presente é necessário a sua inclusão.

Particularidades da instrumentação podem ser vistas nas Figuras 4.7 e 4.8; por exemplo, no início, duas pontas de prova em seqüência (nós 1 e 2), uma para referenciar o evento e a outra, já indicando o comando de início da ação da regra. No bloco de exceção também são visíveis estes dois comandos de escrita juntos em cada opção da exceção, sendo um para caracterizar o bloco de exceção e o outro para o comando interno da opção (nós 8, 9, 10 e 12). Ainda no bloco de exceção, o nó final da regra é referenciado para todas as opções, pois sempre que uma opção for executada a regra finaliza e este final deve ser registrado.

```
CREATE or replace TRIGGER R1

AFTER DELETE OR INSERT OR UPDATE

ON Employee

        DECLARE
e_salaryerror EXCEPTION ;
v_qtde INTEGER ;
        v_emplno employee.emplno %type ;
v_salary employee.salary %type ;

BEGIN

    /*1*/  Ponta_de_prova ('R1', 1, 'tabela', 'EVENTO', 1);

     /*2*/  Ponta_de_prova ('R1', 2, 'tabela', 'SELECT', 1);

    SELECT emplno , salary INTO v_emplno , v_salary

    FROM Employee WHERE emplsupervisor IS NULL ;

    /*3*/  Ponta_de_prova ('R1', 3, 'tabela', 'INSERT', 1);

    INSERT INTO log (timetamp,userid, status, description, tableid , tablekey, oldvalue , newvalue , alert)

    VALUES ( SYSDATE , user , 'S','check president' , 'employee' , NULL , NULL , NULL ,
                'The salary of the president ' || TO_CHAR (v_emplno ) || ' is ' || TO_CHAR ( v_salary ) ) ;

    /*4*/  Ponta_de_prova ('R1', 4, 'tabela', 'SELECT', 1);

    SELECT COUNT ( * ) INTO v_qtde
    FROM Employee WHERE ( salary > v_salary ) ;
   /*5*/  Ponta_de_prova ('R1', 5, 'tabela', 'IF', 1);

    IF ( v_qtde > 0 ) THEN

         /*6*/  Ponta_de_prova ('R1', 6, 'tabela', 'RAISE', 1);

        RAISE e_salaryerror ;

            END IF ;

   /*7*/   Ponta_de_prova ('R1', 7, 'tabela', 'nIfFinal' , 1);
```

Figura 4.7: Representação da Regra Ativa Instrumentada.

```
/*13*/ Ponta_de_prova ('R1', 13, 'tabela', 'END Final', 1);

EXCEPTION

   WHEN NO_DATA_FOUND THEN

            /*8*/ Ponta_de_prova ('R1', 8, 'tabela', 'EXCEPTION', 1);
    /*9*/ Ponta_de_prova ('R1', 9, 'tabela', 'INSERT', 1);

    INSERT INTO log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert )

    VALUES ( SYSDATE , user , 'E' , 'check president' , 'employee' , NULL , NULL , NULL ,
                'The company has no president' ) ;

    /*13*/ Ponta_de_prova ('R1', 13, 'tabela', 'ENDFinal', 1);

    WHEN TOO_MANY_ROWS THEN

    /*8*/ Ponta_de_prova ('R1', 8, 'tabela', 'EXCEPTION', 1);
    /*10*/ Ponta_de_prova ('R1', 10, 'tabela', 'INSERT', 1);

    INSERT INTO log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert )

    VALUES ( SYSDATE , user , 'E' , 'check president' , 'employee' , NULL , NULL , NULL ,
                'The company has too many presidents' ) ;

    /*13*/ Ponta_de_prova ('R1', 13, 'tabela', 'ENDFinal', 1);

    WHEN e_salaryerror THEN

    /*11*/ Ponta_de_prova ('R1', 11, 'tabela', 'EXCEPTION-USER', 1);
    /*11*/ Ponta_de_prova ('R1', 11, 'tabela', 'INSERT');

    INSERT INTO log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert )

    VALUES ( SYSDATE , user , 'E' , 'check president' , 'employee' , NULL , NULL , NULL ,
                'This salary is not permited' ) ;

    /*13*/ Ponta_de_prova ('R1', 13, 'tabela', 'ENDFinal', 1);

   WHEN OTHERS THEN

   /*8*/ Ponta_de_prova ('R1', 8, 'tabela', 'EXCEPTION', 1);
  /*12*/ Ponta_de_prova ('R1', 12, 'tabela', 'INSERT', 1);

   INSERT INTO log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert)

   VALUES ( SYSDATE , user , 'E' , 'check president' , 'employee' , NULL , NULL , NULL ,
                'Error checking president' ) ;

    /*13*/ Ponta_de_prova ('R1', 13, 'tabela', 'ENDFinal', 1);
END ;
```

Figura 4.8: Representação da Regra Ativa Instrumentada.

4.3.5 Geração dos Elementos Requeridos

Um módulo da POKE-TOOL é utilizado para gerar os elementos requeridos na fase de análise estática. Este módulo, o *Newdescr*, é responsável por parte da análise estática do código fonte. Esta análise é fundamental para auxiliar o testador na elaboração dos casos de teste e na aplicação dos critérios Potenciais Usos. Estes critérios são critérios caixa-branca, ou seja, derivam seus requisitos para o teste a partir da estrutura da unidade (Chaim, 1991). São gerados os elementos requeridos a partir das informações dos fluxos de controle, de dados (respectivamente, arquivos <nome da regra>.GFC e TABVARDEF.TES – Figuras 4.4 e 4.5) e a lista de critérios que serão utilizados. Sua saída principal são arquivos com caminhos (e.g., PDUPATHS.TES) e associações (e.g., PUASSOC.TES) requeridas para cada critério. O exemplo desses arquivos do conjunto de elementos requeridos obtidos para cada critério está mostrado nas Figuras 4.9 e 4.10.

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS
POT-DU-CAMINHOS

 1) 1 2 8 12 13
 2) 1 2 8 10 13
 3) 1 2 8 9 13
 4) 1 2 3 8 12 13
 5) 1 2 3 8 10 13
 6) 1 2 3 8 9 13
 7) 1 2 3 4 8 12 13
 8) 1 2 3 4 8 10 13
 9) 1 2 3 4 8 9 13
10) 1 2 3 4 5 7 13
11) 1 2 3 4 5 6 11 13
```

Figura 4.9:Representação do arquivo PDUPATHS.TES.

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

1) <1,(2,8),{ Employee, Log }>
2) <1,(3,8),{ Employee }>
3) <1,(8,12),{ Employee }>
4) <1,(8,12),{ Log }>
5) <1,(8,10),{ Employee }>
6) <1,(8,10),{ Log }>
7) <1,(8,9),{ Employee }>
8) <1,(8,9),{ Log }>
9) <1,(4,8),{ Employee }>
10) <1,(5,7),{ Employee }>
11) <1,(5,6),{ Employee }>
12) <1,(2,3),{ Employee, Log }>
```

Figura 4.10:Representação do arquivo PUASSOC.TES.

4.3.6 Execução dos Casos de Teste e Geração dos Caminhos Exercitados

A etapa de teste inicia-se com a execução dos casos de teste. Os arquivos contendo os elementos requeridos para os critérios Potenciais Usos auxiliam o usuário a projetar os casos de teste. Esses casos de teste devem incluir dados de entrada que são os estados inicial e final do banco de dados, eventos e a saída esperada para estes dados, como está detalhado no Capítulo 5, Seção 5.2.1.

Neste módulo é possível monitorar e obter informações da execução dos casos de teste. Tem como entrada a regra ativa instrumentada e os casos de teste a serem executados. Após a execução dos casos de teste, executa-se uma rotina, escrita em SQL (GERA_TXTORA.SQL), para extrair da tabela INSTRUM os caminhos percorridos na execução de cada caso de teste. Nesta tabela estão os números dos nós, capturados durante a execução do caso de teste, ordenados seqüencialmente. A saída é um arquivo (PATH<numero do caso de teste>.TES) com o caminho exercitado. O arquivo PATH3.TES do exemplo da Seção 5.2 é mostrado na Figura 4.11.

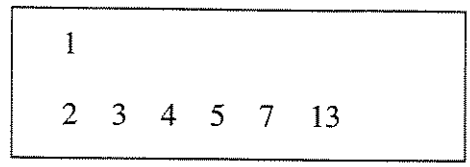


Figura 4.11: Arquivo com representação do arquivo PATH3.TES

4.3.7 Avaliação

A avaliação dos dados obtidos pelo teste inicia-se com a comparação entre resultados obtidos e resultados esperados para determinar se os casos de teste revelaram defeitos. É possível ter uma cobertura do caso de teste de acordo com o critério usado, confrontando elementos requeridos com os elementos exercitados. O módulo avaliador da POKE-TOOL tem como entradas, para cada critério, o conjunto de elementos requeridos[3], gerados no módulo *Newpokeaval*; e os caminhos exercitados, obtidos no módulo de geração dos caminhos exercitados. Como saída tem-se, para cada critério, a cobertura atingida para cada caso de teste, os caminhos ou associações que restam para ser exercitados e os caminhos e associações efetivamente exercitadas. Nas Figuras 4.12 e 4.13 são mostradas as informações obtidas neste módulo.

```
ASSOCIACOES DO CRITERIO TODOS-USOS executadas:

<1,2, c_employee>,<1,4, c_employee>
<2,3, v_emplno>, <2,3, v_salary>
<2,4, v_salary>
<2,(2,3), EXCEPTION>, <2,(2,8), EXCEPTION>
<2,(3,4), EXCEPTION>, <2,(8,9), EXCEPTION>
<2,(8,10), EXCEPTION>
<3,(3,4), EXCEPTION>, <3,(4,5), EXCEPTION>
<4,(5,6), v_qtde>, <4,(5,7), v_qtde>
<4,(4,5), EXCEPTION>,<6,11, e_salaryerror>

Cobertura Total = 59%
```

Figura 4.12: Representação das associações executadas.

[3] Para a avaliação, a POKE-TOOL utiliza uma representação especial dos elementos requeridos na forma de autômatos finitos (Chaim, 1991).

```
ASSOCIACOES DO CRITERIO TODOS-USOS nao
executadas:

<2,(3,8), EXCEPTION>
<2,(8,12), EXCEPTION>
<3,(3,8), EXCEPTION>
<3,(4,8), EXCEPTION>
<3,(8,9), EXCEPTION>
<3,(8,10), EXCEPTION>
<3,(8,12), EXCEPTION>
<4,(4,8), EXCEPTION>
<4,(8,9), EXCEPTION>
<4,(8,10), EXCEPTION>
      <4,(8,12), EXCEPTION>
```

Figura 4.13: Representação das associações não executadas.

4.4 Considerações Finais

Neste capítulo foi apresentada a ferramenta ART-TOOL (*Active Rule Testing Tool*) desenvolvida com o objetivo de apoiar o teste estrutural de regras ativas escritas segundo o padrão SQL-3.

As principais funcionalidades desta ferramenta são:

- Análise do código de uma regra ativa escrita no padrão SQL-3.
- Análise da regra e geração do grafo de fluxo de controle e do grafo de fluxo de dados.
- Geração da versão instrumentada do código fonte da regra em teste.
- Geração do arquivo com informações do caminho exercitado pelo caso de teste.
- Uso de informações da análise estática do código fonte e módulos da ferramenta POKE-TOOL (Chaim, 1991) para gerar os elementos requeridos e avaliar os dados obtidos no teste.

As três primeiras funcionalidades são implementadas como partes da etapa de análise estática pelo módulos: Análises Léxica e Sintática, Geração do Fluxo de Controle, Geração do Fluxo de Dados e Instrumentação. A partir do código fonte da regra ativa são geradas informações do fluxo de controle e do fluxo de dados; é obtida uma versão instrumentada da regra para possibilitar o acompanhamento da execução dos casos de teste. O primeiro módulo da POKE-TOOL – *Newdescr* - é utilizado na fase de análise estática; a partir de informações do fluxo de controle e do fluxo de dados este gera os elementos requeridos. O arquivo com os elementos requeridos auxilia o testador a elaborar os casos de teste e também é necessário para o módulo avaliador. Os caminhos exercitados pelo conjunto de casos de teste são obtidos na fase de teste, esta funcionalidade é implementada pelo módulo Geração dos Caminhos Exercitados. Para a avaliação do teste utiliza-se o módulo da POKE-TOOL, o *Newpokeaval*, que exige como entrada os elementos requeridos para cada critério e os caminhos exercitados; fornece uma análise de cobertura, ou seja, o percentual de elementos requeridos que foram exercitados pelo conjunto de casos de teste.

No Capítulo 5 é apresentado um exemplo de aplicação da ART-TOOL com o objetivo de ilustrar o seu funcionamento.

CAPÍTULO 5

APLICAÇÃO DA ART-TOOL

5.1 Introdução

A aplicação da ART-TOOL é mostrada pelo teste de uma de regra ativa R1, associada a um banco de dados de demonstração baseado em Elmasri e Navathe (2000) e descrito no Apêndice B. O exemplo de regra ativa procura explorar alguns comandos da linguagem SQL e mostra como a ferramenta abstrai os elementos requeridos.

É realçada a automação dos modelos de fluxo de controle, de fluxo de dados e de instrumentação. A aplicação de tais modelos para a geração de elementos requeridos pelos critérios utilizados é avaliada.

Na seqüência é detalhado o processo de teste; a análise estática está descrita na Seção 5.2; o projeto de casos de teste é discutido na Seção5.3; a execução é descrita na Seção 5.4. A análise de cobertura e os resultados do teste da regra R1 são discutidos na Seção 5.5.

A ART-TOOL também foi utilizada para avaliar um conjunto de regras ativas. Inicialmente foram elaborados vários exemplos de regras ativas associadas ao banco de dados de demonstração do Apêndice B. Os exemplos elaborados foram propostos com os mesmos objetivos da regra R1. Também foi utilizado um exemplo real, cedido por uma empresa anônima, composto de cinco regras. Ao todo foram executadas quinze regras. Os resultados destes testes são discutidos na Seção 5.6.

5.2 *Análise Estática da Regra Ativa R1*

A descrição da regra R1 bem como seu código fonte são apresentados na Tabela 5.1. O módulo de análise estática está ilustrada na Figura 5.1 com sua entrada principal e suas respectivas saídas. O dado de entrada é um arquivo texto, contendo o código fonte de uma regra; as regras são consideradas individualmente.

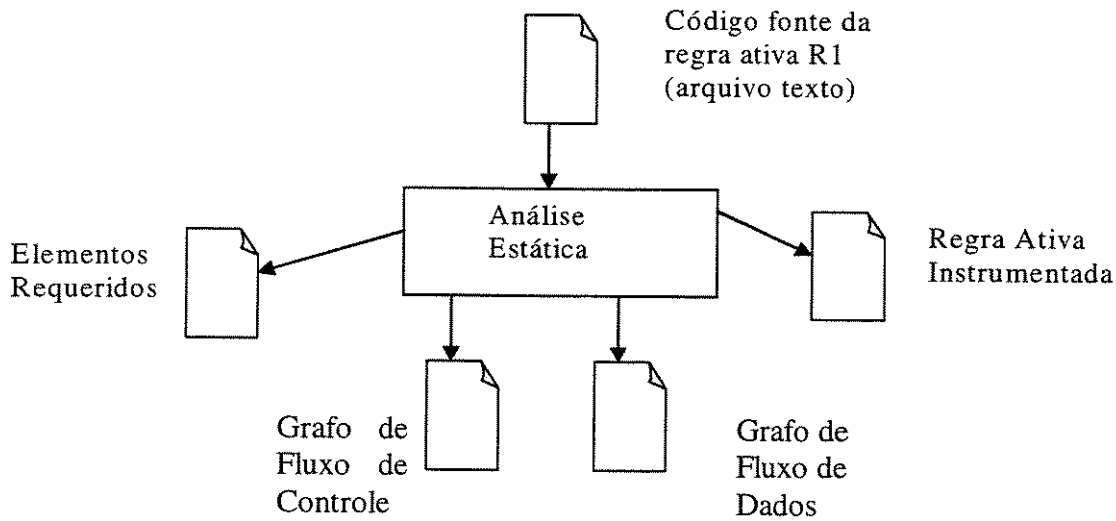


Figura 5.1: Módulo de análise estática da ART-TOOL.

As saídas desta fase incluem: o fluxo de controle apresentado na Figura 5.2; o fluxo de dados apresentado na tabela 5.2; a versão instrumentada da regra; e a geração dos elementos requeridos.

Tabela 5.1: Descrição resumida e código fonte da Regra Ativa R1.

Regra R1 Descrição	Esta regra se inicia a partir de qualquer alteração, inclusão ou exclusão na tabela de empregados. Como nesta tabela é possível que apenas um empregado seja supervisor, a regra verifica a sua existência, analisa seu salário, que deve ser superior a todos os inclusos nesta tabela e o considera em uma tabela de ocorrências.
Código	/* XX */ - Indica o número dos respectivos nós /*1*/ CREATE TRIGGER R1 AFTER DELETE OR INSERT OR UPDATE ON Employee DECLARE e_salaryerror EXCEPTION; v_qtde INTEGER; v_emplno employee.emplno%type; v_salary employee.salary%type; BEGIN /*2*/ SELECT emplno, salary INTO v_emplno, v_salary FROM Employee WHERE emplsupervisor IS NULL; /*3*/ INSERT INTO Log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert) VALUES (SYSDATE, user, 'S', 'check president', 'employee', NULL, NULL, NULL, 'The salary of the president ' || TO_CHAR(v_emplno) || ' is ' || TO_CHAR(v_salary)); /*4*/ SELECT COUNT(*) INTO v_qtde FROM Employee WHERE (salary > v_salary); /*5*/ IF (v_qtde > 0) THEN /*6*/ RAISE e_salaryerror; /*7*/ END IF; /*8*/ EXCEPTION WHEN NO_DATA_FOUND THEN /*9*/ INSERT INTO Log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert) VALUES (SYSDATE, user, 'E', 'check president', 'employee', NULL, NULL, NULL, 'The company has no president'); WHEN TOO_MANY_ROWS THEN /*10*/ INSERT INTO Log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert) VALUES (SYSDATE, user, 'E', 'check president', 'employee', NULL, NULL, NULL, 'The company has too many presidents'); WHEN e_salaryerror THEN /*11*/ INSERT INTO Log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert) VALUES (SYSDATE, user, 'E', 'check president', 'employee', NULL, NULL, NULL, 'This salary is not permited'); WHEN OTHERS THEN /*12*/ INSERT INTO Log (timetamp, userid, status, description, tableid, tablekey, oldvalue, newvalue, alert) VALUES (SYSDATE, user, 'E', 'check president', 'employee', NULL, NULL, NULL, 'Error checking president'); /*13*/ END;

Os aspectos mais relevantes apresentados no fluxo de controle de R1 são os desvios de fluxo de cada comando de manipulação, devido ao fato da regra possuir um bloco de tratamento de exceção. Esta também possui um desvio para um nó específico, onde uma exceção é declarada. Para exemplificar, considere o nó 2, onde pode ocorrer desvio para o nó 8 (Figura 5.2). O nó 2 é um comando SELECT, indicado no código fonte na Tabela 5.1, que tem como objetivo selecionar o supervisor e seu salário na tabela EMPLOYEE. No caso de não existir o supervisor ocorre um desvio para o nó 8, início do bloco de tratamento de exceção, e haverá a seleção para a exceção específica. O predicado de exceção é NO_DATA_FOUND; o controle flui do nó 8 para o nó 9 e a execução da regra acaba no nó 13.

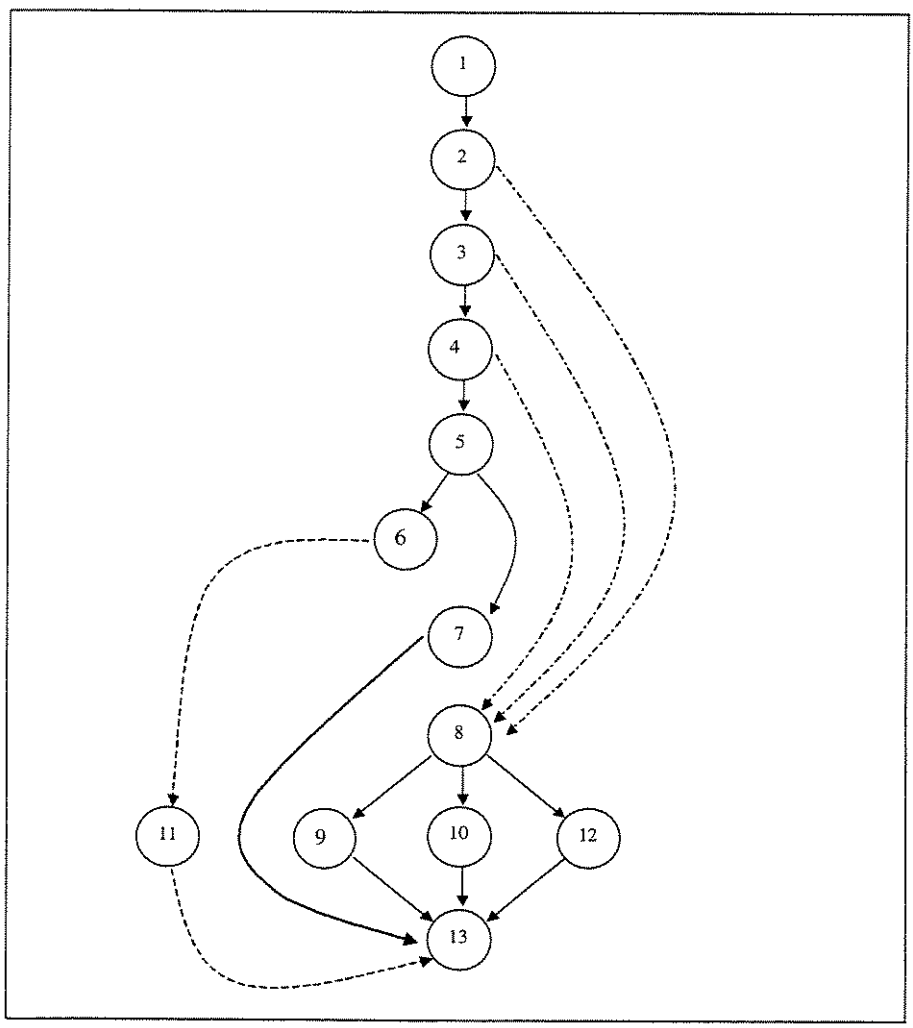


Figura 5.2: Representação do Fluxo de Controle da Regra Ativa R1.

Não são em todos comandos de manipulação que ocorrem exceções. No modelo de fluxo de controle proposto há um desvio para a área de exceção para todo comando de manipulação, podendo assim haver caminhos não executáveis. Para definir exatamente os comandos de manipulação que podem desviar para uma exceção é necessário um estudo detalhado de cada comando que está inserido no código. As soluções para esse problema não são genéricas, uma vez que em comandos de manipulação podem conter outros comandos de manipulação, como no caso de subconsultas.

O fluxo de dados apresenta particularidades específicas de regras ativas. No nó de evento, além das variáveis definidas explicitamente, são definidas as variáveis persistentes que não aparecem neste bloco mas que são referenciadas na ação da regra. A ocorrência implícita de definição dessas variáveis verifica-se a partir do disparo da regra e de todas as tabelas do banco de dados que estão disponíveis para uma manipulação pela regra ativa. Na regra R1, especificamente no nó 1, há a ocorrência de uma definição da variável persistente LOG, que não faz parte do nó de evento, como mostrado na Tabela 5.2. Associações de fluxo de dados são estabelecidas para a variável *exception* a partir dos nós com sua definição explícita até os arcos que incluem uso predicativo da variável. Na Tabela 5.2 observa-se a ocorrência da definição da variável *exception* devido à presença dos comandos de manipulação, como descrito nos nós 2, 3 e 4, pois esta regra possui um bloco de tratamento de exceção. Ocorre uso predicativo da variável *exception*, ocasionado pelos desvios de fluxo para o bloco de exceção, como observado nos arcos (2, 3), (2, 8), (3, 4), (3, 8), (4, 5) e (4, 8). Usos predicativos da variável *exception* também são observados nos arcos (8,9), (8,10) e (8,12), conforme indicado na Tabela 5.2. No caso de exceção definida pelo usuário, ocorre uma definição da variável *e_salaryerror* no nó 6 e um posterior uso predicativo no arco (6, 11).

Tabela 5.2: Representação do fluxo de dados da Regra Ativa R1.

Nó/ Arco	Definição	C-uso	P-uso
1	Employee, Log	Employee	
2	v_emplno,v_salary, exception	Employee	
3	Log, exception	v_emplno, v_salary	
4	v_qtde, exception	Employee, v_salary	
(2,3), (2,8), (3,4), (3,8), (4,5), (4,8)			exception
(5,6),(5,7)			v_qtde
6	e_salaryerror		
(8,9),(8,10), (8,12)			exception
9	Log		
10	Log		
11	Log		
(6, 11)			e_salaryerror
12	Log		

Outro produto da fase estática é a regra instrumentada. A nova versão da regra R1, com comandos de escrita inseridos para possibilitar o acompanhamento da execução dos casos de teste está ilustrada no Capítulo 4, nas Figuras 4.7 e 4.8.

Ao final da análise estática são gerados os elementos requeridos a partir de informações do fluxo de controle e do fluxo de dados; são caminhos ou associações requeridos de acordo com os critérios escolhidos pelo testador, visando ao exercício de diferentes partes da regra. Os elementos requeridos da regra R1 são apresentados nas Tabelas 5.3 e 5.4.

Tabela 5.3: Elementos Requeridos da Regra R1para os critérios especificados.

Critérios	Elementos Requeridos
Todos Nós	1 2 3 4 5 6 7 8 9 10 11 12 13
Todos Arcos	(2,8), (3,8), (4,8), (5,6), (5,7), (8,9), (8,10), (8,12)
Todos Usos	<1,2, Employee>, <1,4, Employee>, <2,3, v_emplno>, <2,3, v_salary>, <2,4, v_salary> <2,(3,4), exception>, <2,(2,8), exception>, <4,(4,8), exception>, <4,(5,6), v_qtde> <2,(8,9), exception>, <2,(8,10), exception >, <2,(8,12), exception> <2,(3,8), exception>, <3,(3,4), exception>, <3,(4,5), exception>, <6,11, e_salaryerror> <3,(3,8), exception>, <3,(4,8), exception>, <4,(4,5), exception>, <4,(5,7), v_qtde>, <3,(8,9), exception>, <3,(8,10), exception >, <3,(8,12), exception > <4,(8,9), exception>, <4,(8,10), exception>, <4,(8,12), exception >

Tabela 5.4: Elementos Requeridos da Regra R1 para os critérios apresentados.

Critérios	Elementos Requeridos
Todos P-Usos	<4,(5,6), v_qtde> , <4,(5,7), v_qtde>, <6,11, e_salaryerror> <2,(2,3), exception>,<2,(2,8), exception>,<2,(3,4), exception> <2,(3,8), exception>,<3,(3,4), exception>, <3,(3,8), exception> <3,(4,5), exception>,<3,(4,8), exception>, <4,(4,5), exception> <2,(8,9), exception>, <2,(8,10), exception>, <2,(8,12), exception> <3,(8,9), exception>, <3,(8,10), exception>, <3,(8,12), exception> <4,(4,8),exception>,<4,(8,9),exception>,<4,(8,10),exception>, <4,(8,12),exception>
Todos Pot-Uso e Todos Pot-Uso/Du	<1,(2,8),{Employee,Log}>, <1,(3,8),{Employee}>, <1,(8,12),{Employee}>, <1,(8,12),{Log}>, <1,(8,10),{Employee}>, <1,(8,10),{Log}>, <1,(8,9),{Employee}>, <1,(8,9),{ Log }>, <1,(4,8),{Employee}>, <1,(5,7),{Employee}>, <1,(5,6),{Employee}>, <1,(2,3),{Employee,Log }> <2,(2,8),{v_emplno,v_salary,exception}>, <2,(3,8),{v_emplno,v_salary}>,<2,(8,12),{exception}> <2,(8,12),{v_emplno,v_salary}>,<2,(8,10),{exception}> <2,(8,10),{v_emplno,v_salary}>,<2,(8,9),{exception}> <2,(8,9),{v_emplno,v_salary}>,<2,(4,8),{v_emplno,v_salary}>, <2,(5,7),{v_emplno,v_salary}>, <2,(2,3),{v_emplno,v_salary,exception}>, <2,(5,6),{v_emplno,v_salary}>, <3,(3,8),{ Log, exception }>, <3,(8,12),{Log}>, <3,(8,12),{exception}>, <3,(8,10),{ Log }>, <3,(8,10),{exception}>, <3,(8,9),{ Log }>, <3,(8,9),{ exception }>, <3,(4,8),{ Log }>, <3,(5,7),{ Log }>, <3,(6,11),{ Log }>, <3,(5,6),{ Log }>, <3,(3,4),{ Log, exception }> <4,(8,12),{ exception }>, <4,(8,12),{ v_qtde }>, <4,(8,10),{ exception }>, <4,(8,10),{ v_qtde }>, <4,(8,9),{ exception }>, <4,(8,9),{ v_qtde }>, <4,(4,8),{ v_qtde, exception }>, <4,(5,7),{ v_qtde, exception }>, <4,(5,6),{ v_qtde, exception }> <6,{e_salaryerror}>, <9,{Log}>, <10,{Log}>, <11,{Log}>, <12,{Log}>
Todos Du-Caminhos	(1 2), (1 2 3 4), (2 3), (2 8), (2 3 4), (2 3 8), (2 8 9), (2 8 10), (2 8 12), (3 4), (3 8), (3 4 5). (3 4 8), (3 8 9), (3 8 10), (3 8 12), (4 5), (4 8), (4 5 6), (4 5 7), (4 8 9), (4 8 10), (4 8 12), (6 11)
Todos Pot-Du Caminhos	(1 2 8 12 13), (1 2 8 10 13), (1 2 8 9 13),(1 2 3 8 12 13), (1 2 3 4 5 7 13), (1 2 8 10 13),(1 2 3 8 9 13), (1 2 3 4 8 12 13), (1 2 3 4 8 10 13), (1 2 3 4 8 9 13), (1 2 3 4 5 6 11 13), (2 8 12 13), (2 8 10 13), (2 8 9 13), (4 5 6 11 13), (2 3 8 12 13), (2 3 8 10 13), (2 3 8 9 13), (2 3 4 8 12 13), (6 11 13), (4 8 9 13), (2 3 4 8 10 13), (2 3 4 8 9 13), (2 3 4 5 7 13), (10 13),(11 13), (12 13) (2 3 4 5 6 11 13), (3 8 12 13), (3 8 10 13), (3 8 9 13), (4 5 7 13), (9 13), (3 4 8 12), (3 4 8 10), (3 4 8 9), (3 4 5 7 13), (3 4 5 6 11), (4 8 12 13), (4 8 10 13)

5.3 *Projeto de Casos de Teste*

Com os elementos requeridos, o testador tem em mãos informações que podem ser usadas para elaborar os casos de teste.

O objetivo é projetar casos de teste que tenham a maior probabilidade de encontrar defeitos com um mínimo de tempo e esforço. Para o teste estrutural de programas convencionais um caso de teste é o conjunto formado por dados de entrada e o resultado esperado para esses dados de entrada.

Os dados de entrada para unidades de programas convencionais são valores atribuídos às variáveis de entrada, ou seja, variáveis locais e globais. Dados de entrada em regras ativas diferem no sentido de que o domínio de entrada não é o mesmo, pois envolve variáveis locais, variáveis de transição e variáveis persistentes.

O estado da base deve ser considerado como parte do conjunto de entrada pois afeta a alteração dos valores das variáveis de transição. Neste sentido, a operação que provocou o evento é também parte integrante do domínio de entrada.

Os eventos, por provocarem a ativação de regras ativas, devem ser considerados na elaboração dos casos de teste. É necessário incluir os eventos possíveis como parte dos casos de teste; a justificativa é que os eventos podem ter defeitos. Por exemplo, uma regra que deve ser disparada somente quando ocorrer uma inclusão poderia estar sendo disparada por exclusão. Em outro caso, uma regra que dispare quando ocorre uma inclusão, alteração ou exclusão, pode estar restrita a ser ativada somente na ocorrência de uma exclusão. Para observar esta classe de defeitos é necessário que o conjunto de casos de teste inclua todos os tipos de eventos (INSERT, DELETE, UPDATE).

Concluindo, um caso de teste de uma regra ativa inclui o estado inicial da base que refere aos valores das variáveis locais, variáveis de transição e variáveis persistentes antes da ocorrência do evento; o evento, as operações de manipulação de dados; e o estado final da base esperado.

A geração dos casos de teste da regra R1 foi feita a partir dos elementos requeridos. Exemplos de casos de teste da regra R1 são descritos sistematicamente a seguir.

Caso de Teste 1

1 - Descrição: C1 - Atualização do salário de um supervisor e inclusão de seu supervisor. Situação não permitida na tabela EMPLOYEE, tabela sem supervisor; visa o exercício de uma exceção.

2 - Evento: UPDATE EMPLOYEE

SET salary = (salary + 250) ,

EmplSupervisor = '981444'

WHERE emplSupervisor IS NULL;

3 - Base de dados antes do disparo da regra:

Tabela: **Employee**

Empno	Salary	EmplSupervisor	
981210	1200,00	981222	
981222	3550,00		} Supervisor
981333	895,00	981222	
:	:	:	

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert
22/03/03	P08	S	Check president	Employee	The salary of the president 981222 is 3550,00
:	:	:	:	:	:

4 - Base de dados esperada após a execução da regra:

Tabela: **Employee**

Empno	Salary	EmplSupervisor
981210	1200,00	981222
981222	3800,00	981444
981333	895,00	981222
:	:	:

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert
22/03/03	P08	S	Check president	Employee	The salary of the president 981222 is 3550,00
23/03/03	P08	E	Check president	Employee	The company has no president.
:	:	:	:	:	:

Caso de Teste 2

1 - Descrição: C2 – Exclusão de um empregado que não exerce a função de supervisor.

A tabela EMPLOYEE encontra-se com dois supervisores, o que não é permitido, provocando uma outra exceção.

Obs: só é possível ter mais de um supervisor na tabela EMPLOYEE se a regra R1 não estiver ativa no momento da inclusão de outros supervisores.

2 - Evento: DELETE FROM EMPLOYEE

WHERE emplno = '981333';

3 - Base de dados antes do disparo da regra:

Tabela: **Employee**

Emplno	Salary	EmplSupervisor
981210	1200,00	981222
981222	3550,00	
981333	895,00	981222
981444	1100,00	
:	:	:

} Supervisor

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert

4- Base de dados esperada após a execução da regra:

Tabela: **Employee**

Emplno	Salary	EmplSupervisor
981210	1200,00	981222
981222	3550,00	
981444	1100,00	
:	:	:

Supervisor

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert
22/03/03	P08	E	Check president	employee	The company has too many presidents.

Caso de Teste 3

1 - Descrição: C3 – Inclusão de um empregado que vai exercer a função de supervisor. É uma ocorrência comum, que acontece a todo instante.

2 - Evento: INSERT INTO EMPLOYEE

VALUES ('981222', 3550.00, NULL);

3 - Base de dados antes do disparo da regra:

Tabela: **Employee**

Empno	Salary	EmplSupervisor

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert

4 - Base de dados esperada após a execução da regra:

Tabela: **Employee**

Empno	Salary	EmplSupervisor
981222	3550,00	

} Supervisor

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert
22/03/03	P08	S	Check president	employee	The salary of the president is 3550,00.

Caso de Teste 4

1 - Descrição: C4 – Inclusão de um empregado com o salário maior do que o salário de seu supervisor. Situação não permitida, o que acarretará uma exceção determinada pelo programador.

2 - Evento: INSERT INTO EMPLOYEE

VALUES ('981444', 4000.00, '981222');

3 - Base de dados antes do disparo da regra:

Tabela: **Employee**

Emplno	Salary	EmplSupervisor
981210	1200,00	981222
981222	3550,00	
981333	895,00	981222
⋮	⋮	⋮

} Supervisor

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert
23/03/03	P08	S	Check president	employee	The salary of the president 981222 is 3550,00.

4 - Base de dados esperada após a execução da regra:

Tabela: **Employee**

Emplno	Salary	EmplSupervisor
981210	1200,00	981222
981222	3550,00	
981333	895,00	981222
981444	4000,00	981222
⋮	⋮	⋮

} Supervisor

Tabela: **Log**

Timetamp	Userid	Status	Description	Tableid	Alert
22/03/03	P08	S	Check president	employee	The salary of the president 981222 is 3550,00.
23/03/03	P08	E	Check president	employee	This salary is not permited

5.4 Execução do Teste com C1, C2, C3 e C4

Com a versão instrumentada, obtida a partir da fase estática, e os casos de teste elaborados, inicia-se a fase de execução. Para executar o teste é necessário observar o estado do banco de dados antes e depois do disparo da regra.

Durante a execução dos casos de teste obtêm-se o caminho exercitado por estes. Para ter uma visão mais completa das partes que foram exercitadas com os casos de teste são mostrados os elementos exercitados de acordo com cada critério, ilustrados nas Tabelas 5.5 e Tabela 5.6. Os elementos exercitados com base nos critérios Potenciais Usos estão ilustrados na Tabela 5.6 e observa-se que poucos elementos requeridos foram exercitados.

Na Tabela 5.5 são mostrados os elementos exercitados quando critérios menos exigentes foram aplicados. Como esta regra possui uma área de tratamento de exceção, o conjunto de elementos requeridos possui várias associações contendo a variável *exception* mas não foi possível exercitar todas as associações desta variável, aspecto este discutido na Seção 5.5.

Tabela 5.5: Elementos exercitados na aplicação dos critérios selecionados.

	Critérios			
Casos de Teste	**Todos Nós**	**Todos Arcos**	**Todos Usos**	**Todos P-Usos**
C1	1 2 8 9 13	(2, 8), (8, 9)	<1,2, Employee>, <2,(2,8), exception> <2,(8,9), exception>	<2,(8,9),exception> <2,(2,8), exception>
C2	1 2 8 10 13	(2, 8), (8,10)	<1,2, Employee>, <2,(2,8), exception> <2,(8,10), exception>	<2,(8,10),exception> <2,(2,8), exception>
C3	1 2 3 4 5 7 13	(5, 7)	<1,2, Employee>, <1,4, Employee> <2,(2,3), exception>, <2,(3,4), exception> <3,(3,4), exception>, <3,(4,5), exception> <4,(4,5), exception>, <2,3, v_emplno> <2,3, v_salary>, <2,4, v_salary> <4,(5,7), v_qtde>	<4,(5,7), v_qtde> <2,(2,3), exception> <2,(3,4), exception> <3,(3,4), exception> <3,(4,5), exception> <4,(4,5), exception>
C4	1 2 3 4 5 6 11 13	(5, 6)	<1,2, Employee>, <1,4, Employee> <2,(2,3), exception>, <2,(3,4), exception> <3,(3,4), exception>,<3,(4,5), exception> <4,(4,5), exception>, <2,3, v_emplno> <2,3, v_salary>, <2,4, v_salary> <4,(5,6),v_qtde>,<6,11,e_salaryerror>	<4,(5,6), v_qtde> <2,(2,3), exception> <2,(3,4), exception> <3,(3,4), exception> <3,(4,5), exception> <4,(4,5), exception>

Tabela 5.6: Elementos exercitados.

	Critérios			
Casos de Teste	**Todos Pot-Usos**	**Todos Pot_Du Caminhos**	**Todos Pot-Uso/Du**	**Todos Du-Caminhos**
C1	<1,(2,8),{Employee, Log}> <1,(8,9),{ Employee }> <1,(8,9),{ Log }> <2,(2,8),{v_emplno,v_salary, exception}> <2,(8,9),{exception}> <2,(8,9),{v_emplno, v_salary}> <9,(,),{ Log }>	1 2 8 9 13 2 8 9 13 9 13	<1,(2,8),{Employee, Log}> <1,(8,9),{ Employee }> <1,(8,9),{ Log }> <2,(2,8),{v_emplno,v_salary, exception}> <2,(8,9),{ exception }> <2,(8,9),{v_emplno,v_salary}> <9,(,),{ Log }>	1 2 2 8 2 8 9
C2	<1,(2,8),{Employee, Log }> <1,(8,10),{ Employee }> <1,(8,10),{ Log }> <2,(2,8),{v_emplno, v_salary, exception}> <2,(8,10),{exception}> <2,(8,10),{v_emplno,v_salary> <10,(,),{ Log }>	1 2 8 10 13 2 8 10 13 10 13	<1,(2,8),{ Employee, Log }> <1,(8,10),{ Employee }> <1,(8,10),{ Log }> <2,(2,8),{v_emplno,v_salary, exception}> <2,(8,10),{ exception }> <2,(8,10),{v_emplno,v_salary> <10,(,),{ Log }>	1 2 2 8 2 8 10
C3	<1,(5,7),{ Employee }> <1,(2,3),{ Employee, Log }> <2,(5,7),{v_emplno,v_salary}> <2,(2,3),{ v_emplno, v_salary, exception }> <3,(5,7),{ Log }> <3,(3,4),{ Log, exception }> <4,(5,7),{ v_qtde, exception }>	1 2 3 4 5 7 13 2 3 4 5 7 13 3 4 5 7 13 4 5 7 13	<1,(5,7),{ Employee }> <1,(2,3),{ Employee, Log }> <2,(5,7),{v_emplno,v_salary}> <2,(2,3),{ v_emplno, v_salary, exception }> <3,(5,7),{ Log }> <3,(3,4),{ Log, exception }> <4,(5,7),{ v_qtde, exception }>	1 2 1 2 3 4 2 3 2 3 4 3 4 3 4 5 4 5 4 5 7
C4	<1,(5,6),{ Employee }> <1,(2,3),{ Employee, Log }> <2,(5,6),{v_emplno,v_salary}> <2,(2,3),{ v_emplno, v_salary, exception }> <3,(6,11),{ Log }> <3,(5,6),{ Log }> <3,(3,4),{ Log, exception }> <4,(5,6),{ v_qtde, exception }> <6,(,),{ e_salaryerror }> <11,(,),{ Log }>	1 2 3 4 5 6 11 13 2 3 4 5 6 11 13 3 4 5 6 11 4 5 6 11 13 6 11 13 11 13	<1,(5,6),{ Employee }> <1,(2,3),{ Employee, Log }> <2,(5,6),{v_emplno,v_salary}> <2,(2,3),{ v_emplno, v_salary, exception }> <3,(6,11),{ Log }> <3,(5,6),{ Log }> <3,(3,4),{ Log, exception }> <4,(5,6),{v_qtde, exception}> <6,(,),{ e_salaryerror }> <11,(,),{ Log }>	1 2 1 2 3 4 2 3 2 3 4 3 4 3 4 5 4 5 4 5 6 6 11

5.5 Análise de Cobertura

Na avaliação são obtidos arquivos com os elementos exercitados, os elementos não exercitados e a cobertura para critério.

Com as informações obtidas do módulo avaliador foi montada a Tabela 5.7 que exibe a cobertura atingida pelos casos de teste, que estão dispostos na tabela em uma ordem crescente de nós exercitados.

Os critérios todos p-usos e todos pot-du caminhos, apresentaram uma porcentagem de cobertura bem menor em relação aos outros critérios aplicados. Não foi possível exercitar as muitas associações requeridas que envolvem a variável *exception*, mostrado na Tabela 5.8.

Tabela 5.7: Cobertura dos critérios avaliados.

	Critérios							
Casos de Teste-R1	Todos Nós	Todos Arcos	Todos Usos	Todos P-Usos	Todos Pot-Usos	Pot-Du Caminhos	Pot – Usos /Du	Du Cami-nhos
1	38 %	25 %	11 %	10 %	14 %	8 %	14 %	12 %
2	38 %	25 %	11 %	10 %	14 %	8 %	14 %	12 %
1 e 2	46 %	38 %	15 %	14 %	24 %	15 %	24 %	16 %
3	54 %	13 %	41 %	29%	14 %	10 %	14 %	36 %
1,2 e 3	77 %	50 %	52 %	43 %	38 %	25 %	38 %	48 %
4	62 %	13 %	44 %	29 %	20 %	15 %	20 %	40 %
1,2,3 e 4	92 %	63 %	59 %	48 %	52 %	40 %	52 %	56 %

Não foi possível elaborar casos de teste que exercitassem todas as associações que envolvem a variável *exception*. O modelo de fluxo de controle pressupõe que cada comando de manipulação pode gerar desvios para todos os tipos de exceções que estão descritas no código. Esta abordagem tem um enfoque "conservador", no sentido de que requer o exercício de todos os tipos de exceções a partir de cada comando de manipulação de dados, embora algumas delas sejam não factíveis.

A partir dos resultados obtidos observa-se que para uma análise mais precisa a respeito dos desvios para a área de exceção é necessário um estudo adicional de cada comando de manipulação. Também é preciso um refinamento do fluxo de controle para determinar caminhos e associações não executáveis. Esta análise está relacionada ao problema de padrões de não executabilidade. Heurísticas que identificam requisitos não executáveis podem ser determinadas a partir da análise semântica da linguagem SQL, tais como as aplicadas em programas convencionais (Vergílio, 1992).

Tabela 5.8: Elementos exercitados e não exercitados para os critérios selecionados.

Casos de Teste C1 a C4			
Todos P-Usos		Todos Pot-Du Caminhos	
Assoc. Requeridas	Exercitadas	Caminhos Requeridos	Exercitados
<2,(2,3), exception> <2,(3,4), exception > <2,(2,8), exception > <2,(3,8), exception > <2,(8,9), exception > <2,(8,10), exception > <2,(8,12), exception > <3,(3,4), exception > <3,(4,5), exception > <3,(3,8), exception > <3,(4,8), exception > <3,(8,9), exception > <3,(8,10), exception > <3,(8,12), exception > <4,(4,5), exception > <4,(4,8), exception > <4,(5,6), v_qtde> <4,(5,7), v_qtde> <4,(8,9), exception > <4,(8,10), exception > <4,(8,12), exception >	**C1** - 2,(8,9), exception 2,(2,8), exception **C2**–2,(8,10),exception 2,(2,8), exception **C3** – 4,(5,6),v_qtde 2,(2,3), exception 2,(3,4), exception 3,(3,4), exception 3,(4,5), exception 4,(4,5), exception **C4** - 4,(5,7),v_qtde 2,(2,3), exception 2,(3,4), exception 3,(3,4), exception 3,(4,5), exception 4,(4,5), exception	1 2 8 12 13 1 2 8 10 13 , 1 2 8 9 13 1 2 3 8 12 13 , 1 2 3 8 10 13 1 2 3 8 9 13 , 1 2 3 4 8 12 13 1 2 3 4 8 10 13 , 1 2 3 4 8 9 13 1 2 3 4 5 7 13 , 1 2 3 4 5 6 11 13 2 8 12 13 2 8 10 13 , 2 8 9 13 2 3 8 12 13 , 2 3 8 10 13 2 3 8 9 13 , 2 3 4 8 12 13 2 3 4 8 10 13 , 2 3 4 8 9 13 2 3 4 5 7 13 , 2 3 4 5 6 11 13 3 8 12 13 , 3 8 10 13 3 8 9 13 , 3 4 8 12 3 4 8 10 , 3 4 8 9 3 4 5 7 13 , 3 4 5 6 11 4 8 12 13 , 4 8 10 13 4 8 9 13 4 5 7 13 , 4 5 6 11 13 6 11 13 , 9 13 10 13, 11 13 12 13	**C1**- 1 2 8 9 13 2 8 9 13 9 13 **C2** - 1 2 8 10 13 2 8 10 13 10 13 **C3** - 1 2 3 4 5 7 13 2 3 4 5 7 13 3 4 5 7 13 4 5 7 13 **C4** -1 2 3 4 5 6 11 13 2 3 4 5 6 11 13 3 4 5 6 11 4 5 6 11 13 6 11 13 11 13

5.6 Análise e Discussão Final

Com o objetivo de mostrar a utilização da ferramenta ART-TOOL no processo de teste de regras ativas, foram testadas quinze regras ativas, divididas em dois conjuntos. O primeiro é composto de regras elaboradas com o intuito de explorar os diversos comandos da linguagem SQL; o segundo constitui um exemplo real composto de cinco regras.

As operações INSERT, DELETE e UPDATE precisam estar incluídas nos casos de teste para testar o evento de disparo da regra.

Um aspecto pertinente é a necessidade de inclusão de um bloco de tratamento de exceção quando comandos de manipulação ou comandos de cursores fazem parte do corpo da regra, para evitar que na ocorrência de problemas a regra seja abortada.

Nos primeiros estudos para a proposta deste trabalho, o modelo de fluxo de dados considerava que a ocorrência da variável cursor no comando CLOSE é uma indefinição de dados. Foi testada uma regra com um defeito inserido – a presença de dois comandos CLOSE para a mesma variável cursor – e o defeito não foi detectado. Assim ficou evidente a necessidade de um enfoque dedicado ao exercício de associações que envolvessem as variáveis dos comandos de cursores, incluindo o comando CLOSE. Então, em um novo modelo de fluxo de dados, estabeleceu-se que a ocorrência da variável cursor neste comando seria uma definição de dados, pois são atribuídos a ela valores nulos; tal abordagem permitiu a identificação de associações de fluxo de dados entre duas ocorrências deste comando para um mesmo cursor, como ilustrado na Figura 5.3, Caso 1.

No comando OPEN também ocorre uma definição da variável cursor; sendo assim, foi proposta uma regra com um fluxo de controle representado na Figura 5.3, Caso 2, em que estão presentes dois comandos OPEN para o mesmo cursor. Foram executados somente quatro casos de teste, conseguindo uma cobertura para os critérios mais fracos de 100%. Para o critério Potenciais Usos foram requeridas e exercitadas as associações que envolviam a variável cursor nos comandos OPEN-FETCH, FETCH-CLOSE e

OPEN-CLOSE, neste conjunto, também estava incluso o OPEN-OPEN, que representa um defeito na regra; com um caso de teste específico foi possível detectar o defeito.

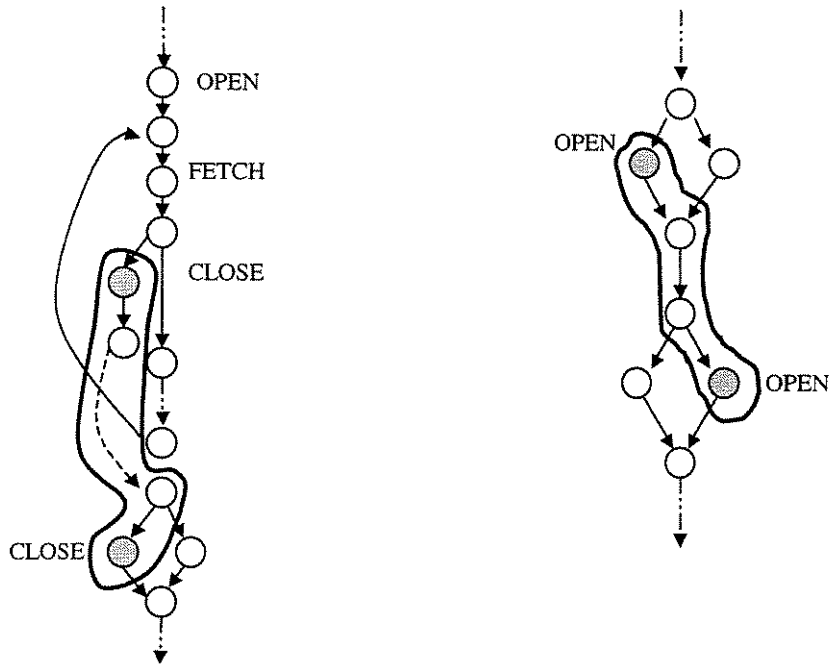


Figura 5.3: Ocorrências com os comandos de cursores.

No conjunto de regras do exemplo real, um fato importante foi notado: nenhuma das cinco regras possui área de tratamento de exceção e, em todas, usam-se cursores e comandos de manipulação. Com a execução dos casos de teste em uma regra foi notada a necessidade da inclusão do bloco de exceção. Trata-se de uma consulta em uma tabela que, em uma situação específica na qual a tabela está vazia, pode causar a não execução da regra.

Defeitos presentes em programas convencionais foram detectados nos exemplos elaborados, dentre eles, os que envolvem variáveis locais, comandos de atribuição com variáveis locais, troca de variáveis locais, ausência de incremento, predicados incorretos, etc. Nas Figuras 5.4 e 5.5 estão ilustrados os códigos com um comando de cursor,

comando de manipulação e comando condicional e nestes estão inseridos defeitos de atribuição com variáveis locais, predicados incorretos e troca de variáveis locais.

```
::::::::::::::::::::::::::::::::::::::::::::::::
LOOP
        FETCH crEmpl INTO v_emplno, v_salary;
        IF (crEmpl%ROWCOUNT > 10) OR (crEmpl%NOTFOUND) THEN
           CLOSE crEmpl;
           EXIT;
        END IF;
        INSERT INTO tempempl  ( emplno, salary )
        VALUES   ( v_emplno, v_salary );
 END LOOP;
::::::::::::::::::::::::::::::::::::::::::::::::
```

Figura 5.4: Exemplo de código de uma regra correta.

```
::::::::::::::::::::::::::::::::::::::::::::::::
LOOP
      FETCH crEmpl INTO v_emplno, v_num;
      IF(crEmpl%ROWCOUNT <v_num)OR (crEmpl%NOTFOUND) THEN
          CLOSE crEmpl;
          EXIT;
      END IF;
      INSERT INTO tempempl  ( emplno, salary )
      VALUES   ( v_depto, v_salary );
 END LOOP;
::::::::::::::::::::::::::::::::::::::::::::::::
```

Figura 5.5: Exemplo de código de uma regra com defeitos.

Em síntese, a aplicação dos modelos para identificação de elementos requeridos pelos critérios utilizados foi capaz de revelar defeitos típicos em comandos de manipulação de dados, inclusive relacionados ao mecanismo de cursor. Adicionalmente, foi evidente a importância da automação dos modelos de fluxo de controle, de fluxo de dados e de instrumentação, para o teste de regras ativas escritas em SQL. Os resultados obtidos pelo uso da ferramenta ART-TOOL indicam que os critérios baseados em fluxo de dados, especificamente os critérios Potenciais Usos, representam um valioso recurso no teste sistemático de regras ativas.

CAPÍTULO 6

CONCLUSÕES

6.1 Síntese da Tese

O uso de regras ativas em bancos de dados relacionais ressalta a necessidade por abordagens de teste sistemático de regras ativas escritas em linguagem SQL. Neste sentido, a proposta deste trabalho é um esforço inicial para a automação de teste de regras ativas escritas em SQL, no contexto de teste estrutural de unidades. Foi construída uma ferramenta, ART-TOOL (*Active Rule Testing Tool*), escrita na linguagem Java, que apóia a aplicação de critérios estruturais baseados em fluxo de dados, especificamente os critérios Potenciais Usos (Maldonado, 1991).

A ferramenta realiza a análise estática do código fonte da regra por meio da qual são obtidos o fluxo de controle, o fluxo de dados e a versão instrumentada da regra. Adotou-se o padrão SQL-3 implementado por muitos sistemas gerenciadores de banco de dados; também foram incorporadas estruturas específicas da linguagem PL/SQL do sistema ORACLE.

Para abstrair os elementos requeridos pelos critérios de teste, o foco principal foram as associações de fluxo de dados de dados persistentes, isto é, as relações (tabelas) de uma base de dados. O mecanismo de cursor, um recurso para a manipulação *tupla-a-tupla* de uma relação, foi especialmente considerado na construção de modelos.

Foram propostos modelos de fluxo de controle, de fluxo de dados e de instrumentação. Esses modelos constituem enfoque promissor, pois desconhece-se a existência de literatura sobre a análise de fluxo de controle e de dados para regras ativas

escritas em SQL (Leitão *et. al*, 2002). Os modelos exploram as partes constituintes de uma regra ativa: evento, condição e ação.

A presença de rotinas de tratamento de exceção em regras ativas elimina a natureza seqüencial de controle em comandos de manipulação de dados. Duas classes de tratamento de exceção foram caracterizadas: definida pelo sistema e definida pelo usuário; abordagens de fluxo de controle foram propostas para cada classe. Para as exceções definidas pelo sistema o modelo de fluxo de controle adota uma abordagem conservadora pois pressupõe que todo comando de manipulação pode resultar em exceção, ocasionando possíveis transferências de controle para toda exceção tratada. A abordagem adotada para o tratamento de exceção em regras ativas é genérica, podendo ser aplicada em outras linguagens que disponham do recurso de tratamento de exceção.

O modelo de fluxo de dados leva em conta a granulosidade, que estabelece a precisão das relações de fluxo de dados (Leitão *et. al*, 2002). A granulosidade de tabela foi adotada para a implementação inicial da ferramenta, desconsiderando-se inicialmente as linhas e colunas afetadas pelas operações de manipulação de dados. Contudo, o trabalho apresenta uma proposta para a identificação de *tuplas*, que poderá ser incorporada à implementação.

O modelo de instrumentação da regra considera os nós conforme o modelo de fluxo de controle. Os caminhos exercitados pela aplicação de casos de teste são registrados em uma tabela do próprio banco de dados, específica para este fim. O enfoque considera a inclusão de tratamento de exceção para aquelas regras que não o possuem; esta medida foi adotada para evitar que a regra seja abortada quando são detectados resultados não previsíveis e, assim, permitir o acompanhamento da execução dos casos de teste até o ponto da ocorrência da exceção.

Para a utilização da ferramenta ART-TOOL no processo de teste são utilizados módulos da ferramenta POKE-TOOL (Chaim, 1991), mais especificamente para gerar os elementos requeridos e para a avaliação do teste. Arquivos específicos de entrada para esses módulos são definidos na ART-TOOL, tornando-a dependente em relação à POKE-TOOL.

Para mostrar a aplicação da ART-TOOL testou-se um exemplo de regra ativa que foi elaborado visando à exploração de comandos de manipulação e tratamento de exceção. Conclui-se que os defeitos que aparecem em comandos de manipulação são perceptíveis pelo exercício de associações de variáveis. Ficou também evidente a necessidade de um bloco de tratamento de exceção quando há comandos de manipulação no código da regra.

Adicionalmente foram testadas dez regras ativas, divididas em dois conjuntos. O primeiro é composto de regras elaboradas com o intuito de explorar os diversos comandos da linguagem SQL; o segundo é um exemplo real composto de cinco regras. Ficou evidente a importância da automação dos modelos de fluxo de controle, de fluxo de dados e de instrumentação para o teste de regras ativas escritas em SQL. Os resultados obtidos pelo uso da ferramenta ART-TOOL indicam que os critérios baseados em fluxo de dados, especificamente os critérios Potenciais Usos, constituem um valioso recurso para o teste sistemático de regras ativas. Os critérios Potenciais Usos requerem associações que envolvam as variáveis dos comandos de cursores (OPEN, FETCH e CLOSE); como discutido na Seção 5.6, o exercício das associações entre ocorrências dos comandos do mesmo cursor torna possível a detecção de defeitos em cursores. Os outros critérios baseados em fluxo de dados não requerem associações que envolvam comandos do mesmo cursor, embora exijam o exercício da variável cursor.

6.2 Contribuições

Abaixo estão relacionadas as principais contribuições deste trabalho:

1. **Esforço inicial para o teste de regras escritas em SQL**: representa a primeira iniciativa para o teste estrutural de regras ativas escritas em SQL.
2. **Modelos de implementação para a automação do teste estrutural de regras ativas**: os aspectos de fluxo de controle, de fluxo de dados e de instrumentação foram estudados, resultando na construção de modelos úteis à automação do processo de teste.

3. **Aplicação de critérios estruturais baseados em fluxo de dados a regras ativas**: a aplicação dos critérios Potenciais Usos, que baseiam-se em fluxo de dados, mostraram-se úteis na revelação de defeitos em regras ativas escritas em SQL.

4. **A ferramenta ART-TOOL**: a ferramenta ART-TOOL viabiliza a aplicação de critérios baseados em fluxo de dados a regras escritas em SQL; constitui um recurso preliminar para apoiar outras atividades de teste.

5. **Modelo de instrumentação para a identificação de *tuplas***: o modelo refina a análise de fluxo de dados pela identificação das linhas afetadas em cada operação de manipulação de dados.

6. **Análise dedicada ao mecanismo de cursor**: em adição ao estudo dos comandos de manipulação de relações de dados, os modelos de implementação abstraem aspectos do mecanismo de cursor.

6.3 Trabalhos Futuros

Os trabalhos futuros pertinentes à pesquisa são discutidos resumidamente a seguir.

1. Aplicação da ferramenta a outros conjuntos de regras, especialmente a regras empregadas em situações reais; avaliação da sua aplicabilidade em grande escala do ponto de vista funcional das regras, sem considerar características de desempenho, utilizando versões reduzidas de teste das bases de dados.

2. Extensão dos modelos de implementação ao tratamento de exceção em blocos aninhados. O modelo proposto insere a exceção no bloco mais externo na ação da regra ativa; contudo, pode existir tratamento de exceção em cada bloco aninhado, o que deve ser considerado nos novos modelos.

3. Implementação do modelo de instrumentação para identificação de *tuplas* e elaboração de critérios específicos para esse refinamento da análise de fluxo de dados.

4. Automação do teste de regras ativas para outros sistemas gerenciadores de banco de dados. É importante observar que os modelos foram propostos considerando o padrão SQL-3, com alguns comandos da linguagem PL/SQL do ORACLE. Pequenas alterações também serão necessárias para outros SGBDs que tenham como base a linguagem SQL.

REFERÊNCIAS BIBLIOGRÁFICAS

Aho, A. V., Sethi, R. e Ullman, J. D., 1995, *Compiladores: Princípios, Técnicas e Ferramentas*, Addison-Wesley Publishing Company, Reading, Massachusetts, USA.

Appel, A. W. e Ginsburg, M., 1997, *Modern Compiler Implementation in Java*, Cambridge University Press, United Kingdom, 1ª edição.

Aranha, M.C.L.F.M., Mendes, N.C., Jino, M. e Toledo, C.M.T., 2001, "*RDBTool: Uma ferramenta de Apoio ao Teste de Bases de Dados Relacionais*", Anais do XI CITS: Conferência Internacional de Tecnologia de Software: Qualidade de Software, Cutitiba, PR, Brasil, pp. 31-43.

Aranha, M.C.L.F.M., 2003, *Sistematização e Uso de Critérios de Teste em Bancos de Dados Relacionais*, Tese de Doutorado em andamento, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas SP, Brasil.

Baralis, E. e Widom, J.,1994, *"An Algebraic Approach to Rule Analysis in Expert Database Systems"*, Proceedings of the Twentieth International Conference on Very Large Databases, pp 475-486, Santiago, Chile. http://citeseer.nj.nec.com/baralis94algebraic.html , visitado em 14/08/2001.

Behrends, H., 1994, "*Simulation-Based Debugging of Active Databases*", Proc. of the 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE 94), Houston, Texas, USA.

Benazet, E., Guehl, H. e Bouzeghoub, M., 1995, "*VITAL: a Visual Tool for Analysis of Rules Behavior in Active Databases*", Proc. of the 2nd Workshop on Rules in Database systems (RIDS 95-LNCS 985), Atenas, Grécia.

Berndtson, M. e Calestam, B., 2000, "*A Uniform Approach for Supporting Active Database Features in UML and OMT* ", http://citeseer.nj.nec.com/447083.html, visitado em 23/08/2001.

BNF SQL3,1998, The BNF Web Club Language SQL, ADA, JAVA, MODULA2, PL/SQL, http://cui.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html visitado em 10/04/2002.

BNF SQL3 syntax,1992, http://www.cs.rit.edu/~mjh1697/SQL/reference.html, visitado em 10/04/2002.

Budd, T. A., 1981, *Mutation Analysis: Ideas, Example, Problems and Prospects*, North-Holland Publishing Company, Chapter Computer Program Testing.

Bueno, P. M. S., 1999, *Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software*, Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, SP, Brasil.

Campin, J., Paton, N. W. e Williams, M. H., 1994, "*A Structured Specification in Active Database System* ", http://citeseer.nj.nec.com/campin95structured.html, visitado em 02/07/2001.

Carniello, A., 2003, *Instrumentação Configurável e Incremental de Programas*, Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, SP, Brasil.

Chaim, M.L., 1991, *POKE-TOOL – Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados*, Dissertação de Mestrado, Faculdade de Engenharia Elétrica, UNICAMP, Campinas, SP, Brasil.

Chaim, M. L., 2001, *Depuração de Programas Baseada em Informação de Teste Estrutural*, Tese de Doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, SP, Brasil.

Chakravarthy, S., Tamizuddin, Z. e Zhou, J., 1995, "*A Visualization and Explanation Tool for Debugging ECA Rules in Active Databases*", Proc. of the 2nd Workshop on Rules in Database Systems (RIDS 95-LNCS 985), Atenas, Grécia.

Chan, H.W.R., Dietrich, S.W. e Urban, S.U., 1997, "*On Control Flow Testing of Active Rules in a Declarative Object-Oriented Framework*", Proceedings of the Third International Workshop on Rules in Database Systems (RIDS '97), Skoevde, Suécia.

Chays, D., Dan, S., Frankl, P.G., Vokolos, F.I. e Weyuker, E.J., 2000, "*A Framework for Testing Database Applications*", Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, EUA.

Chusho, T., 1987, "*Test Data Selection and Quality Estimation Based on the Concept of Essencial Branches for Path Testing*", IEEE Transactions on Software Engineering, vol.13, nº 5, pp. 509-517.

Coupaye,T. e Collet, C., 1998, "*Semantics Based Implementation of Flexible Execution Models for Active Database Systems*". http://citeseer.nj.nec.com/coupaye98semantics.html, visitado em 16/08/2001.

Coupaye, T., Roncancio, C. L. e Bruley, C., 1999, "*Vizar, A Visualization Service for Event-Based Systems*". http://citeseer.nj.nec.com/coupaye99vizar.html, visitado em 16/08/2001.

Davies, R.A., Beynon, R.J.A. e Jones, B.F., 2000, "*Automating the Testing of Databases*", Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, University of Limerick, Irlanda.

DeMillo, R. A., Lipton, R. J. e Sayward, F. G., 1978, *"Hints on Test Data Selection: Help for the Practicing Programmer"*, IEEE Computer, vol.11, n° 4, p. 34-43.

Deutsch, M., 1979, "Verification and Validation", in Software Engineering, (R.Jensen e C.Tonies, eds), Prentice-Hall, pp. 329-408.

Díaz, O., Jaime, A. e Paton, N.,1993, "DEAR: a Debugger for Active Rules ina an object-oriented context", Proc. of the 1st Workshop on Rules in Database Systems (RIDS'93), Edimburgh, Scotland.

Eisenberg, A. e Melton, J., 2000, *"SQL: 1999, Formerly Known as SQL3"* http://dbs.uni-leipzig.de/en/lokal/standards.pdf , visitado em 10/04/2002.

Elmasri, R, Navathe, S.B., 2000, *"Fundamentals of Database System"*, Addison-Wesley.

Fortier, P.J., 1999, *"SQL-3 Implementing the Object-Relational Database"*, McGraw-Hill Enterprise Computing Series.

Frankl, F. G., 1987, *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data*, Tese de Doutorado, Universidade de Nova Iorque, EUA.

Frankl, F.G. e Weyukey, E.J., 1988, *"Data Flow Testing Tool"*, Proc. Softfair II, San Francisco, CA, pp. 46-53.

Frankl, F.G. e Weyukey, E.J., 1988, *"An Applicable Family of Data Flow Testing Criteria"*, IEEE Transactions on Software Engineering, vol. 14, n° 10, pp. 1483-1498.

Fraternali, P., Teniente, E., e Urpí, T., 1997, *"Validating Active Rules by Planning"*. Proceedings of the Third International Workshop on Rules in Database Systems (RIDS '97), Skoevde, Suécia.

Graham, D. R., 1994, *Encyclopaedia of Software Engineering*, John Wiley e Sons, Inc., John J. Marciniack, Nova Iorque, EUA, pp. 1330-3352.

Haraty, R.A., Mansour, N., Daou, B., 2002, *"Regression Testing of Database Applications"*, Proceedings of the 2002 ACM Symposium on Applied Computing, Las Vegas, Nevada, USA.

Harrold, M. J. e Rothermel, G., 1994, *"Performing Data Flow Testing on Classes"*, Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, EUA, pp. 154-163.

Hecht, M. S., 1977, *Flow Analysis of Computer Programs*, Programming Languages Series, Elsevier NorthHolland, Inc., Nova Iorque, EUA.

Herman, P.M., 1976, *"A Data Flow Analysis Approach to Program Testing"*, The Australian Computer Journal, vol. 8, n° 3, pp. 92-96.

Howden, W. E., 1975, "*Methodology for the Generation of Program Test Data*", IEEE Transactions on Computers, vol. 24, n° 5, pp. 554-559.

Jähne, A., Urban, S. D. e Dietrich, S. W., 1996, "*Peard: A Prototype Environment for Active Rule Debugging*", Journal of Intelligent Information Systems, Special Issue on Active Database Systems, Volume 7, Issue 2, pp111-128.

JavaCC e JJTtree (a), Compiladores- IF 120. http://www.cin.ufpe.br/~if120, visitado em 05/10/2002.

JavaCC e JJTtree (b), WebGain. http://www.webgain.com/products/java_cc, visitado em 05/10/2002.

JavaCC e JJTtree (c), WebGain. http://www.webgain.com/products/java_cc/jjtree.html, visitado em 05/10/2002.

JavaCC e JJTtree (d), Others Tools. http://www.cs.purdue.edu/jtb/tools.html, visitado em 05/10/2002.

JavaCC e JJTtree (e). Session plan, MSc PPL Session 2, 2001. http://www.soi.city.ac.uk/~daveb/teaching/P213/ppl2.pdf, visitado em 05/10/2002.

Kappel, G., Kramler, G. e Retschitzegger, W., 2001, "*TriGs Debugger – A Tool for Debugging Active Database Behaivor*", Journal Lecture Notes in Computer Science, vol.2113. http://citeseer.nj.nec.com/409368html, visitado em 16/02/2002.

Laski, J. W. e Korel, B., 1983, "*A Data Flow Oriented Program Testing Strategy*", IEEE Transactions on Software Engineering, vol. 9, n° 3, pp. 347-354.

Leitão , P. S., Cardoso, V. e Jino, M., 2002, *Aspectos de Fluxos de Controle, de Dados e Instrumentação no Contexto de Banco de Dados Ativos*, Relatório Técnico, Departamento de Engenharia de Computação e Automação Industrial, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, S.P., Brasil.

Lyon, N. R., 1977, "*An Automatic Data Generating System for Data Base Simulation and Testing*", Database, vol. 8, n° 4, pp.10-13.

Maldonado, J.C., Chaim, M.L. e Jino, M., 1988 (a), "*Seleção de Casos de Teste Baseada nos Critérios Potenciais Usos*", Anais do II Simpósio Brasileiro de Engenharia de Software, Canela, RS, Brasil, pp. 24-35.

Maldonado, J.C., Chaim, M.L. e Jino, M., 1988 (b), "*Resultados dos Estudos de uma Família de Critérios de Teste de Programa Baseado em Fluxo de Dados*", Relatório Técnico, DCA/RT/001/88 – Departamento de Computação e Automação da Faculdade de Engenharia Elétrica e de Computação – UNICAMP, Campinas, SP, Brasil.

Maldonado, J.C., 1991, *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*, Tese de Doutorado, Faculdade de Engenharia Elétrica, UNICAMP, Campinas, SP, Brasil.

Maldonado, J. C., Chaim, M. L. e Jino, M., 1992, "*Potential Uses Testing Criteria: a Step Towards Bridging the Gap in the Presence of Infeasible Paths*", Proceedings of the XII International Conference of the Chilean Computer Society, Santiago, Chile.

Maldonado, J. C. e Fabbri, S. C. P. F., 2001, "*Verificação e Validação de Software*", In: Rocha, A. R. C., Maldonado, J. C. e Weber, K. C. (Eds.), Qualidade de Software: Teoria e Prática, Prentice-Hall, São Paulo, SP, Brasil, pp. 66-73.

Mannila, H. e Raiha, K.J., 1989, "*Automatic Generation of Test Data for Relational Queries*", Journal of Computer and System Sciences, vol.38, n° 2, pp. 240-258.

Marx, D.I.S. e Frankl, P.G., 1996, "*The Path-wise Approach to Data Flow Testing with Pointer Variables*", International Symposium on Software Testing and Analysis, pp. 135-146.

Montesi, D. e Torlone, R., 1995, "*A Transaction Transformation Approach to Active Rule Processing*". Proceedings of the 11th International Conference on Data Engineering, IEEE Computer Society Press, 109-116.

Myers, G., 1979, *The Art of Software Testing*, Wiley, Nova Iorque, EUA.

Noble, H., 1983, "*The Automatic Generation of Test Data for a Relacional Database*", Information Systems, vol. 8, n° 2, pp. 79-83.

Ntafos, S. C., 1984, "*On Required Element Testing*", IEEE Transactions on Software Engineering, vol. 10, n° 11, pp. 795-803.

Ostrand, T. J. e Balcer, M. J., 1988, "*The Category-partition Method for Specifying and Generating Functional Tests*", Communications of the ACM, vol.31, n° 6, pp. 676-686.

Paduan, E. S., 2000, *Um Modelo Confluente de Execução de Regras em Banco de Dados Relacionais Ativos*, Dissertação de Mestrado, Instituto de Matemática e Estatística, USP, São Paulo, Brasil.

Parrish, A.S., Borie, R. B. e Cordes, D. W., 1993, "*Automated Flow Graph – Based Testing of Object-Oriented Software Modules*", Journal of Systems and Software, vol. 23, pp. 95-109.

Paton, N. W., 1999, *Active Rules in Database Systems*. Springer-Verlag.

Paton, N. W., e Díaz, O., 1999, "*Active Database Systems*". ACM Computing Surveys, Vol 31, No 1, pp. 63-103. Março de 1999.

Pressman, R. S., 1997, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 4ª Ed., Nova Iorque, EUA.

Rapps, S. e Weyuker, E. J., 1982, "*Data Flow Analysis Techniques for Test Data Selection*", International Conference on Software Engineering, Tóquio, Japão, pp. 272-278.

Rapps, S. e Weyuker, E.J., 1985, "*Selecting Software Test Data Using Data Flow Information*", IEEE Transactions on Software Engineering, vol. SE-11, pp. 367-275.

Rocha, A. R. C., Maldonado, J. C. E Weber, K. C., 2001, *Qualidade de Software- Teoria e Prática*, Prentice Hall.

Roper, M. e Rahim, A. R. B. A., 1993, "*Software Testing Using Analysis and Design Based Techniques*". Software Testing, Verification and Reliability, vol. 3, pp 165-179.

Spoto, E.S., 2000, *Critérios de Teste Estrutural para Programas de Aplicação de Banco de Dados Relacional*, Tese de Doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, SP, Brasil.

Standards SQL-3 1994. Index of /sequoia/schema/STANDARDS/SQL3. http://epoch.cs.berkeley.edu:8000/sequoia/schema/STANDARDS/SQL3/, visitado em 10/04/2002.

Taylor, R. N., Levine, D. L. e Kelly, C. D., 1992, "*Structural Testing of Concurrent Programs*", IEEE Transactions on Software. Engineering, vol. 18, nº 3, pp. 206-215.

Ural, H. e Yang, B., 1988, "*A Structural Test Selection Criterion*", Information Processing Letters, vol. 28, nº 3, p.157 à163.

Urman, S., 1997, "*ORACLE 8 – PL/SQL Programming – The Essencial Guide for Every Oracle Programmer*", Osborne, Lisboa: Mc Graw-Hill.

Vaduva, A., 1999, *Rule Development for Active Database Systems*, Tese de Doutorado, Faculdade de Economia, Universidade de Zurique, Zurique.

Vergílio, S.R.,1992, *Caminhos não executáveis: caracterização, previsão e determinação para suporte ao teste de programas*. Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, Brasil.

Vilela, P. R. S., 1998, *Critérios Potenciais Usos de Integração: Definição e Análise*, Tese de Doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, SP, Brasil.

Zaniolo, C., Ceri, S., Faloutsos, C., Snoograss, R. T., Subrahmanian, V. S. e Zicari, R., 1997, *Advanced Database Systems*. Morgan Kaufmann Publishers, Inc.. San Francisco, California, USA.

Yan, S.Y., 1991, *"Declarative Testing of Logic Databases"*, Computers Mathematical Application, Vol.22, No 11. Pp. 39-45.

Weyuker, E. J., 1984, *"The Complexity of Data Flow Criteria for Test Data Selection"*, Information Processing Letters, vol. 19, n° 2.

Woodward, M. R., Heddley, D. e Hennel, M. A., 1980, *"Experience with Path Analysis and Testing of Programs"*, IEEE Transactions on Software Engineering, vol. 6, n° 3, pp. 278-286.

APÊNDICE A

Gramática da Linguagem SQL

Neste apêndice são apresentados os comandos da linguagem SQL que foi utilizada ao longo deste trabalho. Os vários Sistemas Gerenciadores de Banco de Dados Relacional utilizados hoje em dia fazem uso da linguagem SQL padrão existindo, assim, pouca distinção entre eles (Spoto, 2000).

Os comandos da SQL são classificados em duas classes: a DDL, comandos de definição de dados e a DML, comandos de manipulação de dados. A classe DDL inclui os comandos para definição de dados e para controle de acesso a dados. A DML inclui os comandos de manipulação de dados, de recuperação de dados, comandos para processo de transação de dados e para uso de SQL dinâmica. Os comandos considerados neste trabalho seguem o padrão SQL-3 e estão incluídos somente aqueles aceitos em uma regra ativa.

A sintaxe da linguagem SQL adotada neste trabalho está de acordo com a BNF do padrão SQL-3, incluindo também comandos aceitos no ORACLE 8i.

Para uma melhor compreensão inicialmente estão descritos os principais meta símbolos usados, com os seus respectivos significados:

::= significa "é definido como"

| significa "ou"

< > símbolos usados para dar nomes às categorias; distinguem os nomes das regras de sintaxe (chamados também de símbolos não-terminais) dos símbolos terminais que são escritos exatamente como devem ser representadas as regras.

Não-terminal é uma sequência de alternativas consistindo de "*strings*" de terminais ou não-terminais separados pelo meta-símbolo " | " .

[e] usados para incluir itens opcionais.

{ e } meta-símbolos são usados para incluir itens repetitivos (zero ou mais vezes).

Terminais constituídos por um caráter são cercados por aspas (") para distinguí-los dos meta-símbolos.

SINTAXE DE UMA REGRA ATIVA:

<Trigger> ::= <TriggerDeclaration>
<TriggerDeclaration> ::= < EventDeclaration >
[<ConditionSection>]
<TriggeredAction>

EVENTO

<EventDeclaration> ::= CREATE TRIGGER <identifier>
(AFTER | BEFORE)
(INSERT [OR] | DELETE [OR]
| UPDATE [OR] [OF { <atribute> [","] | <identifier>[","] }])
ON <identifier>
[ORDER <number>]
[REFERENCING { (NEW [AS] <identifier >
| OLD [AS] <identifier >
| NEW_ TABLE [AS] <identifier >
| OLD_TABLE) [AS] <identifier > }]
[FOR EACH (ROW | STATEMENT)]

<atribute> ::= <identifier> "." <identifier>

CONDIÇÃO

<ConditionSection> ::= [WHEN "(" (<SQLExpression> ")"]

AÇÃO

<TriggeredAction> ::= [<DeclarationSection> | <Declarations>]
<BeginEndBlock>

Comandos declarativos:

DECLARE

<DeclarationSection> ::= DECLARE <Declarations>

<Declarations> ::= { (LOOKAHEAD(2)< IdentifierDeclaration>
| LOOKAHEAD(2) <CursorDeclaration>) ";" }

<IdentifierDeclaration> ::= <identifier> (<ConstantDeclaration>
| <IntervalDeclaration>
| <VariableDeclaration>
| <ExceptionDeclaration>)

<ConstantDeclaration> ::= CONSTANT <TypeDeclaration> [NOT NULL]
[(([":"] " = ") | DEFAULT) <PlSqlExpression>]

<IntervalDeclaration> ::= INTERVAL (YEAR | MONTH | DAY | HOUR)
["(" <number> ")"] TO (YEAR | MONTH | DAY | HOUR)

<VariableDeclaration> ::= <TypeDeclaration> [NOT NULL]
[(([":"] " = ") | DEFAULT) <PlSqlExpression>]

<ExceptionDeclaration> ::= EXCEPTION

<TypeDeclaration> ::= ({ <BasicDataTypeDeclaration> }
| (<identifier> ("%TYPE"
| "%ROWTYPE"))
| (<atribute> "%TYPE")
| <identifier>

<BasicDataTypeDeclaration> ::= (CHAR | VARCHAR | VARCHAR2
| INTEGER | NUMBER | NATURAL
| REAL | FLOAT | SMALLINT
| INT | NUMERIC | DECIMAL
| DEC | DOUBLE PRECISION
| CHARACTER | CHARACTER VARYING
| CHAR VARYING | BIT
| BIT VARYING | NATIONAL CHARACTER
| NATIONAL CHAR | NCHAR
| NATIONAL CHARACTER VARYING
| NATIONAL CHAR VARYING
| NCHAR VARYING | CHARACTER SET <identifier>)
["(" <number> ["," <number>] ")"]
[ARRAY "(" <number> ["," <number>] ")"]
| DATE | TIME | TIMESTAMP
| TIME WITH TIME ZONE
| TIMESTAMP WITH TIME ZONE
| BOOLEAN

<CursorDeclaration> ::= <identifier> [(SENSITIVE | INSENSITIVE)]
[SCROLL] CURSOR [WITH HOLD]
FOR <SelectStatement>

Obs: esta declaração do cursor é específica para o ORACLE:

CURSOR <identifier> ["(" <ParameterList> ")"] IS <SelectStatement>

< ParameterList > ::= < Parameter> { "," <Parameter>}
<Parameter> ::= <identifier> [[IN] [OUT] <TypeDeclaration>
[(([":"] "=") | DEFAULT) <PlSqlExpression>]]

COMANDOS EXECUTÁVEIS

Manipulação de Dados

INSERT

<insert statement> ::= INSERT INTO < identifier>
["(" <identifier> ["." <identifier>]
{ "," <identifier> ["." <identifier>] }]
(VALUES "(" <PlSqlExpressionList> ")" | <SubQuery>) ";"

UPDATE

<updatestatement> ::= UPDATE <identifier> [<identifier>]
SET <ColumnValues>
[WHERE (<SQLExpression> | CURRENT OF <identifier>)] ";"

<ColumnValues>::= <identifier> | <atribute>] "=" <UpdatedValue>
{ "," <identifier> ["." <identifier>] "=" <UpdatedValue>

<UpdatedValue>::= { "(" <SelectStatement> ")" } | < PlSqlExpression>)

DELETE

<deletestatement> ::= DELETE [FROM]<identifier> [<identifier>]
[WHERE (<SQLExpression> | CURRENT" "OF" <identifier>)] ";"

Recuperação de Dados

SELECT

<selectstatement>::= < SelectWithoutOrder> [< OrderByClause>] [< ForUpdateClause>]

< SelectWithoutOrder>::= SELECT [ALL | DISTINCT] <SelectList>
[<IntoClause>]
<FromClause>
[<JoinedTable>]
[<WhereClause>]
[<ConnectClause>] [<GroupByClause>] [<SetClause>]

<SelectList>::= ("*" | <SelectItem> ["AS" <identifier>]
{"," <SelectItem> ["AS" <identifier> })

<SelectItem>::= (<identifier> ".*"
|<atribute> ".*"
|<SQLSimpleExpression> [<identifier>]

<IntoClause>::= INTO <IntoItem> { "," <IntoItem> }

<IntoItem>::= (<identifier> ["." <identifier>]) | <bind>

<bind> ::= ":" <identifier> { "." <identifier> }

<FromClause>::= FROM (<FromItem> { "," <FromItem> })

<FromItem> ::= (<identifier> [<identifier>] | "DUAL")

<joined table> ::= (<cross join> | <qualified join>)

<cross join> ::= CROSS JOIN <identifier>

<qualified join> ::= [NATURAL] ([INNER | (LEFT | RIGHT | FULL) [OUTER] | UNION])
JOIN <identifier> [". " <identifier>]
([ON <SQLExpression> | USING (<IntoItem>) | <KeyReference>])

<KeyReference> ::= (USING PRIMARY KEY | USING FOREIGN KEY
| USING CONSTRAINT <identifier> [". " <identifier>])
<WhereClause>::= WHERE (<SQLExpression> | <SubQuery>)

<ConnectClause> ::= CONNECT BY <SQLExpression> [STAR WITH <SQLexpression>]

<GroupByClause>::= GROUP BY <SQLExpressionList> [HAVING <SQLExpression>]

<SetClause>::= (((UNION [ALL])
| INTERSECT
| EXCEPT
| MINUS)
| <SelectWithoutOrder>)

<OrderByClause>::= ORDER BY <SQLSimpleExpression> [ASC | DESC]
{"," <SQLSimpleExpression> [ASC | DESC]}

<ForUpdateClause>::= FOR UPDATE OF <identifier> ["." <identifier>]
{"," <identifier> ["."<identifier>}

<SubQuery> ::= <SelectWithoutOrder>

<SQLExpression>::= <SQLAndExpression> {OR <SQLAndExpression>}

< SQLAndExpression>::= <SQLUnaryLogicalExpression>
{ (AND | "..") <SQLUnaryLogicalExpression>}

<SQLUnaryLogicalExpression>::= (<ExistsClause> | ([NOT] <SQLRelationalExpression>)

<ExistsClause>::= [NOT] EXISTS "(" <SubQuery> ")"

<SQLRelationalExpression>::= (("(" <SQLExpressionList> ")"
| ([PRIOR] <SQLSimpleExpression>))
[<SQLRelationalOperatorExpression> | <SQLInClause>
| <SQLBetweenClause> | <SQLLikeClause> | <IsNullClause>]

<SQLExpressionList>::= <SQLSimpleExpression> {"," <SQLSimpleExpression>}

<SQLRelationalOperatorExpression>::= (<Relop> | <Relop1>)
((ANY | ALL | "(" SELECT)
([ALL | ANY] "(" <SubQuery> ")")
| [PRIOR] <SQLSimpleExpression>)

<Relop>::= (">" | "!=" | "#" | "<>")

<Relop1>::= ("=" | ">=" | "<" | "<=")

<SQLInClause>::= [NOT] IN "(" (<SQLExpressionList> | <SubQuery>) ")"

<SQLBetweenClause>::= [NOT] BETWEEN <SQLSimpleExpression> AND <SQLSimpleExpression>

```
<SQLLikeClause>::= [ NOT ] LIKE <SQLSimpleExpression>

<SQLSimpleExpression>::= <SQLMultiplicativeExpression>
                         { ( "+" | "-" | "||" ) <SQLMultiplicativeExpression> }

<SQLMultiplicativeExpression>::= <SQLExpotentExpression>
                                 { ( "*" | "/") <SQLExpotentExpression>}

<SQLExpotentExpression>::= <SQLUnaryExpression> { "**" <SQLUnaryExpression>}

<SQLUnaryExpression>::= ["+" | "-"] <SQLPrimaryExpression>

<SQLPrimaryExpression>::= "NULL" | <FunctionCall>
                                 | <OuterJoinExpression>
                                 | <number>
                                 | ( " ' " {~[" ' "]} " ' " {" ' " {~ [" ' "]} " ' " })
                                 | <bind> | <identifier> | <atribute>
                                 | "(" <PlSqlExpression> ")"
                                | "(" <SubQuery> ")"

<FunctionCall> ::= <identifier> ["." <identifier> ["."<identifier>] ]
                   "(" [ [ DISTINCT | ALL ] ( "*" SQLArguments ) ] ")"

<SQLArguments> ::= <SQLExpressionList>
<OuterJoinExpression>::= <identifier> ["." <identifier> [ "." <identifier> ] ] "(" "+" ")"

<PlSqlExpression>::= <PlSqlAndExpression> { OR < PlSqlAndExpression>}

<PlSqlAndExpression>::= <PlSqlUnaryLogicalExpression> {AND <PlSqlUnaryLogicalExpression>}

<PlSqlUnaryLogicalExpression>::= [NOT] <PlSqlRelationalExpression>

<PlSqlRelationalExpression>::= <PlSqlSimpleExpression>
                               [ (<Relop> | <Relop1>) <PlSqlSimpleExpression>
                                 | <PlSqlInClause>
                                 | <PlSqlBetweenClause>
```

```
                                    | <PlSqlLikeClause>
                                    | <IsNullClause> ]

<PlSqlExpressionList>::= <PlSqlExpression> { "," <PlSqlExpression>}

<PlSqlInClause>::= [ NOT] IN "(" <PlSqlExpressionList> ")"

<PlSqlBetweenClause>::= [NOT] BETWEEN <PlSqlSimpleExpression>
                         AND <PlSqlSimpleExpression>

<PlSqlLikeClause>::= [NOT] LIKE <PlSqlSimpleExpression>

<IsNullClause>::= IS [ NOT ] NULL

<PlSqlSimpleExpression>::= <PlSqlMultiplicativeExpression>
                         { (t = "+" |t = "-" |t = "||") <PlSqlMultiplicativeExpression>}

<PlSqlMultiplicativeExpression>::= <PlSqlExpotentExpression>
                                   { (t = "*" |t = "/") <PlSqlExpotentExpression>}

<PlSqlExpotentExpression>::= <PlSqlUnaryExpression> { "**" < PlSqlUnaryExpression> }

<PlSqlUnaryExpression>::= ( (t = "+" |t = "-") <PlSqlPrimaryExpression> )
                        | <PlSqlPrimaryExpression>

<PlSqlPrimaryExpression>::= NULL
                          |( <identifier> ( %FOUND | %NOTFOUND | %ISOPEN | %ROWCOUNT))
                          |( <identifier> "(" < PlSqlExpressionList> ")" )
                          |( (NEW | OLD ) "." <identifier>)
                          |( SQL (%FOUND | %NOTFOUND | %ISOPEN | %ROWCOUNT))
                          |<identifier>
                          |<number>
                          |(" ' " {~ [" ' "]} " ' " {" ' " {~ [" ' "]} " ' "})
                          | <bind>
                          |<atribute>
                          |"(" PlSqlExpression() ")"
```

<OpenStatement>::= OPEN <identifier> ["(" <PlSqlExpressionList> ")"] ";"

<FetchStatement>::= FETCH <identifier>
INTO (<identifier> | (":" <identifier> ["." <identifier>])
{"," (<identifier> | (":" <identifier> ["." <identifier>])} ";"

<CloseStatement>::= CLOSE <identifier> ";"

Comandos Integrados com SQL

<BeginEndBlock>::= BEGIN [[NOT] ATOMIC]
<SequenceOfStatements>
[<ExceptionBlock>]
END [<identifier>] ";"

<SequenceOfStatements>::= {<PLSQLStatement>}

<ExceptionBlock>::= EXCEPTION {<ExceptionHandler>}

<ExceptionHandler>::= WHEN ({ <identifier> { OR <identifier>} | OTHERS)
THEN <SequenceOfStatements>

<PLSQLStatement>::= (<SubroutineCall> <SubroutineCall>)
| <AssignmentStatement>
| <CaseStatement>
| <ExitStatement>
| <LacoStatement>
| <GotoStatement>
| <IfStatement>
| <LeaveStatement>
| <SignalStatement>
| <NullStatement>
| <RaiseStatement>
| <ReturnStatement>
| <SQLStatement>
| [<DeclarationSection>] <BeginEndBlock>

```
<SQLStatement>::=   <CloseStatement>
                    | <DeleteStatement>
                    | <FetchStatement>
                    | <InsertStatement>
                    | <OpenStatement>
                    | <SavepointStatement>
                    | <SelectStatement>
                    | <UpdateStatement>

<SubroutineCall>::= [CALL] <identifier> ["." <identifier>]
                    [ "(" < PlSqlExpressionList> ")" ] ";"

<AssignmentStatement>::= SET (  (  <bind> [":"] "="  <PlSqlExpression>)
                                |(<identifier> [“.”<identifier>] [":"] "=" <PlSqlExpression>) )  ";"

<CaseStatement> ::= CASE [<SQLExpression> ]
                    { WHEN  <SQLExpression> )
                      THEN   <SequenceOfStatements>}
                    [ ELSE <SequenceOfStatements>]
                    END CASE ";"

<ExitStatement>::= EXIT [ <identifier>] [WHEN <SQLExpression>] ";"

        <LacoStatement>::= [<identifier> ":" ]  ( <ForStatement>  |  <LoopStatement> )

        <ForStatement>::= FOR  <identifier>
                (   ( AS  <SelectStatement> DO <SequenceOfStatements> END FOR ";" )
                  | ( IN [ REVERSE] <PlSqlSimpleExpression> ".."
                      <PlSqlSimpleExpression>  <NormalLoop> ) )

<GotoStatement>::= GOTO <identifier> ";"



<IfStatement>::= IF  <SQLExpression>
                 THEN < SequenceOfStatements>
```

```
                [ { (ELSEIF | ELSIF ) <SQLExpression> THEN <SequenceOfStatements>} ]
                [ ELSE <SequenceOfStatements>]
                (END IF | ENDIF ) ";"

<LoopStatement>::=   <NormalLoop>
                   | <Repeat>
                   | <WhileLoop>
                   | <ForLoop>

<NormalLoop>::= LOOP <SequenceOfStatements> END LOOP [<identifier>] ";"

<Repeat>::= REPEAT <SequenceOfStatements>
            UNTIL <SQLExpression>
            END REPEAT ";"

<WhileLoop>::= WHILE <SQLExpression>
              ( <NormalLoop>
               | DO <SequenceOfStatements> END WHILE ";" )

<ForLoop> ::= <NumericForLoopLookahead>
             ( <NumericForLoop> | <CursorForLoop> )

<NumericForLoopLookahead> ::= FOR <identifier> IN [REVERSE] <PlSqlSimpleExpression> ".."

<NumericForLoop> ::= FOR <identifier> IN [ REVERSE ]
                     <PlSqlSimpleExpression> ".." <PlSqlSimpleExpression> <NormalLoop>

<CursorForLoop> ::= FOR <identifier> IN
                    (  <identifier> [ "(" <PlSqlExpressionList> ")"]
                     | "(" <SelectStatement> ")" )
                    <NormalLoop>

<NullStatement> ::= NULL ";"

<RaiseStatement>::= (RAISE[<identifier>]|RAISE_APPLICATION_ERROR "(" <Arguments> ")") ";"";"
<LeaveStatement>::= LEAVE [<identifier>] ";"
```

<SignalStatement>::= (SIGNAL | RESIGNAL) <identifier> ";"

<ReturnStatement::= RETURN [<PlSqlExpression>] ";"

APÊNDICE B

Demo DB – Um Exemplo de Banco de Dados

Neste apêndice é descrito um banco de dados denominado *Demo DB*, que foi utilizado ao longo deste trabalho, baseado inteiramente em Leitão et. al, 2002.

Demo DB é um banco de dados de demonstração especialmente construído para apoiar os exemplos de regras ativas deste trabalho. Este banco de dados é baseado no exemplo explorado em Elmasri e Navathe, 2000. Uma breve descrição do banco de dados é exibida abaixo:

- a empresa Demo foi fundada em 1911 e hoje atua em todo país no ramo de serviços;
- é constituída por vários departamentos, cada qual possuindo um único empregado gerente que iniciou nesta responsabilidade em uma certa data; é importante saber a quantidade de empregados em cada departamento; um departamento pode ser descentralizado, estando distribuído vários locais;
- todo empregado é unicamente identificado por sua matrícula e é descrito por nome, sexo, data de nascimento, etc.; um empregado pode possuir um único supervisor, o qual sempre possui um salário superior ao seu; todo empregado está alocado a algum departamento;
- existem diversos projetos em andamento, cada qual executado em um único local e controlado por algum departamento; um empregado pode trabalhar para vários projetos, independentemente se são controlados pelo departamento em que ele está alocado, respeitando-se um determinado número de horas semanais, que não pode exceder a um valor limite;
- é pertinente saber o grau de parentesco de cada dependente com seu empregado responsável;

- um registro de algumas transações de interesse deve ser mantido, refletindo a política de negócios de cada momento.

Toda a representação deste banco de dados está ilustrado abaixo na Figura B.1.

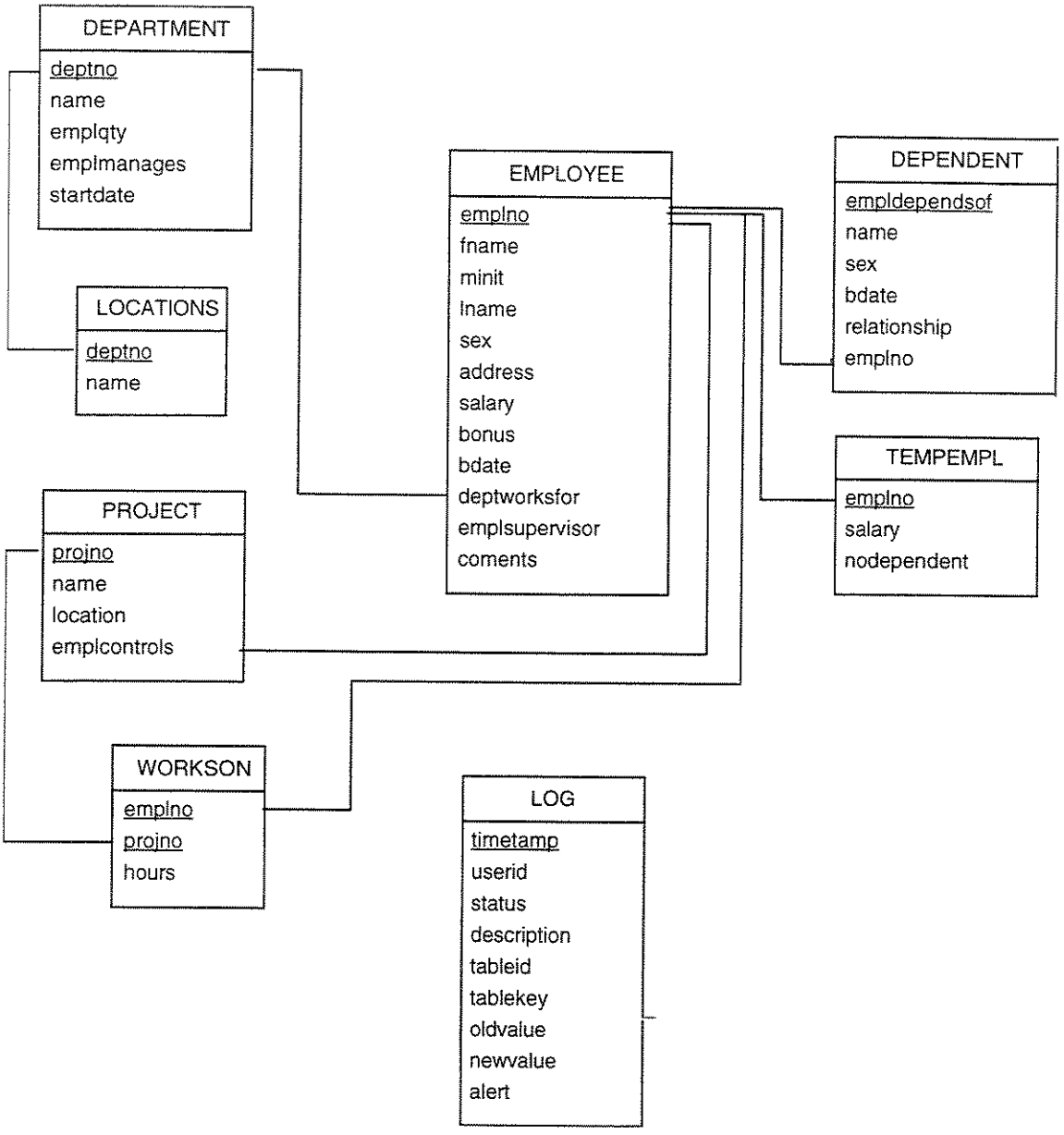


Figura B.1: Representação do banco de dados *Demo DB*.

APÊNDICE C

Exemplos de Regras Ativas

Neste apêndice são mostrados 10 exemplos de regra ativas; 5 regras foram elaboradas com o objetivo de mostrar a utilização da ferramenta ART-TOOL e de explorar o uso dos comandos da linguagem SQL. Estes exemplos podem ser vistos em Leitão (2002). As outras 5 regras testadas foram cedidas por uma empresa.

Para cada exemplo de regra ativa são apresentadas tabelas com uma breve descrição de suas funcionalidades e o seu código. As informações obtidas durante o teste são mostradas na ordem: a representação do fluxo de controle e do fluxo de dados obtidos da análise estática; a tabela com a descrição dos casos de teste executados; a tabela com o número de elementos requeridos executados e não executados de acordo com cada critério; e a tabela com a cobertura dos critérios obtida na avaliação dos testes.

C.1 - Exemplo 1

A regra RA possui comandos de manipulação sem tratamento de exceção e comando de desvio incondicional.

Tabela C-1: Descrição da regra ativa RA.

Regra RA: Quando houver inclusão de um funcionário ou alteração de seu salário não será permitido que este salário seja maior do que o salário de seu supervisor direto; mas se for somente uma alteração do salário, esta não deve permitir que o salário de um funcionário seja modificado em mais de 30%. Deve ser registrado o salário do funcionário se modificado entre 10% e 30%.

Tabela C-2: Código da regra ativa RA.

```
CREATE TRIGGER RA
BEFORE INSERT OR UPDATE OF salary, emplsupervisor
ON employee
FOR EACH ROW
DECLARE v_salary employee.salary%type;

BEGIN
   IF ( :new.emplsupervisor IS NOT NULL ) THEN
           SELECT salary INTO v_salary
           FROM   employee WHERE  ( emplno = :new.emplsupervisor );
           IF ( :new.salary > v_salary ) THEN
                RAISE_APPLICATION_ERROR (-20000, 'Invalid salary update
                   (> supervisor) - employee: ' ||TO_CHAR(:old.emplno));
           END IF;
   END IF;

  IF ( UPDATING ) THEN

      IF ABS(:new.salary - :old.salary) > (:old.salary * 0.3) THEN
         RAISE_APPLICATION_ERROR (-20000, 'Invalid salary update -
            employee: ' || TO_CHAR(:old.emplno));
      ELSIF ABS(:new.salary - :old.salary) > (:old.salary * 0.2) THEN
         INSERT INTO log
            (timetamp, userid, status, description, tableid, tablekey,
                  oldvalue, newvalue, alert)
          VALUES (SYSDATE, user, 'W', 'update salary', 'employee',
                   TO_CHAR(:old.emplno), TO_CHAR(:old.salary),
                   TO_CHAR(:new.salary), 'new value > 20%');
      ELSIF ABS(:new.salary - :old.salary) > (:old.salary * 0.1) THEN
             INSERT INTO log
                 (timetamp, userid, status, description, tableid,
                  tablekey, oldvalue, newvalue, alert)
             VALUES (SYSDATE, user, 'W', 'update salary', 'employee',
                     TO_CHAR(:old.emplno), TO_CHAR(:old.salary),
                     TO_CHAR(:new.salary), 'new value > 10% and < 30%');
      ELSE
             INSERT INTO log
                 (timetamp, userid, status, description, tableid,
                  tablekey, oldvalue, newvalue, alert)
             VALUES (SYSDATE, user, 'S', 'update salary', 'employee',
                TO_CHAR(:old.emplno), TO_CHAR(:old.salary),
                     TO_CHAR(:new.salary), NULL);
     END IF;
  END IF;
END;
```

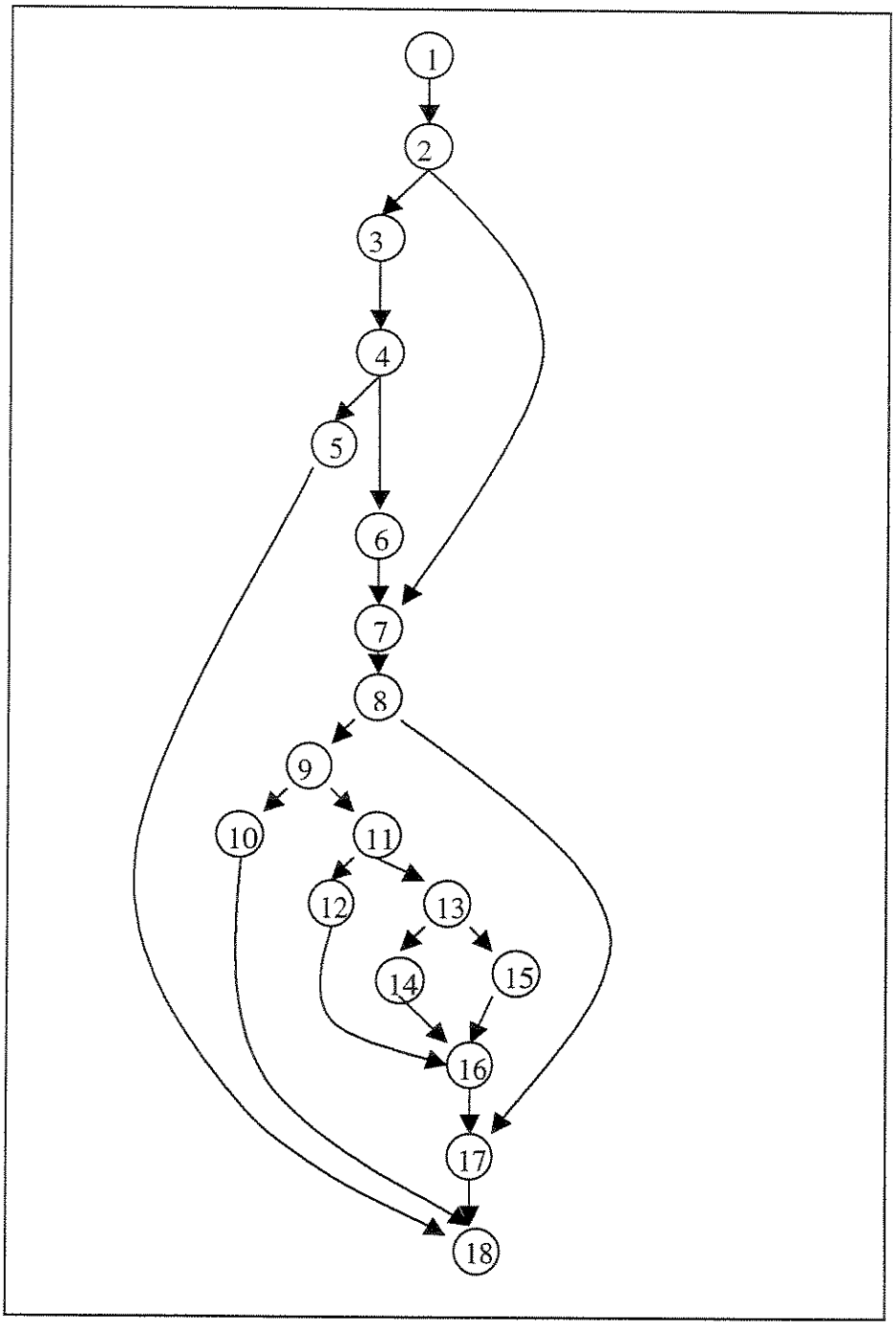


Figura C-1: Representação do fluxo de controle da regra ativa RA.

Tabela C-3: Representação do fluxo de dados da regra ativa RA.

Nó/ Arco	Definição	C_uso	P-uso
1	log employee NEW OLD	employee	
(2,3) (2,7)			NEW
3	v_salary	employee OLD	
(4,5) (4,6)			NEW v_salary
(9,10) (9,11)			NEW OLD
(11,12) (11,13)			NEW OLD
12	Log	OLD NEW	
(13,14) (13,15)			NEW OLD
14	log	OLD NEW	
15	log	OLD NEW	

Tabela C-4: Descrição dos casos de teste da regra ativa RA.

Caso de Teste	Descrição	Evento	Caminho a ser executado
1	Inclusão de um funcionário que não exerce a função de supervisor e com salário menor do que seu supervisor.	Insert	1 2 3 4 6 7 8 17 18
2	Inclusão de um funcionário que não exerce a função de supervidor, mas seu salário é maior do que o salário do se supervisor.	Insert	1 2 3 4 5 18
3	Alteração de 5% do salário de um funcionário que não exerce a função de supervisor.	Update	1 2 3 4 6 7 8 9 11 13 15 16 17 18
4	Alteração de 15% do salário de um funcionário que não exerce a função de supervisor.	Update	1 2 3 4 6 7 8 9 11 13 14 16 17 18
5		Delete	Não há ativação da regra

Tabela C-5: Número de Elementos Requeridos (ER) da regra ativa RA.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	16	2	18
Todos Arcos	6	2	8
Usos	20	6	26
Todos P-Usos	14	4	18
Todos Pot-Uso	12	6	18
Todos Pot-Du-Caminhos	9	11	20
Todos Pot-Uso-Du	12	6	18
Todos Du-Caminhos	11	7	18

Tabela C-6 : Cobertura da regra ativa RA.

Todos Nós	Todos Arcos	Todos Usos	Todos P-Usos	Todos Pot-Usos	Todos Pot-Du-Caminhos	Todos Pot-Uso-Du	Todos Du-Caminhos
89%	75%	77%	78%	67%	45%	67%	61%

Com a aplicação do critério Todos Nós foi possível observar que quando a tabela de funcionários se encontra vazia e ocorre a inclusão de um supervisor é provocada uma exceção devido à falta de dados para a consulta especificada no nó 3. Como a regra RA não possui tratamento de exceção, esta é abortada nessa situação. Nos demais comandos de manipulação que constam nessa regra, comandos INSERT, só é possível ocorrer uma exceção se houver troca de variáveis que compõem a chave primária e estas estiverem com valores iguais.

Foram inseridos defeitos nas variáveis de transição (troca de NEW para OLD). Com a aplicação dos critérios Todos Usos, Todos P-Usos e Todos Pot-Usos foram requeridas e exercitadas associações que envolviam as variáveis de transição, possibilitando assim detectar os defeitos presentes.

C .2 - Exemplo 2

A regra ativa RB testada possui comandos de laço e comandos de manipulação. Esta regra é ativada com a ocorrência de qualquer um dos eventos (INSERT, UPDATE, DELETE).

Tabela C-7: Descrição da regra ativa RB.

Regra RB: com a ocorrência de uma inclusão, exclusão ou alteração esta regra é ativada para determinar o bônus (% do salário) em função da quantidade de horas alocadas em projeto e registra a quantidade de horas em que existe mais empregados trabalhando em projetos.

Tabela C-8: Código da regra ativa RB.

```
CREATE TRIGGER RB
AFTER INSERT OR DELETE OR UPDATE ON workson FOR EACH ROW
DECLARE v_qtdehour INTEGER;
        v_bonus    c_Employee.bonus%type;
        v_qtdeempl INTEGER; v_qtde INTEGER; v_ind INTEGER;
BEGIN
   IF ( DELETING ) OR ( UPDATING ) THEN
      SELECT SUM(hours) INTO v_qtdehour
      FROM   workson WHERE  emplno = :new.emplno;
      IF v_qtdehour <= 10 THEN
         v_bonus := 00.0;
      ELSIF v_qtdehour <= 20 THEN
         v_bonus := 10.0;
      ELSE
         v_bonus := 20.0;
      END IF;
      UPDATE c_Employee
      SET bonus = v_bonus WHERE (emplno=:old.emplno);
   END IF;
   IF ( INSERTING ) OR ( UPDATING ) THEN
      SELECT SUM(hours) INTO v_qtdehour
      FROM workson WHERE  ( emplno = :new.emplno );
      IF v_qtdehour <= 10 THEN
         v_bonus := 00.0;
      ELSIF  v_qtdehour <= 20 THEN
         v_bonus := 10.0;
      ELSE
         v_bonus := 20.0;
      END IF;
      UPDATE c_Employee
      SET    bonus = v_bonus WHERE  ( emplno = :new.emplno );
   END IF;
   v_qtdeempl := 0;
   v_qtdehour := 0;
   v_ind      := 0;
   WHILE (v_ind < 10) LOOP
      v_ind := v_ind + 1;
      SELECT COUNT(*) INTO v_qtde
      FROM   workson
      WHERE  ( hours = v_ind );
      IF ( v_qtde > v_qtdeempl ) THEN
         v_qtdeempl := v_qtde;
         v_qtdehour := v_ind;
      END IF;
   END LOOP;
   INSERT INTO log
    (timetamp,userid,status,description,tableid,tablekey,oldvalue,
     newvalue, alert)
   VALUES (SYSDATE,user,'W','hour category with more employee
            quantity', 'workson',NULL,NULL,NULL,
            TO_CHAR (v_qtdeempl) ||'employees works'||
            TO_CHAR(v_qtdehour)||'hours a week');
END;
```

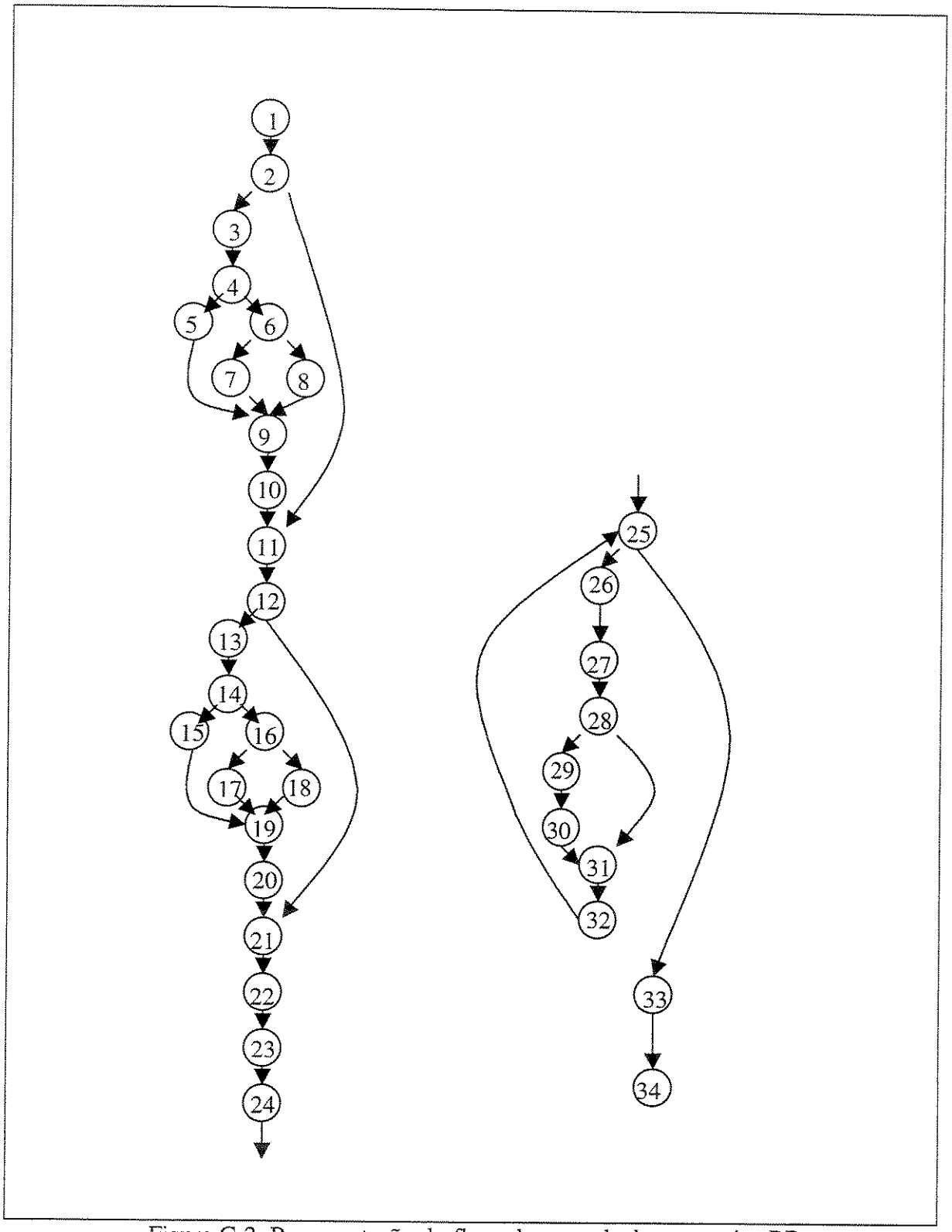


Figura C-2: Representação do fluxo de controle da regra ativa RB.

Tabela C-9: Representação do fluxo de dados da regra ativa RB.

Nó/ Arco	Definição	C-uso	P-uso
1	c_Employee log workson NEW OLD	workson	
3	v_qtdehour	workson NEW	
(4,5) (4,6)			v_qtdehour
5	v_bonus		
(6,7) (6,8)			v_qtdehour
7	v_bonus		
8	v_bonus		
10	c_Employee	c_Employee v_bonus OLD	
13	v_qtdehour	workson NEW	
(14,15) (14,16)			v_qtdehour
15	v_bonus		
(16,17) (16,18)			v_qtdehour
17	v_bonus		
18	v_bonus		
20	c_Employee	c_Employee v_bonus NEW	
22	v_qtdeempl		
23	v_qtdehour		
24	v_ind		
25,26) (25,33)			v_ind
26	v_ind	v_ind	
27	v_qtde	workson v_ind	
(28,29) (28,31)			v_qtde v_qtdeempl
29	v_qtdeempl	v_qtde	
30	v_qtdehour	v_ind	
33	log	v_qtdeempl v_qtdehour	

Tabela C-10: Descrição dos casos de teste da regra ativa RB.

Caso de Teste	Descrição	Evento	Caminho a ser executado
1	Exclusão de uma tupla com um funcionário que trabalha 8 horas no projeto 1, sendo que o total de horas deste passa a ser 7.	Delete	1 2 3 4 5 9 10 11 12 21 22 23 24 25 25 26 27 28 29 30 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 33 34
2	Inclusão de uma tupla com um funcionário que trabalha 4 horas no projeto 1, sendo que o total de horas deste passa a ser 6.	Insert	1 2 11 12 13 14 15 19 20 21 22 23 24 25 26 27 28 29 30 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 33 34
3	Alteração de uma tupla com um funcionário que trabalha 4 horas e passa a trabalhar 8 horas no projeto 1, sendo que o total de horas deste passa a ser 15.	Update	1 2 11 12 13 14 16 17 19 20 21 22 23 24 25 26 27 28 29 30 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 33 34
4	Alteração de uma tupla com um funcionário que trabalha 8 horas e passa a trabalhar 9 horas no projeto 2, sendo que o total de horas deste passa a ser 21.	Update	1 2 11 12 13 14 16 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 26 27 28 31 32 25 33 34

Tabela C11: Número de ERs regra ativa RB.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	27	7	34
Todos Arcos	7	4	11
Todos Usos	28	15	43
Todos P-Uso	11	7	18
Todos Pot-Uso	29	19	48
Todos Pot-Uso	18	32	50
Todos Pot-Du Caminhos	21	84	105
Todos Pot-Uso/ Du	36	72	108
Todos Du-Caminhos	25	46	71

Tabela C-12 : Cobertura da regra ativa RB.

Todos Nós	Todos Arcos	Todos Usos	Todos P-Usos	Todos Pot-Usos	Todos Pot-Du-Caminhos	Todos Pot-Uso-Du	Todos Du-caminhos
79%	64%	65%	61%	43%	19%	32%	35%

Esta regra possui comandos de manipulação e não tem um tratamento de exceção, devido a este fato esta regra poderá ser abortada se a tabela consultada estiver vazia. Esta exceção pode ocorrer em todos os comandos de consulta incluídos nesse código, esta conclusão foi possível com a aplicação do critério de Todos Nós. Foram inseridos defeitos, troca de tabelas em comandos de manipulação que foram detectados também através dos critérios Todos Nós. Os defeitos inseridos nos comandos de manipulação em variáveis locais e de transição foram detectados com a aplicação de critérios, Todos Usos e Todos Potenciais Usos, que exigiram o exercício de associações que continham essas variáveis.

C.3 - Exemplo 3

Este exemplo mostra uma regra ativa, RC, com um cursor, comandos de manipulação sem tratamento de exceção, comandos de laço e comando de desvio

incondicional. Nessa regra, na linha 18, foi inserido um comando para fechar o cursor pela segunda vez; e na linha 27, ao invés da variável v_salary foi colocada a variável v_emplno.

Tabela C-13: Descrição da regra ativa RC.

Regra RC: Após uma exclusão, inclusão ou alteração na tabela de funcionários esta regra dispara e armazena na tabela tempempl os 10 funcionários que possuem maiores salários, e se estes possuirem dependentes, o número total é armazenado também.

Tabela C-14: Código da regra ativa RC.

```
CREATE TRIGGER RC
AFTER DELETE OR INSERT OR UPDATE OF emplno, salary
ON c_Employee
DECLARE
   v_qtde INTEGER;
   v_emplno c_Employee.emplno%type;
   v_salary c_Employee.salary%type;
   CURSOR crEmpl IS
         SELECT emplno, salary
         FROM   c_Employee
         ORDER BY salary DESC;
BEGIN
      DELETE FROM tempempl;
      OPEN crEmpl;
      LOOP
         FETCH crEmpl INTO v_emplno, v_salary;
         IF (crEmpl%ROWCOUNT > 10) OR (crEmpl%NOTFOUND) THEN
             CLOSE crEmpl;
             EXIT;
         END IF;
         SELECT COUNT(*) INTO v_qtde
         FROM  dependent
         WHERE emplno = v_emplno;
         INSERT INTO tempempl
              ( emplno, salary, dependent )
         VALUES
              ( v_emplno, v_emplno, v_qtde );
     END LOOP;
     CLOSE crEmpl;
END;
```

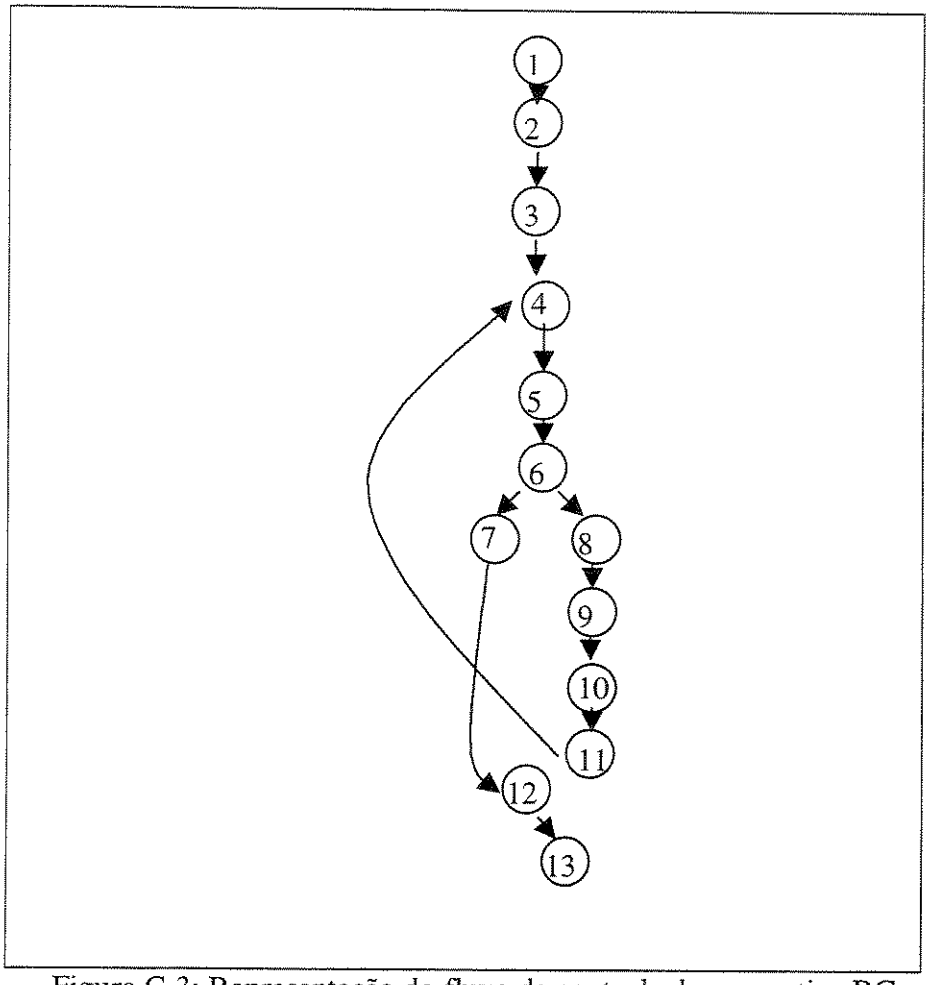


Figura C-3: Representação do fluxo de controle da regra ativa RC.

Tabela C-15: Representação do fluxo de dados da regra ativa RC.

Nó/ Arco	Definição	C-Uso	P-Uso
1	c_Employee tempempl	c_Employee	
2	tempempl		
3	crEmpl	c_Employee	
5	v_emplno v_salary	crEmpl	
(6,7) , (6,8)			crEmpl
10	tempempl	v_emplno v_salary	

Tabela C-16: Descrição dos casos de teste da regra ativa RC.

Caso de Teste	Descrição	Evento	Caminho esperado
1	Inclusão de um funcionário sem dependentes com salário entre os 10 classificados.	Insert	1 2 3 4 5 6 9 1011 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 7 8 13 14
2	Exclusão de um funcionário que estava na lista dos 10 maiores salários.	Delete	1 2 3 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 7 8 13 14
3	Atualização do salário de um funcionário, mas com valor negativo.	Update	1 2 3 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 9 10 11 12 4 5 6 7 8 13 14
4	Tabela de funcionários com apenas um funcionário, alteração de seu salário.	Update	1 2 3 4 5 6 9 10 11 12 4 5 6 7 8 13 14

Tabela C-17: Número de ERs da regra ativa RC.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	13		13
Todos Arcos	2		2
Todos Usos	6		6
Todos P-Uso	2		2
Todos Pot-Uso	16	1	17
Todos Pot-Du Caminhos	7	3	10
Todos Pot-Uso/Du	14	3	17
Todos Du-Caminhos	4	1	5

Tabela C-18: Cobertura da regra ativa RC.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
100%	100%	100%	100 %	95 %	79 %	86 %	89 %

O defeito inserido, a troca de variáveis da linha 27, foi detectado com o exercício dos nós exigido pelo critério Todos Nós e também na aplicação de critérios baseados em fluxo de dados.

O outro defeito, referente ao comando de cursor, com o exercício da associação da variável cursor incluindo os dois comandos de CLOSE, requerida pelo critério Todos Potenciais Usos, foi possível detectar.

C.4 - Exemplo 4

A regra ativa é pequena, mas possui comandos de manipulação sem tratamento de exceção, variáveis de transição e o comando de IF. Nessa regra foram inseridos defeitos. No evento foi inserida a operação DELETE e na linha 15 a variável de transição foi trocada de NEW para OLD.

Tabela C-19: Descrição da regra ativa RD.

Regra RD: Antes de uma inclusão ou alteração na tabela de funcionários esta regra dispara e registra a existência de um empregado com idade inferior a 18 anos.

Tabela C-20: Código da regra ativa RD.

```
CREATE TRIGGER RD
   BEFORE INSERT OR DELETE OR UPDATE OF bdate
   ON Employee
   FOR EACH ROW
DECLARE v_months NUMBER;
BEGIN
   SELECT ADD_MONTHS(:new.bdate, (18*12)) - SYSDATE INTO v_months
    FROM DUAL;
   IF v_months > 0 THEN
      INSERT INTO Log
        (timestamp, userid, status, description, tableid, tablekey,
          oldvalue, newvalue, alert)
      VALUES (SYSDATE, user, 'W', 'update bdate', 'employee',
              TO_CHAR(:old.emplno),
              TO_CHAR(:old.bdate),TO_CHAR(:old.bdate),'age < 18');
   END IF;
END;
```

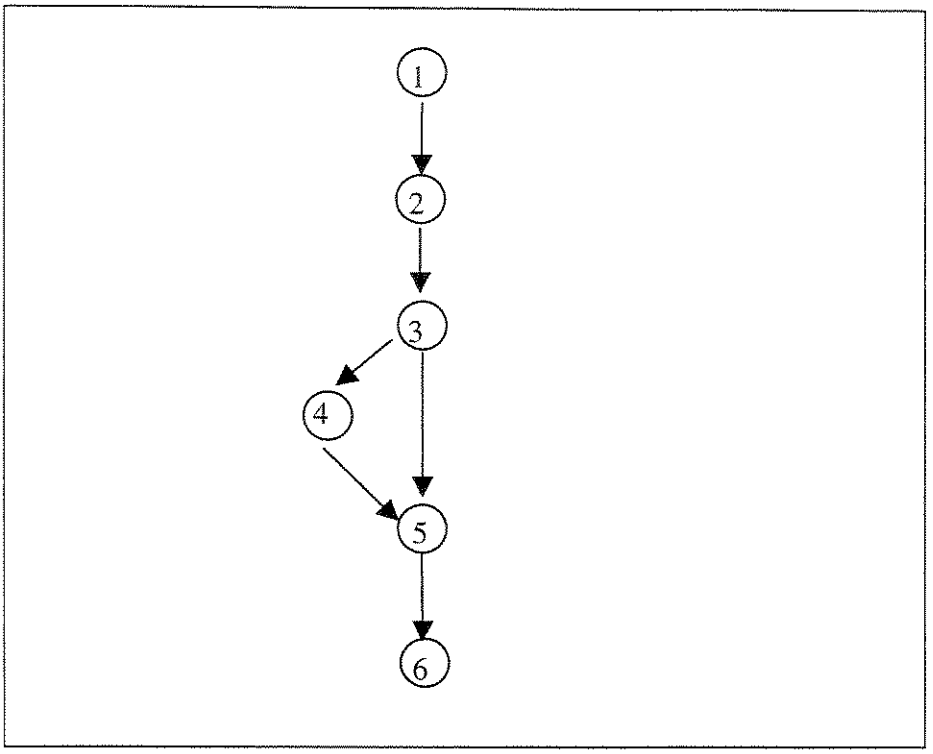


Figura C-4: Representação do fluxo de controle da regra ativa RD.

Tabela C-21: Representação do fluxo de dados da regra ativa RD.

Nó/ Arco	Definição	C-Uso	P-Uso
1	Employee Log, NEW, OLD	Employee	
2	v_months	NEW	
(3, 4), (3,5)			v_months
4	Log	OLD	

Tabela C-22: Descrição dos casos de teste da regra ativa RD.

Caso de Teste	Descrição	Evento	Caminho esperado
1	Inclusão de um funcionário com idade de 18 anos.	Insert	1 2 3 5 6
2	Exclusão de um funcionário com mais de 18 anos.	Delete	A regra não deve ser ativada
3	Atualização da data de nascimento de um funcionário, que é menor de 18 anos.	Update	1 2 3 4 5 6

Tabela C- 23: Número de ERs da regra ativa RD.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	6		6
Todos Arcos	2		2
Todos Usos	4		4
Todos P-Uso	2		2
Todos Pot-Uso	5		5
Todos Pot-Du Caminhos	5		5
Todos Pot-Uso/Du	5		5
Todos Du-Caminhos	4		4

Tabela C- 24: Cobertura da regra ativa RD.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
100%	100%	100%	100 %	100%	100%	100%	100%

Com a inclusão de todos eventos (INSERT, DELETE e UPDATE) no conjunto de casos de teste foi notado o defeito inserido no evento. Com a execução do caso de teste 2 a regra foi ativada e não deveria ser.

Essa regra por ser pequena foi possível um exercício de suas partes obtendo uma cobertura de 100% em todos os critérios aplicados. O defeito da variável de transição só foi detectado com a aplicação dos critérios Todos Nós, Todos Usos e Todos Potenciais Usos. Outro fato notado é que apesar da regra RD possuir comandos de manipulação não é necessária a inclusão de tratamento de exceção, pois não foi possível provocar uma exceção considerando os comandos de manipulação que constam nesse código.

C.5 - Exemplo 5

Esta regra ativa possui o comando FOR, específico da linguagem PL/SQL, em seu código; e comandos de manipulação sem tratamento de exceção.

Tabela C- 25: Descrição da regra ativa RE.

Regra RE: Considerando que funcionários podem trabalhar para projetos de 1 até 10 horas semanais, a regra registra na tabela de ocorrências a faixa de quantidade de horas em que existe mais empregados, independente de empregado e de projeto. Esta regra é ativada após uma inclusão, exclusão ou alteração na tabela de funcionários.

Tabela C- 26: Código da regra ativa RE.

```
CREATE TRIGGER RE
   AFTER INSERT OR UPDATE OR DELETE
   ON workson
DECLARE v_qtdeempl INTEGER;
        v_qtdehour INTEGER;
        v_qtde     INTEGER;
        v_ind      INTEGER;
BEGIN
   v_qtdeempl := 0;
   v_qtdehour := 0;
   FOR v_ind IN 1..10 LOOP
      SELECT COUNT(*) INTO v_qtde
      FROM   workson
      WHERE  ( hours = v_ind );
      IF ( v_qtde > v_qtdeempl ) THEN
         v_qtdeempl := v_qtde;
         v_qtdehour := v_ind;
      END IF;
   END LOOP;
   INSERT INTO log
          (timetamp, userid, status, description, tableid,
             tablekey, oldvalue, newvalue, alert)
   VALUES (SYSDATE, user, 'W', 'hour category with more employee
            quantity', 'workson',NULL, NULL, NULL,
            TO_CHAR(v_qtdeempl) ||'employees works' ||
            TO_CHAR(v_qtdehour) ||'hours a week');
END;
```

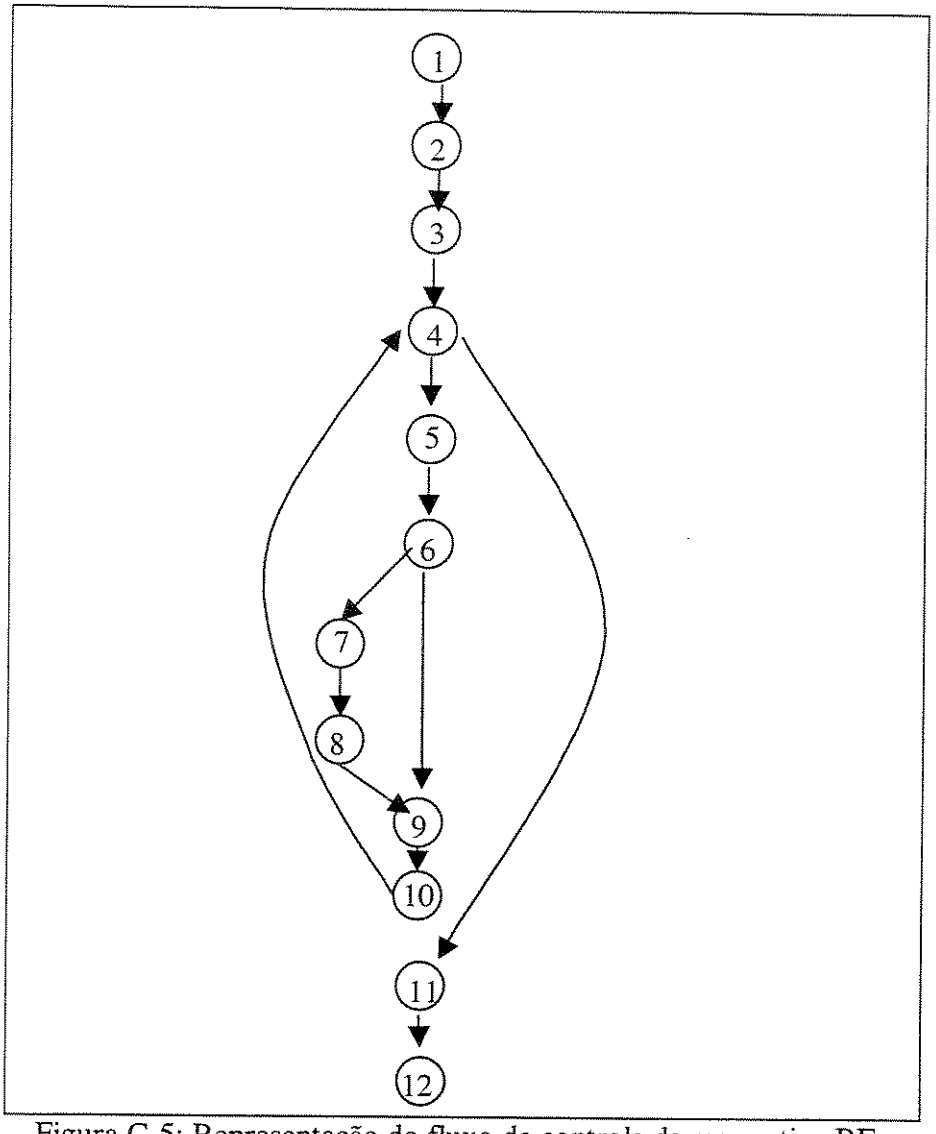


Figura C-5: Representação do fluxo de controle da regra ativa RE.

Tabela C-27: Representação do fluxo de dados da regra ativa RE.

Nó/ Arco	Definição	C-Uso	P-Uso
1	Workson, Log	Workson	
2	v_qtdeempl		
3	v_qtdehour		
4	v_ind		
(4, 5), (4, 11)			v_ind
5	v_qtde	Workson	
(6, 7), (6, 9)			v_qtde, v_qtdeempl
7	v_qtdeempl	v_qtde	
8	v_qtdehour	v_ind	
11	Log	v_qtdeempl, v_qtdehour	

Tabela C-28: Descrição dos casos de teste da regra ativa RE.

Caso de Teste	Descrição	Evento	Caminho esperado
1	Inclusão do primeiro funcionário que trabalha 0 horas.	Insert	1 2 3 4 5 6 9 10 4 5 6 9 10 4 5 6 9 10.....4 5 6 9 10 4 11 12
2	Exclusão de um funcionário que trabalha 2 horas em um projeto.	Delete	1 2 3 4 5 6 7 8 9 10 4 5 6 7 8 9 10 4 5 6 7 8 9 10 .....4 5 6 7 8 9 10 4 11 12
3	Alteração de um funcionário que passa a trabalhar 5 horas em um projeto.	Update	1 2 3 4 5 6 7 8 9 10 4 5 6 7 8 9 10 4 5 6 7 8 9 10 .....4 5 6 7 8 9 10 4 11 12
4	Exclusão do ultimo funcionário que trabalha em projetos. Tabela vazia.	Delete	1 2 3 4 5 6 9 10 4 5 6 9 10 4 5 6 9 10.....4 5 6 9 10 4 11 12

Tabela C- 29: Número de ERs da regra ativa RE.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	12		12
Todos Arcos	3		3
Todos Usos	10		10
Todos P-Uso	5		5
Todos Pot-Uso	28	2	30
Todos Pot-Du Caminhos	18	5	23
Todos Pot-Uso/Du	24	6	30
Todos Du-Caminhos	7	2	9

Tabela C- 30: Cobertura da regra ativa RE.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
100%	100%	100%	100 %	93%	78%	80%	78%

Uma grande parte da regra foi exercitada, obtendo uma cobertura maior que 70% em todos os critérios aplicados. Na execução do caso de teste 4 com a aplicação de critérios baseados em fluxo de controle ficou evidente a necessidade de tratamento de exceção. Esta situação é caracterizada por uma tabela vazia onde são feitas consultas (nó 5) nesta tabela resultando em uma exceção , pois não há dados para consulta.

C.6 – Exemplos Reais

As cinco regras cedidas são responsáveis pela manutenção do estoque. Estão apresentados somente a quantidade de elementos requeridos executados e não executados pelos casos de teste e a cobertura de cada critério, pois a empresa não autorizou a divulgação das regras.

A conclusão do teste feito com este conjunto de regras é discutido no final deste Apêndice.

C.6.1 – Exemplo 6

A regra ER1 é responsável pela manutenção de algumas tabelas quando ocorre a devolução de produtos. Estão incluídos no código da regra ER1 comandos de cursor, comandos de manipulação (INSERT, SELECT e UPDATE) e comando IF. É uma regra composta de 10 nós.

Tabela C- 31: Número de ERs da regra ativa ER1.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	10		10
Todos Arcos	2		2
Todos Usos	8		8
Todos P-Uso	3		3
Todos Pot-Uso	9		9
Todos Pot-Du Caminhos	8		8
Todos Pot-Uso/Du	9		9
Todos Du-Caminhos	5		5

Tabela C- 32: Cobertura da regra ativa ER1.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
100%	100%	100%	100%	100%	100%	100%	100%

C.6.2 – Exemplo 7

ER2 é uma regra ativa que controla a reposição da devolução de produtos que são de consumo interno. É composta de 16 nós e em seu código estão inseridos comandos de cursor, comandos de manipulação, comando IF e variáveis de transição.

Tabela C- 33: Número de ERs da regra ativa ER2.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	16		16
Todos Arcos	2	1	3
Todos Usos	21	1	22
Todos P-Uso	3	1	4
Todos Pot-Uso	12	1	13
Todos Pot-Du Caminhos	10	1	11
Todos Pot-Uso/Du	12	1	13
Todos Du-Caminhos	18	1	19

Tabela C- 34: Cobertura da regra ativa ER2.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
100%	67%	95%	75 %	92%	90%	92%	95%

C.6.3 – Exemplo 8

A regra ativa ER3 é responsável pela entrada de produtos e suas respectivas atualizações. É uma regra composta de 19 nós que possui comandos de manipulação, comando IF e variáveis de transição.

Tabela C- 35: Número de ERs da regra ativa ER3.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	18	1	19
Todos Arcos	3	2	5
Todos Usos	17	3	20
Todos P-Uso	5	3	8
Todos Pot-Uso	10	6	16
Todos Pot-Du Caminhos	7	8	15
Todos Pot-Uso/Du	10	6	16
Todos Du-Caminhos	15	10	25

Tabela C- 36: Cobertura da regra ativa ER3.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
95%	60%	85%	63%	63%	47%	63%	60%

C.6.4 – Exemplo 9

A regra ativa ER4 atualiza uma tabela específica do estoque de acordo com as vendas dos produtos. O seu código inclui comandos de cursor, comandos de manipulação, comando IF e variáveis de transição.

Tabela C- 37: Número de ERs da regra ativa ER4.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	13	1	14
Todos Arcos	1	3	4
Todos Usos	7	8	15
Todos P-Uso	3	5	8
Todos Pot-Uso	6	8	14
Todos Pot-Du Caminhos	5	8	13
Todos Pot-Uso/Du	6	8	14
Todos Du-Caminhos	4	7	11

Tabela C- 38: Cobertura da regra ativa ER4.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
93%	25%	47%	38 %	43%	38%	43%	36%

C.6.5 – Exemplo 10

A regra ativa ER5, após a inclusão de produtos, atualiza todo o estoque. Em seu código estão incluídos 3 cursores, comandos de manipulação, comando IF e variáveis de transição. É uma regra composta de 32 nós.

Tabela C- 39: Número de ERs da regra ativa ER5.

Critérios	Número de ERs executados	Número de ERs não executados	Total de ERs
Todos Nós	32		32
Todos Arcos	4	2	6
Todos Usos	66	2	68
Todos P-Uso	8	2	10
Todos Pot-Uso	50	21	71
Todos Pot-Du Caminhos	35	42	77
Todos Pot-Uso/Du	50	21	71
Todos Du-Caminhos	48	10	58

Tabela C- 40: Cobertura da regra ativa ER5.

Critérios							
Todos Nós	Todos Arcos	Todos Usos	Todos P-Uso	Todos Pot-Uso	Todos Pot-Du Caminho	Todos Pot – Uso /Du	Todos Du-Caminho
100%	67%	97%	80%	70%	44%	70%	83%

C.6.6 – Conclusões dos exemplos ER1 a ER5

A princípio não foram detectados defeitos nessas regras. A aplicação do critério Todos Nós possibilitou verificar se havia defeitos principalmente nos comandos de manipulação, pois estes são representados em nós separados. Com a aplicação dos critérios baseados em fluxo de controle foi possível verificar se havia defeitos em variáveis persistentes. Para verificar as variáveis de transição e variáveis locais foi necessário também a aplicação de critérios baseados em fluxo de dados.

A regra ER5 possui vários cursores e também comandos de manipulação e não possui tratamento de exceção. Na aplicação do critérios baseados em fluxo de controle foi detectada a falta de tratamento de exceção. No primeiro caso trata-se de uma consulta e se a tabela estiver vazia não haverá dados para a consulta, provocando uma exceção. O outro caso é de um comando de atualização em que pode ocorrer a duplicidade de dados, podendo então a regra vir a ser abortada.