Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Teste Baseado na Interação entre Regras Ativas Escritas em SQL

Autor: Plínio de Sá Leitão Júnior
Orientador: Prof. Dr. Mario Jino
Co-orientador: Prof. Dr. Plínio Roberto Souza Vilela

Tese de Doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Doutor em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

Banca Examinadora

Ivan Luiz Marques Ricarte.....DCA/FEEC/UNICAMP
José Mario De Martino.....DCA/FEEC/UNICAMP
Léo Pini Magalhães.....DCA/FEEC/UNICAMP
Marcos Lordello Chaim.....EACH/USP
Sílvia Regina Vergílio.....DINF/UFPR

Campinas, SP
Dezembro/2005

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

Leitão Júnior, Plínio de Sá
L536t Teste baseado na interação entre regras ativas escritas em SQL / Plínio de Sá Leitão Júnior. --Campinas, SP: [s.n.], 2005.

Orientadores: Mario Jino, Plínio Roberto Souza Vilela
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Banco de dados relacionais. 2. Engenharia de software. 3. Programas de computador - Testes. 4. SQL (Linguagem de programação de computador). I. Jino, Mario. II. Vilela, Plínio Roberto Souza. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV Título.

Titulo em Inglês: Testing based on the interaction of SQL rules
Palavras-chave em Inglês: Active database application, Adequacy criterion, Software engineering, Software testing, SQL – based application
Área de concentração: Engenharia de Computação
Titulação: Doutor em Engenharia Elétrica
Banca examinadora: Ivan Luiz Marques Ricarte, José Mario De Martino, Léo Pini Magalhães, Marcos Lordello Chaim e Sílvia Regina Vergílio
Data da defesa: 21/12/2005

Resumo

Bancos de dados ativos têm sido usados como uma alternativa à implementação de parte da funcionalidade de muitas aplicações em diversas áreas de conhecimento. A idéia básica é a resposta automática à ocorrência de eventos pela ativação de tarefas com funções específicas, conduzindo à execução de regras ativas. A despeito do uso difundido dessa tecnologia, pouco esforço de pesquisa tem sido direcionado ao teste de aplicações de bancos de dados ativos. Nesta pesquisa, nós investigamos o uso de uma técnica de teste estrutural para revelar a presença de defeitos, visando à melhoria de qualidade e ao aumento do nível de confiança com relação a este tipo de software.

Uma família de critérios de adequação é proposta e analisada, no âmbito de regras ativas escritas em SQL. Especificamente, um modelo de interação entre regras é elaborado, visando à abstração de associações de interação, as quais formam a base para os requisitos de teste. No contexto de teste estrutural baseado em fluxo de dados, é definido um conjunto de critérios de adequação, denominados Critérios Baseados na Interação entre Regras, que requerem o exercício de associações de interação. Os critérios são uma extensão ao critério *todos usos*, pela exploração de relações de fluxo de dados persistentes devido a interações entre regras. Investigações teóricas e empíricas foram conduzidas, demonstrando que os critérios demonstram habilidade na detecção dos defeitos com complexidade polinomial.

Defeitos e falhas de manipulação foram estudados, enumerados e utilizados em um experimento que avalia a habilidade de detecção de defeitos dos critérios em diferentes granularidades: precisões da análise de fluxo de dados. Uma ferramenta chamada ADAPT-TOOL (***A****ctive* ***D****atabase* ***AP****plication* ***T****esting* ***TOOL*** *for active rules written in SQL*) foi construída para suportar o experimento. Os resultados indicam que: *(i)* a eficácia de detecção de defeitos alcançou 2/3 do conjunto adequado, obtendo-se valores mais elevados para granularidades menos precisas; e *(ii)* a cobertura de associações de interação em granularidades mais precisas não melhora a habilidade de revelar defeitos.

Abstract

Active Rule databases have been used as an alternative to the partial implementation of applications in several knowledge domains. Their principle is the automatic response to events by the activation of tasks with specific functionalities, leading to the execution of active rules. Notwithstanding their widespread use, few research efforts have been concentrated on active database application testing. In this research work we investigate the use of a structural testing technique to reveal the presence of faults, aimed at improving reliability and overall quality of this kind of software.

A family of adequacy criteria is proposed and analysed in the active SQL-based database realm. Specifically, an interaction model between rules is elaborated, in order to abstract interaction associations that form the basis for testing requirements. In the context of data flow based structural testing, a family of adequacy criteria is defined, called Interaction Between Rules based Criteria, that demands the coverage of interaction associations. The criteria are an extension to the *all uses* criterion, by the exploitation of persistent data flow relations associated to rule interaction. Both theoretical and empirical investigations were performed, showing that the criteria posses fault detecting ability with polynomial complexity.

Manipulation faults and failures were studied, enumerated and used in an experiment that evaluates criteria fault detecting ability at different granularities: data flow analysis precisions. A tool called ADAPT-TOOL (***A****ctive* ***D****atabase* ***AP****plication* ***T****esting* ***TOOL*** *for active rules written in SQL*) was built to support the experiment. The results indicate that: *i)* the fault-detecting efficacy was 2/3 of the adequate set, and reaches higher values for the lower data flow analysis precision; and *(ii)* the coverage of interaction association at higher granularities does not improve the fault detecting ability.

A Cyndia e a Giovanna, minha família.

Agradecimentos

A Deus, pela delicadeza da vida.

Ao prof. Mario Jino, pelas oportunidades, pela orientação e pelas conversas que resultaram nesta pesquisa.

Ao prof. Plínio Vilela pelas contribuições a esta pesquisa.

Aos meus pais, Plínio e Eliezita, pelo amor, pela educação e por sua verdadeira amizade.

Aos meus irmãos, por sermos uma família.

A minha esposa, Cyndia, pelo amor, dedicação e sorrisos que iluminam nossas vidas.

A minha filha, Giovanna, por sua inocência e chorinho.

Aos colegas da UNICAMP, em especial aos do grupo de teste de software da FEEC.

Ao CNPq, pelo apoio financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

α	Associação de fluxo de dados
β	Granularidade de fluxo de dados (precisão de fluxo de dados)
Γ	Conjunto de dados definidos e usados
γ	Dados definidos e usados
Δ	(Estado de) Base de dados
Θ	Conjunto de operações
θ	Operação
κ	Versão defeituosa de um conjunto de regras
Λ	Conjunto de casos de teste
λ	Caso de teste
M	Seqüência de mensagens
Π	Conjunto de caminhos
π	Caminho
σ	Lista ou seqüência de regras
τ	Tipo de defeito de manipulação
φ	Tipo de operação

Glossário

Aplicação de Banco de Dados
: Aplicação que manipula dados de uma ou mais bases de dados.

Aplicação Baseada em SQL
: Aplicação que interage com bases de dados, utilizando a SQL.

Bancos de Dados Ativos
: Bancos de Dados que possuem regras que são utilizadas pelo sistema para a monitoração de eventos e, quando tais eventos ocorrem, executam as tarefas implementadas pelas regras.

Caso de Teste
: Combinação de dados de entrada e especificação de resultados esperados.

Critério de Adequação
: Critério que define um conjunto de elementos que devem ser exercitados durante o teste.

Dados Persistentes
: Dados que mantêm seus valores mesmo após o término do processo que os criou.

Defeito
: Deficiência mecânica ou algorítmica que pode gerar um erro.

Elementos Requeridos
: Elementos do software que são requeridos por um critério de teste.

Engano
: Ação humana que introduz um defeito no sistema.

Erro
: Estado incorreto do programa (informação ou comando) que pode levar a uma falha.

Falha
: Evento notável (perceptível) em que um sistema (em execução) viola a sua especificação.

Grafo de Fluxo de Controle
: Grafo que representa a estrutura interna de controle de um programa.

Oráculo

Método (ou ser humano) que, dados um valor de entrada e um resultado observado, determina se o resultado obtido está correto (de acordo com o esperado).

Regra Ativa (Regra ECA)

Mecanismo utilizado em bancos de dados ativos, que é ativado pela ocorrência de eventos, com o intuito de executar uma ação (tarefa da regra) quando uma condição for satisfeita.

SQL

Structured Query Language; linguagem para definição, manipulação e controle de bases de dados relacionais.

Teste

Atividade de execução de um programa, cujo objetivo primário é revelar a presença de defeitos.

Teste Baseado em Análise de Fluxo de Dados

Teste que requer o exercício das interações que envolvam definições de variáveis em um programa e subseqüentes referências a essas definições.

Teste Estrutural

Teste que requer o exercício de elementos da estrutura do software implementado.

Capítulo 1

Introdução

Software is ubiquitous. Automated teller machines, electronic mail, anti-lock braking systems, autopilots, on-line reservations systems, on-line banking, mobile telephones: the list is endless. It is nearly impossible to make it through a day anywhere within the developed world without coming into contact with something containing software. Systems and devices on which modern society depends, depend themselves on software.

(Software System Safety & The NASA Aeronautics Blueprint, 2002
C. Michael Holloway and Kelly J. Hayhurst)

1.1 Contexto

A convivência com software é inevitável no mundo atual. Software afeta nossas vidas em muitos aspectos e tem influenciado nossa cultura, nosso comércio e nossas atividades do dia-a-dia. O transporte, a medicina, a telecomunicação, a indústria e as relações sociais, todos são dependentes do software.

Falhas no funcionamento de software de missão crítica, tal como relacionado ao tráfego aéreo, podem causar danos à vida humana; falhas no funcionamento de software de negócios, tal como relacionado à bolsa de valores, podem afetar a segurança, as finanças e a sobrevivência de negócios. Visto que as conseqüências da presença de defeitos no software são preponderantemente negativas, existe a necessidade por atitudes na direção de buscar o funcionamento correto do software.

Segundo Deutsch (1982), o desenvolvimento de sistemas de software envolve uma série de atividades de produção, nas quais as oportunidades de introdução de defeitos são enormes; por esse motivo o desenvolvimento de software deve ser acompanhado por atividades de garantia de

qualidade. Embora qualidade seja um conceito complexo (Sommerville, 2001), Crosby (1979) afirma que a noção de qualidade está relacionada ao fato de que um produto deveria atender às suas especificações. Pressman (2004) sugere que a especificação de requisitos de software é a base para que a qualidade seja medida e, por outro lado, a busca pela qualidade deve reconhecer que o objetivo essencial é a satisfação dos usuários do software (Galin, 2004); ambas as definições reconhecem que a qualidade não pode ser alcançada sem conformidade com a especificação de requisitos.

No cenário de busca pela melhoria de qualidade, a Engenharia de Software é uma disciplina que propõe métodos, ferramentas e procedimentos visando ao aumento da produção de software comprometida com sua boa qualidade, aplicando princípios de engenharia ao desenvolvimento de software. Pressman (2004) abstrai três fases genéricas para o desenvolvimento de software: *definição*, que identifica os requisitos do sistema e do software; *desenvolvimento*, que estabelece *como* tais requisitos serão alcançados; e *manutenção*, que se concentra nas modificações ocasionadas pela correção de defeitos existentes e por evoluções ambientais ou funcionais. A qualidade é uma questão inerente a todas as fases de desenvolvimento, em que várias atividades são realizadas para melhorá-la, tanto para o processo quanto para o produto desenvolvido.

Verificação e validação são duas atividades centrais de garantia de qualidade de software. A primeira tem como foco se o produto está sendo construído devidamente, observando se o processo de desenvolvimento está correto e de acordo com padrões pré-estabelecidos. A segunda, por sua vez, busca garantir que o produto correto está sendo desenvolvido, se o produto na sua forma corrente está em sintonia com os requisitos originais do usuário. A maior parte do esforço de validação tem sido realizada no final do período de desenvolvimento, quando o produto é testado; decide-se, então, se o produto está de acordo com os requisitos pré-estabelecidos (Vilela, 1998). Teste pode ser visto como a atividade final de verificação e validação dentro da organização de desenvolvimento de software (Jones, 1990).

1.1.1 Teste de Software

Segundo Myers (1979), teste é o processo de executar um programa ou sistema com a finalidade de encontrar defeitos que passaram despercebidos durante a fase de desenvolvimento,

e desempenha um papel importante na garantia de qualidade de software (Zoio, 2005). Teste de software é um método difundido, embora imperfeito, de garantia da qualidade de software (Ammann e Outt, 1994); imperfeito, pois, em geral, o teste não assegura que um software é livre de defeitos; podendo apenas constatar que ele possui defeitos. Quando defeitos são detectados durante a atividade de teste, diz-se que o teste foi bem sucedido (Myers, 1979). Testes devem ser conduzidos de forma rigorosa e sistemática para, caso não revelem defeitos, cresça a confiança do testador com relação à qualidade do software.

Teste é um dos processos mais caros no desenvolvimento e manutenção de produtos de software, com até 30% de recursos sendo alocados a este fim (Hartman, 2002). Esta constatação tem motivado a busca de abordagens sistemáticas de teste que revelem o maior número de defeitos a um baixo custo.

Diversas técnicas de teste têm sido propostas e estão classificadas em *estruturais*, *funcionais*, *baseadas em defeitos* e *baseadas em máquinas de estado finito*. A primeira requer que aspectos da estrutura do software implementado sejam exercitados durante o teste; a segunda estabelece requisitos de teste a partir da especificação funcional do software; a terceira utiliza informação de defeitos típicos para a geração de dados de teste; e a última utiliza a estrutura de máquinas de estado finito e o conhecimento subjacente para derivar requisitos de teste. Essas técnicas são vistas como complementares, pois cada uma delas trata o software sob uma perspectiva distinta; segundo Chaim (2001), em geral tais técnicas podem ser utilizadas tanto para o teste de unidade (teste de cada unidade isoladamente) como para o teste de integração (teste de interação entre as unidades que compõem o software).

Independentemente da técnica utilizada, a atividade de teste possui limitações que são consideradas em geral como problemas indecidíveis (Howden, 1975; Chaim, 2001; Maldonado, 1991; Maldonado e Fabbri; 2001): se existe um algoritmo de propósito geral que possa provar a corretude de um programa; se dois programas computam a mesma função; e se existem dados de teste que executam um específico caminho – seqüência de comandos – do programa (se um caminho é exercitável ou executável). Outras limitações são: correção coincidente (dois dados de teste que exercitam um requisito particular são aplicados a um programa, mas o resultado incorreto é obtido a partir de somente um deles); caminhos ausentes (a ausência de caminhos poderá omitir requisitos de teste estrutural, e, conseqüentemente, o defeito não será revelado); e impossibilidade de testar todas as entradas possíveis de um programa.

As técnicas de teste utilizam critérios que estabelecem requisitos que devem ser satisfeitos durante o teste; tais requisitos são conhecidos como *elementos requeridos*. De forma genérica, um *critério* é um crivo para se tomar decisão; define um conjunto de elementos que devem ser exercitados durante o teste; representam exigências para avaliar o teste. Satisfazer um critério significa exercitar todos os elementos requeridos pelo critério. *Casos de teste* são compostos de dados de entrada e especificação de saídas esperadas. Um critério de teste pode ser usado como *critério de adequação* ou como *critério de seleção*. No primeiro, objetiva-se avaliar um conjunto de casos de teste, sem perder de vista o objetivo de revelar defeitos, fornecendo medidas de cobertura para quantificar a atividade de teste; é um predicado que é usado para determinar se um programa foi suficientemente testado, podendo ser usado como condição de parada para o teste. O último é utilizado para auxiliar à geração de dados de teste, pela seleção de dados de entrada requeridos pelos critérios.

No contexto de teste estrutural, a estrutura interna de um programa é representada por um *grafo de fluxo de controle* (um grafo dirigido também denominado *grafo de programa*), com um único nó de entrada e um único nó de saída, onde cada nó representa uma seqüência de comandos executados em uma dada ordem, e cada arco denota uma transferência de controle entre nós; uma seqüência de nós é um *caminho*. Os critérios estruturais exploram a estrutura de fluxo de controle de programa. Os critérios estruturais mais conhecidos são:

- Critérios baseados em fluxo de controle: o critério *todos nós* requer que todos os comandos do programa sejam exercitados; o critério *todos arcos* requer que toda transferência de controle entre nós seja exercitada; o critério *todos caminhos* exige o exercício de todos os caminhos de um programa.
- Critérios baseados em fluxo de dados: exigem a execução de caminhos do ponto onde um valor foi atribuído a uma variável (definição de dados), até o ponto onde ela foi (potencialmente) utilizada; os critérios de Rapps e Weyuker (1985) e os Critérios Potencias Usos (Maldonado, 1991; Maldonado et al., 1992) são exemplos dessa classe de critérios.

Estudos sobre a complexidade e a relação de inclusão têm sido conduzidos para critérios de teste. A complexidade é definida pelo número de casos de teste necessário para satisfazer o critério no pior caso. Um critério C_1 *inclui* um critério C_2 se, para qualquer programa, todo conjunto de teste que satisfaz C_1 também satisfaz C_2; um critério C_1 *inclui* estritamente um

critério C_2 se C_1 inclui C_2 mas C_2 não inclui C_1 (Rapps e Weyuker, 1985; Frankl e Weyuker, 1988; Maldonado, 1991).

1.1.2 Bancos de Dados Ativos

Segundo Spoto et al. (2005), a manipulação de dados persistentes desempenha um importante papel em soluções implementadas por software. A persistência de dados é um atributo ligado à necessidade de dados não voláteis na execução de aplicações. Aplicações que manipulam dados persistentes são distintas das aplicações convencionais, pois incorporam tais dados ao espaço de entrada e de saída durante a sua execução.

Bancos de dados desempenham um papel de destaque nas operações da maioria das organizações modernas (Chays et al., 2000). Uma *aplicação de banco de dados* é um programa cujo ambiente sempre contém uma ou mais bases de dados (Kapfhammer and Soffa, 2003). A principal motivação deste trabalho é a melhoria de qualidade de aplicações baseadas em SQL. Uma *aplicação baseada em SQL* interage com bases de dados utilizando SQL (*Structured Query Language*), que é a linguagem mais difundida pela comunidade de banco de dados (Fortier, 1999; Elmasri and Navathe, 2003) e extensivamente usada em aplicações de banco de dados. Diferentemente das linguagens imperativas e orientadas a objetos, tais como *C* e *Java*, SQL tem recebido pouca atenção da comunidade acadêmica, apesar do grande montante de código fonte escrito nessa linguagem que necessita de métodos de teste sistemáticos. O uso de bancos de dados relacionais tem-se difundido em aplicações que manipulam dados persistentes e, neste contexto, SQL permanece a linguagem mais aceita e implementada por sistemas de bancos de dados relacionais (Daou et al., 2001).

Bancos de dados convencionais são *passivos*, pois executam apenas comandos de consulta ou transações de mudança de estado submetidos por um usuário ou um programa de aplicação. Para muitas aplicações, entretanto, é importante monitorar situações de interesse e disparar uma resposta quando tais situações ocorrerem. Por exemplo, um sistema de controle de inventário necessita monitorar a quantidade em estoque de itens da base de dados de inventário, de modo que quando a quantidade em estoque estiver abaixo de um dado limite, um processo de emissão de pedidos de compra seja iniciado. Essa conduta poderia ser implementada em um banco de dados passivo de duas maneiras, porém ambas insatisfatórias. Primeiro, a semântica do predicado

que monitora a quantidade em estoque poderia estar embutida em cada programa que atualiza a base de dados de inventário, mas essa é uma abordagem pobre da perspectiva de Engenharia de Software, pois eleva o custo de manutenção e potencializa a existência de versões distintas de uma mesma rotina. Alternativamente, um programa de aplicação pode ser escrito para examinar periodicamente as condições relevantes. Entretanto, se a freqüência de monitoração for muito alta o processo pode tornar-se ineficiente e, se a freqüência for muito baixa, as condições podem não ser detectadas em tempo oportuno.

Um sistema de banco de dados *ativo*, em contraste, é um sistema de banco de dados que monitora as situações de interesse e, quando elas ocorrem, dispara uma resposta apropriada em tempo adequado. A conduta desejada é expressa na forma de *regras ativas*, também denominadas *regras evento-condição-ação* (ou *regras ECA*), as quais são definidas e gravadas na base de dados. Esse enfoque traz o benefício de que as regras ativas podem ser compartilhadas por muitos programas de aplicação, e o sistema de banco de dados pode otimizar sua implementação. A integração do mecanismo de regras ativas a sistemas de banco de dados fornece um mecanismo centralizado, capaz de tratar situações de interesse em tempo oportuno à conduta reativa de acordo com a funcionalidade da aplicação.

O paradigma para regras ECA em sistemas de bancos de dados ativos segue a forma: ***on*** *event* ***if*** *condition* ***then*** *action*. Regras são disparadas pela ocorrência de eventos, tais como operações de mudança de estado da base de dados, ao invés de ciclos periódicos de avaliação de estado da base. Quando os eventos de disparo ocorrem, a condição é avaliada contra o estado da base de dados; se a condição é satisfeita, a ação é executada.

Uma visão geral dos domínios de aplicação para tais sistemas e de seus aspectos de implementação é explorada em (Ceri et al., 2000; Ceri e Widow, 1996; Embury e Gray; 1998). O uso de bancos de dados ativos é comumente classificado em *aplicações internas* e *aplicações externas*. Aplicações externas são responsáveis por tarefas de domínio específico (Chandra e Segev, 1994; Dittrich e Jonscher, 1994; Knolmayer et al., 1994), tais como: administração financeira, controle de inventário, projeto e gerenciamento de distribuição de energia e administração de carga de trabalho. Em aplicações internas, regras são usadas como se extensões da aplicação fizessem parte do sistema gerenciador de banco de dados. Algumas tarefas que podem ser delegadas a regras ativas são: *manutenção de integridade* (Ceri et al., 1994; Ceri e Widow, 1991), em que predicados de restrição de integridade representam transições não

permissíveis entre estados da base de dados, tal que a ação da regra pode cancelar a transação da aplicação ou reparar a consistência da base de dados; *manutenção de dados derivados e de visões materializadas* (Adelberg et al., 1996; Ceri e Widow, 1990; StoneBraker et al., 1990), pela monitoração de eventos de atualização na base de dados que resultem na ação de recomputar valores de dados derivados e de visões materializadas; *suporte à integração de dados* (Ceri e Widow, 1993; Madiraju e Sunderram, 2004), onde visões têm sido propostas como um recurso de integração de dados de fontes múltiplas, distribuídas e heterogêneas, tal como o contexto de *data warehousing* (Hammer et al., 1995); *gerenciamento de versão e replicação* (Kotz-Dittrich e Simon, 1998), onde, usando o mesmo princípio para manter visões, uma modificação de esquema replica definições de uma versão anterior para uma nova versão, propagando alterações ocorridas entre versões; *autorização de acesso a base de dados* (Dittrich e Jonscher, 1994; Jonscher, 1992), para prevenir acessos não autorizados a entidades da base de dados e para gravar *logs* com o registro de tentativas de violação de segurança.

Do ponto de vista da aplicação, parte da sua funcionalidade é especificada em regras, de modo que os fluxos de dados e de controle próprios das regras são abstraídos da aplicação. O controle é desviado dos programas de aplicação para o processamento de regras quando eventos ocorrem e é devolvido à aplicação quando a execução das regras é concluída. O controle oscila entre programas de aplicação e processamento de regras; regras são transparentemente executadas, sem intervenção do usuário.

1.2 Motivação

Iniciativas correntes de pesquisa na área de teste em geral não consideram o processamento existente em regras ativas, o que caracteriza uma carência de abordagens metódicas de teste na busca da melhoria de qualidade dessas aplicações.

Técnicas de teste para programas convencionais têm sido propostas, implementadas e avaliadas, mas relativamente pouco esforço tem sido dedicado ao desenvolvimento de técnicas sistemáticas na direção de corretude de aplicações de banco de dados (Chan e Cheung, 1999; Chays et al., 2000; Kapfhammer and Soffa, 2003; Zhang et al., 2001). As iniciativas mais formalizadas aos objetivos de teste surgiram nos últimos cinco anos, o que demonstra a atualidade de esforços sistemáticos em sintonia com o teste de software. Sobre regras ativas

escritas em SQL, Cardoso (2004) aborda o teste de aplicações de bancos de dados ativos, aplicando os critérios Potenciais Usos ao teste de regras individuais.

Segundo Vaduva (1999), as vantagens de bancos de dados ativos são contrapostas ao custo da complexidade resultante das interações entre execuções autônomas de regras ativas e manipulações de dados submetidas pelo usuário. Mesmo um pequeno número de regras ativas pode ser complexo para entender e gerenciar (Widom e Ceri; 1996). Um defeito está claramente presente se um programa não faz o que é suposto que ele faça, mas defeitos também estão presentes se um programa faz o que não se supõe que ele faça (Myers; 1979); nesse sentido, a interação entre regras pode ser fonte de comportamento incorreto e inesperado, caracterizando uma preocupação pertinente à especificação de casos de testes.

Vários aspectos influenciam a qualidade de um conjunto de regras (Vaduva, 1999): *(i)* interações de regras: regras interagem pelos disparos entre si e pela manipulação de objetos comuns da base de dados; *(ii)* modelo de execução: sistemas de banco de dados possuem políticas para o processamento de regras, tal como a ordem de execução de regras quando várias regras são disparadas: e *(iii)* redundância de regras: as regras de um conjunto podem embutir funcionalidade redundante, podendo em certos cenários ocasionar a execução duplicada da mesma tarefa.

O problema associado a esta pesquisa é a carência de técnicas de teste no contexto de regras ativas escritas em SQL; especificamente, a interação entre regras ativas é usualmente fonte de defeitos em aplicações de banco de dados ativos, sendo preteridas por desenvolvedores dessa classe de aplicações. Sabe-se que defeitos oriundos da integração entre unidades encontram-se nas interfaces dos módulos e em suas interações, e podem constituir até 40 % dos defeitos de um programa (Basili e Perricone, 1984; Harrold e Soffa, 1991), mas não se tem registro sobre defeitos na interação entre regras de um conjunto de regras ativas. Nesse contexto reside a motivação para a proposição de técnicas de teste baseadas na interação entre regras escritas em SQL, buscando revelar a presença de defeitos que porventura não tenham sido descobertos no teste de regras individuais.

1.3 Objetivos

Este trabalho explora a interação entre regras ativas, buscando a melhoria de qualidade de aplicações de bancos de dados ativos pela proposição e análise de critérios de teste, no contexto de regras escritas em SQL. Em especial, a base dos critérios é o conjunto das relações de fluxo de dados entre regras, motivadas pelo disparo entre regras e pela manipulação de entidades comuns da base de dados. Os seguintes objetivos são perseguidos:

- Estudar defeitos e falhas em comandos de manipulação da SQL. Determinar relacionamentos entre tipos de defeito e categorias de falha no contexto de comandos de consulta e de mudança de estado, buscando entender como defeitos propagam-se à saída de comandos.
- Investigar as interações entre regras baseadas no fluxo de dados persistentes entre regras. Caracterizar os vários tipos de interação entre regras ativas escritas em SQL e definir associações de fluxo de dados estabelecidas entre componentes de regras, não necessariamente distintas.
- Realçar a questão da precisão de fluxo de dados persistentes – granularidade – na cobertura de associações de interação. Ensaiar empiricamente a influência da precisão de fluxo de dados na descoberta de defeitos de manipulação ligados à interação entre regras.
- Propor e analisar critérios de teste baseados na interação entre regras, em um contexto complementar ao teste de regras isoladas. Investigar propriedades atribuídas à aplicação dos critérios, abstraindo sua complexidade, comparando-os com outros critérios de teste e exemplificando seu emprego no teste de um conjunto de regras escritas em SQL.
- Oferecer modelos para a implementação da abordagem proposta, visando à automação da aplicação dos critérios baseados na interação entre regras.
- Determinar se existem indícios empíricos de que os critérios propostos contribuem para a descoberta de defeitos de manipulação, bem como sua aplicabilidade no teste de aplicações de bancos de dados ativos.

Em síntese, a abordagem de teste proposta aplica técnicas de teste estrutural baseadas em fluxo de dados, no contexto de dados persistentes do modelo relacional de banco de dados, tendo em vista a descoberta da presença de defeitos a um custo aplicável, pela exploração da interação

entre regras ativas escritas em SQL, visando à melhoria de qualidade de aplicações de bancos de dados ativos.

1.4 Organização

Esta introdução discutiu o contexto no qual o trabalho se insere, a motivação e os objetivos da pesquisa.

No Capítulo 2 é apresentada uma revisão bibliográfica de teste de bancos de dados, incluindo bancos de dados ativos. São também abordados conceitos básicos de teste estrutural e são descritos termos ligados ao modelo relacional, SQL e aspectos inerentes a sistemas que suportam regras ativas.

O Capítulo 3 explora defeitos e falhas de manipulação. A estrutura de comandos SQL de manipulação é estudada a partir de suas construções básicas e uma lista de tipos de defeitos de manipulação é apresentada, utilizando exemplos de comandos corretos e defeituosos. Categorias de falha são estudadas para comandos de consulta e de mudança de estado, visando ao desenvolvimento de um mapeamento para analisar os relacionamentos entre tipos de defeitos e categorias de falha.

As interações entre regras ativas são tratadas no Capítulo 4, onde associações de fluxo de dados são definidas para a proposição de requisitos de teste. É especificado o modelo de interação entre regras ativas escritas em SQL que norteia a abordagem de teste proposta. São também definidos aspectos atribuídos à granularidade das relações de fluxo de dados e às relações de fluxo de dados ocasionadas pela presença de comandos de manipulação. Alguns conceitos são formalizados, incluindo: caso de teste, interface e estado de iniciação de regras ativas.

No Capítulo 5 é definido um conjunto de critérios de adequação – Critérios Baseados na Interação entre Regras – visando ao teste de regras ativas escritas em SQL. As propriedades dos critérios são analisadas, sob a ótica de aspectos ligados à precisão de fluxo de dados, à complexidade e à relação de inclusão. É ainda descrito um experimento que visa a investigar a eficácia dos critérios propostos e a influência da precisão de fluxo de dados na descoberta de defeitos de manipulação.

Modelos de implementação para os Critérios Baseados na Interação entre Regras são propostos no Capítulo 6. São explorados aspectos de fluxo de controle, fluxo de dados, instrumentação e avaliação; a estrutura de uma base de dados de teste é descrita.

No Capítulo 7 são apresentadas uma síntese do trabalho, suas contribuições e as perspectivas de trabalhos futuros.

São ainda incluídos apêndices que complementam o conteúdo da tese: código fonte instrumentado de um conjunto de regras ativas escritas em SQL; modelo conceitual simplificado da base de dados manipulada por esse conjunto de regras; descrição funcional de uma ferramenta, denominada ADAPT-TOOL, que visa a apoiar a aplicação da abordagem de teste proposta; esquema da base de dados de teste utilizado pela ferramenta; e dados da aplicação dos Critérios Baseados na Interação entre Regras.

Capítulo 2

Revisão Bibliográfica

Neste capítulo, as iniciativas de pesquisa em teste de banco de dados são analisadas e discutidas de acordo com seu foco: geração de dados de teste, teste de aplicações de bases de dados e teste de projeto de bases de dados. Inicialmente, é introduzida a terminologia de bancos de dados, mais especificamente os termos do modelo relacional. As técnicas de teste de software são revisadas, sobretudo o teste estrutural baseado em análise de fluxo de dados. Em seguida, é apresentado o suporte que descreve a essência da conduta de regras ativas, onde são abordados os seus modelos descritivos e são instanciados os padrões de programação de regras escritas em SQL. As pesquisas voltadas ao teste de regras ativas são então tratadas e ressaltados os pontos que norteiam cada referência; constata-se a atualidade de esforços de teste sistemáticos para bancos de dados e suas aplicações, bem como a carência de trabalhos que enfoquem a qualidade de regras escritas em SQL (Leitão et al., 2002).

2.1 Terminologia do Modelo Relacional

Alguns termos ligados ao escopo de bases de dados são realçados nesta seção. Um s*istema gerenciador de banco de dados* (DBMS – *Data Base Management System*) é um conjunto de programas que habilita usuários a criar e a manter bancos de dados. Um *banco de dados* é uma coleção de dados relacionados; traduz dados persistentes organizados (Elmasri e Navathe, 2003). Um ou mais bancos de dados em conjunto com o sistema gerenciador de banco de dados constituem um *sistema de banco de dados*; este termo também será utilizado com o

significado de sistema gerenciador de banco de dados. Uma *aplicação* ou *aplicação de banco de dados* consiste de um ou mais programas de aplicação que interagem com sistemas de banco de dados. *Sistemas gerenciadores de bancos de dados ativos* representam DBMSs que incluem a facilidade de conduta reativa a eventos ocorridos.

Bancos de dados relacionais são baseados no *modelo relacional* (Codd, 1970), o qual representa os dados como um conjunto de relações. *Relações* são freqüentemente pensadas como *tabelas*, onde cada linha representa dados sobre uma entidade particular e cada coluna representa um aspecto particular dos dados. Um *esquema de relação* $R(A_1, ..., A_n)$ é o nome de relação e uma lista de *atributos* (identificação de colunas), cada qual com um nome A_i e um domínio $dom(A_i)$. Os *domínios* são tipos de dados atômicos, tal como *integer* e *string*, antes que tipos estruturados, tal como *array*. Uma *relação* ou *estado de relação* do esquema R é um conjunto de *tuplas* (conjunto de linhas da tabela), tal que cada qual constitui um elemento do produto cartesiano $dom(A_1) \times ... \times dom(A_n)$. Um esquema de relação descreve a estrutura dos dados; uma relação descreve o estado dos dados em um particular momento. O esquema de relação, assim como o projeto da base de dados, tipicamente não é modificado com freqüência; o estado de relação é constantemente modificado para refletir mudanças nas entidades do mundo-real modeladas.

Um *esquema de um banco de dados relacional* é um conjunto de esquemas de relações, acrescido de um conjunto de *restrições de integridade*, as quais restringem os possíveis valores dos dados. Um *estado de banco de dados relacional* é um conjunto de estados de relações, tal que as restrições de integridade são satisfeitas. Exemplos de tipos de restrições são: *restrições de domínio*, que especificam possíveis valores de atributos; *restrições de unicidade*, que proíbem valores duplicados; *restrições de obrigatoriedade*, que requerem valores não nulos em atributos; *restrições de integridade referencial*, que impõem a atributos valores dentre os valores existentes em outros atributos, em relações não necessariamente distintas; *restrições de integridade semântica*, que determinam estados consistentes com regras de negócio.

A linguagem mais difundida para sistemas de bancos de dados relacionais é a SQL (*Structured Query Language*), que inclui recursos para a definição e para a manipulação de bases de dados relacionais. SQL possui linguagem de definição de dados (DDL – *Data Definition Language*) para descrição de bases de dados, incluindo restrições de integridade, e linguagem de manipulação de dados (DML – *Data Manipulation Language*) para mudança de estado e consulta

a base de dados. O Capítulo 3 apresenta a estrutura de comandos de manipulação de dados da SQL.

Padrões de programação têm sido desenvolvidos para a indústria de sistemas de banco de dados, pelo ANSI (*the American National Standards Institute*) e pela ISO (*the International Organization for Standardization*). A primeira versão do padrão SQL, denominado SQL-86 ou SQL1, foi publicada em 1986. Em 1989, uma versão revisada do padrão, intitulada SQL-89 ou SQL2, foi publicada; essa versão enriquece a versão SQL-86 com a noção de integridade referencial. A versão contemporânea do padrão, conhecida como SQL3, além de outras extensões significativas, introduziu a noção de ações referenciais, que podem ser consideradas como uma forma limitada de suporte de regra. Mais recentemente, a funcionalidade de banco de dados ativos foi incorporada ao padrão SQL3 na forma de *triggers* – nome dado a regras ativas escritas em SQL (Date e Darwen, 1997).

2.2 Teste de Software

Teste é o processo de executar um programa ou sistema com a finalidade de encontrar defeitos (Myers, 1979). Teste é uma atividade crítica de garantia de qualidade e representa a última revisão de especificação, projeto e codificação. Um teste é bem sucedido quando descobre um defeito ainda não revelado.

Pelo teste, pode-se concluir que o software tem defeitos, em decorrência da percepção de falhas manifestadas; não se pode garantir que ele não possui defeitos. Nesse sentido, é importante distinguir os termos *defeito*, *erro* e *falha*. *Defeito* é um elemento estático, inerente ao programa, ocasionado possivelmente por um engano do programador; é uma deficiência mecânica ou algorítmica que pode gerar um erro e eventualmente manifestar uma falha. *Erro* é um item de informação ou estado inconsistente do programa, resultante do exercício de um defeito. *Falha* é o evento notável (evento perceptível) em que um sistema em execução viola a sua especificação; se o evento não é perceptível, não caracteriza uma falha. Segundo Okun et al. (2002), uma *falha* representa um resultado inaceitável obtido a partir da execução utilizando alguns dados de entrada, ou seja, é uma conduta incorreta que é observável para alguns dados de teste (dados de entrada).

O *domínio de entrada* de um programa é o conjunto de todas as combinações dos valores de entrada. Um programa *P* está correto com respeito a uma especificação funcional *S* se, para qualquer item de dado pertencente ao domínio de entrada *D* do programa *P*, o comportamento do programa está de acordo com o especificado em *S* (Pressman, 2004). Um *caso de teste* é composto por uma entrada e por um resultado esperado. Em geral, deve-se projetar casos de teste que tenham a maior probabilidade de encontrar a maioria dos defeitos com o mínimo de tempo e de esforço (Maldonado, 1991). O mecanismo pelo qual é tomada a decisão se a saída obtida corresponde à saída esperada é denominado *oráculo*; pode-se dizer que o teste é uma atividade dinâmica que expõe um programa à oportunidade de comparar sua conduta de execução a sua especificação funcional.

De forma genérica, um *critério de teste* é um crivo para se tomar decisões. Define um conjunto de elementos que devem ser exercitados durante a aplicação do teste, tendo em vista o objetivo de encontrar defeitos; são exigências que permitem avaliar o teste – *requisitos de teste*. *Critérios de seleção* são utilizados para selecionar os casos de teste e estão ligados à determinação dos dados de entrada que exercitarão os elementos requeridos pelo critério. *Critérios de adequação* são utilizados para avaliar um conjunto de casos de teste. Um critério *C* (critério de adequação) é uma função que, para um dado programa *P*, especificação *S* e conjunto de teste *T*, determina se *T* é *C-adequado* para *(P, S)*; $C: P \times S \times T \rightarrow \{ true, false\}$. Segundo Goodenough e Gerhart (1975), um critério *C* é um *critério válido* se, caso exista algum defeito no programa, existe um conjunto de casos de teste que satisfaz *C* e revela o defeito; um critério *C* é um *critério confiável* se todos os conjuntos de casos de teste que satisfazem *C* são equivalentes; ou seja, se um conjunto revela o defeito, todos revelam o defeito e, se um conjunto não revela um defeito, nenhum conjunto revela o defeito; um critério *C* é um *critério perfeito* se ele é válido e confiável, ou seja, prova que um programa está correto, o que em geral é impossível.

As técnicas sistemáticas de teste de software são classificadas em: *(i)* funcionais, que consideram a especificação funcional do software para derivar requisitos de teste (Howden, 1987; Ostrand e Balcer; 1988; Stocks e Carrington, 1996); *(ii)* baseadas em defeitos, que utilizam informação de defeitos típicos para derivar requisitos de teste (Delamaro et al., 2001; DeMillo et al., 1978); *(iii)* baseadas em máquinas de estado finito, que utilizam a estrutura de máquinas de estado finito e o conhecimento subjacente para derivar requisitos de teste (Maldonado e Fabbri, 2001); e *(iv)* estruturais, também denominadas de teste caixa-branca ou teste caixa-aberta, que se

baseiam no código fonte e sugerem que o testador possui a idéia clara da estrutura de programa. O restante desta seção é dedicado ao teste estrutural de software, sobretudo ao teste baseado em informação de fluxo de dados.

Antes de estudar o teste estrutural, é pertinente ressaltar que usualmente são utilizadas estratégias para a condução da atividade de teste, resultando em vários níveis de aplicação do teste. Os defeitos de lógica e de implementação são revelados no *teste de unidade*; uma *unidade*, ou módulo, representa a menor porção do código de um programa (por exemplo, uma função ou um procedimento). O *teste de integração* visa a identificar os defeitos que surgem quando as unidades são postas para trabalhar em conjunto, os quais não foram revelados durante o teste de unidade; é uma técnica sistemática para integrar os módulos componentes da estrutura do software. O *teste de sistema* considera os diversos elementos componentes do sistema (hardware, software, etc.), objetivando expor cenários funcionais e de desempenho que não estão de acordo com a especificação.

A terminologia apresentada para teste estrutural é baseada em Clarke et al. (1989). A estrutura de fluxo de controle de um módulo *M* (ou unidade *M*) é representada por um *grafo de fluxo de controle*, denotado por $G(M) = (N, E, n_{in}, n_{out})$, que é um grafo dirigido, onde: N é o conjunto de nós; $E \subseteq N \times N$ é o conjunto de arcos; n_{in} e n_{out} são os nós de entrada e de saída, respectivamente. Cada nó do grafo de programa está associado a um bloco de comandos do módulo, que são sempre executados conjuntamente; isto é, se um comando pertencente a um bloco é executado, todos os demais são também executados na ordem dada. Os arcos do grafo de programa representam possíveis transferências de controle entre os nós; o arco (m, n) determina que a execução (exercício) do nó m pode ser imediatamente seguida da execução (exercício) do nó n. Um *caminho* é a seqüência de nós $(n_i, ..., n_k, n_{k+1}, ..., n_j)$, $i \leq k < j$, tal que $(n_k, n_{k+1}) \in E$. Um *caminho completo* sempre inicia em n_{in} e termina em n_{out}.

Os critérios de teste baseados unicamente na estrutura de controle buscam garantir que elementos de fluxo de controle sejam exercitados. O critério *todos nós* requer que todos os comandos do programa sejam exercitados; o critério *todos arcos* requer que toda transferência de controle entre blocos de comandos seja exercitada; o critério *todos caminhos* exige o exercício de todos os caminhos completos de um programa (Howden, 1975; Ntafos, 1988; Woodward et al., 1980). O critério *todos caminhos* é geralmente impraticável, devido ao elevado (talvez infinito) número de caminhos em um grafo de programa. Um caminho é um *caminho simples* se todos os

nós que o compõem, exceto possivelmente o primeiro e o último, são distintos. Em um *caminho livre de laço*, todos os nós são distintos.

As ocorrências de variáveis em um programa podem ser caracterizadas como *definição*, *uso* e *indefinição*. Uma *definição de variável* existe quando um valor é armazenado em uma posição de memória (por exemplo, valor atribuído a variável). Um *uso de variável* ocorre quando um valor estiver sendo utilizado e este não se refere à definição de variável. Uma *indefinição de variável* existe quando não se pode ter acesso ao valor da variável ou sua localização deixa de estar definida na memória. Um uso pode ser um *c-uso* ou um *p-uso*: o primeiro afeta diretamente uma computação ou permite que o resultado anterior de uma computação seja observado; o segundo interfere diretamente no fluxo de controle do programa. Diz-se que um nó *i* possui uma *definição global* da variável *x* se ocorre uma definição de *x* no nó *i* e existe um *caminho livre de definição com respeito a* (c.r.a) *x* para algum nó ou algum arco que possui um uso da variável *x*. Um caminho $(i, n_1, ..., n_m, j)$, $m \geq 0$, que não possui definição da variável *x* nos nós $n_1, ..., n_m$ é dito *caminho livre de definição com respeito a (c.r.a) variável x* do nó *i* até o nó *j* e do nó *i* até o arco (n_m, j). Um caminho $(n_1, n_2, ..., n_j, n_k)$ 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 $(n_1, n_2, ..., n_j, n_k)$ é um caminho simples livre de definição c.r.a *x*; *(2)* ou (n_j, n_k) tem um um *p-uso* de *x*, $(n_1, n_2, ..., n_j, n_k)$ é um caminho livre de definição c.r.a *x* e $(n_1, n_2, ..., n_j)$ é um caminho livre de laço. Um caminho livre de definição $(n_1, n_2, ..., n_j, n_k)$ c.r.a variável *x*, onde o caminho $(n_1, n_2, ..., n_j)$ é um caminho livre de laço e n_1 possui uma definição de *x*, é denominado *potencial-du-caminho* c.r.a *x*.

Os critérios baseados em análise de fluxo de dados requerem que as interações que envolvem definições de variáveis em um programa e subseqüentes (potenciais) referências a essas definições sejam testadas (Herman, 1976; Laski e Korel, 1983; Maldonado et al., 1992; Rapps e Weyuker, 1985). Esses critérios baseiam-se, portanto, para a derivação de casos de teste, nas associações entre a definição de uma variável e os seus possíveis usos subseqüentes; com exceção dos critérios *Potenciais Usos*, eles requerem a ocorrência explícita de um uso de variável para caracterizar e requerer essa interação. Critérios baseados em análise de fluxo de dados são apoiados na intuição de que não se pode ter confiança de que uma variável tenha o valor correto associado em algum ponto no programa, se nenhum teste causa a execução de um caminho do ponto de atribuição ao ponto onde o valor da variável é posteriormente usado (Frankl e Weyuker, 1988). Nos critérios *Potenciais Usos*, os requisitos de teste devem exigir caminhos entre uma

definição e os pontos do programa onde o valor da definição pode ser utilizado; em tais pontos existem *potenciais usos* (Maldonado, 1991; Maldonado et al., 1992).

Abaixo, são descritos alguns critérios das famílias Fluxo de Dados (Rapps e Weyuker, 1985) e Potenciais Usos (Maldonado, 1991; Maldonado et al., 1992).

- **Critério *todos 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, 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 $t \in T$.
- **Critérios *todos du-caminhos***: requer que todos os *du-caminhos* atribuídos às *associações definição-uso* dos tipos *(i, j, x)* e *(i, (j, k), x)* sejam exercitados pelos casos de teste de um conjunto *T*.
- **Critério *todos potenciais-usos***: 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*. A caracterização de uma associação definição-potencial-uso não necessita de um uso explícito de *x* no nó *j* ou no arco *(j, k)*, apenas que *j* ou *(j, k)* seja alcançável por um caminho livre de definição c.r.a *x* a partir de *i*.
- **Critério *todos potenciais-uso/du***: requer que todas as *associações definição-potencial-uso* dos tipos *(i, j, x)* e *(i, (j, k), x)* sejam exercitadas por pelo menos um *potencial-du-caminho* c.r.a *x*.

Os critérios acima definidos consideram o exercício de associações de fluxo de dados a partir da execução de caminhos. A presença de variáveis estruturadas e ponteiros motivou extensões à noção de cobertura de critérios de fluxo de dados (Ostrand e Weyuker, 1991; Vilela et al., 1997). Nesse contexto, um conjunto de teste que executa todos os caminhos de um programa não necessariamente satisfaz critérios baseados em fluxo de dados: o exercício de caminhos não é suficiente para garantir a cobertura de associações de fluxo de dados, pois estas são sensíveis aos dados definidos e aos dados usados em cada ponto do programa. Este aspecto será explorado no Capítulo 5 para *variáveis persistentes* – variáveis ligadas a entidades da base de dados, tal como uma relação – no contexto de cobertura de associações de fluxo de dados persistentes.

Uma propriedade pertinente a critérios de teste refere-se à *aplicabilidade* (Frankl e Weyuker, 1986). Diz-se que um critério *C* satisfaz a propriedade aplicabilidade se e somente se para todo programa *P* existe algum conjunto de teste que é *C-adequado* para *P*; este conceito é motivado pela presença de *caminhos não-executáveis*. Um caminho é *não-executável* (ou *não-exercitável*) se não existe um conjunto de dados de teste que provoque sua execução. Para os critérios baseados em fluxo de controle e em fluxo de dados, é indecidível se existe um conjunto de teste que adequadamente teste um dado programa. Os critérios acima podem ser redefinidos para incluírem somente requisitos de teste executáveis (Frankl e Weyuker, 1988), pela eliminação dos requisitos *não-executáveis*; os elementos requeridos pelos critérios baseados em fluxo de dados redefinidos são (potenciais) associações dos tipos *(i, j, x)* e *(i, (j, k), x) executáveis* e (potenciais) *du-caminhos executáveis*.

2.3 Bancos de Dados Ativos

Conforme foi comentado no Capítulo 1, sistemas gerenciadores de bancos de dados ativos são hábeis para monitorar e para reagir a circunstâncias específicas de relevância para uma aplicação. O uso de bancos de dados ativos é comumente classificado em *aplicações internas* e *aplicações externas*. Aplicações externas são responsáveis por tarefas de domínio específico (Chandra e Segev, 1994; Dittrich e Jonscher, 1994; Knolmayer et al., 1994), tais como: administração financeira, controle de inventário, projeto e gerenciamento de distribuição de energia e administração de carga de trabalho. Em aplicações internas, regras ativas são usadas como se extensões da aplicação fizessem parte do sistema gerenciador de banco de dados. Algumas tarefas que podem ser delegadas a regras ativas são: *manutenção de integridade* (Ceri et al., 1994; Ceri e Widow, 1991), onde predicados de restrição de integridade representam transições não permissíveis entre estados da base de dados, tal que a ação da regra pode cancelar a transação da aplicação ou reparar a consistência da base de dados; *manutenção de dados derivados e de visões materializadas* (Adelberg et al., 1996; Ceri e Widow, 1990; StoneBraker et al., 1990), pela monitoração de eventos de atualizações na base de dados que resultem na ação de recomputar valores de dados derivados e de visões materializadas; *suporte à integração de dados* (Ceri e Widow, 1993; Madiraju e Sunderram, 2004), onde visões têm sido propostas como um

recurso de integração de dados de fontes múltiplas, distribuídas e heterogêneas, tal como o contexto de *data warehousing* (Hammer et al., 1995); *gerenciamento de versão e replicação* (Kotz-Dittrich e Simon, 1998), onde, usando o mesmo princípio para manter visões, uma modificação de esquema replica definições de uma versão anterior para uma nova versão, propagando alterações ocorridas entre versões; *autorização de acesso a base de dados* (Dittrich e Jonscher, 1994; Jonscher, 1992), para prevenir acessos não autorizados a entidades da base de dados e para gravar *logs* de tentativas de violação de segurança.

Embora o benefício de bancos de dados ativos seja reconhecido desde algum tempo atrás, o campo de pesquisa é relativamente recente (Dayal, 1988). Alguns sistemas gerenciadores de bancos de dados comerciais incorporam a facilidade de conduta reativa a eventos ocorridos (DB2, 2005; Informix, 2005; Oracle, 2005; SQL Server, 2005); padrões de programação incluem extensões a SQL voltadas ao uso de regras ativas (Bowman et al., 1997; Date e Darwen; 1997; Fortier, 1999; Kulkarni et al., 1998; Melton e Simon, 1993); regras ativas têm motivado a comunidade acadêmica à construção de protótipos de pesquisa (Ceri et al., 1996; Hanson, 1996; Stonebraker e Kemnitz, 1991; Widow, 1996).

Um sistema de banco de dados ativo é descrito por um *modelo de conhecimento* e por um *modelo de execução* (Paton e Díaz, 1999; Widom e Ceri, 1996). O primeiro modelo descreve a funcionalidade ativa, apresentando um conjunto de propriedades ligadas aos componentes de uma regra ativa; o segundo descreve o comportamento de um conjunto de regras em tempo de execução. Cada sistema de banco de dados ativo, protótipo ou produto, adota um subconjunto dos modelos de descrição (Chakravarthy, 1993). Algumas das propriedades de ambos os modelos são descritas abaixo; em seguida é determinado o suporte existente no padrão SQL3 a regras ativas; por fim, um exemplo de uma regra ativa escrita em SQL é coletado da literatura.

O modelo do conhecimento define propriedades para o evento, a condição e a ação de regra ativa. Para o evento de regra, algumas das propriedades deste modelo são: *fonte do evento*, que determina a natureza e o caminho no qual o evento pode ser detectado (operações na estrutura da base de dados: inserir uma *tupla*; ocorrências de comandos de transação: *rollback* e *commit*; exceções produzidas: tentativa de acesso sem autorização apropriada; e periodicamente ou em algum ponto no tempo); *tipo do evento* (primitivo: provocado por uma ocorrência simples da fonte do evento; e composto: provocado por combinações de eventos primitivos); e *obrigatoriedade do evento* (nenhum, opcional e obrigatório). Para a condição de regra, são

definidas as propriedades: *obrigatoriedade da condição* (nenhuma, opcional e mandatória); e *contexto da condição*, que indica o estado da base de dados em que a condição é avaliada (início da transação vigente, cenário em que ocorreu o evento; momento em que a condição é avaliada). A ação de regra possui as propriedades: *opção da ação*, que define a tarefa da regra (operações na estrutura da base de dados, tal como atualizar a estrutura do banco de dados; informar alguma situação de alerta; realizar um papel alternativo da ação – *do instead*; efetuar alguma chamada externa; cancelar uma transação); e *contexto da ação*, que indica o estado da base de dados disponível para a ação de regra.

Algumas das propriedades do modelo de execução são: *acoplamento evento-condição*, que determina quando a condição é avaliada em relação ao evento que disparou a regra (imediato: a condição é avaliada imediatamente após o evento; adiado: a condição é avaliada na mesma transação do evento; isolado: a condição é avaliada em uma transação distinta do evento); *acoplamento condição-ação*, que indica quando a ação será executada em relação à avaliação da condição (similar ao acoplamento evento-condição: imediato, adiado e isolado); *granularidade de transição*, que sinaliza o relacionamento entre eventos e as regras disparadas (*tupla*: se uma ocorrência de evento dispara uma regra; conjunto: se um conjunto de ocorrências de evento dispara uma regra); *política de ciclo*, que especifica a conduta quando eventos são provocados pela apreciação da condição ou pela execução da ação de regra (iterativa: não suspende a condição ou a ação da regra; recursiva: suspende a condição ou ação da regra e o controle é desviado para a regra disparada); e *prioridade*, que determina a próxima regra a ser disparada quando o evento de várias regras foi provocado (numérica: cada regra possui um valor de prioridade absoluto; relativo; explicita prioridades entre regras; dinâmica: baseado nas regras mais recentemente disparadas; nenhuma prioridade).

A noção de regras ativas é uma das principais extensões introduzidas no padrão SQL3. Uma regra ativa em SQL3, denominada *trigger*, é ativada por uma transição de estado na base de dados. A propriedade *fonte do evento* do modelo de conhecimento limita-se a operações na estrutura da base de dados, onde o evento consiste em mudanças de estado de uma relação particular, na forma de *tipo de evento* primitivo. A condição de regra pode ser constituída por qualquer predicado SQL arbitrário, incluindo subconsultas e funções definidas pelo usuário (Kulkarni et al., 1998). Baralis et al. (1998) apresentam o *modelo evento-condição-ação* em sintonia com a noção de *triggers*: o modelo ECA para regras ativas é uma tripla de componentes:

(i) o *conjunto de eventos* é o conjunto de operações de manipulação de dados sendo monitoradas; *(ii)* a condição é um predicado que referencia o estado corrente da base de dados e os *valores de transição* da regra; e *(iii)* a *ação* é uma seqüência de operações de manipulação de dados. Os *valores de transição* associados a uma dada execução de uma regra ativa são dados transientes que estão sendo inseridos, modificados ou excluídos por uma operação monitorada pela regra. A Figura 2.1 apresenta a sintaxe SQL3 para a criação de uma *trigger*, conforme (Kulkarni et al., 1998).

Sobre a Figura 2.1, vale comentar algumas cláusulas e produções em relação às propriedades dos modelos de conhecimento e de execução. A produção *<trigger action time>* influencia as propriedades *contexto da condição* e *contexto da ação*, pois determina se a regra será disparada antes ou após a aplicação da operação de disparo na base de dados. A produção *<trigger event>* define os tipos de operação de mudança de estado que provocam o disparo da regra. A cláusula *FOR EACH { ROW | STATEMENT }* determina a propriedade *granularidade de transição*. A produção *<search condition>* não inclui operações de mudança de estado, diferentemente da produção *<triggered SQL statement>*; nesse sentido, o disparo entre *triggers*, não necessariamente distintas, pode ocorrer somente a partir da ação de regra, não podendo a consideração da condição de regra provocar eventos de disparo. O disparo entre regras em conjunto com a política de ciclo dificulta o entendimento da interação entre regras e a previsão do estado final após sua execução (Ramakrishnan, 1998). Segundo Kulkarni et al. (1998), SQL3 define como imediatos os acoplamentos *evento-condição* e *condição-ação* e a propriedade *política de ciclo* é recursiva. Quando múltiplas regras são elegíveis para disparo, a ordem de execução é baseada na ordem ascendente do tempo de criação de *triggers*.

Sob a ótica dos componentes de regra, a notação r_i (e_i, c_i, a_i) indica que a regra r_i é descrita pelo evento e_i, pela condição c_i e pela ação a_i. No contexto de fluxo de controle, similarmente a programas convencionais, uma regra ativa *r* é representada por um grafo dirigido, dado por $G(r) = (N, A, e, x)$, onde: *N* representa o conjunto de nós da regra; *A* denota o conjunto de arcos; o nó de entrada (evento da regra) é identificado por *e*; e *x* representa o nó de saída; diz-se ainda que $N_a(r)$ é o conjunto de nós que representam a ação da regra *r* (Leitão et al.; 2002). A Figura 2.2 mostra um exemplo de regra ativa escrita em SQL; no início de cada linha está a indicação do número de nó correspondente no grafo de fluxo de controle, o qual é ilustrado na Figura 2.3. A apresentação dessas figuras visa a ilustrar a representação de regras ativas escritas

em SQL por grafos de fluxo de controle; não se tem a intenção de especificar neste ponto como esta representação é alcançada (a Subseção 6.1 especifica o modelo de fluxo de controle para regras escritas em SQL). Sobre a Figura 2.3, os nós 1, 2 e 7 representam, respectivamente, o evento da regra , a condição da regra e o nó de saída do grafo; os nós 3 a 7 representam a ação da regras; e as linhas tracejadas representam transferências de controle para o nó de saída do grafo ocasionadas por exceção na execução de comandos de manipulação da SQL.

```
<trigger definition> ::=
        CREATE TRIGGER <trigger name>
        <trigger action time>
        <trigger event> ON <table name>
        [ REFERENCING <old or new values alias list> ]
        <trigger action>

<trigger action time> ::= BEFORE | AFTER

<trigger event> ::= INSERT | DELETE | UPDATE [ OF <column name list> ]

<old or new values alias list> ::= <old or new values alias> ……

<old or new values alias> ::=
        OLD [AS] <identifier> |
        NEW [AS] <identifier |
        OLD_TABLE [AS] <identifier> |
        NEW_TABLE [AS] <identifier>

<trigger action> ::=
        [ FOR EACH { ROW | STATEMENT } ]
        [ <trigger condition> ]
        <triggered SQL statement>

<trigger condition> ::=
        WHEN <left paren> <search condition> <right paren>

<triggered SQL statement> ::=
        <SQL procedure statement> |
        BEGIN ATOMIC
        { <SQL procedure statement> <semicolon> } ……
        END
```

Figura 2.1 – Sintaxe do padrão SQL3 para a criação de *triggers* (Kulkarni et al., 1998).

```
*01*   CREATE TRIGGER Reorder
*01*   AFTER UPDATE OF PartOnHand ON Inventory
*02*   WHEN (:New.PartOnHand < :New.ReorderPoint)
*01*   FOR EACH ROW
*01*       DECLARE NUMBER X;
*01*       BEGIN
*03*          SELECT COUNT(*) INTO X
*03*          FROM PendingOrders
*03*          WHERE Part = :New.Part;
*04*          IF X = 0 THEN
*05*             INSERT INTO PendingOrders
*05*             VALUES (:New.Part, :New.OrderQuantity, SYSDATE);
*06*          END IF;
*07*       END;
```

Figura 2.2 – Exemplo de regra ativa escrita em SQL (Zaniolo et al., 1997).

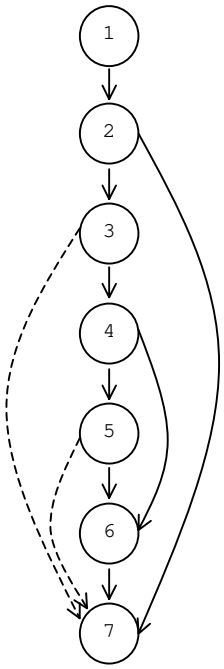


Figura 2.3 – Grafo de fluxo de controle obtido da regra da Figura 2.2.

2.4 Teste de Bancos de Dados

Pesquisas sobre teste de banco de dados podem ser classificadas em três categorias de teste (Leitão et al, 2002): *(i)* teste de aplicações de banco de dados; *(ii)* teste de projeto de banco de dados; *(iii)* e geração de bases de dados para teste. Na primeira, estão incluídos programas com SQL embutida e rotinas escritas em SQL que utilizam extensões de fluxo de controle, tal

como regras ativas; a segunda tem como foco avaliar o esquema da base de dados frente às especificações funcionais para a base de dados; e a terceira envolve a construção de bases de dados na forma de dados de teste para as categorias de teste anteriores.

Em adição, alguns trabalhos citados na literatura sobre teste de sistemas de banco de dados estão na linha de avaliação de desempenho. Nesse contexto, vários *benchmarks* têm sido desenvolvidos com o objetivo de avaliar sistemas gerenciadores de banco de dados (Carey et al., 1993; TPC, 2005). Na pesquisa descrita em (Slutz, 1998), que foi conduzida pela equipe de pesquisa da empresa *Microsoft*, comandos SQL válidos são gerados aleatoriamente e executados em vários sistemas que possuem bases de dados idênticas. O objetivo é comparar o desempenho de sistemas de gerenciamento de bancos de dados pela execução dos comandos SQL produzidos; neste trabalho, as saídas obtidas pelos comandos são também verificadas para cada sistema visando à apreciação relativa da presença de defeitos.

Alguns dos trabalhos explorados nesta seção não possuem foco claro na área de teste, pois seus autores não utilizaram a sistemática dessa área; havia a preocupação com a qualidade de bases de dados e de suas aplicações, mas os métodos de teste não foram aplicados. As iniciativas mais formalizadas aos objetivos de teste surgiram nos últimos cinco anos, o que demonstra a atualidade de esforços sistemáticos em sintonia com o teste de software. Existe consenso sobre a necessidade de abordagens sistemáticas de teste que explorem a estrutura e a semântica das operações de acesso a base de dados (Chan e Cheung, 1999; Chays et al., 2000; Kapfhammer e Soffa, 2003; Suárez-Cabal e Tuya, 2004; Zhang et al., 2001), aspecto este não coberto por abordagens de teste tradicionais.

A Tabela 2.1 sintetiza as pesquisas identificadas para teste de banco de dados, em relação às categorias acima mencionadas; os trabalhos específicos sobre o teste de regras ativas não estão incluídos nesta tabela e são apresentados na Seção 2.5. A primeira coluna lista os trabalhos relacionados ao teste de banco de dados; as colunas segunda, terceira e quarta indicam se um trabalho refere-se às categorias geração de bases de teste, teste de aplicações de banco de dados e teste de projeto de banco de dados, respectivamente; a última coluna comenta sucintamente a pesquisa. É natural que uma pesquisa seja enquadrada em mais de uma categoria, sobretudo quando se refere à geração de bases de teste, pois neste caso os dados de teste gerados objetivam alguma categoria de teste. Cada uma das pesquisas sumarizada na Tabela 2.1 é descrita a seguir.

Tabela 2.1 – Classificação de pesquisas relacionadas ao teste de banco de dados.

Trabalhos sobre teste de banco de dados	Ger.	Apl.	Proj.	Conteúdo sucinto
(Aranha, 2005)			X	Uma família de critérios objetiva apoiar a descoberta de defeitos na estrutura lógica de bases de dados relacionais; são propostos requisitos para o teste das restrições de relação e de relacionamento.
(Aranha et al., 2000)			X	Aborda uma ferramenta que automatiza os critérios propostos em (Aranha, 2005).
(Chan e Cheung, 1999)	X	X		Usa técnicas caixa-branca convencionais para a geração de dados de teste. Transforma comandos SQL em procedimentos de linguagem de propósito geral.
(Chays et al., 2000)	X	X		Com o auxílio de uma ferramenta, é aplicada uma técnica, similar ao particionamento por equivalência (Ostrand e Balcer, 1988), para a geração de bases de dados para o teste de aplicações que usam SQL na manipulação de dados.
(Chays et al., 2002) (Chays e Deng, 2003)	X	X		Apresentam uma ferramenta que implementa a abordagem de geração de dados de teste proposta por Chays et al. (2000).
(Daou et al., 2001)	X	X		Focando-se no teste de regressão, uma análise de impacto determina os componentes de software afetados por uma manutenção, visando a extrair de um conjunto os casos de teste que exercitam tais componentes; aplicam-se algoritmos para a redução de casos de teste.
(Davies et al., 2000)	X		X	É efetuada a geração de dados válidos e inválidos em relação às restrições de integridade de atributos individuais, sem considerar os relacionamentos entre entidades da base de dados.
Kapfhammer e Soffa (2003)		X		É proposta uma família de critérios de adequação, extensões ao critério *todos usos*, para as várias granularidades de análise de fluxo de dados persistentes; não é feita análise se os critérios propostos são reveladores de defeito.
(Leitão et al., 2005)		X		Investiga o relacionamento entre tipos de defeito e categorias de falha em comandos de manipulação escritos em SQL.
(Lyons, 1977)	X	X	X	Um compilador recebe especificações escritas em linguagem *english-like* e gera um programa escrito em linguagem PL/I para gerar arquivos de dados.
(Mannila e Räihä, 1989)	X	X		Constrói bases de teste para comandos de consulta individuais, pela geração de dados representativos das dependências funcionais obtidas das especificações das relações atribuídas a cada consulta.
(Noble, 1983)	X		X	Define uma sintaxe para a estrutura de uma base de dados e disponibiliza funções para a geração de dados apoiadas na estrutura definida; explora a dependência entre relações da base.
(Robbert e Maryanski, 1991)			X	Uma ferramenta realiza a geração de um plano de teste para o teste de restrições de um modelo E-R.
(Spoto et al., 2000) (Spoto, 2000) (Spoto et al;, 2005)		X		A pesquisa é pioneira na análise de relações de fluxo de dados persistentes pela manipulação via SQL. São propostos critérios de teste intra- e inter-modular para aplicações de banco de dados relacional.
(Suárez-Cabal e Tuya, 2004)		X		É definida uma medida de cobertura para comandos de consulta SQL (*select*), visando a usá-la na forma de critério de adequação; baseia-se nas condições existentes nos predicados para a seleção de *tuplas*, buscando cobrir os valores verdadeiro e falso em cada uma de suas diferentes combinações.
(Zhang et al., 2001)	X		X	A abordagem define uma propriedade que deve ser satisfeita pela execução de comandos SQL embutidos em programas de aplicação; a propriedade é escrita na forma de expressões, que devem ser usadas por solucionadores de restrições para a geração de instâncias de banco de dados.

Aranha (2005) introduz uma família de critérios de teste para a estrutura lógica de uma base de dados relacional. Os requisitos de teste são estabelecidos para o teste de relação e para o teste de relacionamento; o primeiro é o exercício de uma unidade da base de dados – uma relação – para detecção de defeitos na estrutura dos atributos e de suas definições; o segundo objetiva revelar problemas nos relacionamentos entre relações de uma base de dados, através do exercício de chaves. Os elementos explorados pelos critérios incluem: atributos sem restrições explícitas, ou seja, atributos sem restrições de integridade referencial, de domínio e de valores nulos; atributos com restrições explícitas; e atributos com restrições potenciais de integridade referencial, tais como chave primária potencial e chave estrangeira potencial. Os requisitos também exigem que determinados tipos de operação SQL sejam executados sobre os elementos da base; por exemplo, casos de teste são construídos para executar os comandos *select*, *update*, *insert* e *delete* sobre os componentes da base. Uma ferramenta foi construída para apoiar a aplicação dos critérios: *RDBTool – Relational DataBase Testing Tool* (Aranha et al., 2000). A ferramenta realiza análise estática no esquema da base de dados, para obter as definições e as declarações formais sobre os seus componentes descritivos; tais dados são então utilizados para uma avaliação estática da base de dados, produzindo um guia para o testador. Os elementos requeridos são determinados e os comandos SQL para testar cada elemento da base são construídos. A ferramenta também possui uma fase dinâmica, o teste aplicado é avaliado: determina a fatia do conjunto adequado que é reveladora de defeito.

Chan e Cheung (1999) propõem o mecanismo *WHODATE* (*WHite bOx Database Application TEsting*) voltado para a geração de casos de teste de aplicações de banco de dados com SQL embutida. A motivação dos autores é a constatação de que técnicas tradicionais não consideram a semântica de comandos SQL na geração de casos de teste, tratando-os como caixas-pretas. Comandos SQL são transformados em fragmentos de programação em linguagens de propósito geral (tais como C e C++), permitindo o uso de técnicas caixa-branca convencionais para a geração de casos de teste. Devido à grande variação entre conjuntos de comandos SQL suportados por diferentes sistemas de banco de dados, tais comandos são primeiramente convertidos em *RAEs* (*Relational Algebraic Expressions*) que, por sua vez, são compostas por *RAOs* (*Relational Algebraic Operations*), tais como seleção, projeção, união, diferença e produto cartesiano (em alguns casos não é possível transformar comandos SQL em *RAEs* usando *RAOs* fundamentais; por exemplo, devido ao uso da cláusula *order by* da SQL). As operações

relacionais em *RAEs* são então traduzidas para um conjunto de procedimentos em C++. Cada operação algébrica fundamental em *RAEs* é transformada em um procedimento que aceita uma ou mais relações como entrada e retorna uma nova relação (alguns procedimentos podem também receber outros parâmetros, tal como o predicado de seleção). O grafo de fluxo de cada comando SQL será composto dos grafos de fluxo de todos os procedimentos das *RAEs* (em teste caixa-branca convencional, cada comando SQL é tratado como um único nó na geração de casos de teste). Após *RAEs* terem sido transformadas, diferentes técnicas caixa-branca podem ser usadas para produzir casos de teste a partir da agregação dos procedimentos transformados com os comandos hospedeiros. É observado que, no caso de cobertura de comando e de desvio, a abordagem proposta gera mais ou menos o mesmo número de casos de teste da abordagem caixa-branca tradicional; na cobertura de caminho, a abordagem proposta gera mais casos de teste. Segundo os autores, os casos de teste gerados ajudam a revelar defeitos relacionados a estados particulares de banco de dados. Apesar de não ser explicitamente mencionado no artigo, conclui-se que o caminho executado de um procedimento resultante de um comando SQL é dependente do estado da base de dados. Os autores não utilizaram teste baseado em informação de fluxo de dados na avaliação da abordagem proposta.

Em (Chays et al., 2000) são realçados alguns aspectos pertinentes à atividade de teste no contexto de bancos de dados: objetos do teste – programa de aplicação, esquema, segurança, precisão de dados e sistema gerenciador; natureza de linguagens usadas em programas de aplicação; inclusão dos estados de banco de dados nos espaços de entrada e de saída para o teste de programas de aplicação; demanda por abordagens dedicadas ao teste de programas de aplicação; e dificuldades pertinentes ao uso de bases de dados reais durante a atividade de teste. O artigo explora a geração de bases de dados específicas, visando ao teste de aplicações de banco de dados que utilizam a SQL para a manipulação de dados, usando uma técnica baseada em especificação e similar ao particionamento por equivalência (Ostrand e Balcer, 1988). A abordagem proposta é a geração de dados especialmente para o propósito de teste de aplicações de banco de dados e a execução do teste em um ambiente isolado. Neste sentido, povoam-se as tabelas com dados válidos (dados que respeitam as restrições de integridade) e interessantes, segundo indicações do testador. O testador identifica importantes categorias de dados, fornece valores de amostra para cada categoria e particiona cada categoria em grupos. Uma ferramenta baseada em *Postgres* (Stonebraker e Rowe, 1986) constrói casos de teste pela combinação de

diferentes escolhas das várias categorias, gerando *tuplas* para as tabelas da base de dados. Para os dados gerados, são respeitadas as restrições de obrigatoriedade de valor (*not null*), de unicidade de valor (*unique*) e de integridade referencial, obtidas pela ferramenta a partir dos meta-dados da base de dados. Os autores buscam incorporar heurísticas para apoiar a geração de estados de banco de dados, com o objetivo de manter um número razoável de casos de teste e evitar a geração de combinações impossíveis. Do ponto de vista da aplicação, não são explorados critérios de adequação para a atividade de teste.

AGENDA – *A (test) GENerator for Database Applications* – é uma ferramenta voltada ao teste de aplicações de banco de dados (Chays et al., 2002; Chays e Deng, 2003), cuja geração de base de dados de teste é baseada nas idéias propostas em (Chays et al., 2000), acima mencionadas. O testador fornece valores representativos, organizados em grupos, para os atributos da base de dados. Guiada por heurísticas, a ferramenta utiliza tais grupos de valores para produzir *templates* de teste, isto é, casos de teste abstratos. As heurísticas são usadas para controlar a explosão do número de *templates* e para selecionar tipos particulares de *template*; por exemplo, explorar valores limite e forçar a inclusão de valores nulos e duplicados, de acordo com o esquema da base de dados. O testador então atribui saídas esperadas da aplicação para o teste de cada *template*. A ferramenta instancia os *templates* com valores de teste, executa casos de teste e auxilia o testador na verificação se as saídas e o novo estado da base de dados estão consistentes com a saída esperada.

Daou et al. (2001) propõem uma abordagem para o teste de regressão de aplicações de banco de dados. Inicialmente, é identificado o tipo da modificação feita: alterações realizadas nos módulos de banco de dados – modificações de programas que acessam o banco de dados – e alterações nos componentes de banco de dados – modificações na estrutura de tabelas do banco de dados. Em seguida é realizada uma análise de impacto da modificação, para identificar os componentes afetados e para selecionar o conjunto preliminar de casos de teste que exercitam os componentes afetados. São então aplicados dois algoritmos para a redução de casos de teste: *graph walk technique*, que se baseia no fluxo de controle e utiliza a dependência existente entre comandos e os vários componentes de banco de dados; e *call graph firewall*, que explora as dependências de fluxo de dados inter-procedimental e adota o nível coluna (atributo de tabela) para a granularidade da análise de fluxo de dados (Leung e White, 1990). Uma ferramenta foi construída para apoiar a abordagem proposta e utilizada em um experimento que utilizou a

linguagem PL/SQL do sistema *Oracle*. Os resultados demonstraram que a taxa de redução de casos de teste está ligada ao número de módulos afetados devido a cada modificação e à distribuição de casos de teste dentro dos módulos; o número de módulos afetados por modificação depende do nível de interação entre os módulos e dos vários componentes de banco de dados.

Uma abordagem para geração de bases de dados é desenvolvida em (Davies et al., 2000). A idéia é a produção de arquivos texto com uma combinação de valores válidos e inválidos do ponto de vista das restrições da base de dados. Esse enfoque é justificado pelos autores como um mecanismo para o teste das restrições de integridade de dados definidas no esquema de banco de dados durante a geração automatizada de dados. A abordagem é simples: a estrutura das tabelas é lida para realizar a análise sintática das regras de validação de atributos individuais, tais como lista de valores, faixa de valores e padrão de caracteres; são então produzidos arquivos com dados que satisfazem e que não satisfazem as restrições da base de dados; os arquivos são utilizados para povoar a base de dados; falhas ocorridas durante a carga da base de dados são capturadas e gravadas em um arquivo *log*, que será posteriormente analisado contra o projeto original. As idéias foram inicialmente aplicadas em um protótipo desenvolvido para uma base de dados do sistema *Microsoft Access*. Até a data de publicação do artigo, o protótipo tratava isoladamente cada tabela da base de dados, onde nenhuma integridade referencial era checada. Os autores não esclarecem os critérios de parada para o teste de esquema da base de dados.

Kapfhammer e Soffa (2003) propuseram uma família de critérios de adequação para o teste intraprocedimental de aplicações de banco de dados, baseando-se na informação de fluxo dos dados persistentes manipuláveis pelas aplicações. Especificamente, os critérios propostos são extensões ao critério *todos usos*, focados exclusivamente nas relações de fluxo de dados atribuídas às entidades de banco de dados relacional, explorando níveis de precisão – granularidade – para a análise de fluxo de dados. São propostos critérios para as granularidades: *banco de dados*, *relação*, *atributo*, *registro (tupla)* e *valor de atributo*. A partir da análise estática do código fonte de uma aplicação, as associações de fluxo de dados persistentes são abstraídas para os níveis *banco de dados*, *relação* e *atributo*. Para os níveis *tupla* e *valor de atributo*, é estabelecida uma abordagem conservadora para enumerar o conjunto de entidades que podem ser definidas e usadas em cada ponto de interação com o banco de dados; pode-se observar, neste caso, que o elenco de associações de fluxo de dados abstraído é dependente do estado da base de

dados usada como entrada em cada caso de teste. Foram realizados dois experimentos em aplicações escritas em linguagem *Java*, que utilizaram a SQL para a manipulação de dados persistentes. Tais estudos empíricos revelaram que, mesmo para um número reduzido de *tuplas*, o número de elementos requeridos pelos critérios tende a ser elevado para os níveis *tupla* e *valor de atributo*; também, apesar de alguns desses elementos já serem exercitados no teste de adequação do critério *todos usos* focado somente nas variáveis de programa, algumas dessas associações de fluxo de dados persistentes requerem casos de testes adicionais. Adicionalmente, o artigo é pioneiro ao abordar as ocorrências de definição de entidades de banco de dados sem subseqüentes ocorrências de uso – a presença da definição de dados persistente não implica que deva existir uma contrapartida de uso dentro da aplicação; no estudo empírico, observou-se um elevado número dessas definições. O trabalho não menciona estudos para determinar se os critérios propostos são reveladores de defeito.

No contexto de aplicações que utilizam a SQL, Leitão et al. (2005) discutem aspectos sobre falhas em comandos de manipulação e apresentam os resultados de uma investigação direcionada ao entendimento sobre o relacionamento entre tipos de defeito e categorias de falha. Uma lista de tipos de defeito é apresentada, organizada a partir da estrutura de comandos de consulta (*select*) e de mudança de estado (*insert, delete* e *update*). O conceito de categoria de falha de manipulação é examinado e é caracterizada a noção de saída da execução de operações de manipulação. Um experimento é conduzido para entender como defeitos de manipulação são propagados à saída de comandos. O conteúdo deste artigo é discutido em detalhes no Capítulo 3.

Lyons (1977) desenvolveu um recurso para apoiar a geração automática de dados, denominado *ADG* (*Automatic Data Generating*). *ADG* é um compilador que recebe como entrada uma especificação que descreve as características do arquivo de dados a ser gerado. A saída é um programa escrito em linguagem *PL/I* que, quando executado, criará um arquivo com as características desejadas. A linguagem fonte permite especificar o nome do arquivo, a quantidade de registros do arquivo, o nome de um campo (atributo), o tipo de campo (tipo de dado), o tamanho de um campo, o conjunto de valores possíveis para um campo, uma função de geração de valores para campos e algumas fórmulas para lei de formação de valores de um campo. O artigo apresenta um exemplo para a geração de um arquivo, com alguns dados com defeito em relação a regras pré-estabelecidas. Esse é um aspecto interessante do uso desse gerador, pois permite a geração de valores válidos e inválidos. O autor ressalta algumas aplicações para o

compilador *ADG*: em ambientes educacionais, habilitar a manipulação de grandes arquivos de dados e a simulação de problemas; evitar o uso de dados reais por questões de segurança e de privacidade; permitir a criação de situações problema no arquivo; fornecer dados de auditoria para a comparação de sistemas. O enfoque do trabalho segue um paradigma anterior à tecnologia de banco de dados, pois a definição de dados (meta-dados) está presente nos programas de aplicação e não no banco de dados propriamente dito.

Mannila e Räihä (1989) descrevem um mecanismo para povoar um banco de dados, de modo que um conjunto pré-definido de dependências funcionais seja satisfeito. Dada uma consulta (*query*) à base de dados, são abstraídas as especificações do conjunto de relações atribuídas à consulta; o projetista da base de dados identifica dependências funcionais dessas relações, utilizando-as para criar os dados das relações, de modo que sejam representativos das dependências existentes. O método objetiva gerar dados de teste para cada comando individual de consulta a base de dados. Os dados de teste podem ser extraídos de bases pré-existentes, reduzindo-as aos dados suficientes às dependências envolvidas. É utilizado um método para a produção de bases de dados reduzidas, visando a operações *select-project-join* onde cada esquema de relação aparece uma única vez na consulta (Mannila e Räihä; 1985). Apesar do enfoque ser o teste de consultas a base de dados, o artigo enfatiza que as relações resultantes podem expor imperfeições no projeto da base de dados, tais como dependências funcionais defeituosas, ausentes ou indesejáveis. Não são explorados aspectos de fluxo de controle e de dados da aplicação.

Noble (1983) concebeu uma abordagem para a geração de dados para um banco de dados relacional. Para tal, foram desenvolvidos recursos sintáticos que combinam aspectos ligados a restrições da base de dados e a funções dirigidas ao domínio de valores para a geração de dados. É implementado um sistema, denominado *QIKSYS*, capaz de entender construções de uma linguagem para estabelecer a definição de atributos, *tuplas*, relações, dependência entre relações e outras restrições típicas em bancos de dados relacionais; também é possível estabelecer listas de valores e faixas de valores permitidas a atributos, em adição ao tipo de dado de cada atributo. Após definir a estrutura dos dados, os arquivos de dados atribuídos às relações da base de dados são gerados pelo uso de funções dedicadas, tais como *GENREL* e *GENLINK*, que geram um dado número de *tuplas* para uma relação e *tuplas* para as relações envolvidas em uma particular dependência, respectivamente. O autor ressalta a seqüência de geração de *tuplas* envolvidas em

uma relação de dependência; as relações são classificadas, segundo aspectos de dependência: relações base, que definem alguma restrição com outras relações dependentes; relações dependentes, que possuem restrição existencial com outras relações; e relações derivadas, que são criadas a partir de outras relações por meio de operações da álgebra relacional. O que se observa é um esforço razoável para conhecer e escrever toda a estrutura de uma base de dados, que constitui cópia do esquema da base acrescido de aspectos do domínio de valores para geração de dados. Contudo, é possível gerar dados para um subconjunto das relações da base de dados, reduzindo o esforço para escrever a definição de dados.

Em (Robbert e Maryanski, 1991), os autores situam o trabalho no teste de aplicações de banco de dados, mas o que realmente se observa é a elaboração de um *plano de teste*, termo utilizado pelos autores, baseando-se no modelo conceitual da base de dados. O modelo de teste é baseado no paradigma *entidade-relacionamento (E-R)* proposto por Chen (1976), mas, segundo os autores, pode ser aplicado a outros paradigmas. Casos de teste podem ser projetados para testar as construções do modelo *E-R* e demais restrições do banco de dados; teste exaustivo requer o exercício de todas as construções deste modelo. Um plano de teste é elaborado por um programa, a partir do dicionário e de outras definições do banco de dados. O plano não é um documento genérico, mas um procedimento detalhado usando termos definidos no esquema do banco de dados em teste. Este plano orienta o testador sobre condições excepcionais, extremas, valores de fronteira, valores válidos, valores inválidos, entidades e relacionamentos de um banco de dados. Na abordagem sugerida, o gerador de plano de teste aponta os testes que devem ser conduzidos. Os testadores executam cada teste indicado e gravam os resultados em um arquivo de *log* ou relatório de teste. A equipe de projeto deve definir a validade dos resultados para, por exemplo, determinar se valores válidos estão sendo aceitos e valores inválidos estão sendo rejeitados. A inclusão de um gerador facilita a obtenção de planos de teste atualizados, a cada manutenção da estrutura do banco de dados. É sugerida a aplicação da abordagem proposta ainda na fase de projeto do banco de dados. Não são fornecidos esclarecimentos sobre a seleção de casos de teste e sobre a determinação do ponto de parada de teste; a descrição da proposta é abstrata. A definição do critério de teste utilizado também não é explícita.

A análise de relações de fluxo de dados persistentes pela manipulação de dados via SQL foi inicialmente abordada em (Spoto et al., 2000; Spoto, 2000; Spoto et al., 2005). O trabalho é focado no teste estrutural de programas de aplicação de banco de dados relacional com SQL

embutida. Dois modelos de fluxo de dados são propostos: intra-modular e inter-modular, onde um módulo é encarado como um programa de aplicação. O modelo de fluxo de dados intra-modular aplica-se ao teste de unidade e ao teste de integração das unidades de um programa. O inter-modular destina-se à integração dos programas que compõem uma aplicação. Duas abordagens são propostas para o teste de integração intra-modular: teste baseado no grafo de chamadas e teste baseado na dependência de dados. Critérios de teste baseados nos modelos intra-modular e inter-modular são apresentados e discutidos com exemplos de sua aplicação. A abordagem para o teste de integração de unidades baseada no grafo de chamadas é inspirada na pesquisa de Vilela (1998), que abstrai o caminho de integração entre duas unidades de programa, o qual é composto por subcaminhos concatenados da unidade chamadora e da unidade chamada. A essência do trabalho situa-se na extensão de abordagens de teste de fluxo de dados, dedicadas a programas convencionais, a programas de aplicação de banco de dados relacional. Os comandos SQL para manipulação de dados foram estudados e os tipos de ocorrência de variáveis para dados persistentes foram abstraídos. Os resultados da aplicação dos critérios em programas reais indicam que demandam um número pequeno de casos de teste, apesar de sua complexidade de ordem exponencial.

Suárez-Cabal e Tuya (2004) definem uma medida de cobertura para comandos de consulta SQL (*select*), visando a usá-la na forma de critério de adequação para o teste de aplicações que acessam bases de dados. A abordagem baseia-se nas condições presentes nos predicados para a seleção de *tuplas*, buscando cobrir os valores verdadeiro e falso em cada uma de suas diferentes combinações. Qualquer consulta na aplicação é primeiramente normalizada, seguindo uma construção sintática segundo o padrão SQL3, conforme notação a *BNF* (*Backus Naur Form*) adotada; o resultado é a localização na cláusula *from* de predicados atribuídos à operação de junção e na cláusula *where* são postos os demais predicados de seleção de *tuplas*. É proposta a construção de uma árvore de cobertura visando a auxiliar a avaliação de condições; cada nível da árvore representa uma condição da consulta, iniciando pela cláusula *from* (quando existem operações de junção), seguindo com as condições da cláusula *where*, na mesma ordem em que estão postas na consulta. São considerados os cenários de avaliação de condições da esquerda para a direita e da direita para a esquerda, e de ocorrência de valores nulos para cada operador de uma condição. Segundo os autores, a árvore de cobertura é um recurso poderoso para a inserção de *tuplas* quando se objetiva alcançar uma dada cobertura, e para reduzir o tamanho de

uma base existente conservando a mesma medida de cobertura da base de dados original. O artigo menciona a existência de situações impossíveis na árvore de cobertura devido a restrições no esquema da base de dados, as quais impactam o cálculo de cobertura, mas não comentam estratégias para identificá-las. A pesquisa não explora quaisquer fluxos de controle e de dados na aplicação. Uma característica que limita a abordagem é a ocorrência de comandos de mudança de estado na aplicação, pois os comandos de consulta foram tratados de forma isolada; a base de dados que alcança cada comando de consulta pode ser distinta daquela utilizada como dado de teste. Os autores mencionam a necessidade de se reduzir o número de nós da árvore de cobertura, pois é comum a utilização de predicados de seleção de *tuplas* compostos por um número elevado de condições.

Zhang et al. (2001) estudam a geração de instâncias de banco de dados, a partir da semântica de comandos SQL embutidos em programas de aplicação de banco de dados. O artigo descreve uma ferramenta de suporte que gera um conjunto de restrições, que representam, coletivamente, uma propriedade contra a qual o programa deve ser testado. Instâncias de banco de dados para o teste de programas podem ser derivadas pela resolução do conjunto de restrições gerado pela ferramenta, usando solucionadores de restrição. Para efeito de geração de dados de teste, as restrições representam fórmulas que descrevem o relacionamento entre a instância de banco de dados de entrada e a tabela resultante. Exemplos de propriedades para um determinado comando SQL de consulta a base de dados são: o comando não retorna qualquer linha e o comando produz valores numéricos negativos. Em outras palavras, a entrada da ferramenta é composta por um comando SQL, pela definição de esquema de banco de dados e por uma propriedade que representa o requisito do testador. A saída é um conjunto de restrições que podem ser fornecidas a solucionadores de restrição; os autores usaram a *BoNuS* (Zhang, 2000), uma ferramenta de propósito geral para a solução de restrições envolvendo variáveis de diferentes tipos. Se tais restrições puderem ser satisfeitas, obtêm-se as instâncias desejadas de banco de dados, as quais devem estar consistentes com as restrições do esquema da base de dados.

2.5 Teste de Regras Ativas

Foram encontradas na literatura poucas iniciativas de pesquisas voltadas ao teste de regras ativas. O único trabalho focado na sistemática de teste aplicada a regras escrita em SQL é (Cardoso, 2004), que instancia o conceito de potencial associação (Maldonado, 1991) ao teste de regras isoladas.

2.5.1 Abordagem de Chan et al.

O trabalho de Chan et al. (1997) explora o teste de regras ativas escritas em *CDOL – Comprehensive Declarative Object Language* (Urban et al., 1997), uma linguagem baseada na aplicação de regras ativas. Segundo os autores, o objetivo é adaptar técnicas de teste para o teste de aplicações de bancos de dados ativos. A abordagem envolve o teste de fluxo de controle intra-regra e inter-regra.

Uma contribuição é o estudo de fluxo de controle para regras declarativas (a natureza declarativa da linguagem dificulta a abstração de fluxo de controle) e sua utilização em geração de dados de teste. No fluxo de controle intra-regra, a condição de regra é normalizada e são abstraídos três tipos de fluxo de controle: fluxo do evento para a condição; fluxo da condição para ação; e possíveis desvios para o nó de saída a partir da condição de regra. Durante o teste, é requerido o exercício de cada um desses fluxos, similar ao critério *todos os ramos*. No fluxo de controle inter-regra, as regras são consideradas *duas-a-duas* (cada par de regras é tratado separadamente), onde se pretende exercitar cada possível disparo entre regras. Não se considera o teste de *clusters* de regras – um subconjunto de regras relacionadas a funções da aplicação ou a dependências entre regras; assim, basta exercitar cada potencial disparo, independentemente de seu contexto de execução. Ao final, é fornecida uma arquitetura funcional de uma ferramenta para o teste de fluxo de controle *CDOL*; a ferramenta possui um simulador de instâncias de banco de dados usadas durante os testes.

Segundo os autores, esta pesquisa representa o primeiro esforço dedicado ao teste de regras ativas em sistemas de banco de dados. Para Vaduva (1999), as idéias propostas neste trabalho não podem ser aplicadas diretamente a linguagens imperativas utilizadas em condição e

em ação de regra, pois as análises de fluxo de controle intra-regra são dependentes da natureza declarativa da linguagem em questão.

2.5.2 Método de Vaduva

O trabalho de Vaduva (1999) é focado principalmente na interação entre regras ativas e seus efeitos. A pesquisa é conduzida em *SAMOS* (Gatziu et al., 1995), um sistema gerenciador de banco de dados orientado a objetos. É introduzida uma nova abordagem de teste, baseada na idéia de cobertura de tantos cenários de execução quanto possível, executando diferentes seqüências de regra. É dada atenção à geração de dados necessários para cada caso de teste: eventos externos e estados de banco de dados. Uma ferramenta apóia a geração de dados, a execução de casos de teste e a depuração de regras. A idéia é ter, para cada seqüência de regra, um estado de banco de dados inicial que permita, quando eventos necessários ocorrerem, a execução de seqüência de regra completa.

O critério de teste, denominado *cobertura de seqüência*, parte da idéia intuitiva de se testar várias seqüências de regras, independentemente de haver ou não haver disparo entre regras, com a intenção de revelar interações inadequadas entre regras ou falsas sucessões de execução de regras. Neste caso, o alvo para cada caso de teste é a execução de uma certa seqüência de regras, dentre as regras de um dado conjunto. Os elementos estruturais das regras são ignorados na cobertura de uma seqüência. Para clarear a descrição do método, são dadas duas definições:

Definição:

> Dado um conjunto de regras $R = \{r_1, r_2, ..., r_n\}$, uma seqüência de regras de tamanho k, $k \leq n$, é uma lista ordenada de k regras $\rho = (r_i, ..., r_j)$, $1 \leq i, j \leq n$. A lista dá a ordem na qual as regras são disparadas quando a seqüência é executada.

Definição:

> Uma seqüência de regras ρ_k de tamanho k está incluída na seqüência ρ_l de tamanho l, com $k, l \in N$, denotado por $\rho_k \subseteq \rho_l$, quando: 1) $k \leq l$, e 2) se $\rho_k = (r_i, ..., r_j)$, quando $\rho_l = (r_i, ..., r_j, ..., r_m)$, significando que os primeiros k elementos de ρ_l são exatamente os elementos de ρ_k.

Cada caso de teste é determinado por uma seqüência de regras, isto é, a execução de um caso de teste objetiva a execução da seqüência de regras correspondente. Obviamente, é impossível executar todas as seqüências de regras, devido ao aumento de custo do teste. O conjunto de casos de teste é então restrito às combinações de regras tal que, em cada combinação, nenhuma regra seja executada mais de uma vez. Se o conjunto de regras R possui n elementos, $R = \{r_1, r_2, ..., r_n\}$, então, utilizando a notação da autora, $\wp(R) = \{R_i \subseteq R, i = 1, ..., 2^n\}$ é o conjunto de todos os subconjuntos de R. Isto significa que a cardinalidade do conjunto de casos de teste Λ é:

$$|\Lambda| = C_n^1.1! + C_n^2.2! + ... + C_n^n.n! = (n! / (n-1)!) + (n! / (n-2)!) + ... + n!$$

O conjunto Λ contém todas as possíveis permutações de elementos de $R_i \in \wp(R)$, $1 \leq i \leq 2^n$; mesmo assim, o número de casos de testes é extremamente grande; por exemplo, para $n = 5$, então $|\Lambda| = 325$ se todos os casos de teste tiverem sucesso no exercício da seqüência pretendida. Foram eliminadas as seqüências de regras não alcançáveis ou irrelevantes, tais como: ***(i)*** se o disparo entre regras resulta na transferência imediata de controle entre regras, então existem seqüências não alcançáveis; por exemplo, se r_1 dispara r_2, então todas as seqüências contendo (r_1, r_i, r_2), $1 \leq i \leq n$, $i \neq 1,2$, nunca poderão ser executadas; e ***(ii)*** se o teste visa a análise de interações, então as seqüências cujas regras, de acordo com a análise estática de R, não disparam ou são disparadas por outras regras e não manipulam os mesmos objetos, poderiam ser descartadas durante o teste.

A abordagem também buscou otimizar o processo de teste. A idéia é que a execução de uma seqüência $(r_1, r_2, ..., r_n)$ fosse considerada na execução de todas as seqüências incluídas. Portanto, os casos de teste gerados para cobrir a seqüência $(r_1, r_2, ..., r_n)$ seriam utilizados para cobrir todas as seqüências incluídas, tais como (r_1, r_2) e (r_1, r_2, r_3). Utilizando a notação da autora, o conjunto Λ' foi derivado do conjunto Λ, tal que Λ focasse as seqüências de regras mais longas:

$$\Lambda' \subset \Lambda\ ,\ \Lambda' = \{\ \text{seqüências de regra } \rho_i \mid\ \forall \rho_i\ ,\ \rho_j \in \Lambda\ ,\ \rho_i \subset \rho_j\ \}$$

Portanto, dados de teste em Λ podem ser automaticamente usados em Λ': se as seqüências de regra exercitadas por Λ são cobertas, então as seqüências de regras exercitadas por Λ' também serão cobertas.

2.5.3 Aplicação dos Critérios Potenciais Usos

Motivada pela necessidade por abordagens de teste sistemático para bancos de dados ativos, Cardoso (2004) contribuiu para a automação do teste de regras ativas escritas em SQL, no contexto de teste estrutural de unidades. Foi construída uma ferramenta, *ART-TOOL* (*Active Rule Testing Tool*), que apóia a aplicação de critérios estruturais baseados em Fluxo de Dados, especificamente os critérios *Potenciais Usos* (Maldonado, 1991; Maldonado et al., 1992).

Foram apresentados modelos de fluxo de controle, de fluxo de dados e de instrumentação, a partir das especificações contidas em (Leitão et al., 2002). Tais modelos são aplicados na análise estática de código fonte. Sobre o modelo de fluxo de dados, a ferramenta automatiza somente a granularidade *relação*, que constitui uma abordagem conservadora em comparação com granularidades que representam maior precisão de fluxo de dados. A noção de granularidade de fluxo de dados é abordada no Capítulo 4.

Atenção especial foi dada à presença de rotinas de tratamento de exceção, as quais eliminam a natureza seqüencial dos comandos de manipulação de dados persistentes. Duas classes de tratamento de exceção foram caracterizadas: definida pelo sistema e definida pelo programador; 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 é genérica, podendo ser estendida a outras linguagens que disponham do recurso de tratamento de exceção.

Ressaltou-se também a utilização de variáveis do tipo *cursor*. Cursores explícitos e implícitos são derivados da necessidade de se tratar *tupla*-a-*tupla* o resultado de uma operação de manipulação, diferentemente da álgebra relacional. A cobertura de potenciais associações de fluxo de dados para as variáveis do tipo *cursor* foi efetiva na determinação de defeitos atribuídos ao manuseio defeituoso dessas variáveis.

A ferramenta monitora o fluxo de dados sem identificar quais entidades persistentes são afetadas pelas operações de acesso a bases de dados. Assim, aplica a essência das relações de fluxo de dados focando-se apenas nos caminhos exercitados. O texto da dissertação inclui uma proposta para a identificação de *tuplas* afetadas, a qual foi extraída de (Leitão et al., 2002).

Basicamente, exporta os predicados de seleção de *tuplas* dos comandos SQL para os comandos de instrumentação de fluxo de dados inseridos no código da regra.

A ferramenta foi aplicada a várias regras escritas em SQL. Segundo a autora, 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. Os resultados obtidos pelo uso da ferramenta 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.

2.6 Considerações Finais

Neste capítulo, as iniciativas de pesquisa sobre teste de banco de dados foram analisadas e classificadas segundo as categorias: teste de aplicações de banco de dados, teste de projeto de banco de dados e geração de bases de teste. Observou-se que a sistemática de teste de software foi inserida apenas recentemente nas pesquisas focadas na qualidade de bases de dados e de suas aplicações, e que o teste de aplicações que usam a SQL tem sido abordado pela comunidade acadêmica.

Características de sistemas gerenciadores de banco de dados ativos foram descritas segundo os modelos de conhecimento e de execução. Identificou-se que o suporte SQL a regras ativas, segundo o padrão SQL3, abrange um subconjunto das propriedades desses modelos, aspecto este utilizado pelo modelo de interação entre regras explorado nesta pesquisa, que é introduzido no Capítulo 4.

Foi constatado que o teste de regras escritas em SQL foi alvo de apenas uma pesquisa, (Cardoso, 2004), a qual é focada na aplicação dos Critérios Potenciais Usos ao teste de regras isoladas. Especificamente, modelos de implementação desses critérios descritos em (Leitão et al., 2002) apoiaram a construção de uma ferramenta, a qual automatiza a cobertura de associações de fluxo de dados em nível de granularidade *relação*, mas não alcança às demais precisões de fluxo de dados, as quais são descritas no Capítulo 4.

Nos próximos capítulos são estudados a estrutura dos comandos de manipulação da SQL, aspectos de defeitos e de falhas em tais comandos e é delineado o modelo de interação entre regras que utiliza informação de fluxo de dados persistentes.

Capítulo 3

Defeitos e Falhas em Manipulação de Dados Persistentes

O conhecimento sobre defeitos e falhas de *software* é importante para promover o desenvolvimento de programas menos defeituosos, guiar a depuração de programas e nortear a proposição de abordagens de teste, na direção de melhoria de qualidade e de redução de custos.

Defeitos são uma parte inevitável de produção de *software*. Defeitos podem existir sem causar dano, quando nunca alcançados. Quando a execução do programa a partir de uma dada entrada alcança o local do defeito, uma falha pode ocorrer. Um mesmo defeito pode ocasionar diferentes categorias de falha, dependendo da entrada utilizada. Para alcançar a qualidade de *software* desejada, é imperativo entender o relacionamento entre defeitos e falhas em uma linguagem de programação específica, o que está intrinsecamente associado ao conhecimento das construções sintáticas da linguagem e como estas são usadas para obter a semântica desejada.

Assim, o elenco de defeitos e suas possíveis falhas são dependentes da linguagem de programação utilizada. Diferentes linguagens de programação possuem aspectos sintáticos e semânticos próprios, que afetam a percepção cognitiva do testador sobre sua complexidade. Esse fato conduz a tipos específicos de defeitos para linguagens de programação distintas. Tal cenário motiva a investigação para entender que tipos de defeitos de *software* podem ocorrer quando uma linguagem particular é utilizada. Em nosso contexto – aplicações de banco de dados ativos que usam a SQL – operações de manipulação de dados podem ser fonte de defeitos em muitos programas. Uma operação de manipulação modifica o estado da base de dados ou consulta o banco de dados para obter e formatar dados de saída. Essas interações diretas entre programas e

bases de dados são materializadas por comandos específicos da SQL, mencionados como *comandos de manipulação*.

A enumeração de defeitos de *software*, também denominada de *esquema de defeitos*, é construída de acordo com propósitos específicos, tais como: identificar causas de erros, tomar decisão durante o desenvolvimento de *software*, elaborar perfis de desenvolvimento e guiar a atividade de teste. Esses esquemas podem ser simples ou mais elaborados. Por exemplo, um esquema simples agrupa defeitos como principal e secundário, e um esquema mais elaborado refina a descrição do defeito para estabelecer níveis sucessivos, dividindo-os em categorias mais especializadas.

Chillarege et al. (1992) apresentam um esquema de classificação patrocinado pela IBM, objetivando promover análise e aprendizagem sobre o processo de *software* que trata de detecção, correção e prevenção de defeitos. A meta é extrair informação sobre defeitos e usar tal informação na avaliação de partes de um processo de desenvolvimento de *software*, com o intuito de tomar ações corretivas. A proposta original define uma classificação ortogonal de defeitos (ODC – *Orthogonal Defect Classification*), que envolve a identificação das categorias *gatilho*, *tipo* e *qualificador* para cada defeito. O *gatilho* indica o evento que promoveu a descoberta da presença do defeito. O *tipo* identifica a categoria do defeito, tal como *interface* ou *algoritmo*. O *qualificador* envolve adjetivos tais como *ausente* e *incorreto*. Esse esquema de defeitos tem sido evoluído pela inclusão de novas categorias de defeito adicionalmente às três originais, em resposta às evoluções das técnicas de *software*. Algumas categorias são identificadas no momento da descoberta do defeito e outras em tempo de correção do defeito.

Kelly e Shepard (2001) desenvolveram um esquema de classificação de defeitos, baseando-se em Chillarege et al. (1992), para avaliar e comparar a efetividade de técnicas de inspeção de *software*. De acordo com os autores, o esquema ODC não é adequado para propósitos de inspeção, visto que os tipos de defeito devem refletir o problema no tempo em que se percebe o defeito no código: o tipo de defeito deve relatar o código antes que a atividade de correção tenha iniciado. Comparado ao ODC, o número de tipos de defeito foi substancialmente expandido e novos qualificadores foram adicionados, tais como *supérfluo* e *obscuro*. Ambos os esquemas acima são definidos usando julgamento subjetivo. O primeiro tenta responder a questão "o que você estava pensando quando descobriu o defeito?"; o segundo está relacionado à pergunta "que tarefa você estava realizando quando descobriu o defeito?".

Uma enumeração mais completa de defeitos foi proposta em Beizer (1990). Neste esquema, as principais categorias abrangem defeitos funcionais, passando por defeitos de integração e de interfaces internas, até defeitos de definição e de execução de testes. Cada categoria principal é detalhada em níveis de suficiência para incluir a descrição de todos os seus possíveis defeitos. Não se objetiva apresentar minúcias deste esquema, mas ressaltar que este serve como ponto de referência para a construção de novos esquemas de defeito.

De acordo com Beizer (1990), não existe uma forma universal para enumerar e categorizar defeitos. Defeitos são difíceis de enumerar e um dado defeito pode estar em uma ou outra categoria, dependendo do seu histórico de ocorrências e do ponto de vista do testador. Adicionalmente, os aspectos inerentes às linguagens de programação determinam o elenco de defeitos possíveis e caracterizam a propensão a defeitos de programação. Mais importante que buscar a enumeração e a classificação de defeitos "corretas", é usar algum esquema de defeitos para nortear estratégias de teste.

Leitão et al. (2005) propuseram uma enumeração de defeitos de manipulação, fundamentando-se na estrutura de comandos SQL. Este esquema foi utilizado para estudar a propagação de defeitos à saída da execução desses comandos. Nenhuma outra publicação de teste de software é focada na enumeração de defeitos e de falhas de manipulação em aplicações baseadas na SQL. Este capítulo baseia-se no conteúdo deste artigo; contudo não tenciona construir o *framework* de enumeração de defeitos e falhas, mas ser um recurso para apoiar o entendimento das causas e das conseqüências de incorretudes em comandos SQL de manipulação. Ênfase é dada ao entendimento de falhas, como se apresentam, como se relacionam aos defeitos de manipulação e que fatores influenciam sua manifestação.

O capítulo é organizado conforme descrito a seguir. A Seção 3.1 explora a estrutura de comandos SQL de manipulação a partir de suas construções básicas, organizando-a em itens estruturais; esses itens representam um passo na direção de entender e de agrupar defeitos para avaliar a saída de execução de comandos. Uma lista de tipos de defeitos de manipulação é apresentada na Seção 3.2, utilizando exemplos de comandos corretos e defeituosos; categorias de falha são introduzidas para comandos de consulta e de mudança de estado, em adição a dois conjuntos de dados que capturam a noção de saída de execução de comandos. A Seção 3.3 considera o desenvolvimento de um mapeamento para analisar o relacionamento entre tipos de defeitos e categorias de falha, onde são construídas bases de dados e são executados comandos

SQL defeituosos. A Seção 3.4 abstrai propriedades de bases de dados que se mostraram reveladoras de defeito na execução de comandos da seção anterior.

3.1 Comandos de Manipulação de Dados

A estrutura geral para a manipulação de dados em SQL é apresentada nesta seção e foi inspirada em um estudo teórico dos comandos de manipulação baseado nos fundamentos da SQL, tal como tratado por Elmasri e Navathe (2003) e Fortier (1999). Tal estrutura foi organizada em itens estruturais, visando a entender que defeitos podem ocorrer em cada item estrutural. Manipulação de dados é implementada pelos comandos *insert*, *delete*, *update* e *select*, para manipular bases de dados por inserção, exclusão, modificação e consulta de dados, respectivamente. O comando *select* é mencionado como *comando de consulta* e os demais como *comandos de mudança de estado*. A estrutura geral do comando *select* é composta pelos itens estruturais *[s1]* a *[s6]*, conforme mostrado na Tabela 3.1:

Tabela 3.1: Estrutura para o comando *select*.

Item estrutural	**Descrição**
$[s_1]$	Lista ordenada de expressões para computar os valores dos atributos retornados
$[s_2]$	Lista de nomes das relações usadas como fonte de dados
$[s_3]$	Predicado para a seleção de *tuplas*
$[s_4]$	Lista de expressões para agrupamento de dados
$[s_5]$	Predicado para a seleção de grupos de dados
$[s_6]$	Lista ordenada de argumentos para ordenação de dados

A forma mais simples do comando *select* possui os itens estruturais *[s1]* e *[s2]*; os demais itens são usados de acordo com a semântica da consulta. Para ilustrar, considere a seguinte consulta:

```
[s1]   select     salary, count(*), sum(bonus)
[s2]   from       empl
[s3]   where      (salary + bonus) > 1050
[s4]   group by   salary
[s5]   having     count(*) > 1
[s6]   order by   salary desc
```

No exemplo dado, a consulta usa a relação *empl* como fonte de dados (item *[s2]*). O mecanismo de seleção restringe empregados cujo salário adicionado do bônus é superior a 1050 (item *[s3]*). Esses empregados são agrupados com base em seus salários; em outras palavras, os empregados de cada grupo possuem o mesmo valor para o atributo *salary* (item *[s4]*). Os grupos que possuem mais de uma *tupla* são então selecionados (item *[s5]*). Para os grupos de empregados selecionados, são computados o salário, a quantidade de empregados e o somatório de bônus (item *[s1]*). Os dados abstraídos são ordenados por salário em ordem descendente (item *[s6]*).

A estrutura dos comandos cuja execução modifica o estado da base de dados é apresentada na Tabela 3.2, a qual inclui um exemplo para cada comando visando a ilustrar a semântica dos seus itens estruturais. O comando *insert* tem duas possíveis construções: uma que menciona explicitamente os valores de dados a serem inseridos na base de dados e outra que obtém tais valores a partir da execução de uma subconsulta.

Tabela 3.2: Estruturas dos comandos *insert*, *delete* e *update*.

Comando	**Item estrutural**	**Descrição**	**Exemplo**
insert (1)	$[i_1]$	Nome de relação	*insert into empl* *(emplno, name, salary, bonus)* *values (1234 , 'ana' , 1060 , 35)*
	$[i_2]$	Lista ordenada de atributos	
	$[i_3]$	Lista ordenada de valores de atributos	
insert (2)	$[i_1]$	Nome de relação	*insert into empl* *(emplno, name, salary, bonus)* *select custno, name, salary, 0 from customer*
	$[i_2]$	Lista ordenada de atributos	
	$[i_4]$	Subconsulta	
Delete	$[d_1]$	Nome de relação	*delete from empl* *where (salary + bonus) > 1050*
	$[d_2]$	Predicado para a seleção de *tuplas*	
Update	$[u_1]$	Nome de relação	*update empl* *set salary = salary * 1.01, bonus = bonus * 1.10* *where (salary + bonus) > 1050*
	$[u_2]$	Lista de atribuições de valores a atributos	
	$[u_3]$	Predicado para a seleção de *tuplas*	

Os itens estruturais não exploram detalhes sintáticos dos comandos da SQL, mas focam nos componentes semânticos da linguagem para a manipulação de dados. Algumas cláusulas, tais como *union* e *intersect*, não são cobertas pela estrutura de manipulação de dados apresentada,

visto que o interesse maior é explorar defeitos nas construções básicas de cada comando de manipulação. As subconsultas são consultas implicitamente ou explicitamente embutidas em qualquer comando de manipulação, onde sua estrutura é a mesma apresentada na Tabela 3.1.

3.2 Defeitos e Falhas de Manipulação

Enumerar tipos de defeitos é a base para a compreensão do porquê existem defeitos em uma dada linguagem de programação. Tal atividade requer conhecimento das construções básicas disponíveis na linguagem de programação correspondente. Esta seção é focada na enumeração de tipos de defeitos em comandos de manipulação baseando-se nos itens estruturais mostrados na Seção 3.1.

A terminologia normalmente usada em teste de programas é estendida para comandos individuais de manipulação de dados. Um *defeito* (*fault*) é definido como um item incorreto presente em um comando SQL de manipulação. Um *comando defeituoso* é um comando SQL que possui, no mínimo, um defeito. Um *erro* é um estado de execução inconsistente causado pela execução de um comando SQL defeituoso. Uma *falha* ocorre quando um defeito é propagado para a saída do comando SQL defeituoso, resultando em uma saída distinta da esperada[1]. *Categorias de falha* abstraem os modos como falhas são manifestadas na execução de comandos SQL defeituosos.

Os possíveis tipos de defeito em comandos de manipulação são introduzidos e identificados seguindo o termo *[xm]-n*, onde: *xm* refere-se ao item estrutural tal como mostrado nas Tabelas 3.1 e 3.2; e *n* é um número seqüencial para os tipos de defeito que estão relacionados ao mesmo item estrutural. Esta seção apresenta, para cada tipo de defeito, um identificador, uma descrição e um exemplo. Na descrição, o termo *tabela* denota uma relação da base de dados. O exemplo ilustra a versão correta e uma versão incorreta de um comando de manipulação. É possível entender a semântica de um tipo de defeito pela observação da sua descrição e pela comparação entre as duas versões do comando exemplo. O esquema de banco de dados usado nos exemplos é intuitivo e, por isso, foi omitido dessa discussão.

[1] A noção de funcionalidade esperada na especificação de software é estendida à saída esperada da execução de comandos de manipulação.

O comando *select* possui uma estrutura mais complexa do que os comandos *insert*, *delete* e *update*, conforme apresentado nas Tabelas 3.1 e 3.2, resultando em uma numerosa lista de tipos de defeito para este comando: mais da metade dos tipos de defeito são atribuídos ao comando *select* em relação aos demais comandos de manipulação. A Tabela 3.3 enumera os tipos de defeito para o comando *select*; apesar de sua extensão, o conteúdo desta tabela é auto-explicativo, devido à simplicidade dos exemplos existentes. A descrição apresentada na segunda coluna utiliza um qualificador e um local relativo ao item estrutural do comando; por exemplo, os tipos de defeito *[s1]-1* e *[s1]-4* referem-se aos qualificadores *ausente* e *incorreto*, respectivamente, no item estrutural *lista ordenada de expressões para computar os valores dos atributos retornados*.

Tabela 3.3: Lista de tipos de defeito para o comando *select*.

Ident.	Descrição	Comando correto	Comando incorreto
$[s_1]$-1	Uma expressão está ausente da lista ordenada de expressões para computar os valores dos atributos retornados	*select emplno, name, salary* *from empl*	*select emplno, salary* *from empl*
$[s_1]$-2	Uma expressão está indevidamente presente na lista ordenada de expressões para computar os valores dos atributos retornados	*select emplno, name, salary* *from empl*	*select emplno, name, salary, salary + bonus* *from empl*
$[s_1]$-3	A ordem de expressões na lista ordenada de expressões para computar os valores dos atributos retornados está incorreta	*select emplno, name, salary, salary + bonus* *from empl*	*select emplno, name, salary + bonus, salary* *from empl*
$[s_1]$-4	Uma expressão incorreta está na lista ordenada de expressões para computar os valores dos atributos retornados	*select emplno, name, salary* *from empl*	*select emplno, name, bonus * 0.5* *from empl*
$[s_2]$-1	Um nome de tabela está ausente da lista de nomes de tabelas usadas como fonte de dados	*select e.name, e.salary* *from empl e, dept d*	*select e.name, e.salary* *from empl e*
$[s_2]$-2	Um nome de tabela está indevidamente presente na lista de nomes de tabelas usadas como fonte de dados	*select e.name, e.salary* *from empl e*	*select e.name, e.salary* *from empl e, dept d*
$[s_2]$-3	Um nome de tabela incorreto está na lista de nomes de tabelas usadas como fonte de dados	*select e.name, d.salary* *from empl e, dept d*	*select e.name, d.salary* *from empl e, depn d*
$[s_3]$-1	O predicado para a seleção de *tuplas* está ausente	*select emplno, salary* *from empl* *where salary > 1050*	*select emplno, salary* *from empl*
$[s_3]$-2	O predicado para a seleção de *tuplas* está indevidamente presente	*select emplno, salary* *from empl*	*select emplno, salary* *from empl* *where salary > 1050*
$[s_3]$-3	O predicado para a seleção de *tuplas* está incorreto	*select emplno, salary* *from empl* *where salary > 1050*	*select emplno, salary* *from empl* *where salary < 1200*
$[s_4]$-1	A lista de expressões para agrupamento de dados está ausente	*select salary* *from empl* *group by salary*	*select salary* *from empl*
$[s_4]$-2	A lista de expressões para agrupamento de dados está indevidamente presente	*select salary* *from empl*	*select salary* *from empl* *group by salary*
$[s_4]$-3	Uma expressão está ausente da lista de	*select salary, count(*)*	*select salary, count(*)*

	expressões para agrupamento de dados	*from empl* *group by salary, bonus*	*from empl* *group by salary*
[s_4]-4	Uma expressão está indevidamente presente na lista de expressões para agrupamento de dados	*select salary, count(*)* *from empl* *group by salary*	*select salary, count(*)* *from empl* *group by salary, bonus*
[s_4]-5	Uma expressão incorreta está presente na lista de expressões para agrupamento de dados	*select salary, count(*)* *from empl* *group by salary, bonus*	*select salary, count(*)* *from empl* *group by salary, salary - bonus*
[s_5]-1	O predicado para a seleção de grupos de dados está ausente	*select salary, count(*)* *from empl* *group by salary* *having count(bonus) > 1*	*select salary, count(*)* *from empl* *group by salary*
[s_5]-2	O predicado para a seleção de grupos de dados está indevidamente presente	*select salary, count(*)* *from empl* *group by salary*	*select salary, count(*)* *from empl* *group by salary* *having count(bonus) > 1*
[s_5]-3	O predicado para a seleção de grupos de dados está incorreto	*select salary, count(*)* *from empl* *group by salary* *having count(bonus) > 1*	*select salary, count(*)* *from empl* *group by salary* *having count(salary) > 1*
[s_6]-1	A lista ordenada de argumentos para ordenação de dados está ausente	*select emplno, name, salary* *from empl* *order by salary*	*select emplno, name, salary* *from empl*
[s_6]-2	A lista ordenada de argumentos para ordenação de dados está indevidamente presente	*select emplno, name, salary* *from empl*	*select emplno, name, salary* *from empl* *order by salary*
[s_6]-3	A ordem de argumentos na lista ordenada de argumentos para ordenação de dados está incorreta	*select emplno, name, salary* *from empl* *order by salary, name*	*select emplno, name, salary* *from empl* *order by name, salary*
[s_6]-4	Um argumento incorreto está na lista ordenada de argumentos para ordenação de dados	*select emplno, name, salary* *from empl* *order by salary, name*	*select emplno, name, salary* *from empl* *order by salary desc, name*

As Tabelas 3.4, 3.5 e 3.6 introduzem as listas de tipos de defeito para os comandos *insert*, *delete* e *update*, respectivamente.

A propagação do efeito do defeito para a saída de comando, denominada *falha*, está relacionada a problemas bem conhecidos, instanciados por Chays et al. [2000] para o teste de aplicações de banco de dados: *controlabilidade*, que busca colocar o sistema em um estado desejado antes da execução de um caso de teste; e *observabilidade*, que trata da observação do estado de sistema após a execução de casos de teste.

Dois conjuntos de dados são introduzidos a seguir, *conjunto de dados retornados* e *conjunto de dados definidos*, visando a capturar a noção de saída da execução de comandos de manipulação[2]; ambos são denominados *conjuntos de saída*. Eles são conjuntos tabulares (bi-dimensionais), mas não constituem relações de acordo com os conceitos do modelo relacional

[2] A noção da saída de execução de comandos individuais denota os dados obtidos pela execução de comandos de manipulação, com precisão suficiente para decidir se uma falha foi produzida.

(Codd, 1970), visto que um desses conjuntos pode possuir linhas duplicadas. Apesar disso, tais conjuntos são descritos abaixo usando terminologia do modelo relacional, tal como *tupla* e *atributo*.

O *conjunto de dados retornados* está relacionado com a execução de comandos de consulta, enquanto o *conjunto de dados definidos* está relacionado com a execução de comandos de mudança de estado. Cada execução de comando de manipulação está associada a duas versões dos comandos de saída: a esperada e a obtida. Se o conjunto de saída obtido for distinto do esperado, então um defeito foi propagado para a saída de execução do comando e ocorreu a manifestação de falha. As formas nas quais a versão obtida pode diferir da versão esperada são denominadas *categorias de falha de manipulação*.

Tabela 3.4: Lista de tipos de defeito para o comando *insert*.

Ident.	**Descrição**	**Comando correto**	**Comando incorreto**
$[i_1]$-1	O nome de tabela está incorreto.	*insert into empl* *(emplno, name, salary)* *values (8888, 'mary smith', 1050)*	*insert into depn* *(emplno, name, salary)* *values (8888, 'mary smith', 1050)*
$[i_2]$-1	Um atributo está ausente da lista ordenada de atributos	*insert into empl* *(emplno, name, salary, bonus)* *values (8888, 'mary smith', 1050, 12)*	*insert into empl* *(emplno, name, salary)* *values (8888, 'mary smith', 1050)*
$[i_2]$-2	Um atributo está indevidamente presente na lista ordenada de atributos	*insert into empl* *(emplno, name, salary)* *values (8888, 'mary smith', 1050)*	*insert into empl* *(emplno, name, salary, bonus)* *values (8888, 'mary smith', 1050, 12)*
$[i_3]$-1	Um valor incorreto está presente na lista ordenada de valores de atributos	*insert into empl* *(emplno, name, salary, bonus)* *values (8888, 'mary smith', 1050, null)*	*insert into empl* *(emplno, name, salary, bonus)* *values (8888, 'mary smith', 1050, 22)*
$[i_3]$-2	A ordem dos valores na lista ordenada de atributos está incorreta	*insert into empl* *(emplno, name, salary, bonus)* *values (8888, 'mary smith', 1050, 22)*	*insert into empl* *(emplno, name, salary, bonus)* *values (8888, 'mary smith', 22, 1050)*
$[i_4]$-1	Existe um defeito na subconsulta	*insert into empl* *(emplno, name, salary)* *select 9999, name, salary* *from depn* *where emplno = 1111*	*insert into empl* *(emplno, name, salary)* *select 9998, name, salary* *from depn* *where emplno = 1111*

Tabela 3.5: Lista de tipos de defeito para o comando *delete*.

Ident.	Descrição	Comando correto	Comando incorreto
$[d_1]$-1	O nome de tabela está incorreto	*delete from empl* *where name like 'ann%'*	*delete from depn* *where name like 'ann%'*
$[d_2]$-1	O predicado para a seleção de *tuplas* está ausente	*delete from empl* *where name like 'ann%'*	*delete from empl*
$[d_2]$-2	O predicado para a seleção de *tuplas* está indevidamente presente	*delete from empl*	*delete from empl* *where name like 'ann%'*
$[d_2]$-3	O predicado para a seleção de *tuplas* está incorreto	*delete from empl* *where name like 'ann%'*	*delete from empl* *where salary between 800 and 1060*

Tabela 3.6: Lista de tipos de defeito para o comando *update*.

Ident.	Descrição	Comando correto	Comando incorreto
$[u_1]$-1	O nome de tabela está incorreto	*update empl* *set salary = salary * 1.1* *where salary < 550*	*update depn* *set salary = salary * 1.1* *where salary < 550*
$[u_2]$-1	Uma atribuição de valor está ausente da lista de atribuições de valores a atributos.	*update empl* *set salary = salary * 1.1,* *bonus = bonus * 1.1* *where salary < 550*	*update empl* *set salary = salary * 1.1* *where salary < 550*
$[u_2]$-2	Uma atribuição de valor está indevidamente presente na lista de atribuições de valores a atributos	*update empl* *set salary = salary * 1.1* *where salary < 550*	*update empl* *set salary = salary * 1.1,* *bonus = bonus * 1.1* *where salary < 550*
$[u_2]$-3	Uma expressão incorreta está presente no lado direito de uma atribuição de valor da lista de atribuições de valores a atributos	*update empl* *set salary = salary * 1.1* *where salary < 550*	*update empl* *set salary = 1.1* *where salary < 550*
$[u_2]$-4	Um nome incorreto de atributo está presente do lado esquerdo de uma atribuição de valor da lista de atribuições de valores a atributos	*update empl* *set salary = salary * 1.1* *where salary < 550*	*update empl* *set bonus = salary * 1.1* *where salary < 550*
$[u_3]$-1	O predicado para a seleção de *tuplas* está ausente	*update empl* *set bonus = bonus * 1.1* *where salary < 550*	*update empl* *set bonus = bonus * 1.1*
$[u_3]$-2	O predicado para a seleção de *tuplas* está indevidamente presente	*update empl* *set salary = salary * 1.1*	*update empl* *set salary = salary * 1.1* *where salary < 550*
$[u_3]$-3	O predicado para a seleção de *tuplas* está incorreto	*update empl* *set salary = salary * 1.1* *where salary < 550*	*update empl* *set salary = salary * 1.1* *where salary > 1200*

Um comando de manipulação está defeituoso quando o conjunto de saída obtido por sua execução diferir do conjunto esperado em pelo menos uma categoria de falha de manipulação. As categorias de falha de manipulação são: *lista de atributos*, *ordem de atributos*, *número de tuplas*, *lista de valores de atributos* e *ordem de tuplas*. A categoria *lista de atributos* descreve os atributos do conjunto esperado; esta categoria está relacionada ao domínio de atributo e à semântica de atributo. A categoria *ordem de atributos* é significante quando a ordem dos

atributos caracteriza as *tuplas* do conjunto de saída. A categoria *número de tuplas* define a cardinalidade de *tuplas* do conjunto esperado. A categoria *lista de valores de atributos* estabelece, para todas as *tuplas*, o valor esperado para cada atributo, sem considerar a ordem de tais atributos. A categoria *ordem de tuplas* define de forma não ambígua a ordem esperada das *tuplas* do conjunto de saída, quando esta ordem for relevante.

Definição: ***Conjunto de dados retornados***

Cada execução de um comando *select* resulta em um conjunto de *tuplas*, nomeado neste estudo de *conjunto de dados retornados*. Este conjunto é descrito por uma lista ordenada de atributos, $(A^r_1, A^r_2, ..., A^r_p)$, onde A^r_i, $1 \leq i \leq p$, é mencionado como *atributo retornado*. As expressões usadas para computar os valores de atributos retornados são tipicamente compostas por operadores aritméticos e lógicos, funções pré-definidas para acessar e para formatar dados de tipos especiais, funções de sumarização, subconsultas, etc., dificultando portanto sua formalização. Se várias execuções são realizadas para um dado comando *select*, então são obtidos vários conjuntos de dados retornados, um para cada execução realizada. A presença de um defeito é descoberta em um comando *select* quando pelo menos um de seus conjuntos de dados retornados obtidos diferir do conjunto esperado, em relação a quaisquer das categorias de falha; nesse sentido, todas as categorias de falha são aplicáveis a comandos de consulta.

Considere o comando *select* apresentado na Seção 3.1. A lista de atributos é composta por 3 elementos, os quais são computados pelas expressões *salary*, *count(*)* e *sum(bonus)*, e os valores obtidos para cada *tupla* são retornados de acordo com essa seqüência de atributos. O número de *tuplas* e a ordem de valores em cada *tupla* dependem do comando propriamente dito e do estado da base de dados anterior à execução do comando. A ordem de *tuplas* é relevante neste comando, baseando-se nos valores do atributo *salary* de forma descendente.

Definição: ***Conjunto de dados definidos***

A execução dos comandos *insert*, *delete* e *update* pode modificar o estado da base de dados e *conjunto de dados definidos* é a denominação para o conjunto de *tuplas* relacionadas aos dados efetivamente inseridos, excluídos e modificados pela execução desses comandos. Este conjunto é descrito por uma lista de atributos, $(A^d_1, A^d_2, ..., A^d_s)$,

onde A^d_i, $1 \leq i \leq s$, é mencionado como *atributo definido*. Considere um comando *C* de mudança de estado, tal que sua execução pode modificar o estado da relação *r*. Se *C* é um comando *insert*, a lista dos atributos definidos pela execução de *C* é composta por todos os atributos do esquema da relação *r*. Se *C* é um comando *delete*, a lista dos atributos definidos pela execução de *C* é composta pelos atributos da chave primária da relação *r*. Se *C* é um comando *update*, onde o conjunto de saída é referente à imagem de banco de dados após a execução de comando, a lista de atributos definidos pela execução de *C* é determinada pelos atributos com valores explicitamente atribuídos na estrutura de comando (conforme item estrutural *[u_2]* na Tabela 3.2), em adição aos atributos da chave primária da relação *r*. É importante notar que os atributos da chave primária da relação estão sempre incluídos na lista de atributos definidos, visto que são necessários para checar se a operação de mudança de estado foi aplicada nas tuplas "certas".

Por exemplo, considere o seguinte comando:

```
update empl
set  salary = salary * 1.1, bonus = 12
where bonus is null
```

Se a chave primária de *empl* é representada unicamente pelo atributo *emplno*, a lista de atributos definidos é composta pelos atributos *emplno*, *salary* e *bonus*.

Similarmente ao conjunto de dados retornados, se várias execuções são realizadas para um mesmo comando de mudança de estado, então são obtidos vários conjuntos de dados definidos, um para cada execução realizada. Entretanto, a presença de um defeito é descoberta em um comando de mudança de estado quando pelo menos um de seus conjuntos de dados definidos obtidos diferir do conjunto esperado em relação às seguintes categorias de falha: *lista de atributos*, *número de tuplas* e *lista de valores de atributos*; as categorias de falha *ordem de atributos* e *ordem de tuplas* não são consideradas neste caso, pois são irrelevantes para caracterizar o estado da base de dados após a execução do comando.

Em síntese, comandos de consulta e comandos de mudança de estado diferem na noção de falha de execução de comando. Nos primeiros, uma falha é manifestada quando o conjunto de dados retornados é distinto do esperado; nos últimos, uma falha é manifestada quando o conjunto de dados definidos é distinto do esperado (o estado da base de dados após a execução do

comando difere do esperado). Adicionalmente, as categorias de falha *ordem de atributos* e *ordem de tuplas* são exclusivas para a consulta de dados, visto que a ordem de colunas e de linhas do conjunto resultante de uma consulta pode ser relevante dentro da aplicação de banco de dados em que o comando está inserido.

3.3 Mapeamento entre Defeitos e Falhas

Esta seção explora a construção de um mapeamento e analisa o relacionamento entre defeitos e falhas de manipulação. O entendimento desse relacionamento foi construído seguindo três etapas: *i)* estudo da semântica dos tipos de defeitos baseando-se no conhecimento dos itens estruturais dos comandos de manipulação, tendo em mente a projeção do efeito do defeito para a saída da execução de comandos; *ii)* construção preliminar de um mapeamento entre tipos de defeitos e suas falhas possíveis, resultando em um recurso útil às atividades de teste e de depuração; e *iii)* execução de comandos defeituosos para evoluir o mapeamento. O objetivo é construir um único mapeamento para todos os tipos de defeitos de manipulação, que englobe comandos de consulta e de mudança de estado.

Baseando-se na lista de tipos de defeito introduzida na seção anterior, um conjunto de comandos defeituosos foi elaborado para cada tipo de defeito de manipulação, onde cada comando defeituoso possuía um único tipo de defeito. Os comandos de cada conjunto exploram variações para cada tipo de defeito. Por exemplo, considere os comandos defeituosos abaixo, os quais possuem defeitos do tipo *[s_6]-4* tal como identificado na Tabela 3.3, onde diferentes construções do comando *select* são usadas para o tipo de defeito. A versão correta dos comandos defeituosos possui a cláusula de ordenação *order by salary, name*.

```
select emplno, name, salary     select emplno, name, salary     select emplno, name, salary
from empl                       from empl                       from empl
order by salary desc, name      order by salary, name desc      order by salary, bonus
```

Os comandos defeituosos foram executados para observar as categorias das falhas manifestadas; nesse sentido diz-se que categorias de falha foram manifestadas (também se diz que categorias de falha foram exercitadas ou cobertas). Bases de dados foram preparadas para a

execução dos comandos, de modo que buscassem cobrir todas as categorias de falha para cada tipo de defeito. Se qualquer categoria de falha ainda não tiver sido exercitada para um tipo de defeito, então novas bases de dados foram preparadas, pela inserção, exclusão e modificação de *tuplas*, até que todas as categorias de falhas fossem cobertas. O mesmo raciocínio foi aplicado aos comandos defeituosos, pois novos comandos defeituosos foram preparados com a intenção de cobrir todas as categorias de falha para um mesmo tipo de defeito. Em ambos os casos, após não se conseguir a manifestação de alguma categoria de falha, foi assumido que tal categoria não é aplicável ao tipo de defeito em questão. Como resultado, mais de 180 comandos defeituosos foram executados para todos os tipos de defeito, usando bases de dados implementadas no sistema gerenciador de banco de dados *Oracle*. Nenhuma tentativa foi realizada para minimizar o número de comandos defeituosos em relação a cada tipo de defeito. O esquema das bases de dados usadas era simples e similar ao utilizado nos exemplos das Tabelas 3.3 a 3.6. Um banco de dados composto por 3 relações, com 2 a 6 *tuplas* foi restaurado antes da avaliação de cada comando defeituoso, e no mínimo 3 bases de dados foram utilizadas na execução de cada comando. Como conseqüência, o mapeamento evoluiu para o conteúdo da Tabela 3.7. A primeira coluna lista as categorias de falha para os conjuntos de dados retornados e de dados definidos, em relação aos comandos de consulta e de mudança de estado, respectivamente. A segunda coluna apresenta, para cada categoria de falha, a lista de tipos de defeito que podem causar a cobertura da categoria de falha.

Tabela 3.7: Mapeamento entre tipos de defeito e categorias de falha.

Categoria de falha	**Lista de tipos de defeito**
Lista de atributos	$[s_1]$-1, $[s_1]$-2, $[s_1]$-4, $[i_1]$-1, $[d_1]$-1, $[u_1]$-1, $[u_2]$-1, $[u_2]$-2, $[u_2]$-4
Ordem de atributos	$[s_1]$-3
Número de *tuplas*	$[s_2]$-1, $[s_2]$-2, $[s_2]$-3, $[s_3]$-1, $[s_3]$-2, $[s_3]$-3, $[s_4]$-1, $[s_4]$-2, $[s_4]$-3, $[s_4]$-4, $[s_4]$-5, $[s_5]$-1, $[s_5]$-2, $[s_5]$-3, $[d_2]$-1, $[d_2]$-2, $[d_2]$-3, $[i_4]$-1, $[u_3]$-1, $[u_3]$-2, $[u_3]$-3
Lista de valores de atributos	$[s_2]$-1, $[s_2]$-2, $[s_2]$-3, $[s_3]$-1, $[s_3]$-2, $[s_3]$-3, $[s_4]$-3, $[s_4]$-4, $[s_4]$-5, $[i_2]$-1, $[i_2]$-2, $[i_3]$-1, $[i_3]$-2, $[i_4]$-1, $[d_2]$-3, $[u_2]$-3, $[u_3]$-1, $[u_3]$-2, $[u_3]$-3
Ordem de *tuplas*	$[s_6]$-1, $[s_6]$-2, $[s_6]$-3, $[s_6]$-4

Nas operações de mudança de estado, os comandos defeituosos que violavam restrições de integridade da base de dados, tais como chave primária e chave estrangeira, foram

descartados. Tais comandos foram substituídos por comandos que respeitavam as restrições, evitando que todos os tipos de defeito para essa classe de comandos estivessem relacionados à categoria de falha *número de tuplas*.

A compilação de comandos SQL pelo sistema de gerenciador de banco de dados eliminou muitos comandos defeituosos, reduzindo portanto o número de possíveis defeitos de manipulação. Por exemplo, considere o tipo de defeito *[i_3]-2*, onde a ordem dos valores na lista ordenada de atributos está incorreta. Se a ordem incorreta dos valores resulta em conflitos de tipos de dados e nenhuma conversão implícita é possível de ser realizada pelo sistema, a análise estática do comando pode detectar essa situação e retornar uma mensagem descrevendo o defeito.

Mesmo construindo um único mapeamento, alguns aspectos entre comandos de consulta e de mudança de estado são distintos. Por exemplo, os atributos do conjunto de dados definidos são qualificados pelo nome da relação, em adição ao nome de atributo propriamente dito. No tipo de defeito *[i_1]-1*, a operação de mudança de estado é executada para uma relação em vez de modificar o estado de outra relação, aplicando portanto a operação a uma lista de atributos incorreta. O mesmo raciocínio é estendido aos tipos de defeito *[d_1]-1* e *[u_1]-1*. Ainda, defeitos localizados na lista de atribuições de valores a atributos (item estrutural *[u_2]* na Tabela 3.2) do comando *update* poderiam alterar a lista de atributos do conjunto de dados definidos, ocasionando também falhas da categoria *lista de atributos*. Nos comandos de consulta, essa categoria de falha está relacionada a defeitos na lista ordenada de expressões para computar os valores dos atributos retornados.

A categoria de falha *ordem de atributos* é causada por um único tipo de defeito: *[s_1]-3*. Falhas desse tipo são difíceis para descobrir quando a ordem de domínios dos atributos não é alterada. Apesar de estar ligada a somente um tipo de defeito, não significa que se deva alocar pouco esforço de depuração para o defeito existente. Em geral, qualquer defeito pode ser propagado à saída da execução do comando e o dano causado pelo defeito é dependente do seu contexto de execução.

A categoria *ordem de tuplas* não considera qualquer mecanismo implícito de ordenação de dados. Por exemplo, algumas implementações automaticamente ordenam o conjunto de dados retornados a partir dos valores obtidos na lista de expressões usadas para agrupamento de dados. Se uma ordenação de *tuplas* é esperada no conjunto de dados retornados, então o comando deverá possuir explicitamente uma cláusula para a ordenação de dados. Os tipos de defeito *[s_6]-1* a *[s_6]-*

4 compartilham a mesma categoria de falha e a percepção de que seus efeitos são proporcionais à cardinalidade de *tuplas* do conjunto de saída esperado.

O tipo de defeito *[i_4]-1* está associado à categoria *lista de valores de atributos* desde que o defeito esteja localizado em uma subconsulta explícita, a qual é usada como fonte de dados para a(s) *tupla(s)* sendo inserida(s). Alguns programadores preferem usar subconsultas antes que operações de junção, mesmo que ambas possuam a mesma semântica. Subconsultas embutidas de forma implícita ou explícita em qualquer comando de manipulação devem ser analisadas na ótica de comandos de consulta, observando seus tipos de defeito e suas categorias de falha.

Um tipo de defeito inicialmente assumido durante o estudo semântico dos itens estruturais dos comandos de manipulação estava associado à ordem incorreta das expressões usadas para agrupamento de dados no comando *select*. Contudo, foi observado que tais expressões são usadas em conjunto como se elas estivessem concatenadas, e sua ordem não possuía relevância para computar os argumentos de agrupamento de dados. Esse raciocínio não é aplicável aos argumentos para ordenação de *tuplas*.

Os tipos de defeito atribuídos à categoria de falha *lista de atributos* poderiam também estar associados à categoria de falha *lista de valores de atributos*, visto que esses tipos de defeito podem resultar em valores indevidos nas *tuplas* do conjunto de saída (ou valores corretos para atributos indevidos). Tal consideração foi omitida do mapeamento de dados, pois a categoria *lista de atributos* está relacionada ao domínio de atributo e à semântica de atributo. A atenção primária do testador poderia ser a lista de atributos retornados ou definidos, antes que a lista de valores de atributos. Entretanto, o que comumente ocorre é a checagem única da lista de valores de atributos. É pertinente também estar atento ao significado do atributo e aos possíveis valores assumidos pelo atributo.

Tipos de defeito relativos a um mesmo item estrutural não compartilham os resultados da análise de propagação de defeitos, pois podem exercitar distintas categorias de falha. Este é o caso do item estrutural *[s_1]*, onde o tipo de defeito *[s_1]-3* produz uma particular categoria de falha, em contraste com os outros tipos de defeito desse item estrutural. No item *[s_2]*, um único tipo de defeito, *[s_2]-3*, está relacionado a múltiplas categorias de falha. Por outro lado, todos os tipos de defeito do item estrutural *[s_6]* compartilham a mesma categoria de falha. Embora itens estruturais não estejam ligados completamente à semântica das categorias de falhas, eles organizam os tipos de defeito em grupos e auxiliam a compreensão das falhas ocorridas.

No contexto de análise de comandos individuais, a propagação de defeito para a saída de comando é controlada pelo tipo de defeito e pelo comando defeituoso, em adição ao estado da base de dados. Por exemplo, considere o tipo de defeito *[s_4]-4* na Tabela 3.3. Dependendo do estado da base de dados, as categorias *número de tuplas* ou *lista de valores de atributos*, ou ambas, serão exercitadas pela execução do comando. Vale lembrar que o domínio de bases de dados é em geral infinito, o que realça a necessidade do uso de abordagens sistemáticas para o teste de aplicações de banco de dados.

As categorias de falha *número de tuplas* e *lista de valores de atributos* são causadas por uma grande diversidade de tipos de defeitos, que englobam comandos de consulta e de mudança de estado; mais que 50% dos tipos de defeito listados nas Tabelas 3.3 a 3.6 são a causa da manifestação dessas categorias de falha. Adicionalmente, alguns desses tipos de defeito estão associados a ambas categorias de falha, tal como *[s_4]-4*, *[d_2]-3*, *[u_3]-2*, e na maioria dos casos este fato é observado para o comando *select* (9 de 13). Esta pode ser uma evidência de que o comando *select* possui grande propensão a defeitos ligados a essas categorias, que poderiam demandar maior atenção do testador. Não obstante, quase todas as categorias de falha são exercitáveis por mais de um tipo de defeito. Tais características tornam pouco provável a localização do defeito no primeiro momento em que sua existência é descoberta. Isto corrobora que o caminho falha-para-defeito é não trivial e o conhecimento sobre os possíveis defeitos de manipulação é crucial para a programação SQL sem defeitos e para o teste de aplicações de banco de dados.

3.4 Cardinalidade, Diversidade e Nulidade

A seleção de bases de dados reveladoras de defeito de manipulação requer conhecimento sobre os tipos de defeitos e as categorias de falha em comandos de manipulação. Na evolução do mapeamento entre defeitos e falhas houve a construção de bancos de dados, visando a maximizar a lista de categorias de falha exercitáveis para cada tipo de defeito. Nesse sentido, foi observado que as bases de dados construídas para a execução de comandos defeituosos possuíam propriedades comuns para cada tipo de defeito, e que estas, quando presentes, aumentavam a

possibilidade de que falhas fossem manifestadas. Esta seção introduz tais propriedades, apresentando exemplos para ilustrar sua utilização.

Dentro do contexto de abstração de propriedades para bases de teste, Zhang et al. (2001) estudaram a geração de instâncias de banco de dados, a partir da semântica de comandos SQL embutidos em programas de aplicação; os autores reconhecem que poucos estudos têm sido dedicados à geração de dados para o teste de programas de aplicação de banco de dados que usam a SQL. O artigo descreve uma ferramenta de suporte que gera um conjunto de restrições, que representam, coletivamente, uma propriedade contra a qual o programa deve ser testado. Exemplos de propriedades para um determinado comando SQL de consulta a base de dados são: o comando retorna nenhuma linha e o comando produz valores numéricos negativos. A entrada da ferramenta é composta por um comando SQL, pela definição de esquema de banco de dados e por uma propriedade que representa o requisito para o testador. Se a propriedade puder ser satisfeita, obtêm-se instâncias de banco de dados, as quais devem estar consistentes com as restrições do esquema da base de dados. O trabalho é limitado a comandos de consulta e a restrições focadas no resultado da consulta: restrições simples de domínio para atributos de certos tipos de dados e restrições de cardinalidade de *tuplas*. Os autores reconhecem a impossibilidade da completa geração automática de bases de dados para teste, mas realçam a importância de iniciativas de suporte automatizado à geração de dados para o teste de aplicações de banco de dados.

Em nosso estudo, três propriedades foram identificadas como reveladoras da presença de defeitos de manipulação: *cardinalidade*, *diversidade* e *nulidade*. A primeira refere-se ao número de *tuplas* (linhas) das relações (tabelas) de entrada usadas na execução de um comando de manipulação; a segunda denota a heterogeneidade de valores para os atributos das relações de entrada; e a última atenta para a existência de valores nulos nos atributos das relações de entrada. Sobre tais propriedades, é importante ressaltar que, diferentemente da abordagem de Zhang et al. (2001), a referência de observação situa-se nos dados de entrada para a execução de comandos, que são aplicadas a comandos de consulta e de mudança de estado.

Para ilustrar as propriedades, considere o fragmento de código correto, sem defeitos, da ação de uma regra ativa da Figura 3.1. Nessa figura, o nó *i* denota uma consulta ao salário e ao departamento de alocação do empregado identificado pelo conteúdo da variável *v_emplno*; o nó *j* representa uma consulta ao salário atribuído ao gerente do departamento identificado pelo

conteúdo da variável *v_deptno*; um comando de mudança de estado é representado no nó *k*, onde ao salário do gerente do departamento identificado pelo conteúdo da variável *v_deptno* atribui-se o conteúdo da variável *v_salary*; e o nó *l* indica uma consulta aos empregados cujo salário é superior a 2000, restringindo o nome e o salário acrescido de 10%, ordenando os dados de modo ascendente por nome do empregado.

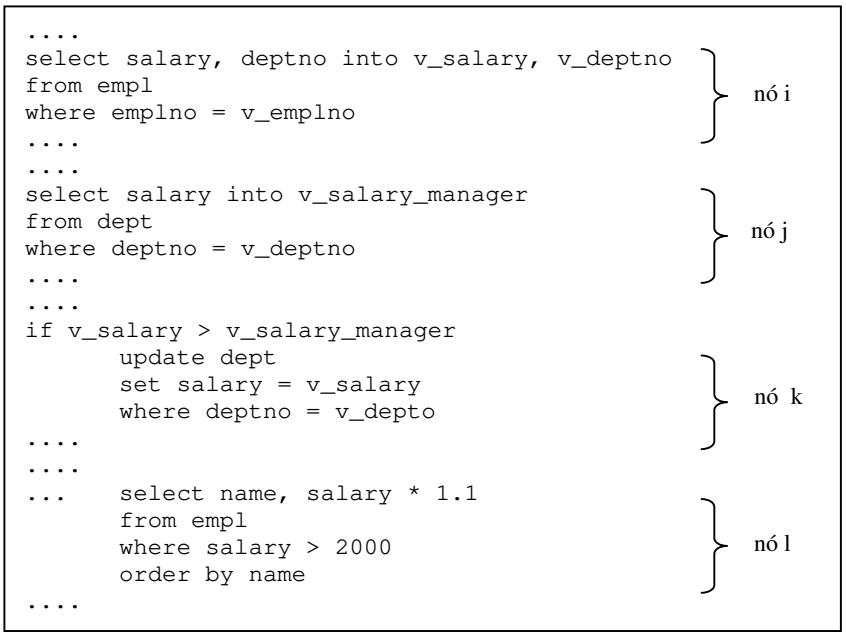


Figura 3.1: Fragmento de código da ação de uma regra ativa.

Seja uma versão incorreta do comando atribuído ao nó *i* do grafo da Figura 3.1:

select salary, deptno into v_salary, v_deptno

from empl, proj

where emplno = v_emplno

Um defeito do tipo *[s_2]-2* localiza-se na cláusula *from*, onde um nome de tabela está indevidamente presente na lista de nomes de tabelas usadas como fonte de dados. Para que uma falha seja manifestada devido à execução da versão incorreta do comando, é requerida que a relação *proj* possua a propriedade *cardinalidade* com valor "zero *tuplas*" ou "várias *tuplas*" (pelo menos duas *tuplas*), pois o número de *tuplas* do conjunto de saída do comando é proporcional à cardinalidade da relação *proj*. Seja uma outra versão incorreta do comando atribuído ao nó *i* da Figura 3.1:

select salary, deptno into v_salary, v_deptno
from empl
where emplno = v_emplno
and bonus is not null

Neste caso, é caracterizada a existência de um defeito do tipo *[s_3]-3*: o predicado para a seleção de *tuplas* está incorreto. A manifestação da falha requer a presença de *tuplas* na relação *empl* com diversidade de valores para o atributo *bonus*, incluindo valor nulo; em adição, se a propriedade *cardinalidade* para a relação *empl* possui o valor "várias *tuplas*", as chances para testador perceber a presença do defeito são aumentadas. Seja uma versão defeituosa do comando atribuído ao nó *l*:

*select name, salary * 1.1*
from empl
where salary > 1000
order by salary desc

Neste comando, é constatada a presença de defeitos do tipo *[s_3]-3* e *[s_6]-4*: o predicado para a seleção de *tuplas* está incorreto e um argumento incorreto está na lista ordenada de argumentos para ordenação de dados, respectivamente. A heterogeneidade de valores para os atributos *name* e *salary* eleva a chance da manifestação da falha e possivelmente aumenta a percepção do testador para a presença do defeito do tipo *[s_3]-3*. Em adição, A heterogeneidade de valores para o atributo *salary* e a propriedade *cardinalidade* com valor "várias *tuplas*" para a relação *empl* favorece a manifestação da falha atribuída ao defeito do tipo *[s_6]-4*. O comando abaixo representa uma versão defeituosa do comando referente ao nó *k*:

update dept
*set salary = v_salary * 1.1*
where deptno <> v_deptno

Defeitos do tipo *[u_2]-3* e *[u_3]-3*, uma expressão incorreta está presente no lado direito de uma atribuição de valor da lista de atribuições de valores a atributos da relação e o predicado para a seleção de *tuplas* está incorreto, respectivamente, estão presentes no comando acima. A cardinalidade "alguma *tupla*" (pelo menos uma *tupla*) e a diversidade de valores para os atributos *salary* e *deptno*, incluindo valores nulos se possíveis para tais atributos, representam critérios

para seleção de dados da relação *empl* que elevam a chance de manifestação de falhas na execução deste comando.

Valores nulos representam situações especiais para os atributos de um banco de dados. Segundo Elmasri e Navathe (2003), valores nulos possuem três diferentes interpretações: valor desconhecido, o valor existe, mas não é conhecido; valor não disponível, o valor existe, mas não pode ser obtido – valor retido; ou atributo não aplicável, valor indefinido para a *tupla*. Exemplos para tais interpretações são: a data de nascimento de um empregado é desconhecida, o telefone residencial de um empregado não foi divulgado e a data de egresso de um empregado ainda trabalhando na empresa é não aplicável, respectivamente. Em termos computacionais, valores nulos podem ser vistos como valores desconhecidos, onde em geral qualquer computação com valores desconhecidos produzirá um valor também desconhecido. Por exemplo, a expressão *(salary * 1.1 + bonus)* resultará em valor nulo se pelo menos um dos atributos envolvidos, *salary* ou *bonus*, possuir valor nulo; o predicado *(salary > 2000)* será avaliado como falso para todas as *tuplas* cujo valor do atributo *salary* é nulo.

A propriedade *nulidade* é aplicável aos atributos da(s) relação(ões) de entrada usados, diretamente ou indiretamente, na computação dos valores dos atributos da relação de saída; tais atributos, também denominados de *atributos manipulados*, são os atributos da relação de entrada referenciados explicitamente ou implicitamente no comando de manipulação ou listados diretamente como atributos do conjunto de saída. Por denotar situações especiais e com aspectos computacionais próprios, a ocorrência de valores nulos representa uma oportunidade para a introdução de defeitos. Algumas dessas oportunidades envolvendo os atributos manipulados são: expressões aritméticas e lógicas localizadas em qualquer item estrutural de comando de manipulação; predicados para a seleção de dados; e funções de sumarização. Tipicamente, sistemas gerenciadores de banco de dados disponibilizam recursos para tratar valores nulos em comandos de manipulação, possibilitando que a presença de tais valores seja prevista pelo programador; como exemplo, são citados: funções embutidas que substituem valores nulos, cláusulas dedicadas a predicados que indagam a existência de valores nulos e mecanismos de sumarização que ignoram *tuplas* que possuem valores nulos em certos atributos.

Durante a construção do mapeamento entre tipos de defeitos e categorias de falha, foi observado que defeitos ligados à presença de valores nulos podem estar situados na maioria (10 de 15) dos itens estruturais dos comandos de manipulação. Para o comando *select*, identificou-se

essa possibilidade para os itens estruturais *[s_1]*, *[s_3]*, *[s_4]*, *[s_5]* e *[s_6]* e para os comandos de mudança de estado foram relacionados os itens *[i_3]*, *[i_4]*, *[d_2]*, *[u_2]* e *[u_3]*. Excetuando-se a categoria de falha *ordem de atributos*, as demais categorias de falha são exercitáveis por defeitos atribuídos à existência de valores nulos. Dessa forma, requerer valores nulos para os atributos manipulados pela execução de comandos constitui uma diretriz pertinente para revelar a presença de defeitos de muitos dos tipos listados para os comandos de manipulação.

A diversidade de valores para os dados de teste foi tratada por Chays et al. (2000), em sua abordagem para a geração de bases de dados visando ao teste de aplicações de banco de dados que usam a SQL. Relações da base de dados são povoadas com dados que respeitam as restrições de integridade e que são interessantes, segundo julgamento do testador. Tais dados são obtidos a partir de indicações do próprio testador, pelo fornecimento de amostras de dados organizadas em categorias. Casos de teste são então construídos a partir da combinação dos valores das várias categorias. É enfatizada a importância de dados que cubram uma variedade de diferentes características das entidades da base de dados, mas não existe uma sistemática para medir a diversidade de dados, nem uma política para garantir um grau mínimo de diversidade.

Similarmente à propriedade *nulidade*, a propriedade *diversidade* é aplicável aos atributos manipulados. A heterogeneidade de valores para os atributos manipulados, em adição à observância das cardinalidades “zero tuplas”, “alguma tupla” e “várias tuplas” para a(s) relação(ões) de entrada, possibilitam que diversos cenários de execução sejam aplicados a um dado comando de manipulação, elevando a possibilidade de manifestação da falha e da percepção do testador para essa manifestação. Na construção do mapeamento entre defeitos e falhas, pode-se afirmar que a propriedade *diversidade* pode apoiar a ocorrência de falhas ocasionadas por defeitos ligados a todos os tipos listados na Seção 3.2 para os comandos de manipulação. Em adição, foi percebida a necessidade de uma medida para apreciar o nível de adequação da diversidade aplicada às execuções de cada comando defeituoso; tal medida seria computada para cada base de dados (caso de teste) e para o conjunto de bases de dados utilizadas na execução de um dado comando (conjunto de casos de teste).

3.5 Considerações Finais

Este capítulo discutiu aspectos ligados a defeitos e falhas de manipulação, e apresentou o resultado de uma investigação sobre o relacionamento entre tipos de defeito e categorias de falha. As construções básicas de comandos SQL de manipulação foram examinadas e organizadas em itens estruturais, com a intenção de agrupar tipos de defeito. O conceito de falha de manipulação foi estudado, introduzindo-se dois conjuntos de dados que capturam a noção de saída da execução de comandos: conjunto de dados retornados, o qual está relacionado a comandos de consulta (comando *select*) e conjunto de dados definidos, que se refere a comandos de mudança de estado (comandos *insert*, *delete* e *update*). Categorias de falha foram caracterizadas em nível de comandos individuais, permitindo o entendimento de como defeitos de manipulação são propagados para a saída de execução de comandos. Uma lista de tipos de defeitos de manipulação foi apresentada para cada item estrutural, incluindo exemplos de comandos corretos e defeituosos.

Um mapeamento entre tipos de defeitos e categorias de falha de manipulação foi construído a partir da execução de um conjunto de comandos defeituosos, visando a exercitar todos tipos de categorias de falha para cada tipo de defeito. Para a execução de tais comandos, foi necessário construir bases de dados de entrada. Foram abstraídas algumas propriedades atribuídas a essas bases de dados: cardinalidade, diversidade e nulidade; observou-se que a presença dessas propriedades contribui para a descoberta de defeitos.

A análise do mapeamento entre defeitos e falhas de manipulação aponta que: *i)* existe um relacionamento muitos-para-muitos entre defeitos e falhas de manipulação, o que dificulta a identificação do tipo de defeito a partir do conhecimento da categoria de falha ocorrida; *ii)* a manifestação das categorias de falha é dependente do tipo de defeito, do comando defeituoso e da base de dados de entrada; *iii)* as propriedades cardinalidade, diversidade e nulidade representam requisitos reveladores de defeito na geração de bases de dados de entrada; e *iv)* o conhecimento de defeitos e falhas de manipulação é crucial para a programação SQL e para o teste de aplicações de banco de dados.

Algumas contribuições deste capítulo são: introdução de uma lista de tipos de defeito de manipulação, representando o primeiro elemento de uma investigação na direção de se propor uma abordagem sistemática de teste para aplicações de banco de dados; apresentação do conceito

de saída da execução de comandos de manipulação, buscando capturar dados suficientes para o projeto de casos de teste de comandos SQL; proposição de um elenco de categorias de falha de manipulação, constituindo um recurso para a determinação se a saída produzida pela execução de casos de teste está correta; construção e validação preliminar de um mapeamento entre tipos de defeito e categorias de falha, aumentando o conhecimento entre defeitos e falha de manipulação para as atividades de teste e de depuração de aplicações de banco de dados; e abstração de propriedades potencialmente reveladores de defeito, no que diz respeito às bases de dados de entrada.

Capítulo 4

Interação entre Regras Baseada em Fluxo de Dados

Neste capítulo é investigada a interação entre regras e estabelecido um modelo de interação que serve de base aos critérios propostos no Capítulo 5. Tal modelo é explorado na definição de associações de interação entre regras escritas em SQL. Vários tipos de interação entre regras ativas são estudados, baseando-se no fluxo de dados persistentes entre componentes de regras não necessariamente distintas. A natureza persistente dos dados possibilita a existência de relações de fluxo de dados entre regras, independente da ocorrência de disparo entre regres; em adição, o disparo entre regras estabelece novos fluxos de dados devido à transferência de controle entre regras.

A precisão de fluxo de dados tem sido abordada pela literatura para, por exemplo, tratar variáveis agregadas e do tipo ponteiro (Horgan e London, 1991; Ostrand e Weyuker, 1991; Vilela et al., 1997), utilizando a conjectura de que a análise de fluxo de dados mais precisa aumenta a eficácia do teste (Marx e Frankl, 1996). A precisão de fluxo de dados persistentes é uma preocupação recente no teste de aplicações de banco de dados (Kapfhammer e Soffa., 2003; Leitão et al., 2002). Neste capítulo explora-se tal questão e estabelece-se um modelo de fluxo de dados para comandos de manipulação da SQL, onde associações persistentes são caracterizadas pela ocorrência desses comandos e sua cobertura pode ser alcançada segundo vários níveis de precisão.

4.1 Fluxo de Dados Persistentes

Similarmente a variáveis de programa, as ocorrências de variáveis ligadas a objetos (entidades) de banco de dados – variáveis ligadas a dados persistentes, também denominadas *variáveis persistentes* ou *variáveis de banco de dados* – são classificadas como *definição* e *uso*. Devido à natureza persistente, não se observa indefinição de dados e toda definição e uso são globais aos programas de aplicação: os estados das variáveis são mantidos após a execução do programa e permanecem disponíveis para qualquer programa de aplicação. A notação *ddef* é aplicada para a *definição de dados persistentes*, ou simplesmente *definição persistente*: *ddef(i) = { variáveis v | v é uma variável de banco de dados que é definida no nó i }*. A definição de uma variável persistente é uma *definição persistente*.

O uso persistente ocorre em algum nó de manipulação de dados da SQL e pode afetar o fluxo de controle nos arcos de saída desses nós, quando houver tratamento de exceção (Spoto, 2000). Esse enfoque é enriquecido pela inclusão de ocorrências de comandos de seleção de fluxo de controle de programa, os quais podem possuir uso de dados persistentes, conforme exemplificado abaixo:

if exists (select id from empl where emplno = :emplno)

No exemplo acima, o predicado de seleção de fluxo utiliza uma consulta a base de dados, ocasionando que o uso de dados persistentes é diretamente determinante na escolha do próximo comando a ser executado. Assim, usos persistentes ocorrem nos arcos de saída dos nós da SQL que realizam consulta implícita ou explícita a dados persistentes. A notação *duso* é aplicada para o *uso de dados persistentes*, ou simplesmente *uso persistente*: *duso(i,j) = { variáveis v | v é uma variável de banco de dados que é usada no arco (i,j) }*. O uso de uma variável persistente é um *uso persistente*.

Uma *associação de fluxo de dados persistentes*, *associação persistente*, *associação-ddef-duso*, ou simplesmente *ddua*, é dada pela tripla *[i, (j,k), v]*, onde: $v \in ddef(i)$; $v \in duso(j,k)$; e existe um caminho livre de definição c.r.a *v* de *i* até *(j,k)*. Equivalentemente, se *ddu(v,i)* = { arcos *(j,k)* | $v \in ddef(i)$, $v \in duso(j,k)$ e existe um caminho livre de definição c.r.a *v* de *i* até *(j,k)* }, uma associação persistente é dada pela tripla *[i, (j,k), v]*, onde $(j,k) \in ddu(v,i)$.

4.1.1 Granularidade de Fluxo de Dados Persistentes

Relações de fluxo de dados ligadas a entidades de banco de dados são estabelecidas a partir de níveis de granularidade das variáveis de banco de dados: a definição e o uso posterior são observados para a mesma granularidade (Daou et al., 2001). A *granularidade* é uma propriedade que determina o nível de precisão da análise de fluxo de dados e estabelece as relações de fluxo de dados das variáveis persistentes. Segundo Kapfhammer e Soffa (2003), a granularidade pode ser definida segundo vários níveis: *banco de dados*, *relação* (tabela), *atributo* (coluna), *tupla* (linha) e *valor de atributo*.

No nível de granularidade *relação* mapeiam-se as variáveis persistentes em relações do banco de dados, não importando quais *tuplas* ou atributos foram afetados por operações de manipulação. Esse enfoque simplifica a análise e reduz o número de variáveis, pois considera cada relação como uma única variável. Contudo, estabelece uma abordagem conservadora, pois toda definição de variável seguida de uso dessa variável é uma associação de fluxo de dados, independentemente se estão manipulando dados distintos dentro da relação. Cardoso (2004) construiu uma ferramenta para a aplicação de critérios baseados em fluxo de dados em regras ativas escritas em SQL e utilizou a granularidade *relação* em sua implementação.

No nível de granularidade *atributo* exploram-se as ocorrências de definição e de uso para os atributos de relações, sem se preocupar em saber quais *tuplas* foram manipuladas. Esta granularidade representa uma maior precisão do que o nível *relação*, pois distingue os atributos de cada relação; associações de fluxo de dados podem ser caracterizadas quando a interseção dos dois conjuntos de colunas manipuladas, respectivamente, atribuídos a uma ocorrência de definição seguida de uma ocorrência de uso, não for vazia. Uma vantagem para essa abordagem é que o número de colunas de uma relação é fixo e pode-se determinar suas ocorrências de definição e de uso estaticamente. Aranha (2005) usa essa abordagem para definir requisitos de teste de esquema de bases de dados relacionais, onde os casos de teste buscam exercitar atributos e restrições associadas a atributos. Daou et al. (2001) propõem uma estratégia para o teste de regressão de aplicações de banco de dados e usam a granularidade em nível coluna para determinar os módulos de banco de dados afetados por modificações na definição de componentes de banco de dados.

O nível de granularidade *tupla* é adotado quando se busca refinar a análise de fluxo de dados para saber quais as *tuplas* afetadas em cada operação de manipulação de dados. Em geral, não se pode determinar estaticamente que *tuplas* serão alvo de uma dada operação, devido à complexidade que pode ser atingida nos predicados de seleção de linhas (Vaduva, 1999) e à dependência ao estado de banco de dados corrente (Leitão et al., 2002); ou seja, esta é uma questão indecidível. No contexto de bancos de dados relacionais, em geral uma *tupla* representa uma entidade do mundo real, uma associação entre entidades ou algum aspecto particular de uma entidade. Sendo assim, utilizar granularidade em nível linha significa atingir as entidades do mundo real manipuladas por operações de banco de dados.

Os níveis de granularidade *banco de dados* e *valor de atributo* representam lados antagônicos de precisão de análise de fluxo de dados. No primeiro, todo o banco de dados é convencionado como uma única variável e as ocorrências de definição e de uso podem ser conhecidas por análise estática. O último é o nível com maior precisão de análise de fluxo de dados e representa uma composição dos níveis *tupla* e *atributo*, herdando a indecidibilidade na identificação de *tuplas* inerente à granularidade em nível *tupla*.

Leitão et al. (2002) propuseram preliminarmente um modelo de instrumentação de *tuplas*, capaz de abstrair a identificação das *tuplas* afetadas pela execução de operações de manipulação de dados. Tal modelo representa a primeira iniciativa que trata conjuntos de dados afetados por operações da SQL, utilizando as próprias estruturas de dados do modelo relacional para armazenar os dados atribuídos à instrumentação de *tuplas*. A evolução deste modelo é descrita no Capítulo 6. Em (Spoto, 2000), para cada comando que caracteriza uma definição (persistente) de tabela, existe um comando específico para indicar a *tupla* utilizada; entretanto, a implementação desse esforço na direção da granularidade em nível *tupla* utiliza variáveis de programa para a identificação da linha afetada por um comando.

Kapfhammer e Soffa (2003) definem critérios de teste baseados em análise de fluxo de dados para aplicações de banco de dados explorando os vários níveis de granularidade; são introduzidas extensões ao critério *todos-usos* para dados persistentes no contexto do modelo relacional. Nesta abordagem, o elenco de associações de fluxo de dados é dependente do estado inicial da base de dados: o tamanho da base de dados afeta a determinação de elementos requeridos, pois associações são abstraídas para toda entidade possível da base de dados. Para as granularidades *tupla* e *valor de atributo*, é estabelecida uma abordagem conservadora para

enumerar os objetos de banco de dados que podem ser definidos e usados. Os resultados do estudo empírico realizado com dois programas escritos em linguagem *Java* utilizando o sistema de banco de dados *MySQL* são: o número de elementos requeridos tende a ser elevado para os níveis de granularidade *tupla* e *valor de atributo*, mesmo usando bases de dados reduzidas; apesar de alguns desses elementos requeridos já serem exercitados no teste de adequação ao critério *todos-usos* focado somente nas variáveis de programa, algumas associações de fluxo de dados persistentes requerem casos de teste adicionais. Em síntese, o número de elementos requeridos aumenta à medida em que se aplica mais precisão na análise de fluxo de dados e casos de teste específicos são necessários ao exercício de associações de fluxo de dados persistentes.

A cobertura de associações de fluxo de dados persistentes é afetada pela granularidade da análise de fluxo de dados adotada. A cobertura de uma *associação-ddef-duso* em nível de granularidade β é alcançada quando a interseção dos conjuntos *ddef* e *duso* estabelecidos em nível β é distinta do conjunto vazio e pelo menos um caminho livre de definição c.r.a variável persistente for executado. Nesse sentido, adotou-se uma *abordagem precisa* para a noção de redefinição de dados persistentes: a redefinição persistente ocorre em nível *valor de atributo*, independentemente da granularidade utilizada para a análise de cobertura da associação persistente. A motivação para esta abordagem é a determinação exata dos dados definidos que alcançam os usos persistentes, para então efetuar a avaliação de cobertura em cada granularidade de fluxo de dados.

Uma *ddua* denotada por *[i, (j,k), v]* é coberta para a granularidade β se existem dados de entrada capazes de executar pelo menos um caminho completo que inclua algum caminho π iniciado no nó *i* e terminado no arco *(j,k)*, desde que: *(i)* $v \in ddef(i) \cap duso(j,k)$ para a granularidade β; e *(ii)* π seja um caminho livre de definição c.r.a *v* considerando a abordagem precisa de redefinição de dados. Associações que são exercitadas para uma dada granularidade podem não estar sendo cobertas em outras granularidades. Por exemplo, considere os comandos abaixo, onde *(1)* e *(2)* representam definição e uso de dados persistentes, respectivamente:

(1) *update empl*
*set bonus = bonus * 1.1*
where salary < :var_1

(2) *select emplno from empl*
where salary > :var_2

O conteúdo das variáveis *var_1* e *var_2* são determinantes para a cobertura de associações estabelecidas a partir dos comandos *(1)* e *(2)*. Para a precisão de fluxo de dados *tupla*, é requerido

que a definição e o uso ocorram em pelo menos uma mesma *tupla*; ou seja, o predicado *(var_2 < var_1)* deve ser verdadeiro para que possam existir relações *empl* capazes de atender tal requisito; para a granularidade *relação*, a qual é menos exigente do que a precisão *tupla*, este predicado não é exigido.

Uma questão pertinente é se a cobertura de associações persistentes em granularidades mais precisas agrega valor à habilidade de descoberta de defeitos. Nesse sentido, o Capítulo 5 explora o requisito de teste *(critério, granularidade)*, investigando empiricamente se as granularidades *relação*, *atributo*, *tupla* e *valor de atributo* são suplementares à descoberta de defeitos no contexto de regras ativas escritas em SQL.

4.1.2 Fluxo de Dados em Comandos de Manipulação

Na análise de fluxo de dados para variáveis persistentes, é mandatório basear-se em alguma granularidade para a determinação dos dados definidos e usados em conseqüência de execuções dos comandos de manipulação. A Tabela 4.1 apresenta uma análise das ocorrências de definição e de uso para tais comandos, com inspiração nas proposições de Leitão et al. (2002). A primeira coluna indica o comando de manipulação; as colunas segunda e terceira determinam, de acordo com a granularidade, os dados definidos no nó de ocorrência do comando e os dados usados nos arcos de saída do nó de ocorrência do comando, respectivamente. O termo *relação consultada* denota uma relação usada como fonte de dados em uma operação de consulta; o termo *relação definida* refere-se à relação que possivelmente será atualizada por uma operação de mudança de estado. São referenciados conjuntos de dados, seguindo a nomenclatura B_g, onde: B assume a notação D e U para os conjuntos de dados definidos e usados, respectivamente; e g assume a notação b, r, a, t e va para as granularidades *banco de dados*, *relação*, *atributo*, *tupla* e *valor de atributo*, respectivamente.

O conteúdo da Tabela 4.1 possui diferenças em comparação a trabalhos anteriores. Spoto (2000) considera suas análises em nível *relação* e determina que todos comandos de mudança de estado – *insert*, *delete* e *update* – possuem definição e usos persistentes, de forma independente do predicado de seleção de *tuplas*; também inclui o comando *create view*, o qual não constitui um comando de manipulação, como ocorrência de definição persistente. No modelo de fluxo de dados persistentes de Kapfhammer e Soffa (2003), os tipos de ocorrências de variáveis persistentes nos comandos *delete* e *update* são dependentes da granularidade de fluxo de dados;

por exemplo, se a cláusula *where att_1 > 200* for utilizada em um comando *update*, definição e uso persistentes ocorrem na granularidade *atributo*, mas na granularidade *relação* existe somente definição persistente.

Tabela 4.1: Análise das ocorrências de definição e de uso para os comandos SQL de manipulação.

Comando	**Definição Persistente**	**Uso Persistente**
Select		*(U_b)* Bases de dados que possuem as relações consultadas *(U_r)* Relações consultadas *(U_t) Tuplas* manipuladas nas relações consultadas *(U_a)* Atributos das relações consultadas referenciados, implicitamente ou explicitamente, no comando de manipulação *(U_{va})* Dados dos atributos em U_a pertencentes às *tuplas* em U_t
Insert	*(D_b)* Base de dados que possui a relação definida *(D_r)* Relação definida *(D_t) Tuplas* inseridas na relação definida *(D_a)* Atributos da relação definida *(D_{va})* Dados dos atributos em D_a pertencentes às *tuplas* em D_t	
Update	*(D_b)* Base de dados que possui a relação definida *(D_r)* Relação definida *(D_t) Tuplas* modificadas na relação definida *(D_a)* Atributos da relação definida que são referenciados no lado esquerdo de uma atribuição na lista de atribuições de valores a atributos *(D_{va})* Dados dos atributos em D_a pertencentes às *tuplas* em D_t	*(U_b)* Base de dados que possui a relação definida *(U_r)* Relação definida *(U_t) Tuplas* modificadas na relação definida *(U_a)* Atributos da relação definida que são referenciados, implicitamente ou explicitamente, no comando de manipulação *(U_{va})* Dados dos atributos em U_a pertencentes às *tuplas* em U_t
Delete	*(D_b)* Base de dados que possui a relação definida *(D_r)* Relação definida *(D_t) Tuplas* excluídas da relação definida *(D_a)* Atributos da relação definida *(D_{va})* Dados dos atributos em D_a pertencentes às *tuplas* em D_t	*(U_b)* Base de dados que possui a relação definida *(U_r)* Relação definida *(U_t) Tuplas* excluídas da relação definida *(U_a)* Atributos da relação definida que são referenciados, implicitamente ou explicitamente, no comando de manipulação *(U_{va})* Dados dos atributos em U_a pertencentes às *tuplas* em U_t

Sobre a cobertura de associações persistentes em uma dada granularidade, considere a Figura 4.1, que ilustra um hipotético fluxo de controle para um fragmento de código em SQL

para manipular a relação *empl* do banco de dados *company*, a qual é descrita pelo esquema *empl (id, name, salary, bonus)*. Neste exemplo, são abstraídas as associações persistentes α_1, α_2 e α_3, especificadas, respectivamente, como *[k, (m, n), empl]*, *[k, (n, succ(n)), empl]* e *[m, (n, succ(n)), empl]*, onde *succ(n)* representa um nó sucessor do nó *n.* Pode-se afirmar: as associações α_1 e α_2 somente poderão ser cobertas nas precisões *tupla* e *valor de atributo* se, quando o controle alcançar o nó *k*, existirem *tuplas* na relação *empl* que atendam aos requisitos dessas granularidades; contudo, não existem bases de dados de entrada que ocasionem a cobertura da associação α_3 em tais granularidades.

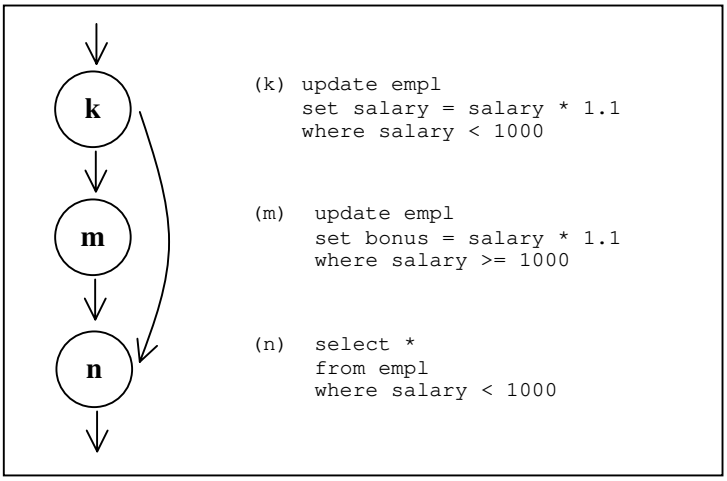


Figura 4.1: Exemplo de fluxo de controle de um fragmento de código com comandos SQL.

Para complementar a análise de fluxo de dados persistentes, vale considerar um tipo especial de uso persistente, denominado de *uso persistente implícito* ou simplesmente *uso implícito*. Esta categoria de uso existe na execução de qualquer operação de mudança de estado, pois em geral o sistema gerenciador de banco de dados realiza uma verificação para determinar se restrições de entidade podem estar sendo violadas; por exemplo, se foi realizada uma tentativa para a inserção de valores duplicados da chave primária. Nesse sentido, todo o conteúdo da relação é usado, visto que restrições são verificadas para o conjunto de entidades da relação. Como conseqüência, associações de fluxo de dados podem ser abstraídas para esta classe de uso: se uma operação mudança de estado efetua uso implícito do conjunto de dados de uma relação, então este uso pode ser afetado por qualquer definição anterior desta relação; tais associações são mencionadas como *associações persistentes implícitas*, *associações implícitas* ou *dduas*

implícitas. Duas ocorrências de operação de mudança de estado, não necessariamente distintas, podem estabelecer uma *relação persistente implícita*, caracterizando uma associação de fluxo de dados devido ao uso implícito de dados persistentes.

Usos implícitos não estão no foco desta pesquisa, não sendo, portanto, considerados na abstração de associações persistentes introduzidas na Seção 4.4. A despeito disso, abaixo é apresentada uma estratégia para a aplicação de associações implícitas, ressaltando suas limitações:

- a cobertura de associações implícitas é alcançada nos níveis de granularidade *banco de* dados e *relação*, podendo-se determinar estaticamente os caminhos para cobrir as associações; a cobertura em níveis de granularidade mais precisos é indiferente, pois se todas as *tuplas* são usadas, então qualquer *tupla* definida anteriormente será suficiente para o exercício da associação;
- se a aplicação do caso de teste λ exercita os caminhos determinados estaticamente para cobrir a associação α, e a execução da operação do nó de definição não afeta qualquer *tupla* (ou seja, nenhuma *tupla* é definida), α será coberta por λ, pois é considerado que as *tuplas* sendo implicitamente usadas são as mesmas que persistiram após a execução da operação de mudança de estado.

4.2 Definições para Regras Ativas Escritas em SQL

Esta seção trata aspectos pertinentes a regras ativas escritas em SQL, para os quais não se encontrou definição explícita na literatura. São explorados os conceitos: evento, interface, estado de iniciação, caso de teste, seqüência e conjunto de casos de teste. As definições são realizadas no contexto de sistemas de banco de dados relacional, e delineiam o modelo de regras ativas para o padrão SQL3.

Definição: ***Evento de Regra Ativa***

O *evento* (ou *evento de disparo*) de uma regra ativa *r* é descrito pela quádrupla $E(r) = \langle T, \{ \varphi_1, \varphi_2, ..., \varphi_n \}, GT, TD \rangle$, onde: *T* é a identificação do esquema atribuído à relação *t* cuja mudança de estado dispara a execução de *r*; $\{ \varphi_1, \varphi_2, ..., \varphi_n \}$ é o conjunto que descreve os

tipos de operação de mudança de estado, também denominada *operação de disparo*, onde a execução de uma operação de quaisquer desses tipos, aplicada à relação *t*, pode provocar o disparo de *r*; *GT* representa a propriedade *granularidade de transição*, que indica se a regra será disparada para cada *tupla* afetada por sua operação de disparo (*row-level trigger*), ou se a regra será disparada uma única vez para o conjunto de dados afetados pela operação de disparo (*statement-level trigger*); e *TD* representa a propriedade *tempo de disparo*, que indica se a regra será disparada antes ou após a execução da operação de disparo (*before trigger* e *after trigger*, respectivamente), ou se a regra será disparada em substituição à operação de disparo (*instead of trigger*).

Por exemplo, na declaração:

create trigger trigger_employee
before insert, update of salary, update of bonus
on employee
for each row
begin
.....
end;

o evento de disparo da regra *trigger_employee*, dado por $E(trigger\_employee) = \langle employee, \{ insert, update\ of\ salary, update\ of\ bonus \}, row\text{-}level\ trigger, before\ trigger \rangle$, é provocado quando qualquer operação do tipo *insert*, *update of salary* ou *update of bonus* for aplicada à relação *employee*; a propriedade *granularidade de transição* aponta que a regra será disparada para cada linha afetada pela operação de disparo; e a propriedade *tempo de disparo* indica que a regra será disparada antes que a mudança de estado, devido à execução da operação de disparo, seja aplicada à base de dados.

Definição: ***Interface de Regra Ativa***

A *interface* de uma regra ativa *r* é representada pela descrição de evento e pela descrição de sua condição. De modo geral, é dada pela quíntupla $I(r) = \langle T, \{ \varphi_1, \varphi_2, ..., \varphi_n \}, GT, TD, C \rangle$, onde *C* refere-se ao predicado associado à condição da regra; note que os quatro primeiros elementos da quíntupla descrevem o evento de regra: $I(r) = \langle E(r), C \rangle$. A

inclusão da parte condição de regra na representação de interface ocorre para fins de teste em modo imediato para o acoplamento evento-condição. Estratégias de teste que buscam revelar defeitos na interface de regra devem requerer casos de teste que executem a ação de regra (a condição é avaliada como verdadeira) e que não executem a ação de regra (a condição é avaliada como falsa).

Definição: ***Variável de Regra Ativa***

No contexto de regras ativas escritas em SQL, o termo *variável de regra ativa* refere-se aos seguintes tipos: *(i) variáveis locais*: são as variáveis explicitamente declaradas pela cláusula *declare* e possuem escopo limitado ao bloco *b* em que foram declaradas (e blocos internos a *b*); *(ii) variáveis de sistema*: são variáveis mantidas pelo sistema e disponíveis para uso pelos processos de bancos de dados; podem representar dados com escopo limitado ao processo em execução (por exemplo, estado de exceção da última manipulação de dados executada) ou denotar informação global a todos os processos (por exemplo, número de conexões correntes com o banco de dados); *(iii) variáveis persistentes (variáveis de banco de dados)*: são variáveis referentes a dados ou meta-dados de um banco de dados; são ditas persistentes pois seus valores mantêm-se após o término do processo que as definiu; *(iv) variáveis de transição*: são variáveis relativas aos dados afetados pela operação que provocou o evento de disparo da regra, possuindo escopo limitado à regra; sua estrutura é do tipo *tupla* ou relação; no primeiro caso, a regra é disparada para cada linha afetada pela operação de disparo da regra; no segundo caso, o seu valor possui o conjunto de linhas afetadas pela operação de disparo da regra, quando a regra é disparada uma única vez independentemente da quantidade de linhas afetadas; e *(v) variáveis cursor*: são variáveis associadas à facilidade de manipulação *tupla-por-tupla* do conjunto resultante de uma consulta (operação de recuperação de dados persistentes); a exemplo das variáveis locais, variáveis do tipo *cursor* devem ser explicitamente declaradas pela cláusula *declare*.

A Seção 4.1 introduziu associações de fluxo de dados oriundas de variáveis persistentes referentes a dados (bases de dados). No contexto deste trabalho, a intenção é propor e avaliar a aplicação de técnicas de teste baseadas em tais associações para regras escritas em SQL;

especificamente, serão estudadas as interações baseadas em fluxo de dados entre regras ativas. *Variáveis locais* e *variáveis de transição* possuem escopo local e são abordadas por Cardoso (2004) em sua ferramenta para o teste de regras isoladas. Apesar de serem tipicamente locais às regras, variáveis *cursor* possuem manipulação persistente em tempo de iniciação do cursor; neste caso, a abertura do cursor representa um comando de manipulação do tipo *select*.

Definição: ***Estado de Iniciação de Regra Ativa***

A execução de regras ativas é realizada pelo sistema gerenciador de bases de dados, de acordo com o seu mecanismo próprio de detecção de eventos ocorridos e de escalonamento de regras disparadas. Ao ser executada uma regra, é atribuído um estado inicial para a regra, denominado de *estado de iniciação*. Dentre os fatores associados ao estado de iniciação, são ressaltados: *(i) os dados afetados pela operação que provocou o disparo da regra, denominados de dados de transição*; tais dados representam a mudança de estado que provocou o disparo da regra e estão disponíveis localmente à regra em tempo de sua execução; podem ser explicitamente referenciados no código da condição e da ação da regra; por exemplo, na instrução *IF (:new.salary > :old.salary)*, o predicado avalia se a operação de disparo elevou o valor do atributo *salary*; *(ii) o tipo de operação que provocou o evento de disparo da regra*; este dado pode compor um predicado no código da condição e da ação da regra; por exemplo, se *{ insert, update(salary) , update(bonus) }* descreve os tipos de operação de disparo de uma regra, então a instrução *IF UPDATING(SALARY)* poderia ser usada para determinar precisamente o tipo de operação que provocou o disparo da regra; e *(iii) a base de dados no momento de consideração do evento, da condição e da ação da regra*; representa o contexto de pertinência para cada componente de regra; por exemplo, a base de dados disponível durante a avaliação da condição da regra representa um estado anterior à operação que provocou o evento de disparo da regra (neste caso, a mudança de estado atribuída à operação de disparo ainda não foi aplicada à base de dados).

Os dados mencionados em *(i)* dependem do estado da base de dados no momento da operação de disparo, da semântica da operação de disparo e da propriedade *granularidade de transição* da regra. O componente *(ii)* é um atributo da operação de disparo. Os dados disponíveis

no componente *(iii)* dependem do estado da base de dados no momento de execução da operação de disparo, da semântica da operação de disparo e da propriedade *tempo de disparo* da regra.

Os espaços de entrada e de saída no teste de programas de aplicação de banco de dados incluem os estados da base de dados, em adição aos parâmetros de entrada e de saída da aplicação. Isto tem um substancial impacto na noção do que é um caso de teste, de como gerar casos de teste e de como verificar os resultados produzidos pela execução dos casos de teste (Chays et al., 2000).

Definição: ***Caso de Teste de Regra Ativa***

Um *caso de teste* λ para uma regra ativa *r* é dado pela quádrupla $\langle \theta, \Delta_0, \Delta_1, M \rangle$, onde: θ é a operação que provocou o evento de disparo de *r*; Δ_0 é o estado da(s) base(s) de dado(s) antes da execução de θ (e de *r*); Δ_1 é o estado da(s) base(s) de dado(s) após a execução de *r*; e *M* é a seqüência $\langle m_1, m_2, ..., m_k \rangle$, k ≥ 0, de mensagens emitidas durante a execução de *r*. Assim, pode-se dizer que a entrada de um caso de teste é dada por $\langle \theta, \Delta_0 \rangle$ e a sua saída é dada por $\langle \Delta_1, M \rangle$. Se a execução de *r* resultar no disparo de outras regras (ou da mesma regra), então Δ_1 e *M* referem-se ao obtido pela seqüência de regras executadas a partir de *r*.

Em regras cuja propriedade *tempo de disparo* indicar que a regra será disparada após a aplicação da operação de disparo à base de dados, é esperado que os estados anterior e posterior à aplicação de um caso de teste sejam distintos ($\Delta_0 \neq \Delta_1$), devido à natureza de mudança de estado da operação de disparo. Em adição, a execução das operações de manipulação de dados persistentes, que estão presentes na ação da regra, podem também realizar mudança de estado na base de dados. Se $\Delta_0 = \Delta_1$, a execução da regra foi possivelmente abortada, cancelando as operações de mudança de estado aplicadas à base de dados ocorridas na transação de execução da regra.

Definição: ***Seqüência de Casos de Teste***

Uma *seqüência de casos de teste* para uma regra ativa *r*, denotada por $\langle \lambda_1, ..., \lambda_t \rangle$, é descrita pela quádrupla $\langle \langle \theta_1,..., \theta_t \rangle, \Delta_0, \langle \Delta_1, ..., \Delta_t \rangle, \langle M_1, ..., M_t \rangle \rangle$, onde: $\langle \theta_1,..., \theta_t \rangle$

é a seqüência de operações de disparo de *r*, tal que θ_k representa a operação de disparo de *r* em λ_k; Δ_0 é o estado inicial da(s) base(s) de dados antes da aplicação da seqüência de casos de teste; $<\Delta_1, ..., \Delta_t>$ é a seqüência de estados da(s) base(s) de dados, tal que Δ_k representa o estado da(s) base(s) de dados imediatamente após a aplicação de λ_k e Δ_{k-1} representa o estado da(s) base(s) de dados imediatamente antes da execução de λ_k; e $<M_1,..., M_t>$ é a seqüência de seqüências de mensagens emitidas, tal que M_k representa a seqüência de mensagens em λ_k.

Definição: ***Conjunto de Casos de Teste***

Um *conjunto de casos de teste*, denotado por $\{ \lambda_1, ..., \lambda_t \}$, denota a aplicação de casos de teste de forma independente, onde o estado de entrada da(s) base(s) de dados para um caso de teste é independente do estado obtido pela aplicação de qualquer caso de teste anterior. A distinção entre seqüência e conjunto de casos de teste ressalta o papel da base de dados no teste de regras ativas. Se a base de dados resultante da aplicação de um caso de teste é utilizada no caso de teste seguinte, então a ordem de aplicação de casos de teste pode afetar os dados de saída.

As definições de conjunto e de seqüência de casos de teste são pertinentes ao teste de aplicações de bancos de dados. A persistência de dados possui duas implicações: a atividade de teste pode corromper a base de dados e, portanto, seqüências de casos de teste não são indicadas para a utilização de bases de produção; a restauração da base de dados de entrada após a aplicação de um caso de teste é requerida quando se aplica um conjunto de casos de teste. Chays et al. (2000) advogam a utilização de bases de dados sintéticos, com o intuito de preservar a consistência da base de dados e de selecionar dados de teste direcionados ao exercício de elementos requeridos.

4.3 Interação entre Regras Ativas

O comportamento de um conjunto de regras reflete a reação do sistema em tempo de execução: regras ativas são disparadas em resposta a eventos ocorridos na execução de aplicações

de banco de dados e na execução de regras ativas propriamente ditas. As vantagens de bancos de dados ativos são atenuadas pelo custo da complexidade resultante das interações entre execuções autônomas de regras ativas e manipulações de dados submetidas pelo usuário (Chan et al., 1997). Mesmo um pequeno número de regras ativas pode ser complexo para entender e gerenciar (Widow e Ceri, 1996). Tal complexidade conduz a um comportamento imprevisível, não determinístico, difícil de ser estruturado (Vaduva, 1999).

Análise estática tem sido usada na investigação de propriedades de um conjunto de regras, buscando prever o comportamento de um sistema em tempo de execução (Aiken et al., 1995; Baralis et al., 1998). As propriedades mais amplamente estudadas para um conjunto de regras são abaixo descriminadas, conforme descrito por Zaniolo et al. (1997):

- Terminação: Em qualquer transação definida pelo usuário que resulte no processamento de regras, o processamento termina, produzindo um estado final.
- Confluência: Em qualquer transação definida pelo usuário que resulte no processamento de regras, o processamento termina, produzindo um único estado final, independentemente da ordem de execução das regras disparadas.
- Determinismo observável: Em adição à confluência, para qualquer transação definida pelo usuário que resulte no processamento de regras, todas as ações visíveis ao usuário são as mesmas, tais como as mensagens exibidas para apresentar estados intermediários ou para sinalizar situações de alerta.

Mesmo atendendo às propriedades terminação, confluência e determinismo observável, não é garantido que um conjunto de regras esteja correto. A interação entre regras pode ser fonte de comportamento incorreto e inesperado, caracterizando um aspecto pertinente para a especificação de casos de testes. Em particular, o comportamento de um conjunto de regras é representado em três aspectos:

- O fluxo de controle intra-regra é determinado pela seqüência de operações executadas durante o processamento de uma única regra, seguindo a ordem: detecção de evento, avaliação de condição e execução de ação. Chan et al. (1997) introduzem uma abordagem para o teste intra-regra para regras escritas em uma linguagem declarativa e orientada a

objetos; apesar de não existirem construções de fluxo de controle em uma linguagem declarativa, os autores abstraem que o controle flui do evento para a condição e da condição para a ação. Leitão et al. (2002) preocupam-se com o fluxo de controle em regras escritas em SQL; o fluxo de controle é representado para comandos de manipulação, para rotinas de tratamento de exceção e para as extensões estruturadas da SQL, as quais inserem construções similares a linguagens imperativas, tais como seleção simples, seleção múltipla e laços.

- O fluxo de controle inter-regra reflete a transferência de controle de uma regra para outra. O fluxo de controle é dependente do modelo de execução do sistema de regras utilizado, especialmente da política de ciclo e dos modos de acoplamento evento-condição e condição-ação (conforme descrito na Seção 2.3):
    - este trabalho adota *política de ciclo recursiva e modos de acoplamento imediatos* para o modelo de fluxo de controle inter-regra; ou seja, a execução de uma regra é interrompida quando o evento de disparo de alguma regra é provocado e o controle é transferido imediatamente para a regra disparada.
- Interações de regras são dependências que podem aparecer entre regras em tempo de execução. Dadas duas regras, r_i e r_j, não necessariamente distintas, descritas por (e_i, c_i, a_i) e (e_j, c_j, a_j), respectivamente, as interações entre regras são definidas como:
    - *interação A/E (Ação/Evento)*: a execução de a_i provoca e_j, ou um componente de e_j;
    - *interação A/C (Ação/Condição)*: a execução de a_i modifica os mesmos objetos da base de dados acessados durante a avaliação de c_j;
    - *interação A/A (Ação/Ação)*: as execuções de a_i e de a_j acessam os mesmos objetos da base de dados; especificamente, a_i modifica um ou mais objetos da base de dados acessados durante a execução de a_j.

Um outro tipo de interação, o qual não é de interesse neste trabalho, ocorre quando r_i ativa ou desativa r_j. Neste caso, comandos dedicados são utilizados para explicitamente habilitar ou desabilitar o disparo de r_j quando e_j for provocado.

O fluxo de controle intra-regra pode conduzir a interações entre regras: um caso de teste aplicado a uma regra resulta na execução de caminhos em que a ação de regra pode provocar

eventos ou modificar o valor verdade da condição de outras regras. A interação *A/E* afeta o fluxo de controle inter-regra, pois contribui para o disparo entre regras. O fluxo de controle inter-regra impacta na ordem de execução de regras e, portanto, afeta a existência de interações de regra, visto que as interações entre regras são unidirecionais: uma interação entre duas regras existe se as partes de regra correspondentes são executadas em uma certa ordem.

Definição: ***Grafo de interação entre regras ativas***

Considere R um conjunto de regras ativas. Um grafo de interação entre regras é denotado por $G = (N, E)$, onde: N representa o conjunto de nós, em que cada nó denota uma regra ativa r que interage com outras regras de R ou consigo mesma, tal que $r \in R$; e E representa o conjunto de arcos unidirecionais que denotam a existência de pelo menos uma interação entre duas regras, não necessariamente distintas.

Três grafos podem ser abstraídos a partir do conjunto R, denominados *grafos de interação*:

- *Grafo AE*, descrito por $G^{AE} = (N^{AE}, E^{AE})$, onde: N^{AE} representa o conjunto de nós, que denotam regras de R que participam de interações *A/E* entre si; e E^{AE} representa o conjunto de arcos, em que cada arco é um arco dirigido que denota a presença de pelo menos uma interação *A/E* entre duas regras de R, não necessariamente distintas. Este grafo é também denominado de *grafo de disparo* (Baralis, 1998).
- *Grafo AC*, descrito por $G^{AC} = (N^{AC}, E^{AC})$, onde: N^{AC} representa o conjunto de nós, que denotam regras de R que participam de interações *A/C* entre si; e E^{AC} representa o conjunto de arcos, em que cada arco é um arco dirigido que denota a presença de pelo menos uma interação *A/C* entre duas regras de R, não necessariamente distintas.
- *Grafo AA*, descrito por $G^{AA} = (N^{AA}, E^{AA})$, onde: N^{AA} representa o conjunto de nós, que denotam regras de R que participam de interações *A/A* entre si; e E^{AA} representa o conjunto de arcos, em que cada arco é um arco dirigido que denota a presença de pelo menos uma interação *A/A* entre duas regras de R, não necessariamente distintas.

Como exemplo, considere um conjunto composto por 3 regras: $R = \{ r_i, r_j, r_k \}$. A Figura 4.2 mostra os grafos *AE*, *AC* e *AA* para este conjunto de regras. Os grafos ilustram os vários

caminhos de interação das regras. Em 4.2(a), uma execução da regra r_i pode provocar o evento de disparo da regra r_j; em 4.2(b), uma execução da regra r_i pode modificar o resultado da avaliação da condição da regra r_k, e vice-versa; em 4.2(c), uma execução da regra r_k pode influenciar operações de manipulação de dados em posteriores execuções desta regra.

Vale ressaltar que os grafos *AE*, *AC* e *AA* exploram interações diretas entre regras, ou seja, duas-a-duas. A intenção não é capturar características controladas pelo sistema gerenciador de banco de dados, mas explorar as interações entre regras ligadas a aspectos de fluxo de controle e de fluxo de dados.

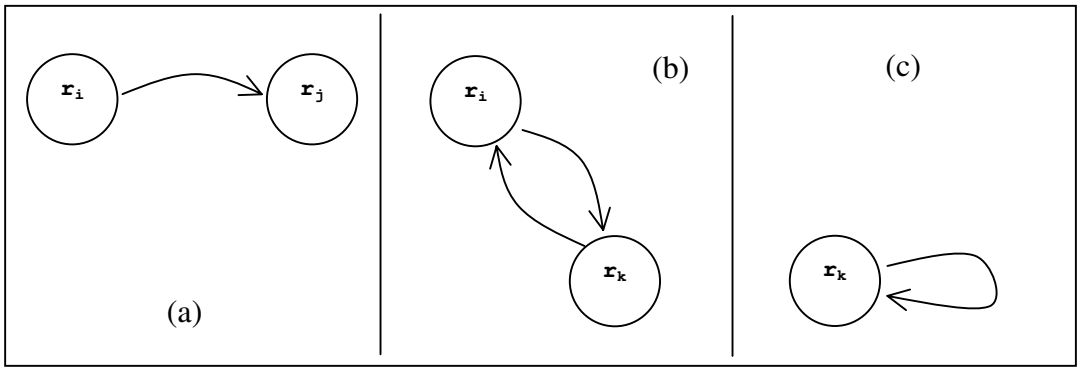


Figura 4.2 – Exemplos de grafos de interação para o conjunto de regras $R = \{ r_i, r_j, r_k \}$: (a) grafo AE; (b) grafo AC; e (c) grafo AA.

Baralis (1998) introduziu o grafo de ativação, que busca ilustrar se o resultado da avaliação da condição de uma regra pode ser modificado de falso para verdadeiro após a execução de outra regra, ou se este resultado pode ser verdadeiro sob a ótica de uma regra afetar a si mesma. O autor apresenta situações especificas, em que se pode atribuir um arco ao grafo de ativação; contudo, muitos arcos são pessimisticamente incluídos devido à dificuldade, ou à impossibilidade, de inferir o valor verdade da condição de regra.

O grafo *AC* é distinto do grafo de ativação, pois busca captar o efeito da interação entre regras na condição de regra, sem se preocupar com a terminação de sua execução. Se um arco está presente no grafo *AC*, então ele conecta duas regras, não necessariamente distintas, onde a execução da ação da primeira regra pode alterar o valor verdade da condição da segunda regra, independentemente do valor resultante. No contexto de teste de regras isoladas, as estratégias de teste sistemáticas deveriam requerer casos de teste em que a condição de regra seja avaliada

como verdadeira e como falsa; a idéia é exercitar contextos de ativação e de não ativação da ação de regra. Estendendo esta noção à interação de regras, interações *A/C* deveriam ser exercitadas para ambos os contextos, de modo que a ativação e a não ativação da ação de regra seja explorada para cada interação *A/C*.

A questão da indecidibilidade está presente nos grafos de interação. A presença de um arco em tais grafos denota a possibilidade de interação; as interações *A/E*, *A/C* e *A/A* são interações potenciais. Segundo Vaduva (1999), em geral é indecidível saber se uma operação executada dentro da ação de uma regra manipula precisamente os mesmos objetos de outras regras. Ou seja, a completa predição das interações *A/E*, *A/C* e *A/A* é um problema indecidível. Não é possível ter certeza, para todos os estados de base de dados e para todos os possíveis comandos nas ações de regra, inclusive comandos de fluxo de controle, se uma interação irá ocorrer.

Escopo transacional e origem de eventos representam aspectos pertinentes à interação entre regras ativas. Uma transação de banco de dados é uma unidade lógica de processamento composta por uma ou mais operações que devem ser executadas atomicamente. As operações de uma regra são executadas dentro do escopo transacional da operação que provocou o seu evento de disparo. Regras ativas são inicialmente disparadas por eventos provocados na aplicação de banco de dados e a execução de regras pode provocar disparo entre regras. Neste sentido, transações são iniciadas por aplicações de banco de dados e se estendem à seqüência de operações executadas por regras disparadas, incluindo disparo entre regras; denominamos esta característica por *extensibilidade transacional*. Eventos provocados por operações existentes em regras ativas são classificados como *eventos internos*; *eventos externos* são provocados por operações na aplicação de banco de dados propriamente dita. Eventos internos caracterizam interações *A/E* e qualquer execução de regras devido ao disparo entre regras ocorre dentro do escopo da transação atribuída à operação do seu evento externo. Interações *A/C* e *A/A* podem ocorrer em uma mesma transação ou em transações separadas.

Considere os grafos de fluxo de controle para as regras r_i e r_j exibidos na Figura 4.3. A execução da operação atribuída ao nó q na ação da regra r_i pode provocar o evento de disparo da regra r_j, estabelecendo uma interação *A/E* entre as regras r_i e r_j (esta característica é ilustrada na Figura 4.3 pela linha pontilhada em negrito). As operações atribuídas aos nós p e m modificam o estado da relação t; e a operação associada ao nó s resulta no uso desta relação no arco (s, x_j).

Interações *A/A* são estabelecidas entre as regras r_i e r_j para os pares $<p, (s, x_j)>$ e $<m, (s, x_j)>$. Se a execução da operação representada pelo nó *q* provoca o evento de disparo de r_j, transferindo o controle para esta regra, então a execução de ambas as regras e o exercício da interação *A/A* referente ao par $<p, (s, x_j)>$ ocorrem no escopo da mesma transação, devido à extensibilidade transacional.

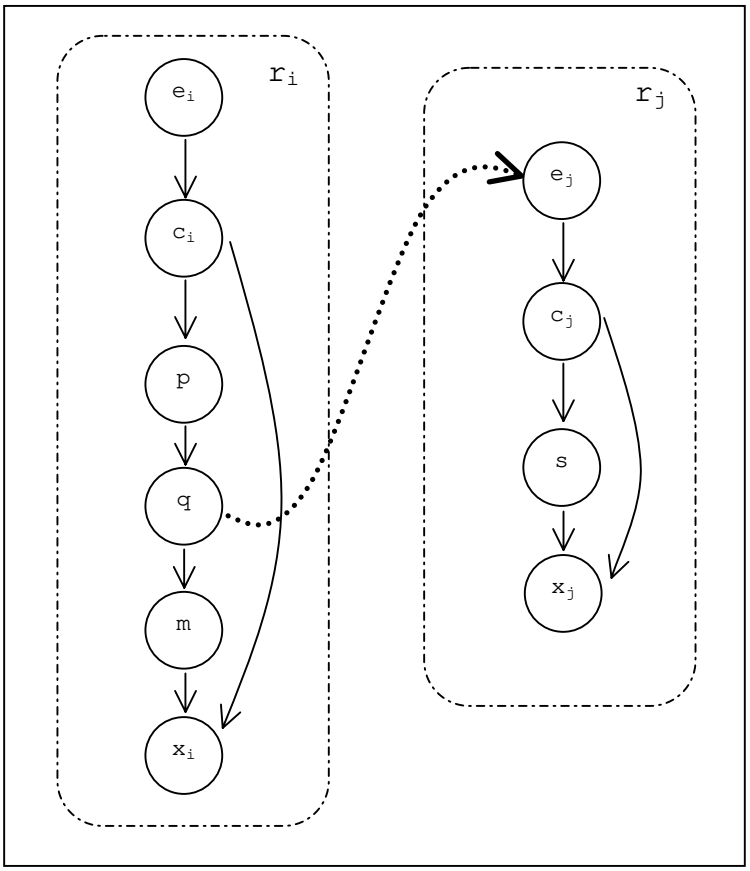


Figura 4.3 – Exemplo de interação A/E entre as regras r_i e r_j; o nó *q* na ação da regra r_i refere-se a uma operação cuja execução pode provocar o evento de disparo de r_j.

Vilela (1998) introduziu o conceito de potencial associação de fluxo de dados interprocedimentais, explorando as chamadas de procedimentos entre unidades de programa; a chamada de procedimentos estabelece novos caminhos de execução, compostos pela concatenação de subcaminhos pertencentes às unidades chamadora e chamada, estabelecendo oportunidades para novas relações de fluxo de dados entre as unidades. São estabelecidas associações de chamada e de retorno, dependendo se o local da definição de dados está na

unidade chamadora ou na unidade chamada, respectivamente. Essa abordagem considera a integração das unidades de programa, realizada duas-a-duas (*pairwise*). Spoto (2000) estendeu essa idéia para o teste de aplicações de banco de dados, contemplando a definição e o uso de variáveis do tipo tabela; os fluxos de controle resultantes de chamadas de procedimentos entre unidades foram explorados para abstrair relações de fluxo de dados persistentes. Para o caso de regras ativas e no modelo de interações adotado – política de ciclo recursiva e modos de acoplamento imediatos, onde a execução de uma regra é interrompida quando o evento de disparo de alguma regra é provocado e o controle é transferido para esta regra –, interações *A/E* podem resultar em novas interações *A/C* e *A/A*, similarmente às associações de chamada e de retorno inicialmente definidas por Vilela (1998), com a particularidade de as interações resultantes ocorrerem sempre no escopo de uma mesma transação. No exemplo da Figura 4.3, a interação *A/E* entre as regras r_i e r_j resulta na interação *A/A* entre as regras r_i e r_j referente ao par $< p, (s, x_j) >$, onde as ocorrências de definição e de uso estão em uma mesma transação.

Spoto (2000) afirma que uma variável tabela definida por uma unidade de programa mantém-se viva ao longo do tempo. Esta característica resulta em associações de fluxo de dados entre duas unidades de programa que foram executadas em momentos distintos, sem a necessidade de um comando de chamada de uma para a outra; tais associações são denominadas associações por dependência de dados, sendo exercitadas por execuções independentes de pares de unidades (execução da unidade que define a variável tabela e, em seguida, execução da unidade que usa a variável tabela). Para o contexto de regras ativas, a execução independente de regras, seguindo uma ordem estabelecida, é utilizada para o exercício de interações *A/C* e *A/A*. Considere a interação *A/A* entre as regras r_i e r_j referente ao par $< m, (s, x_j) >$ na Figura 4.3. Esta interação existe em execuções independentes das regras r_i *e* r_j, respeitando a seqüência $< r_i , r_j >$, visto que o fluxo de controle ocasionado pelo disparo entre regras não estabelece a existência de caminhos que são iniciados no nó *m* e que alcancem o arco (s, x_j). Tais execuções independentes podem ocorrer dentro de uma mesma transação ou em transações separadas. A execução em transações separadas e concomitantes pode levar a problemas de concorrência na execução de transações (por exemplo, se duas transações acessam o mesmo item de dados, é possível que haja acesso a dados modificados e pendentes de confirmação). Aspectos de concorrência não fazem parte do contexto deste trabalho; o foco de interesse é a interação direta entre regras ativas, buscando revelar a presença de defeitos pelo exercício dessas interações; nesse sentido, as

execuções independentes de regras ocorrem de forma serial: primeiro a regra que possui a definição de dados persistente, seguida da regra que possui o uso de dados persistentes.

4.4 Associações de Interação

Esta seção introduz associações de fluxo de dados persistentes para a interação entre regras, denominadas de *associações de interação*. O disparo entre regras não utiliza parâmetros reais ou variáveis globais, diferentemente da chamada convencional de procedimentos. O conjunto de variáveis usadas como entrada e como saída da execução de uma regra é composto unicamente por variáveis persistentes (as variáveis de banco de dados). Interações dependentes do disparo entre regras e interações baseadas unicamente na dependência de dados são oriundas de relações de fluxo de dados persistentes.

Interações *A/E* podem ser entendidas como uma forma de relação de fluxo de dados entre regras. Em seu modelo de fluxo de dados, Leitão et al. (2002) estabeleceram que no nó de entrada do grafo de uma regra – o nó atribuído ao evento de regra – são observadas as seguintes ocorrências de definição e de uso: definição das variáveis de transição da regra e uso do conjunto de dados afetados pela operação de mudança de estado; ou seja, às variáveis de transição é atribuído o conjunto de dados definidos pela operação de disparo. Considere os grafos de duas regras r_i e r_j, apresentados na Figura 4.3. A execução da operação atribuída ao nó q define um novo estado para a relação t; essa mudança de estado provoca o evento de disparo de r_j. Nesse sentido, existe um uso de t no disparo da regra r_j. O disparo de r_j a partir de r_i resulta em uma relação de fluxo de dados c.r.a t para o par $\prec q, (e_j, c_j) \succ$. Assim, interações *A/E* são representadas pelo tipo de associação de fluxo de dados descrito abaixo:

(i) ***associação-ddef-duso-AE***: é uma quíntupla *[x, (y, z), v, r', r'']*, onde *r'* e *r''* são regras, não necessariamente distintas, de um conjunto *R* de regras ativas; *v* é uma variável de banco de dados; *x* representa um nó da ação da regra *r'* que contém definição da variável *v*: $x \in N_a(r') \wedge v \in ddef(x)$; *y* representa o nó de evento da regra *r''*; *(y, z)* é um arco que possui uso de *v*: $v \in duso(y, z)$; e o uso da variável *v* no arco *(y, z)* ocorre em decorrência do disparo de *r''* a partir da definição de *v* em *x*, devido à atribuição de valores que define

as variáveis de transição em *r''*. Utilizando a semântica dos elementos da quíntupla *[x, (y, z), v, r', r'']* acima descritos, define-se a função ***ddu-AE (r', r'', v, x)*** que retorna os arcos *(y, z)*, tal que $v \in ddef(x)$ e $v \in duso(y, z)$.

Note que a definição acima não restringe explicitamente a existência de pelo menos um caminho livre de definição c.r.a *v* de *x* para *(y, z)*. No modelo de regras adotado, o exercício de interações *A/E* determina que a execução de uma regra seja interrompida para o início imediato da execução da regra disparada, tornando implícito que não haverá redefinição dos dados definidos. Contudo, é mandatório que a regra disparada seja executada para que os dados definidos sejam usados.

As interações baseadas unicamente na dependência de dados são introduzidas a seguir e, para serem cobertas, requerem a execução de subcaminhos ligados às regras que participam da interação. Para definir as associações de fluxo de dados atribuídas às interações *A/C* e *A/A*, considere o grafo da regra *r* dado por $G(r) = (N, A, e, x)$, conforme introduzido no Capítulo 2. Um *subcaminho* em $G(r)$ é uma seqüência finita de nós $p = (n_1, n_2, ..., n_k)$ tal que, para todo i, $1 \leq i < k$, $(n_i, n_{i+1}) \in A$. Um subcaminho formado pela concatenação de dois subcaminhos p_1 e p_2 é denotado por $p_1 . p_2$. Um *subcaminho inicial* de *r* é um subcaminho cujo primeiro nó é o nó de entrada de *r*. Um *subcaminho final* de uma regra é um subcaminho cujo último nó é o nó de saída de *r*. Um caminho completo em $G(r)$ é ao mesmo tempo um subcaminho inicial e final. Um *subcaminho iniciado em um arco* é um subcaminho cujos dois primeiros nós são os nós que compõem o arco. Um *subcaminho terminado em um arco* é um subcaminho cujos dois últimos nós são os nós que compõem o arco. O subcaminho $(n_i, n_1, ..., n_m, n_k)$, $m \geq 0$, que não contenha definição da variável *v* nos nós $n_1, ..., n_m$ é chamado de *subcaminho livre de definição c.r.a v* do nó n_i ao nó n_k e do nó n_i até o arco (n_m, n_k).

(ii) ***associação-ddef-duso-AC***: é uma quíntupla *[x, (y, z), v, r', r'']*, onde *r'* e *r''* são regras, não necessariamente distintas, de um conjunto *R* de regras ativas; *v* é uma variável de banco de dados; *x* representa um nó da ação da regra *r'* que contém definição da variável *v*: $x \in N_a(r') \wedge v \in ddef(x)$; *y* representa o nó da condição da regra *r''*; *(y, z)* é um arco que possui uso de *v*: $v \in duso(y, z)$; existem um subcaminho final no grafo de *r'* iniciado no nó *x* e um subcaminho inicial no grafo de *r''* terminado no arco *(y, z)*, denotados,

respectivamente, *p'* e *p''*, tal que *p' . p''* é um subcaminho livre de definição c.r.a *v*. Utilizando a semântica dos elementos da quíntupla *[x, (y, z), v, r', r'']* acima descritos, define-se a função ***ddu-AC (r', r'', v, x)*** que retorna os arcos *(y, z)*, desde que exista pelo menos um subcaminho *p' . p''* livre de definição c.r.a *v*, onde $v \in ddef(x)$ e $v \in duso(y, z)$.

(iii)***associação-ddef-duso-AA***: é uma quíntupla *[x, (y, z), v, r', r'']*, onde *r'* e *r''* são regras, não necessariamente distintas, de um conjunto *R* de regras ativas; *v* é uma variável de banco de dados; *x* representa um nó da ação da regra *r'* que contém definição da variável *v*: $x \in N_a(r') \wedge v \in ddef(x)$; *y* representa um nó da ação da regra *r''*: $y \in N_a(r'')$; *(y, z)* é um arco que possui uso de *v*: $v \in duso(y, z)$; existem um subcaminho final no grafo de *r'* iniciado no nó *x* e um subcaminho inicial no grafo de *r''* terminado no arco *(y, z)*, denotados, respectivamente, *p'* e *p''*, tal que *p' . p''* é um subcaminho livre de definição c.r.a *v*. Utilizando a semântica dos elementos da quíntupla *[x, (y, z), v, r', r'']* acima descritos, define-se a função ***ddu-AA (r', r'', v, x)*** que retorna os arcos *(y, z)*, desde que exista pelo menos um subcaminho *p' . p''* livre de definição c.r.a *v*, onde $v \in ddef(x)$ e $v \in duso(y, z)$.

A Figura 4.4 exemplifica as interações *A/C* e *A/A* entre duas regras r_i e r_j. Os nós *p* e *q* possuem definição das relações *t'* e *t''*, respectivamente. Nos arcos (c_j, s) e (c_j, x_j) ocorre uso de *t'* e no arco (s, x_j) ocorre uso de *t''*. As quíntuplas $[p, (c_j, s), t', r_i, r_j]$ e $[p, (c_j, x_j), t', r_i, r_j]$ constituem duas associações do tipo *associação-ddef-duso-AC*, desde que os subcaminhos $(p, q, m, x_i) . (e_j, c_j, s)$ e $(p, q, m, x_i) . (e_j, c_j, x_j)$ sejam subcaminhos livres de definição c.r.a *t'*. A quíntupla $[q, (s, x_j), t'', r_i, r_j]$ constitui uma associação do tipo *associação-ddef-duso-AA*, desde que o subcaminho $(q, m, x_i) . (e_j, c_j, s, x_j)$ seja um subcaminho livre de definição c.r.a *t''*.

A SQL e seus padrões não prevêem a definição de dados persistentes no nó associado à condição de regra. Os usos de dados persistentes nos arcos de saída do nó de condição de regra promovem a existência de associações do tipo *associação-ddef-duso-AC*, mas a impossibilidade de ocorrer definição de dados persistentes neste nó representa o porquê de não se estabelecer associações ligadas a interações condição-ação e condição-condição.

As definições a seguir consideram as associações estabelecidas a partir das transferências de controle resultantes do disparo entre regras. São caracterizados os locais da definição e do uso

persistentes, se estão na regra disparante ou na regra disparada, dando origem a associações de chamada e de retorno.

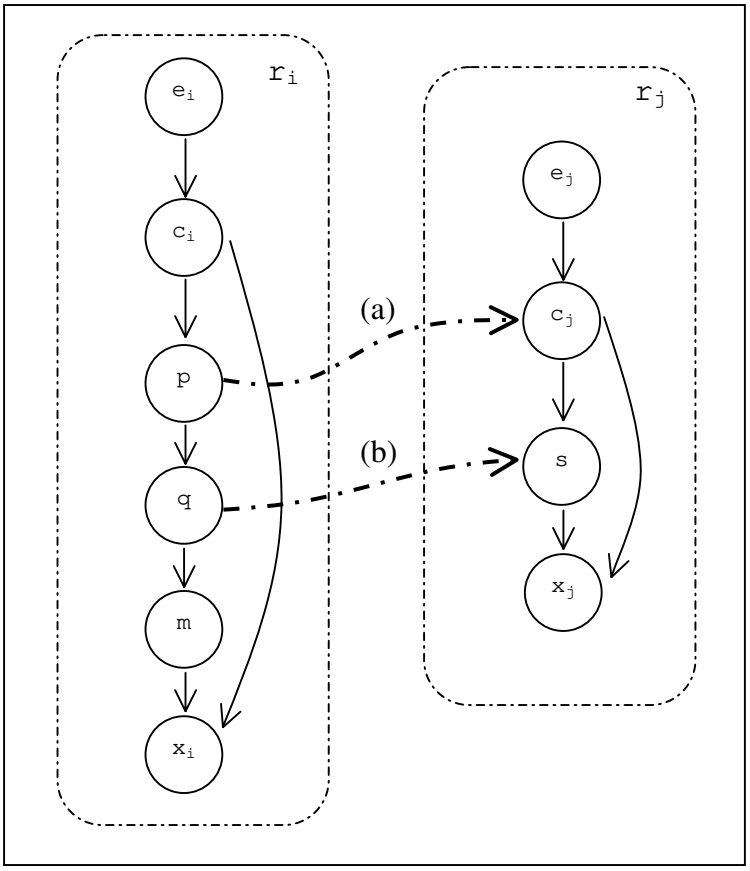


Figura 4.4 – Exemplo de interações A/C e A/A entre as regras r_i e r_j: (a) interação A/C para os pares < $p,(c_j; s)$ > e < $p,(c_j; x_j)$ >; (b) interação A/A para o par < $q,(s, x_j)$ >.

(iv)***associação-ddef-duso-AC de chamada***: é uma sêxtupla *[x, (y, z), v, m, r', r'']*, onde *r'* e *r''* são regras, não necessariamente distintas, de um conjunto *R* de regras ativas; *v* é uma variável de banco de dados; *x* e *m* representam nós da ação da regra *r'*, não necessariamente distintos: $x \in N_a(r') \wedge m \in N_a(r')$; *x* contém definição da variável *v*: $v \in ddef(x)$; *m* possui uma operação capaz de provocar o evento de disparo de *r''*; *y* representa o nó da condição da regra *r''*; *(y, z)* é um arco no grafo de *r''* que possui uso de *v*: $v \in duso(y, z)$; existem um subcaminho no grafo de *r'* iniciado em *x* e terminado em *m*, e um subcaminho inicial no grafo de *r''* terminado no arco *(y, z)*, denotados, respectivamente, por *p'* e *p''*, tal que *p' . p''* é um subcaminho livre de definição c.r.a *v*. Utilizando a

semântica dos elementos da sêxtupla *[x, (y, z), v, m, r', r'']* acima descritos, define-se a função ***ddu-AC de chamada (r', r'', v, x, m)*** que retorna os arcos *(y, z)*, desde que exista pelo menos um subcaminho *p' . p''* livre de definição c.r.a *v*, onde $v \in ddef(x)$ e $v \in duso(y, z)$.

(v) ***associação-ddef-duso-AA de chamada***: é uma sêxtupla *[x, (y, z), v, m, r', r'']*, onde *r'* e *r''* são regras, não necessariamente distintas, de um conjunto *R* de regras ativas; *v* é uma variável de banco de dados; *x* e *m* representam nós da ação da regra *r'*, não necessariamente distintos: $x \in N_a(r') \wedge m \in N_a(r')$; *x* contém definição da variável *v*: $v \in ddef(x)$; *m* possui uma operação capaz de provocar o evento de disparo de *r''*; *y* e *z* representam nós da ação da regra *r''*: $y \in N_a(r'') \wedge z \in N_a(r'')$; *(y, z)* é um arco no grafo de *r''* que possui uso de *v*: $v \in duso(y, z)$; existem um subcaminho no grafo de *r'* iniciado em *x* e terminado em *m*, e um subcaminho inicial no grafo de *r''* terminado no arco *(y, z)*, denotados, respectivamente, por *p'* e *p''*, tal que *p' . p''* é um subcaminho livre de definição c.r.a *v*. Utilizando a semântica dos elementos da sêxtupla *[x, (y, z), v, m, r', r'']* acima descritos, define-se a função ***ddu-AA de chamada (r', r'', v, x, m)*** que retorna os arcos *(y, z)*, desde que exista pelo menos um subcaminho *p' . p''* livre de definição c.r.a *v*, onde $v \in ddef(x)$ e $v \in duso(y, z)$.

(vi)***associação-ddef-duso-AA de retorno***: é uma sêxtupla *[x, (y, z), v, m, r', r'']*, onde *r'* e *r''* são regras, não necessariamente distintas, de um conjunto *R* de regras ativas; *v* é uma variável de banco de dados; *x* representa um nó da ação da regra *r''* que contém definição da variável *v*: $x \in N_a(r'') \wedge v \in ddef(x)$; *y* e *z* representam nós da ação da regra *r'*: $y \in N_a(r') \wedge z \in N_a(r')$; *(y, z)* é um arco no grafo de *r'* que possui uso de *v*: $v \in duso(y, z)$; *m* é um nó da ação da regra *r'* que possui uma operação capaz de provocar o evento de disparo de *r''*; existem um subcaminho final no grafo de *r''* iniciado em *x* e um subcaminho no grafo de *r'* iniciado em um nó sucessor de *m*, *succ(m)*, e terminado no arco *(y, z)*, denotados, respectivamente, por *p'* e *p''*, tal que *p' . p''* é um subcaminho livre de definição c.r.a *v*. Utilizando a semântica dos elementos da sêxtupla *[x, (y, z), v, m, r', r'']* acima descritos, define-se a função ***ddu-AA de retorno (r', r'', v, x, m)*** que retorna os

arcos *(y, z)*, desde que exista pelo menos um subcaminho *p' . p''* livre de definição c.r.a *v*, onde $v \in ddef(x)$ e $v \in duso(y, z)$.

As associações *(iv)*, *(v)* e *(vi)* são resultantes de fluxos de controle estabelecidos pelo disparo entre regras ativas. Retornando à Figura 4.3, se o nó *p* no grafo da regra r_i possui uma definição da variável persistente *t* e o arco (s, x_j) no grafo da regra r_j possui um uso de *t*, então a sêxtupla $[p, (s, x_j), t, q, r_i, r_j]$ constitui uma associação do tipo *associação-ddef-duso-AA de chamada*, desde que o subcaminho $(p, q) . (e_j, c_j, s, x_j)$ seja um subcaminho livre de definição c.r.a *t*. Em um outro contexto, se o nó *s* no grafo da regra r_j possui uma definição da variável persistente *t* e o arco (m, x_i) no grafo da regra r_i possui um uso de *t*, então a sêxtupla $[s, (m, x_i), t, q, r_i, r_j]$ constitui uma associação do tipo *associação-ddef-duso-AA de retorno*, desde que o subcaminho $(s, x_j) . (m, x_i)$ seja um subcaminho livre de definição c.r.a *t*.

As associações *(i)*, *(iv)*, *(v)* e *(vi)* requerem o disparo entre regras para sua cobertura. Seja *r* uma regra de um conjunto de regras *R*; o exercício de um nó *n* da regra *r*, o qual representa alguma operação de mudança de estado de um tipo capaz de disparar regras de *R*, incluindo *r*, não garante que haverá disparo entre regras, pois é requerido que $ddef(n) \neq \varnothing$. Esse aspecto é peculiar às regras ativas escritas em SQL e é distinto das abordagens em (Spoto, 2000; Vilela; 1998), as quais utilizam o grafo de chamada entre unidades, baseando-se no exercício de nós para a transferência de controle entre unidades.

O Capítulo 5 inclui um exemplo completo das associações de interação abstraídas de um conjunto de regras ativas.

4.5 Considerações Finais

Neste capítulo foi delineado o modelo de interação entre regras escritas em SQL adotado por esta pesquisa. Foram tratadas as diversas formas de interação entre regras, onde os componentes de uma regra interagem com os componentes de outra regra, ou da mesma regra, tal que são estabelecidas relações de fluxo de dados entre componentes de regras não necessariamente distintas.

Associações de interação foram introduzidas, explorando o fluxo de dados persistentes entre regras. As associações são derivadas do disparo entre regras e da manipulação das mesmas entidades da base de dados. O disparo entre regras modifica o fluxo de controle, transferindo-o entre regras, dando oportunidade a novas relações de fluxo de dados resultantes da concatenação entre subcaminhos das regras envolvidas. Em adição, regras são executadas em uma certa ordem, sem que ocorra disparo entre regras, obedecendo-se uma seqüência estabelecida por definição e uso persistentes.

Enfatizou-se o conceito de associação persistente, fundamentada nos fluxos de dados oriundos da presença dos comandos de manipulação da SQL. Cada comando foi analisado para determinar as ocorrências de definição e de uso persistentes, e foram ressaltadas as diferenças desse modelo de fluxo de dados em relação a outras iniciativas da literatura.

Uma característica essencial evidenciada no contexto de associações persistentes é a precisão de fluxo de dados, também denominada granularidade, que é adotada para a cobertura dessas associações. A granularidade impacta o teste, visto que demanda dados de teste específicos para cada precisão de fluxo de dados. O Capítulo 5 apresenta resultados empíricos da comparação entre granularidades na cobertura de associações de interação quanto à descoberta de defeitos e enfatiza aspectos de executabilidade e relação de inclusão.

Capítulo 5

Critérios de Teste Baseados na Interação entre Regras Ativas

A execução de um conjunto de regras pode levar à descoberta de defeitos não detectados durante o teste individual de regras. A preocupação com o comportamento de um conjunto de regras, em adição ao teste de cada regra isoladamente, caracteriza a necessidade por uma abordagem estruturada para a especificação e a aplicação de critérios de teste a regras ativas. Os Critérios de Teste Baseados na Interação entre Regras, propostos neste capítulo, buscam apoiar a descoberta de defeitos que porventura não tenham sido detectados pelo teste individual das regras de um dado conjunto. Foram definidos critérios para explorar as associações de interação introduzidas no Capítulo 4, onde são destacadas duas abordagens: interação norteada unicamente pela dependência de dados persistentes e interação derivada do disparo entre regras.

5.1 Definições de Critérios de Teste

Nesta seção, é definida uma família de critérios de teste baseados na interação entre regras ativas. Nesse sentido, um critério *C* (critério de adequação) é uma função que, para um dado conjunto de regras *R*, especificação *S* e conjunto de teste *T*, determina se *T* é *C-adequado* para *(R, S)*; ou seja, determina se *R* foi adequadamente testado com respeito a *S* por *T* de acordo com *C*.

Para a definição de critérios de teste ligados às associações de interação que exploram o fluxo de controle resultante do disparo entre regras – *associação-ddef-duso-AE*, *associação-ddef-*

duso-AC de chamada, *associação-ddef-duso-AA de chamada* e *associação-ddef-duso-AA de retorno* – considere que a aplicação de um conjunto de casos de teste proporciona a dupla $<\Pi, \Gamma>$, onde: Π é o conjunto de caminhos resultantes, tal que $\Pi = \{ \pi_1, ..., \pi_p \}, p > 0$; e Γ determina os dados persistentes definidos e usados ao longo de cada caminho de Π. Considere um conjunto de regras R com especificação S; sejam duas regras r' e r'' de R, não necessariamente distintas, descritas pelos grafos $G(r') = (N', A', e', x')$ e $G(r'') = (N'', A'', e'', x'')$, respectivamente; então qualquer $\pi_k \in \Pi$ é composto por $(e', ..., m) . (e'', ..., x'') . (succ(m), ..., x')$, onde m é o nó que representa a operação de r' que provocou o evento de disparo de r'', e $succ(m)$ é um nó sucessor de m em r'. Seja o conjunto $\Gamma = \{ \gamma_1, ..., \gamma_p \}$, onde γ_k representa os dados persistentes definidos e usados ao longo do caminho π_k.

- **Critério *todos-dusos-AE***: a dupla $<\Pi, \Gamma>$ satisfaz o critério *todos-dusos-AE* para (R, S), se e somente se, para cada par de regras r' e r'' de R, não necessariamente distintas, para cada nó m da ação da regra r' que possui uma operação capaz de provocar o evento de disparo de r'', Π e Γ incluírem todos os subcaminhos $(m) . (e'')$ para toda variável persistente v, tal que $v \in ddef(m)$ e $v \in duso(e'', succ(e''))$. Equivalentemente, em termos de associação de interação, a dupla $<\Pi, \Gamma>$ deve incluir, para cada nó x da ação da regra r' que possui uma operação capaz de provocar o evento de disparo de r'', todas as associações $[x, (y,z), v, r', r'']$ de R do tipo *associação-ddef-duso-AE*, tal que $(y,z) \in ddu\text{-}AE (r', r'', v, x)$.

- **Critério *todos-dusos-AC de chamada***: a dupla $<\Pi, \Gamma>$ satisfaz o critério *todos-dusos-AC de chamada* para (R, S), se e somente se, para cada par de regras r' e r'' de R, não necessariamente distintas, para todo nó x da ação da regra r', para toda variável persistente v definida em x e para todo nó m da ação da regra r' que possui uma operação capaz de provocar o evento de disparo de r'', incluir pelo menos um subcaminho $p' . p''$ livre de definição c.r.a v do nó x para todo arco (y,z) que possui uso de v, onde y representa o nó da condição de r'', tal que p' é um subcaminho iniciado em x e terminado em m, e p'' é um subcaminho inicial de r'' terminado em (y,z). Equivalentemente, em termos de associação de interação, a dupla $<\Pi, \Gamma>$ deve incluir, para cada nó m da ação

da regra *r'* que possui uma operação capaz de provocar o evento de disparo de *r''*, todas as associações *[x, (y,z), v, m, r', r'']* de *R* do tipo *associação-ddef-duso-AC de chamada*, tal que *(y,z)∈ ddu-AC de chamada (r', r'', v, x, m)*.

- **Critério *todos-dusos-AA de chamada***: a dupla *<Π, Γ>* satisfaz o critério *todos-dusos-AA de chamada* para *(R, S)*, se e somente se, para cada par de regras *r'* e *r''* de *R*, não necessariamente distintas, para todo nó *x* da ação da regra *r'*, para toda variável persistente *v* definida em *x* e para todo nó *m* da ação da regra *r'* que possui uma operação capaz de provocar o evento de disparo de *r''*, incluir pelo menos um subcaminho *p' . p''* livre de definição c.r.a *v* do nó *x* para todo arco *(y,z)* que possui uso de *v*, onde *y* e z representam nós da ação de *r''*, tal que *p'* é um subcaminho iniciado em *x* e terminado em *m*, e *p''* é um subcaminho inicial de *r''* terminado em *(y,z)*. Equivalentemente, em termos de associação de interação, a dupla *<Π, Γ>* deve incluir, para cada nó *m* da ação da regra *r'* que possui uma operação capaz de provocar o evento de disparo de *r''*, todas as associações *[x, (y,z), v, m, r', r'']* de *R* do tipo *associação-ddef-duso-AA de chamada*, tal que *(y,z)∈ ddu-AA de chamada (r', r'', v, x, m)*.

- **Critério *todos-dusos-AA de retorno***: a dupla *<Π, Γ>* satisfaz o critério *todos-dusos-AA de retorno* para *(R, S)*, se e somente se, para cada par de regras *r'* e *r''* de *R*, não necessariamente distintas, para todo nó *x* da ação da regra *r''*, para toda variável persistente *v* definida em *x* e para todo nó *m* da ação da regra *r'* que possui uma operação capaz de provocar o evento de disparo de *r''*, incluir pelo menos um caminho *p'' . p'* livre de definição c.r.a *v* do nó *x* para todo arco *(y,z)* que possui uso de *v*, onde *y* e z representam nós da ação de *r'*, tal que *p'* é um subcaminho final de *r''* iniciado em *x*, e *p''* é um subcaminho em *r''* iniciado em *succ(m)* e terminado em *(y,z)*. Equivalentemente, em termos de associação de interação, a dupla *<Π, Γ>* deve incluir, para cada nó *m* da ação da regra *r'* que possui uma operação capaz de provocar o evento de disparo de *r''*, todas as associações *[x, (y,z), v, m, r', r'']* de *R* do tipo *associação-ddef-duso-AA de retorno*, tal que *(y,z)∈ ddu-AA de retorno (r', r'', v, x, m)*.

Sobre as associações de interação baseadas unicamente na dependência de dados persistentes – *associação-ddef-duso-AC* e *associação-ddef-duso-AA* – considere que a aplicação de um conjunto de casos de teste proporciona a dupla $<\Pi, \Gamma>$, onde: Π é o conjunto de caminhos resultantes, tal que $\Pi = \{ \pi_1, ..., \pi_p \}$, $p > 0$; e Γ determina os dados persistentes definidos e usados ao longo de cada caminho completo de Π. Considere um conjunto de regras R com especificação S; sejam duas regras r' e r'' de R, não necessariamente distintas, descritas pelos grafos $G(r') = (N', A', e', x')$ e $G(r'') = (N'', A'', e'', x'')$, respectivamente; então qualquer $\pi_k \in \Pi$ é composto por $(e', ..., x') \,.\, (e'', ..., x'')$, resultante de execuções em seqüência de r' e r''. Seja o conjunto $\Gamma = \{ \gamma_1, ..., \gamma_p \}$, onde γ_k representa os dados persistentes definidos e usados ao longo do caminho π_k.

- **Critério *todos-dusos-AC***: a dupla $<\Pi, \Gamma>$ satisfaz o critério *todos-dusos-AC* para (R, S), se e somente se, para cada par de regras r' e r'' de R, não necessariamente distintas, para todo nó x da ação da regra r' e para toda variável persistente v definida em x, incluir pelo menos um caminho $p' \,.\, p''$ livre de definição c.r.a v do nó x para todo arco (y,z) que possui uso de v, onde y representa o nó de condição de r'', tal que p' é um subcaminho final de r' iniciado em x, e p'' é um subcaminho inicial de r'' terminado em (y,z). Equivalentemente, em termos de associação de interação, a dupla $<\Pi, \Gamma>$ deve incluir todas as associações $[x, (y,z), v, r', r'']$ de R do tipo *associação-ddef-duso-AC*, tal que $(y,z) \in ddu\text{-}AC(r', r'', v, x)$.

- **Critério *todos-dusos-AA***: a dupla $<\Pi, \Gamma>$ satisfaz o critério *todos-dusos-AC* para (R, S), se e somente se, para cada par de regras r' e r'' de R, não necessariamente distintas, para todo nó x da ação da regra r' e para toda variável persistente v definida em x, incluir pelo menos um caminho $p' \,.\, p''$ livre de definição c.r.a v do nó x para todo arco (y,z) que possui uso de v, onde y e z representam nós da ação de r'', tal que p' é um subcaminho final de r' iniciado em x, e p'' é um subcaminho inicial de r'' terminado em (y,z). Equivalentemente, em termos de associação de interação, a dupla $<\Pi, \Gamma>$ deve incluir todas as associações $[x, (y,z), v, r', r'']$ de R do tipo *associação-ddef-duso-AA*, tal que $(y,z) \in ddu\text{-}AA(r', r'', v, x)$.

Para ilustrar os requisitos dos *critérios baseados na interação entre regras*, considere os códigos fonte das regras ativas *trg001*, *trg002* e *trg005*, exibidos nas Figuras 5.1, 5.2 e 5.3, onde no início de cada linha está o número do nó correspondente, e cujos grafos de fluxo de controle estão representados nas Figuras 5.4, 5.5 e 5.6, respectivamente; a Seção 6.1 apresenta o modelo de fluxo de controle para regras escritas em SQL. Tais regras manipulam uma base de dados, que é descrita pelos seguintes esquemas de relação: *tab001 (key001, att001_1, att001_2, att001_3)*, *tab002 (key002, att002_1)*, *tab003 (key003, att003_1)*, *tab004 (key004, att004_1, att004_2)* e *tab005 (key004, att005_1)*. Os elementos requeridos pelos critérios *todos-dusos-AA*, *todos-dusos-AC*, *todos-dusos-AE*, *todos-dusos-AA de chamada*, *todos-dusos-AA de retorno* e *todos-dusos-AC de chamada* são apresentados nas Figuras 5.7, 5.8, 5.9, 5.10, 5.11, e 5.12, respectivamente.

No restante deste capítulo, os critérios propostos serão referenciados de forma indistinta como *Critérios Baseados na Interação entre Regras*, *Critérios de Interação* e *Critérios Baseados nas Associações de Interação*.

```
*01* CREATE OR REPLACE TRIGGER trg001
*01*     AFTER INSERT OR DELETE OR UPDATE ON tab001
*01*     FOR EACH ROW
*01* DECLARE v_exists INTEGER;
*01* BEGIN
*02*      UPDATE tab003
*02*      SET    att003_1 = att003_1 + NVL(:new.att001_2, 0) - NVL(:old.att001_2, 0)
*02*      WHERE  key003 = NVL(:new.key001, 0)
*02*      OR     key003 = NVL(:old.key001, 0);
*03*      IF DELETING THEN
*04*         DELETE FROM tab002
*04*         WHERE  key002 = :old.key001;
*05*      ELSE
*05*         BEGIN
*06*            SELECT COUNT(*) INTO v_exists
*06*            FROM   tab002
*06*            WHERE  key002 = :new.key001;
*07*            IF :new.att001_3 >= (:new.att001_2 - :new.att001_1) THEN
*08*               IF v_exists > 0 THEN
*09*                  DELETE FROM tab002
*09*                  WHERE  key002 = :new.key001;
*10*               END IF;
*11*            ELSE
*11*               IF v_exists > 0 THEN
*12*                  UPDATE tab005
*12*                  SET    att005_1 = :new.att001_2 - :new.att001_1 -:new.att001_3
*12*                  WHERE  key005 = :new.key001;
*13*               ELSE
*13*                  INSERT INTO tab002
*13*                     (key002, att002_1)
*13*                  VALUES
*13*                     (:new.key001, :new.att001_2 - :new.att001_1 -:new.att001_3);
*14*               END IF;
*15*            END IF;
*17*         EXCEPTION
*18*             WHEN OTHERS THEN
*19*                  RAISE_APPLICATION_ERROR (-20002,'Ins/upd error: ' || sqlerrm);
*16*         END;
*20*      END IF;
*21*      SELECT COUNT(*) INTO v_exists
*21*      FROM   tab004
*21*      WHERE  key004 = NVL(:new.key001, 0)
*21*      OR     key004 = NVL(:old.key001, 0);
*22*      IF v_exists > 0 THEN
*23*         UPDATE tab004
*23*         SET    att004_2 = att004_2 + NVL(:new.att001_2, 0) - NVL(:old.att001_2, 0)
*23*         WHERE  key004 = NVL(:new.key001, 0)
*23*         OR     key004 = NVL(:old.key001, 0);
*24*      END IF;
*25* END trg001;
```

Figura 5.1 – Regra ***trg001***, exemplo de regra ativa escrita em SQL.

```
*01* CREATE OR REPLACE TRIGGER trg002
*01* AFTER INSERT OR DELETE OR UPDATE ON tab002
*01* FOR EACH ROW
*01* DECLARE v_count INTEGER;
*01* BEGIN
*02*      SELECT COUNT(*) INTO v_count
*02*      FROM   tab003
*02*      WHERE  key003 = NVL(:new.key002, 0)
*02*      OR     key003 = NVL(:old.key002, 0);
*03*      IF DELETING OR UPDATING THEN
*04*         UPDATE tab004
*04*         SET att004_1  = att004_1 * ( 1 + v_count / 10 )
*04*         WHERE key004 = :old.key002;
*05*      END IF;
*06*      IF INSERTING OR UPDATING THEN
*07*         UPDATE tab004
*07*         SET att004_1  = att004_1 * ( 1 - v_count / 10 )
*07*         WHERE key004 = :new.key002;
*08*      END IF;
*09* END trg002;
```

Figura 5.2 – Regra ***trg002***, exemplo de regra ativa escrita em SQL.

```
*01* CREATE OR REPLACE TRIGGER trg005
*01* BEFORE INSERT OR UPDATE ON tab005
*01* FOR EACH ROW
*02* WHEN (EXISTS(SELECT key003 FROM tab003 WHERE att003_1 IS NULL))
*03* BEGIN
*03*      RAISE_APPLICATION_ERROR (-20003,'Ins/upd error: ' || sqlerrm);
*04* END trg005;
```

Figura 5.3 – Regra ***trg005***, exemplo de regra ativa escrita em SQL.

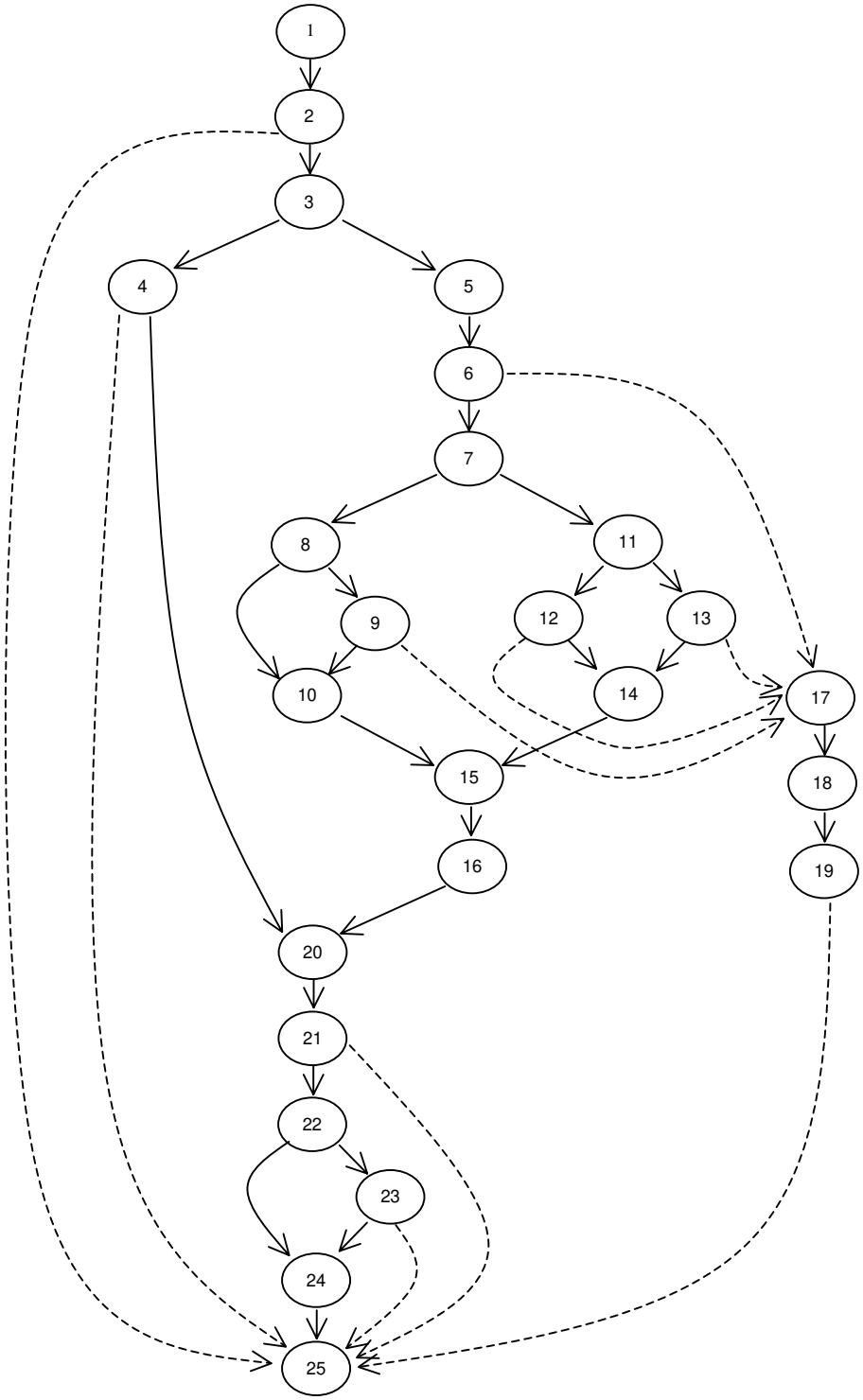


Figura 5.4 – Grafo de Fluxo de Controle da regra ***trg001***, que foi apresentada na Figura 5.1.

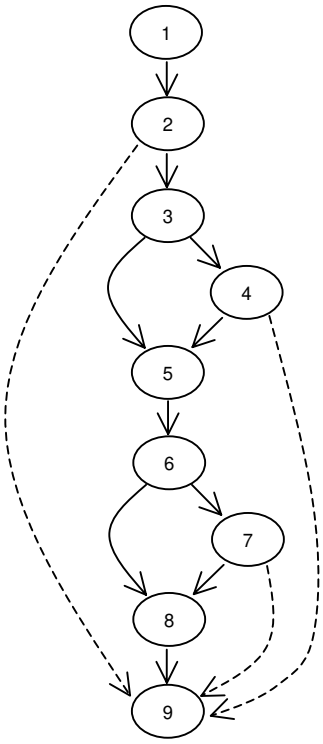


Figura 5.5 – Grafo de Fluxo de Controle da regra ***trg002***, que foi apresentada na Figura 5.2.

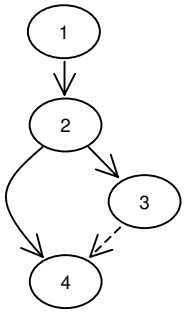


Figura 5.6 – Grafo de Fluxo de Controle da regra ***trg005***, que foi apresentada na Figura 5.3.

```
(1) [ 4 , (4,20) , tab002 , trg001 , trg001 ]
(2) [ 4 , (4,25) , tab002 , trg001 , trg001 ]
(3) [ 4 , (6,7) , tab002 , trg001 , trg001 ]
(4) [ 4 , (6,17) , tab002 , trg001 , trg001 ]
(5) [ 4 , (9,10) , tab002 , trg001 , trg001 ]
(6) [ 4 , (9,17) , tab002 , trg001 , trg001 ]
(7) [ 9 , (4,20) , tab002 , trg001 , trg001 ]
(8) [ 9 , (4,25) , tab002 , trg001 , trg001 ]
(10) [ 9 , (6,7) , tab002 , trg001 , trg001 ]
(11) [ 9 , (6,17) , tab002 , trg001 , trg001 ]
(12) [ 9 , (9,10) , tab002 , trg001 , trg001 ]
(13) [ 9 , (9,17) , tab002 , trg001 , trg001 ]
(14) [ 13 , (4,20) , tab002 , trg001 , trg001 ]
(15) [ 13 , (4,25) , tab002 , trg001 , trg001 ]
(16) [ 13 , (6,7) , tab002 , trg001 , trg001 ]
(17) [ 13 , (6,17) , tab002 , trg001 , trg001 ]
(18) [ 13 , (9,10) , tab002 , trg001 , trg001 ]
(19) [ 13 , (9,17) , tab002 , trg001 , trg001 ]
(20) [ 2 , (2,3) , tab003 , trg001 , trg001 ]
(21) [ 2 , (2,25) , tab003 , trg001 , trg001 ]
(22) [ 2 , (2,3) , tab003 , trg001 , trg002 ]
(22) [ 2 , (2,9) , tab003 , trg001 , trg002 ]
(23) [ 23 , (21,22) , tab004 , trg001 , trg001 ]
(24) [ 23 , (21,25) , tab004 , trg001 , trg001 ]
(25) [ 23 , (23,24) , tab004 , trg001 , trg001 ]
(26) [ 23 , (23,25) , tab004 , trg001 , trg001 ]
(27) [ 23 , (4,5) , tab004 , trg001 , trg002 ]
(28) [ 23 , (4,9) , tab004 , trg001 , trg002 ]
(29) [ 23 , (7,8) , tab004 , trg001 , trg002 ]
(30) [ 23 , (7,9) , tab004 , trg001 , trg002 ]
(31) [ 12 , (12,14) , tab005 , trg001 , trg001 ]
(32) [ 12 , (12,17) , tab005 , trg001 , trg001 ]
(33) [ 4 , (21,22) , tab004 , trg002 , trg001 ]
(34) [ 4 , (21,25) , tab004 , trg002 , trg001 ]
(35) [ 4 , (23,24) , tab004 , trg002 , trg001 ]
(36) [ 4 , (23,25) , tab004 , trg002 , trg001 ]
(37) [ 4 , (4,5) , tab004 , trg002 , trg002 ]
(38) [ 4 , (4,9) , tab004 , trg002 , trg002 ]
(39) [ 4 , (7,8) , tab004 , trg002 , trg002 ]
(40) [ 4 , (7,9) , tab004 , trg002 , trg002 ]
(41) [ 7 , (21,22) , tab004 , trg002 , trg001 ]
(42) [ 7 , (21,25) , tab004 , trg002 , trg001 ]
(43) [ 7 , (23,24) , tab004 , trg002 , trg001 ]
(44) [ 7 , (23,25) , tab004 , trg002 , trg001 ]
(45) [ 7 , (4,5) , tab004 , trg002 , trg002 ]
(46) [ 7 , (4,9) , tab004 , trg002 , trg002 ]
(47) [ 7 , (7,8) , tab004 , trg002 , trg002 ]
(48) [ 7 , (7,9) , tab004 , trg002 , trg002 ]
```

Figura 5.7 – Elementos requeridos pelo Critério *todos-dusos-AA*.

```
(49) [ 2 , (2,3) , tab003 , trg001 , trg005 ]
(50) [ 2 , (2,4) , tab003 , trg001 , trg005 ]
```

Figura 5.8 – Elementos requeridos pelo Critério *todos-dusos-AC*.

```
(51) [  4 , (1,2) , tab002 , trg001 , trg002 ]
(52) [  9 , (1,2) , tab002 , trg001 , trg002 ]
(53) [ 13 , (1,2) , tab002 , trg001 , trg002 ]
(54) [ 12 , (1,2) , tab005 , trg001 , trg005 ]
```

Figura 5.9 – Elementos requeridos pelo Critério *todos-dusos-AE*.

```
(55) [ 2 , (2,3) , tab003 , 4, trg001 , trg002 ]
(56) [ 2 , (2,9) , tab003 , 4, trg001 , trg002 ]
(57) [ 2 , (2,3) , tab003 , 9, trg001 , trg002 ]
(58) [ 2 , (2,9) , tab003 , 9, trg001 , trg002 ]
(59) [ 2 , (2,3) , tab003 , 13, trg001 , trg002 ]
(60) [ 2 , (2,9) , tab003 , 13, trg001 , trg002 ]
```

Figura 5.10 – Elementos requeridos pelo Critério *todos-dusos-AA de chamada*.

```
(61) [ 4 , (21,22) , tab004 , 4, trg001 , trg002 ]
(62) [ 4 , (21,25) , tab004 , 4, trg001 , trg002 ]
(63) [ 4 , (23,24) , tab004 , 4, trg001 , trg002 ]
(64) [ 4 , (23,25) , tab004 , 4, trg001 , trg002 ]
(65) [ 4 , (21,22) , tab004 , 9, trg001 , trg002 ]
(66) [ 4 , (21,25) , tab004 , 9, trg001 , trg002 ]
(67) [ 4 , (23,24) , tab004 , 9, trg001 , trg002 ]
(68) [ 4 , (23,25) , tab004 , 9, trg001 , trg002 ]
(69) [ 4 , (21,22) , tab004 , 13, trg001 , trg002 ]
(70) [ 4 , (21,25) , tab004 , 13, trg001 , trg002 ]
(71) [ 4 , (23,24) , tab004 , 13, trg001 , trg002 ]
(72) [ 4 , (23,25) , tab004 , 13, trg001 , trg002 ]
(73) [ 7 , (21,22) , tab004 , 4, trg001 , trg002 ]
(74) [ 7 , (21,25) , tab004 , 4, trg001 , trg002 ]
(75) [ 7 , (23,24) , tab004 , 4, trg001 , trg002 ]
(76) [ 7 , (23,25) , tab004 , 4, trg001 , trg002 ]
(77) [ 7 , (21,22) , tab004 , 9, trg001 , trg002 ]
(78) [ 7 , (21,25) , tab004 , 9, trg001 , trg002 ]
(79) [ 7 , (23,24) , tab004 , 9, trg001 , trg002 ]
(80) [ 7 , (23,25) , tab004 , 9, trg001 , trg002 ]
(81) [ 7 , (21,22) , tab004 , 13, trg001 , trg002 ]
(82) [ 7 , (21,25) , tab004 , 13, trg001 , trg002 ]
(83) [ 7 , (23,24) , tab004 , 13, trg001 , trg002 ]
(84) [ 7 , (23,25) , tab004 , 13, trg001 , trg002 ]
```

Figura 5.11 – Elementos requeridos pelo Critério *todos-dusos-AA de retorno*.

```
(85) [ 2 , (2,3) , tab003 , 12, trg001 , trg005 ]
(86) [ 2 , (2,4) , tab003 , 12, trg001 , trg005 ]
```

Figura 5.12 – Elementos requeridos pelo Critério *todos-dusos-AC de chamada*.

5.2 Análise de Propriedades

O intuito desta seção é a análise de algumas propriedades dos critérios propostos. São consideradas a relação de inclusão, explorando os intervenientes granularidade de análise de fluxo de dados e presença de não executabilidade, e a complexidade dos critérios, tendo como foco o número de elementos requeridos para cobri-los no pior caso.

A abstração de associações persistentes é alcançada pela análise estática do código fonte das regras do conjunto em teste. No entanto, a determinação da executabilidade de associações de interação em qualquer granularidade de fluxo de dados é, em geral, uma questão indecidível, visto que perpassa a questão da executabilidade de caminhos. Adicionalmente, existem aspectos peculiares à cobertura de associações de interação, os quais serão tratados nesta seção, especificamente ligados à precisão de fluxo de dados em que as associações são cobertas.

Definição:

Uma associação de interação α é *exercitável* (ou *executável*) na granularidade de fluxo de dados β se: *(i)* existe algum caminho completo executável π que a cobre; e *(ii)* os dados manipulados em π exercitam o fluxo de dados de α na precisão β.

No contexto de associações persistentes, a cobertura de associações requer a análise dinâmica dos dados definidos e usados, ao longo dos caminhos percorridos durante a aplicação de casos de teste, pois em geral não é possível determinar estaticamente quais entidades da base de dados serão afetadas pela execução de comandos de manipulação: se alguma entidade será manipulada, e, em caso positivo, quais entidades serão manipuladas. Tal característica está em sintonia com a constatação de Ostrand e Weyuker (1991) para programas escritos em linguagens com ponteiros: um conjunto de teste que exercita todos os caminhos de um programa não necessariamente exercita cada associação *def-uso*, em contraste com a teoria original (Rapps e Weyuker, 1985), que garantia a cobertura de todas as associações a partir do exercício de todos os caminhos. A efetividade de um critério de teste depende não somente dos caminhos selecionados, mas também dos dados de teste para tais caminhos (Clarke et al., 1989).

Visando a reduzir o número de casos de teste e, portanto, diminuir o custo do teste, Marré e Bertolino (1996) afirmam que existe uma ordenação natural entre as associações de fluxo de

dados em um programa: algumas associações são cobertas mais facilmente que outras, visto que cobrir as últimas implica em cobrir as primeiras. O intuito é focar atenção na geração de casos de teste para cobrir um conjunto mínimo de associações, as quais incluem as demais associações, sem perda de eficácia do teste. Segundo os autores, a associação *x* inclui (*subsumes*) a associação *y* quando todo caminho que cobre *x* também cobre *y*.

Definição:

Instanciando as idéias de Marré e Bertolino (1996) para o contexto de associações de interação e, sendo mais abrangente do que o exercício de caminhos: *(i) ddua* α_x *inclui ddua* α_y *na granularidade* β se todo conjunto de teste que cobre α_x em β também cobre α_y em β; *(ii) ddua* α_x *inclui estritamente ddua* α_y *na granularidade* β se α_x inclui α_y em β, mas α_y não inclui α_x em β; e *(iii) ddua* α_x *é incomparável com ddua* α_y *na granularidade* β se α_x não inclui α_y em β, nem α_y inclui α_x em β. Ortogonalmente: *(iv) ddua* α *na granularidade* β_x *inclui ddua* α *na granularidade* β_y se todo conjunto de teste que cobre α em β_x também cobre α em β_y; *(v) ddua* α *na granularidade* β_x *inclui estritamente ddua* α *na granularidade* β_y se α em β_x inclui α em β_y, mas α em β_y não inclui α em β_x; e *(vi) ddua* α *na granularidade* β_x *é incomparável com ddua* α *na granularidade* β_y se α em β_x não inclui α em β_y, nem α em β_y inclui α em β_x.

Os teoremas 5.1 a 5.5 utilizam a *abordagem precisa* para a noção de redefinição de dados persistentes, a qual foi introduzida na Seção 4.1: a redefinição persistente ocorre em nível *valor de atributo*, independentemente da granularidade utilizada para a análise de cobertura da associação persistente. A motivação para esta abordagem é a determinação exata dos dados definidos que alcançam os usos persistentes, para então efetuar a avaliação de cobertura em cada granularidade de fluxo de dados.

Teorema 5.1: A associação de interação α na granularidade *atributo* inclui estritamente α na granularidade *relação*.

Prova: A dupla $<\Pi, \Gamma>$ cobre uma associação de interação α entre as regras *r'* e *r''* na granularidade β se o conjunto de caminhos Π inclui o exercício de algum caminho π que

cubra α e o conjunto de dados manipulados $\gamma \in \Gamma$ ao longo de π exercita o fluxo de dados de α na precisão β. Especificamente para a granularidade *relação*, o exercício de algum caminho que cubra a associação, o que inclui o exercício das ocorrências de definição e de uso que caracterizam a associação, é suficiente para a cobertura neste nível de precisão de fluxo de dados, independentemente dos dados manipulados (por exemplo, não é necessário certificar-se se um mesmo atributo definido está sendo usado). Uma condição necessária para a dupla $<\Pi, \Gamma>$ cobrir α na granularidade *atributo* é que algum caminho π exercite a associação tal que, ao longo de π, não seja vazia a interseção entre os conjuntos de atributos definidos e usados com respeito à variável persistente atribuída à associação, nos pontos das ocorrências de definição e de uso que caracterizam α, respectivamente. Neste sentido, ao se cobrir a associação em nível *atributo*, estar-se-á cobrindo-a também em nível *relação*. Por outro lado, a cobertura em nível *relação* não garante que a mesma seja coberta em nível *atributo*, visto que não é exigido que as ocorrências de definição e de uso persistentes recaiam sobre um mesmo atributo.

Teorema 5.2: A associação de interação α na granularidade *tupla* inclui estritamente α na granularidade *relação*.

Prova: Sejam os conjuntos Tup_d *e* Tup_u de *tuplas* definidas e usadas, respectivamente, nas ocorrências de definição e de uso que caracterizam a associação de interação α, pela aplicação do caso de teste λ. Para que um conjunto de casos de teste Λ cubra α em nível *tupla*, é necessário que a condição $Tup_d \cap Tup_u \neq \varnothing$ seja satisfeita ao longo de um caminho decorrente de Λ. Desta forma, considerando-se que serão aplicados casos de teste capazes de cobrir α em nível *tupla*, então a associação também será coberta em nível *relação*, pois foi exercitado o fluxo de dados para a relação que possui as *tuplas* dos conjuntos Tup_d *e* Tup_u. No caso de se cobrir α em nível *relação*, a citada condição não é requerida, não garantindo, portanto, que α será coberta em nível *tupla*.

Teorema 5.3: A associação de interação α na granularidade *atributo* é incomparável com α na granularidade *tupla*.

Prova: As coberturas de uma associação de interação nos níveis *atributo* e *tupla* endereçam aspectos distintos de fluxo de dados. Considere que a associação de interação α refere-se ao fluxo de dados da variável persistente rel_x. Seja o caso de teste λ_1, o qual cobre a associação α para os atributos att_1 e att_3 da relação rel_x, pela definição das *tuplas* tup_5 e tup_{15} e uso das *tuplas* tup_8 e tup_9; considere também o caso de teste λ_2, que cobre α para as *tuplas* tup_8 e tup_9 da relação rel_x, pela definição do atributo att_3 e uso do atributo att_1. O caso de teste λ_1 cobre a associação nas precisões *relação* e *atributo*, enquanto que λ_2 cobre α nos níveis *relação* e *tupla*. No entanto, λ_1 não cobre α no nível *tupla*, nem λ_2 cobre α no nível *atributo*, provando que α na granularidade *atributo* é incomparável com α na granularidade *tupla*.

Teorema 5.4: A associação de interação α na granularidade *valor de atributo* inclui estritamente α nas granularidades *atributo* e *tupla*.

Prova: Em adição aos casos de teste λ_1 eλ_2, que foram introduzidos na prova do Teorema 5.3, considere o caso de teste λ_3 que cobre a associação α, onde os exercícios das ocorrências de definição e de uso que caracterizam α efetuam, respectivamente, a definição dos atributos att_1 e att_3 nas *tuplas* tup_5 e tup_{15} da relação rel_x e o uso dos atributos att_3 e att_5 nas *tuplas* $tup1_5$ e tup_{25} da relação rel_x. A prova deste teorema fundamenta-se no fato do nível de granularidade *valor de atributo* incorporar simultaneamente as restrições de fluxo de dados dos níveis *atributo* e *tupla* sendo, portanto, o nível mais exigente. Nesse sentido, λ_3 cobre α na precisão *valor de atributo*, o que resulta em λ_3 também cobrir α nos níveis *atributo* e *tupla*; no entanto, o nível *valor de atributo* não é alcançado na cobertura de α pelos casos de teste λ_1 eλ_2.

Teorema 5.5: As relações de inclusão estabelecidas pelos teoremas 5.1, 5.2, 5.3 e 5.4 são válidas mesmo na presença de não executabilidade.

Prova: Conforme foi mencionado anteriormente, a executabilidade de uma associação de interação α na granularidade de fluxo de dados β está relacionada às questões: se existem

dados de teste capazes de executar pelo menos um caminho completo que cubra α; e se os dados manipulados ao longo deste caminho exercitam α na precisão β. O teorema não foca a executabilidade de um conjunto de associações (por exemplo, para distinguir as associações executáveis dentre as requeridas por um dado critério de teste), mas se refere à consideração individual de associações, ou seja, sua executabilidade para cada granularidade de fluxo de dados. Nesse sentido, existe uma hierarquia quanto à executabilidade de precisão de fluxo de dados, a qual coincide com a hierarquia da relação de inclusão entre granularidades. Por exemplo, se a associação de interação α na granularidade *atributo* inclui estritamente α na granularidade *relação*, e se α é não exercitável na precisão *relação*, então α também é não exercitável na precisão *atributo*. Esta afirmativa decorre da dependência entre os predicados de fluxo de dados próprios de cada granularidade, a qual baseia a relação de inclusão entre granularidades. Portanto, se existe uma relação de inclusão entre duas granularidades para uma dada associação de interação α, tal que *α na granularidade β_x inclui estritamente α na granularidade β_y*, se α é não exercitável *na granularidade β_y*, então α também é não exercitável *na granularidade β_x*.

O par composto por critério baseado na interação entre regras e granularidade de fluxo de dados, denotado por *(C, β)*, determina um crivo para a atividade de teste. Quando se busca satisfazer o critério C_x na granularidade β_y, estabelece-se um novo requisito de teste. Na Figura 5.13, as notações *(C, b)*, *(C, r)*, *(C, a)*, *(C, t)* e *(C, va)* denotam requisitos de teste para as granularidades *banco de dados*, *relação*, *atributo*, *tupla* e *valor de atributo*, respectivamente; e *C* refere-se a um critério de teste baseado na interação entre regras. As setas indicam uma relação de inclusão entre requisitos de teste: se uma seta inicia em *(C, β_1)* e termina em *(C, β_2)*, então todo conjunto de teste que satisfaz *(C, β_1)* também satisfaz *(C, β_2)*, mas o inverso não é verdadeiro; em outras palavras, *(C, β_1) inclui estritamente (C, β_2)*. Esta relação de inclusão é mantida mesmo na presença de requisitos não exercitáveis. As relações de inclusão entre requisitos de teste contidas na Figura 5.13 são um corolário das provas dos Teoremas 5.1 a 5.5, visto que os critérios propostos são baseados nas associações de interação tratadas em tais teoremas.

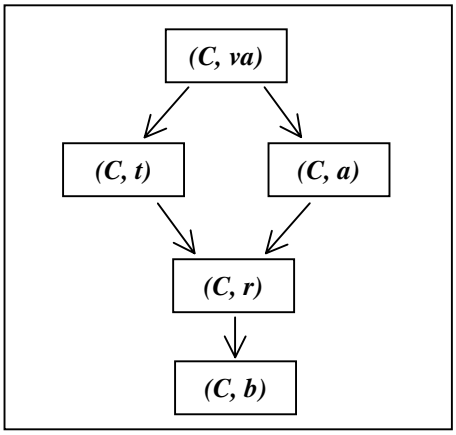


Figura 5.13: Relação de inclusão entre requisitos de teste definidos pelo par *(critério, granularidade)*.

Definição:

Uma associação de interação α possui *executabilidade exclusiva* na granularidade β, se e somente se: *(i)* α é executável em β; e *(ii)* α é não executável em qualquer granularidade mais precisa do que β. São identificados três cenários para executabilidade exclusiva: *(1)* α é executável no nível *relação* e é não executável nos níveis *tupla*, *atributo* e *valor de atributo*; *(2)* α é executável no nível *tupla* e é não executável no nível *valor de atributo*; e *(3)* α é executável no nível *atributo* e é não executável no nível *valor de atributo*. Este conceito será utilizado nas análises efetuadas na Seção 5.3.

Os teoremas que seguem buscam comparar critérios de teste pelo estabelecimento de uma relação de inclusão entre critérios. Dados o critério C, o programa P e a especificação S, a expressão $T \in C(P, S)$ significa que o conjunto de teste T satisfaz C com respeito à (P, S); no contexto de interação entre regras, P e S constituiriam um conjunto de regras e sua especificação funcional, respectivamente. O *critério* C_1 *inclui o critério* C_2, se e somente se, para todo (P, S), qualquer conjunto de teste T que cobre C_1 também cobre C_2. O *critério* C_1 *inclui estritamente o critério* C_2, se e somente se, C_1 inclui C_2, mas C_2 não inclui C_1. O *critério* C_1 *é incomparável com o critério* C_2, se e somente se, C_1 não inclui C_2, nem C_2 inclui C_1.

Teorema 5.6: Os critérios baseados nas associações de interação são incomparáveis com o critério *cobertura de seqüência* (Vaduva, 1999).

Prova: A abordagem de Vaduva (1999) para o teste de regras, a qual foi descrita no Capítulo 2, busca a execução de tantas seqüências de regras quanto possível, onde todo requisito de teste é identificado por uma seqüência de regras extraída de um conjunto de regras em teste. Para qualquer conjunto de regras *R*: *(i)* as exigências do critério *cobertura de seqüência* ignoram o fluxo de controle em cada regra executada e o exercício de quaisquer associações de fluxo de dados, inclusive as associações de interação entre regras; e *(ii)* os critérios baseados na interação entre regras estabelecem requisitos de fluxo de dados persistentes para pares de regras, cujo cumprimento em geral pode ser atingido por seqüências específicas de regras, as quais representam um subconjunto das possíveis seqüências obtidas a partir de *R*. Portanto, os critérios baseados nas associações de interação e o critério *cobertura de seqüência* endereçam aspectos disjuntos de um conjunto de regras sendo, portanto, incomparáveis.

Teorema 5.7: Os critérios baseados nas associações de interação são incomparáveis com o critério *todos os caminhos*.

Prova: Conforme foi mencionado no Capítulo 4, a literatura comenta os vários níveis de granularidade para a cobertura de associações persistentes, onde são estabelecidos requisitos de teste do tipo *(critério, granularidade)*, para os quais é necessário monitorar os dados definidos e usados ao longo dos caminhos executados pela aplicação de casos de teste. Tal característica está em sintonia com a constatação de Ostrand e Weyuker (1991) para programas escritos em linguagens com ponteiros: um conjunto de teste que exercita todos os caminhos de um programa não necessariamente exercita cada associação *def-uso*. O critério *todos os caminhos* não objetiva atender a requisitos do tipo *(critério, granularidade)*, visto que o aspecto *granularidade* requer conhecimento sobre os dados definidos e usados, além dos caminhos percorridos. Por outro lado, a cobertura de todas as associações de interação pode ser alcançada sem o exercício dos caminhos completos desprovidos de qualquer comando de manipulação de dados persistentes. Em adição, os critérios representados pelas associações de interação baseadas unicamente na dependência de dados persistentes – *todos-dusos-AC* e *todos-dusos-AA* – são cobertos por

conjuntos de teste que não exigem a transferência de controle entre regras ativas, não sendo, portanto, satisfeitos por conjuntos de teste adequados ao critério *todos os caminhos*.

Corolário: Os critérios baseados nas associações de interação são incomparáveis com critérios baseados unicamente em fluxo de controle, tais como *todos os nós* e *todos os arcos*.

Teorema 5.8: Os critérios baseados nas associações de interação são incomparáveis com os *critérios de fluxo de dados* propostos por Rapps e Weyuker (1985) e por Frankl e Weyuker (1988) e com os *critérios potenciais usos* introduzidos por Maldonado (1991) e por Maldonado et al. (1992).

Prova: As famílias de *critérios de fluxo de dados* e os *critérios potenciais usos* abstraem requisitos de teste baseados em associações e em potenciais associações de fluxo de dados para variáveis de programa, respectivamente. Entretanto, tais critérios, originalmente, não exigem o exercício de relações de fluxo de dados persistentes. Considere as regras *trg001*, *trg002* e *trg005*, conforme exibido nas Figuras 5.1, 5.2 e 5.3, respectivamente. Em ambas as famílias de critérios, não são observadas relações de fluxo de dados entre regras, pois as variáveis de programa – *v_exists* na regra *trg001* e *v_count* na regra *trg002* – são de escopo local às regras em que foram declaradas, não alcançando, portanto, o fluxo inter-regra. Em adição, a cobertura das associações requeridas pelos *critérios de fluxo de dados* – *<6, (8,9), v_exists>*, *<6, (8,10), v_exists>*, *<6, (11,12), v_exists>*, *<6, (11,13), v_exists>*, *<21, (22,23), v_exists>*, *<21, (22,24), v_exists>*, *<2, (4,5), v_count>*, *<2, (4,9), v_count>*, *<2, (7,8), v_exists>* e *<2, (7,9), v_count>* – e pelos *critérios potenciais usos* é independente do fluxo de dados persistentes, tornando-os incomparáveis com os critérios baseados em associações de interação.

Teorema 5.9: Os critérios baseados nas associações de interação são incomparáveis com os critérios para aplicações de banco de dados relacional propostos por Spoto (2000).

Prova: A abordagem de Spoto (2000) possui um conceito próprio de *definição persistente* em aplicações de banco de dados. Para estabelecer uma associação definição-uso, é requerido o exercício de dois nós, os quais assinalam a definição persistente: o nó que

possui o comando que define os dados persistentes (por exemplo, *insert*, *delete* e *update*) e o nó que possui o comando que sanciona a transação de banco de dados (comando *commit*). Assim, toda definição de dados é primeiramente confirmada no âmbito da base de dados para, então, se estabelecerem relações de fluxo de dados entre essas ocorrências de definição persistente e potenciais usos, por caminhos livres de definição c.r.a variáveis persistentes. Considere as unidades de programa *X* e *Y* de uma aplicação de banco de dados representada no grafo da Figura 5.14, que utiliza a notação de Spoto (2000): os nós circulares denotam comandos da linguagem hospedeira; e os nós retangulares denotam comandos da SQL. Se dados de teste forem aplicados tal que: o caminho $(1\ 2\ 3\ 5\ 7)_X$. $(1\ 2\ 3\ 4)_Y$. $(8)_X$ seja exercitado, e os critérios baseados na interação entre regras sejam cobertos, estar-se-á deixando de cobrir os critérios para aplicações de banco de dados relacional, pois o exercício do nó *4* da regra *X* é necessário para a definição persistente de *tab001*, demonstrando que estes critérios não são incluídos pelos critérios baseados na interação entre regras. Por outro lado, se o caminho $(1\ 2\ 3\ 4\ 5\ 7)_X$. $(1\ 2\ 3\ 4)_Y$. $(8)_X$ for exercitado, a cobertura da associação caracterizada pela definição persistente *<2,4>* em *X* e pelo uso no arco *(3,4)* em *Y* c.r.a *tab001* pode ser alcançada, mas as coberturas das associações de interação entre regras não são requisitadas para os dados de teste (por exemplo, pelo disparo de regras a partir do nó *2* da regra *X*), resultando, portanto, que os critérios para aplicações de banco de dados não incluem os critérios baseados na interação entre regras.

Teorema 5.10: Os critérios baseados nas associações de interação são incomparáveis entre si.

Prova: Os critérios propostos buscam exercitar aspectos disjuntos de um conjunto de regras, tornando-os incomparáveis entre si. Para realizar a demonstração do teorema, considere os elementos requeridos para os critérios *todos-dusos-AA*, *todos-dusos-AC*, *todos-dusos-AE*, *todos-dusos-AA de chamada*, *todos-dusos-AA de retorno* e *todos-dusos-AC de chamada* apresentados nas Figuras 5.7 a 5.12, respectivamente. As análises a seguir são realizadas de acordo com o alcance sintático de caminhos, analisando critérios dois-a-dois; não se objetiva realizar um exame completo dos critérios, comparando cada critério com todos os outros, mas apresentar algumas evidências que levem à inferência da validade do teorema.

- *Os critérios todos-dusos-AA e todos-dusos-AC são incomparáveis*: o primeiro pode ser satisfeito sem o disparo da regra *trg005*, condição necessária à satisfação do segundo (requisitos *(49)* e *(50)*); o segundo pode ser satisfeito a partir do exercício dos caminhos $(1\ 2\ 3\ 5\ 6\ 7\ 8\ 10\ 15\ 16\ 20\ 21\ 22\ 24\ 25)_{trg001}$. $(1\ 2\ 3\ 4)_{trg005}$ e $(1\ 2\ 3\ 5\ 6\ 7\ 8\ 10\ 15\ 16\ 20\ 21\ 22\ 24\ 25)_{trg001}$. $(1\ 2\ 4)_{trg005}$, os quais não inclui o nó *9* da regra *trg001*, cujo exercício é necessário à cobertura das associações *(5)*, *(6)* e *(7)*, as quais são requeridas pelo primeiro.
- *Os critérios todos-dusos-AC e todos-dusos-AC de chamada são incomparáveis*: o primeiro critério pode ser satisfeito pela execução dos caminhos $(1\ 2\ 3\ 5\ 6\ 7\ 11\ 13\ 14\ 15\ 16\ 20\ 21\ 22\ 24\ 25)_{trg001}$. $(1\ 2\ 3\ 4)_{trg005}$ e $(1\ 2\ 3\ 5\ 6\ 7\ 11\ 13\ 14\ 15\ 16\ 20\ 21\ 22\ 24\ 25)_{trg001}$. $(1\ 2\ 4)_{trg005}$, mas não cobre as associações *(85)* e *(86)*, requeridas pelo segundo critério, que exigem o disparo da regra *trg005* a partir da execução da operação atribuída ao nó *12* da regra *trg001*; o segundo critério pode ser satisfeito pela execução dos caminhos $(1\ 2\ 3\ 5\ 6\ 7\ 11\ 12)_{trg001}$. $(1\ 2\ 4)_{trg005}$. $(14\ 15\ 16\ 20\ 21\ 22\ 24\ 25)_{trg001}$ e $(1\ 2\ 3\ 5\ 6\ 7\ 11\ 12)_{trg001}$. $(1\ 2\ 3\ 4)_{trg005}$. $(14\ 15\ 16\ 20\ 21\ 22\ 24\ 25)_{trg001}$, mas não executa qualquer subcaminho final de *trg001* iniciado no nó 2, exigido para a cobertura das associações *(49)* e *(50)*, as quais são requeridas pelo primeiro.
- *Os critérios todos-dusos-AA de chamada e todos-dusos-AA de retorno são incomparáveis*: o primeiro critério requer o exercício do uso de dados persistentes nos arcos *(2,3)* e *(2,9)* da regra *trg002* (requisitos *(55)* a *(60)*) cujo exercício pode ser alcançado sem passar pelos nós *4* e *7* da regra *trg002*, os quais são necessários à cobertura dos requisitos do segundo critério (requisitos *(61)* a *(84)*); por outro lado os nós *4* e *7* da regra *trg002* podem ser exercitados sem incluir o arco *(2,9)*, o qual é exigido pelas associações *(56)*, *(58)* e *(60)* requeridas pelo primeiro critério.

Dessa forma, pode-se estabelecer cenários de fluxo de controle que demonstrem que os critérios baseados na interação entre regras são incomparáveis entre si.

A complexidade de um critério é um conceito pertinente à atividade de teste, sobretudo para apoiar a estimativa do custo de aplicação do critério. A complexidade é estabelecida pelo número de casos de teste exigidos pelo critério para o pior caso, embora, na prática, este cenário seja pouco provável de ocorrer. Dados um critério de interação entre regras *C* e qualquer

conjunto de regras *R* e especificação *S*, se existir um conjunto de casos de teste *Λ* que seja *C–adequado* para *(R, S)*, então existe um conjunto de casos de teste *Λ'* tal que a cardinalidade de *Λ'* é menor ou igual à complexidade do critério *C*.

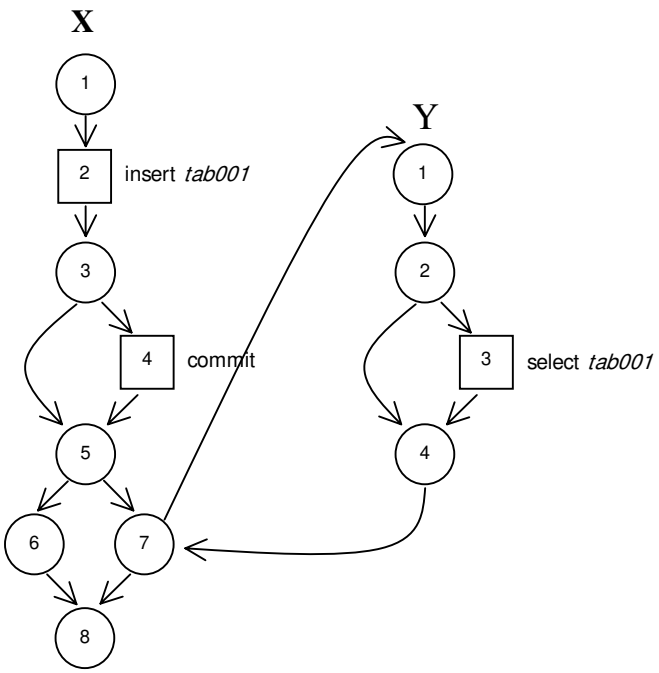


Figura 5.14: Exemplo de grafo de fluxo de controle entre duas unidades de programa (*X* e *Y*), usando a notação de Spoto (2000).

Segundo Maldonado (1991), a complexidade dos critérios *todos os usos* é 2^t, onde a estrutura que maximiza o número de casos de testes é dada por um grafo de programa que representa uma seqüência de *t* comandos *if-then-else* dentro de um laço. As análises realizadas pelo autor também determinaram que a complexidade dos critérios *potenciais usos* é da ordem de 2^t. Apesar dos critérios baseados nas associações de interação serem, em sua essência, uma extensão dos critérios *todos os usos* para o contexto de interação entre regras ativas escritas em SQL, eles não herdam a complexidade dos critérios *todos os usos*, conforme demonstrado abaixo.

Na Subseção 5.3.5, serão discutidos alguns dos resultados de um experimento para a aplicação dos critérios propostos. É importante realçar a constatação de que casos de teste reveladores de defeito estão relacionados à cobertura de associações de interação nas diversas precisões de fluxo de dados. Tal constatação vai ao encontro do objetivo primário da atividade de teste – revelar a presença de defeitos – e representa uma contribuição importante às iniciativas de

pesquisa que buscam estender abordagens de teste baseadas em fluxo de dados para o contexto de bancos de dados relacionais.

Considere os grafos de regras ativas da Figura 5.15, que possuem exemplos para iniciar uma discussão sobre a complexidade dos critérios baseados na interação entre regras. No contexto de teste de unidade, no grafo exibido na Figura 5.15(a) são estabelecidas 32 associações de interação para a variável persistente *tab001*. Vale ressaltar que o comando representado pelo nó *3* não caracteriza uma redefinição dos dados persistentes definidos no nó *2*, pois o requisito *(critério, granularidade)* não é satisfeito apenas pelo exercício de caminhos. Note que o limite 2^t para os critérios baseados em fluxo de dados é dado em função da estrutura de controle, sem considerar os dados persistentes definidos e usados em conseqüência da aplicação de casos de teste. Em 5.15(b), procura-se realçar que associações de interação são estabelecidas somente para entidades de banco de dados e que a presença de estruturas de seleção de controle não determina a existência de associações dessa natureza. Não são observadas associações de interação para a variável persistente *tab003*, devido à ausência de ocorrências de definição de dados persistentes, mesmo existindo uma ocorrência do comando *select* no nó 2.

Para demonstrar a complexidade dos critérios propostos, considere trechos dos grafos das regras *X* e *Y*, não necessariamente distintas, mostrados na Figura 5.16. Os nós representam comandos de manipulação de dados; a seta tracejada em negrito constitui a ocorrência de disparo entre regras, as outras setas tracejadas representam os arcos para as exceções ocorridas na execução de comandos de manipulação, as setas pontilhadas denotam a continuidade da seqüência de nós e as demais setas complementam o fluxo de controle intra-regra; e, por fim, as linhas que unem os nós das regras *X* e *Y* indicam a existência de interação entre regras baseada em fluxo de dados persistentes, onde o sentido desse fluxo é indicado na parte inferior de cada grafo.

É importante ressaltar que os critérios baseados na interação entre regras não possuem sua complexidade diretamente atrelada ao fluxo de controle intra-regra, pois: *(i)* cada elemento requerido é uma associação de interação, cuja cobertura é alcançada por algum caminho livre de definição (não são requeridos *du-caminhos*); *(ii)* em geral, a redefinição de dados persistentes não é determinada pelo exercício de um nó (ou de um caminho), pois a definição e o uso de dados persistentes são sensíveis aos dados de teste; e *(iii)* os caminhos completos que não incluem quaisquer nós representativos de comandos de manipulação não são em geral exercitados por

casos de teste reveladores de defeito. Nesse sentido, os grafos da Figura 5.16 não objetivam evidenciar qual estrutura de fluxo de controle intra-regra apóia a maximização do número de elementos requeridos (tal fluxo é tratado de forma genérica), mas explorar a interação entre regras que maximiza o número de associações de interação.

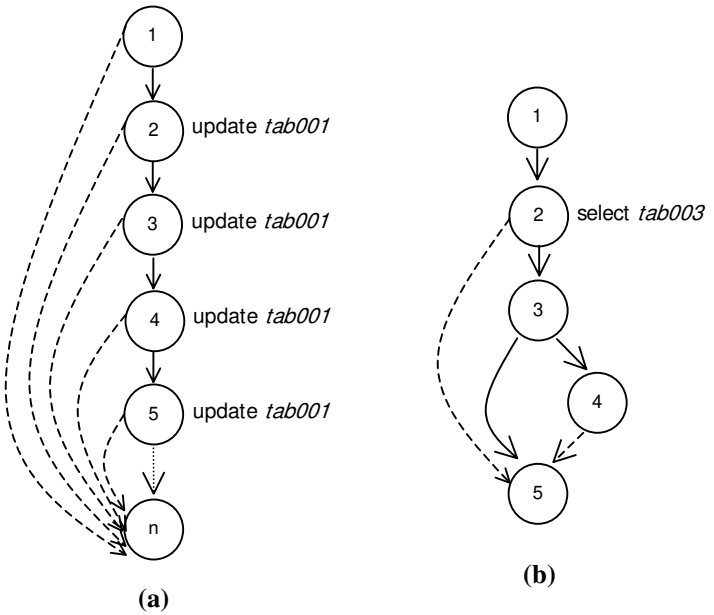


Figura 5.15: Exemplos para a discussão sobre a complexidade dos critérios baseados na interação entre regras.

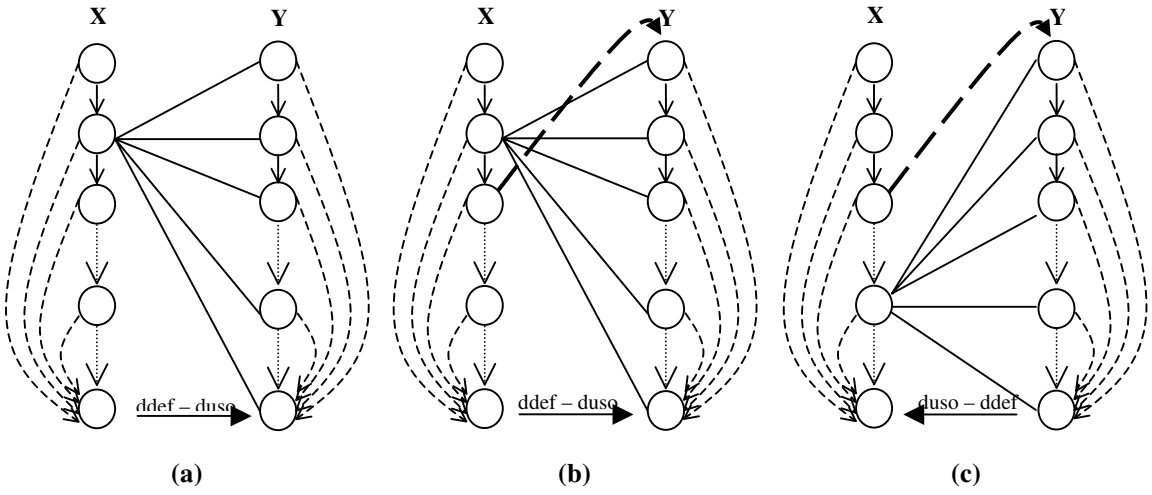


Figura 5.16: Exemplos para a discussão sobre a complexidade dos critérios baseados na interação entre regras, com fluxo de controle intra-regra genérico.

Visando a maximizar o número de associações de interação, considere que todo nó nos grafos da Figura 5.16 constitui um comando de manipulação que caracteriza uso e definição de dados (por exemplo, comando *update*); devido a esse enfoque, são abstraídas associações de interação entre a definição que há no próprio nó e os usos existentes em seus arcos de saída, onde, por exemplo, sua cobertura possivelmente ocorrerá em disparos seqüenciais da regra que possui o nó. Para introduzir a complexidade dos critérios propostos, seja *m* o número de nós que possuem definição e uso de dados persistentes para uma dada entidade de banco de dados.

Teorema 5.11: Os critérios baseados nas associações de interação possuem complexidade de ordem polinomial.

Prova: ***(I)*** A complexidade para o critério *todos-dusos-AA* é dada por $\mathbf{2.m^2}$. No grafo em 5.16(a), cada nó possui interação com os dois arcos de saída de cada nó existente, incluindo ele mesmo. Considere o grafo G^{m-1}, possuindo $m - 1$ nós que possuem definição e uso de dados persistentes, cuja complexidade é C^{m-1} é $2.(m-1)^2$. Ao se acrescentar um novo comando de manipulação neste grafo, constituindo o grafo G^m, serão estabelecidas $2.m$ novas associações de interação entre o nó sendo incluído para os arcos de saída de todos os nós, e $2.(m-1)$ novas associações de interação entre os nós até então existentes para os arcos de saída do nó sendo incluído; portanto a complexidade de G^m é calculada $C^m = C^{m-1} + 2.m + 2.(m - 1) = 2.m^2$. ***(II)*** A complexidade para o critério *todos-dusos-AC* é dada por $\mathbf{2.m}$, pois o uso de dados persistentes ocorre especificamente nos dois arcos de saída da condição de regra. Nos grafos em 5.16(b) e 5.16(c), as relações de fluxo de dados persistentes exploram as transferências de controle entre regras, com associações de interações estabelecidas a partir dos fluxos de controle de chamada e de retorno, respectivamente. ***(III)*** A complexidade para o critério *todos-dusos-AA de chamada* é dada por $\mathbf{2.(m^3 - m^2)}$. Se em 5.16(b) todos os nós possuem operações de disparo entre regras, então em cada disparo existem $m - 1$ nós com definição de dados persistentes, os quais podem interagir com o uso de dados persistentes que ocorre nos dois arcos de saída dos *m* nós do grafo; portanto, $C^m = 2.m.(m - 1).m = 2.(m^3 - m^2)$. ***(IV)*** Esta complexidade também é estabelecida para o critério *todos-dusos-AA de retorno*, onde o fluxo de dados ocorre no sentido inverso (da regra *Y* para a regra *X*), conforme exibido em 5.16(c). ***(V)*** Para o critério *todos-dusos-AC de chamada*, a complexidade é dada por $\mathbf{2.(m^2 - m)}$, se

todos os nós possuem operações de disparo entre regras, então em cada disparo existem *m – 1* nós com definição de dados persistentes, os quais podem interagir com o uso de dados persistentes que ocorre nos dois arcos de saída do nó atribuído à condição de regra; portanto, $C^m = 2.m.(m-1) = 2.(m^2 - m)$.

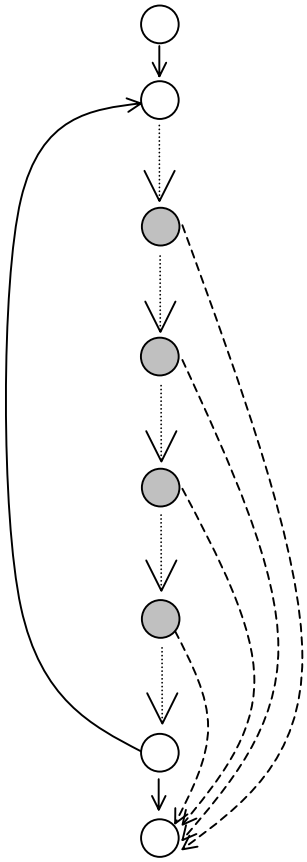


Figura 5.17: Exemplo de fluxo de controle intra-regra que apóia a maximização do número de associações de interação.

A Figura 5.17 mostra um exemplo de fluxo de controle intra-regra que apóia a maximização do número de associações de interação, onde os nós preenchidos representam comandos de manipulação de dados. Vale ressaltar que, em requisitos do tipo *(critério, granularidade)*, a redefinição de dados é sensível aos dados manipulados em cada ocorrência de manipulação; o exercício de caminhos não é suficiente para determinar a redefinição de dados persistentes. O cenário de fluxo de controle intra-regra necessário para alcançar a complexidade dos critérios *todos-dusos-AA de chamada* e *todos-dusos-AC de chamada* é que, para todo disparo

entre regras, *m – 1* definições de dados persistentes (em todos os nós de manipulação de dados, exceto o nó que provocou o disparo entre regras) alcancem os arcos da regra disparada que possuem uso de dados persistentes, seguindo o fluxo da transferência de controle entre regras. Nesse sentido, se N_s é o conjunto de nós de um grafo de regra, onde qualquer elemento de N_s seja um nó que representa comandos de manipulação, então para todo nó $n_s \in N_s$, o fluxo de controle intra-regra deve permitir que exista algum caminho π que exercite todos os nós de N_s, tal que o último nó em π pertencente a N_s seja n_s, e o exercício de n_s provoque o disparo entre regras (para o critério *todos-dusos-AA de retorno*, é requerido que o nó n_s seja o primeiro nó de N_s em π).

5.3 Aplicação dos Critérios Propostos

O teste de regras baseado na interação entre regras tem o objetivo primário de detectar a presença de defeitos que porventura não tenham sido revelados durante o teste individual de regras. Essa categoria de teste pode ser vista como uma técnica para sistematizar a execução integrada de um conjunto de regras. Sobre o teste individual de regras, Leitão et al. (2002) e Cardoso (2004) propuseram a aplicação de técnicas baseadas em fluxo de dados no contexto do teste de regras isoladas. Apesar de terem sido testadas individualmente, quando as regras são colocadas para interagir umas com as outras, podem surgir falhas em resposta a essa interação. Ainda, segundo Widow e Ceri (1996), um pequeno conjunto de regras pode ser complexo para entender e para gerenciar, e sua execução conjunta requer abordagens de garantia de qualidade na direção da sua corretude.

Algumas categorias de defeitos ligados à interação entre regras são:

- *Interface de regra*: constituem defeitos situados na descrição do evento ou na operação de sua condição; descrição da operação de disparo incorreta, operação de disparo incorreta, disparo indevido entre regras e condição de regra indevidamente presente são exemplos de defeito dessa categoria.
- *Funcionalidade*: a execução conjunta de várias regras, com ou sem disparo explícito entre elas, resulta em uma funcionalidade indevida implementada pelo conjunto de regras; regras com operação em duplicata, princípios de negócio não complementados

corretamente pela execução conjunta de regras e operação indevida nas relações de fluxo de dados persistentes entre regras são exemplos de defeitos dessa categoria.

Nos exemplos acima, o defeito localiza-se primariamente nas ocorrências de manipulação que participam de associações de interação ou na declaração de *interface* propriamente dita. O modelo de interação entre regras ativas, que foi introduzido no Capítulo 4, é fundamentado em ocorrências de manipulação de dados que determinam associações persistentes entre regras ativas escritas em SQL. Essas ocorrências são susceptíveis a defeitos de manipulação, tal como qualquer outra ocorrência de manipulação. O experimento descrito nesta seção busca a descoberta de defeitos nessas ocorrências, os quais podem dar origem a interações imperfeitas entre regras ativas.

5.3.1 Objetivos

Este estudo endereça duas questões para a aplicação dos Critérios Baseados na Interação entre Regras. As seguintes hipóteses são investigadas:

(h1) A cobertura das associações de interação em níveis mais precisos de granularidade agrega valor a sua habilidade para a detecção de defeitos de manipulação, em adição à elevação do custo do teste: os defeitos de manipulação se manifestam nos vários níveis de granularidade de forma suplementar.

(h2) Os critérios baseados nas associações de interação representam critérios de adequação que apóiam à descoberta de defeitos na interação entre regras ativas.

5.3.2 Aspectos de Análise

A comparação entre os vários níveis de granularidade perpassa o aumento de custo do teste e conduz à discussão sobre a influência da granularidade da análise de fluxo de dados persistentes na descoberta da presença de defeitos de manipulação. A Figura 5.13 ilustra o grafo da relação de inclusão para a cobertura de associações de interação em distintas granularidades e intui que a adequação a níveis mais precisos de granularidade eleva o custo do teste. Essa relação de inclusão determina que os domínios de entrada para a cobertura de qualquer associação em distintas granularidades são passíveis de comparação, excetuando-se a comparação entre as

granularidades *tupla* e *atributo*. O aumento de custo não se dá pelo incremento do número de elementos requeridos, mas pela redução do domínio de entrada referente à cobertura de associações em cada granularidade. Sejam $D_{\beta 1}$ e $D_{\beta 2}$ os domínios de entrada adequados a uma particular associação em *(R, S)* nas granularidades β_1 e β_2, respectivamente; então selecionar elementos de $D_{\beta 1}$ é mais difícil do que encontrar elementos de $D_{\beta 2}$ quando $D_{\beta 1} \subseteq D_{\beta 2}$.

A investigação sobre a descoberta de defeitos usa os espaços de entrada adequados a cada granularidade para observar se as regiões desses espaços são suplementares para revelar a presença de defeitos de manipulação. Os tipos de defeito introduzidos no Capítulo 3 capturam a essência dos defeitos possíveis em comandos de manipulação e são focados nos itens estruturais desses comandos e em qualificadores de incorretude, tais como *ausente* e *incorreto*. Seja *Fault(D)* o conjunto de defeitos em *(R, S)* possíveis de serem revelados por algum elemento do domínio de entrada *D*, tal que qualquer elemento de *Fault(D)* é um defeito de manipulação. Se $D_{\beta 1} \subset D_{\beta 2}$, dizemos que $D_{\beta 1}$ e $D_{\beta 2}$ são suplementares com respeito à descoberta de defeitos se, e somente se, para todo *(R, S)* e toda associação de interação em *(R, S)*, $Fault(D_{\beta 2} - D_{\beta 1}) \neq Fault(D_{\beta 1})$ e $Fault(D_{\beta 1}) \subseteq Fault(D_{\beta 2})$.

A hipótese *(h2)* investiga se a abordagem de teste proposta ampara a descoberta de defeitos na interação entre regras escritas em SQL. Em um experimento similar, Kapfhammer e Soffa (2003) ensaiam empiricamente sua abordagem de teste baseada em relações de fluxo de dados manipulados via SQL, verificando aspectos ligados a custo, tal como quantidade de associações de fluxo de dados, mas não analisam se os critérios propostos apóiam a descoberta de defeitos existentes em programas de aplicação de banco de dados. Muitos fatores estão envolvidos na apreciação empírica se um critério de adequação apóia a descoberta de defeitos. No caso dos critérios baseados na interação entre regras ativas, dentre vários fatores, são citados os seguintes: a técnica adotada para a geração de dados de teste; os dados de entrada gerados a partir da técnica adotada; o conjunto de regras ativas utilizadas no teste; os tipos de defeito monitorados; e os defeitos existentes nas regras.

5.3.3 Descrição do Experimento

O primeiro aspecto é a busca por regras representativas da população de regras escritas em SQL. Observou-se que empresas relutam em disponibilizar suas regras ativas, pois estas

incorporam a tecnologia do negócio em questão; mesmo assim, o contato com algumas empresas foi positivo, pois foi reforçada a importância do uso de *triggers* em aplicações de banco de dados, e que esforços adicionais eram alocados ao teste de regras que possuíam alguma interação.

O experimento foi conduzido utilizando o sistema gerenciador de banco de dados *Oracle*; tal sistema tem sido usado pela comunidade acadêmica em seus experimentos (Daou et al., 2001) e suporta o modelo adotado pela tese para a interação de regras ativas, conforme descrito na Seção 4.3. O sistema *Oracle* utilizado no experimento possui um banco de dados exemplo, o qual é comumente mencionado na documentação deste sistema; o Apêndice B ilustra um modelo conceitual inspirado neste banco de dados. As regras presentes no banco de dados foram estendidas para compor o conjunto de regras R_x, o qual possui aspectos de interação e semântica compatível com o esquema da base de dados. A Figura 5.18 apresenta os grafos de interação para o conjunto R_x. O Apêndice A apresenta o código fonte de tais regras, acrescido de comandos de instrumentação, conforme os modelos de implementação introduzidos no Capítulo 6.

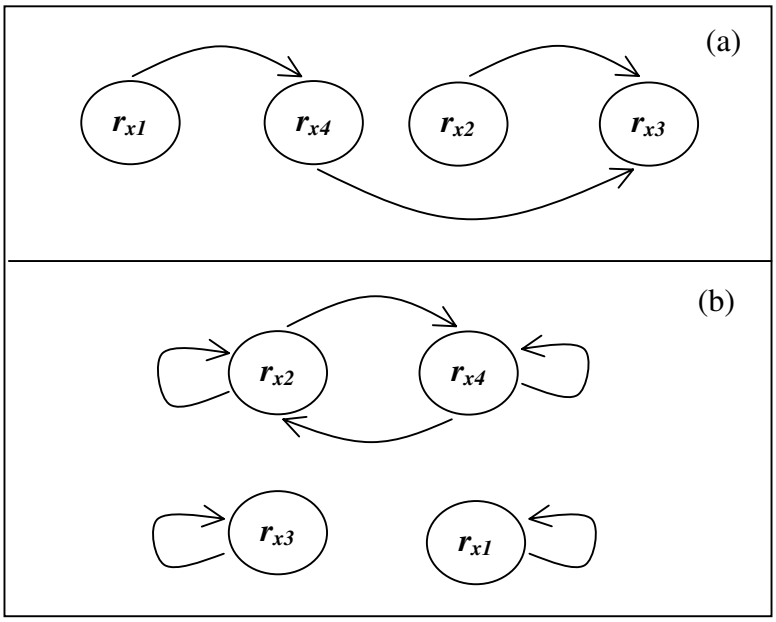


Figura 5.18 – Grafos de interação para o conjunto de regras $R_x = \{ r_{x1}, r_{x2}, r_{x3}, r_{x4} \}$, cuja denominação implementada é R_x = *{ TRG_COMMIT_ITEMS, TRG_INVENTORY_QUANTITY, TRG_PRODUCT_PENDING, TRG_PRODUCT_QUANTITY }*: (a) grafo *AE*; (b) grafo *AA*.

A abordagem para a geração de bases para teste utilizada no experimento é inspirada em Ostrand e Balcer (1988) e em Chays et al. (2000). Basicamente, são atribuídos valores para cada

domínio, que são combinados para compor relações de entrada, garantindo que restrições de integridade dessas relações sejam satisfeitas, ou seja, são produzidos estados válidos do banco de dados. A partir das ocorrências dos comandos de manipulação e da descrição de esquema da base de dados, é possível abstrair os atributos pertinentes à execução dos comandos; tais atributos, definidos no Capítulo 3 como *atributos manipulados*, são os atributos da(s) relação(ões) de entrada referenciados explicitamente ou implicitamente no comando de manipulação ou listados diretamente como atributos do conjunto de saída. O testador deve fornecer uma coleção de valores para cada atributo, usando a premissa de que é mais significativo ao teste conferir resultados a partir de dados ligados a situações típicas de uso. Os valores são usados para gerar dados que cobrem uma variedade de combinações e que satisfazem as regras de integridade da base de dados.

O experimento é conduzido para o teste das regras do conjunto R_x, conforme descrito a seguir.

(1) Criar novas versões do conjunto de regras R_x, pela introdução de um defeito para caracterizar cada versão criada. A introdução de defeitos ocorrerá de forma aleatória:

- **(1a)** identificar todas as ocorrências de manipulação determinantes de associações de interação entre regras do conjunto R_x; ou seja, tomar as ocorrências de manipulação atribuídas à definição e ao uso persistentes em cada associação de interação em R_x;
- **(1b)** para cada tipo de defeito de manipulação τ, do elenco de tipos de defeito explorado no Capítulo 3, proceder conforme abaixo:
  - **(1b)-1** selecionar aleatoriamente uma ocorrência de manipulação dentre as identificadas em *(1a)*;
  - **(1b)-2** introduzir um defeito do tipo τ na ocorrência de manipulação selecionada em *(1b)-1*; se não for possível introduzir o defeito nesta ocorrência (por exemplo, o defeito se refere à cláusula de ordenação incorreta e o comando não possui cláusula de ordenação), retornar ao passo *(1b)-1* para a seleção de outra ocorrência de manipulação.
- **(1c)** Deverá ser garantido que em toda ocorrência de definição e de uso de dados persistentes determinante de associações de interação seja introduzido pelo menos um defeito de manipulação; contudo, os tipos de defeito listados no Capítulo 3 referenciam

as construções sintáticas básicas dos comandos de manipulação, o que poderá resultar na impossibilidade de se inserir algum defeito devido à inexistência de comandos que suportem o seu tipo.

Cada versão defeituosa κ do conjunto R_x é caracterizada pela presença de um defeito em algum comando de manipulação determinante de associação de interação entre regras. A aleatoriedade na seleção de ocorrências é usada para evitar a parcialidade na inserção de defeitos.

(2) Executar as versões instituídas a partir do conjunto R_x, segundo as diretrizes a seguir. Em cada versão defeituosa κ do conjunto de regras, para toda associação de interação α determinada pelo comando defeituoso nesta versão e para cada granularidade β de análise de fluxo de dados:

- **(2a)** gerar base de teste, inspirando-se na abordagem descrita por Chays et al. (2000); a Seção 2.4 descreve esta abordagem;
- **(2b)** elaborar comandos de manipulação para o disparo das regras do conjunto R_x, de modo que cubra todos os tipos de operação que disparam as regras cuja execução envolve a cobertura de α;
- **(2c)** aplicar os dados de teste para cobrir α em β; se a associação não foi coberta para esta granularidade, repetir os passos *(2a)* e *(2b)* para gerar novos dados de teste até que se cubra α em β, ou se conclua que a associação é não exercitável nesta granularidade; a idéia é buscar dados de teste adequados às granularidades *relação*, *tupla*, *atributo* e *valor de atributo*;
- **(2d)** se α foi coberta em β, avaliar se uma falha foi manifestada devido à presença do defeito.

Note que toda versão defeituosa κ do conjunto de regras R_x é testada para cada granularidade de fluxo de dados. Isto permite capturar a influência da granularidade na descoberta de defeitos de interação entre regras.

5.3.4 Aplicação do Experimento

Todas as regras do conjunto R_x estiveram habilitadas ao disparo durante a execução de cada caso de teste, visando a estabelecer a operação real da aplicação de banco de dados. Não se buscou isolar as regras envolvidas nas associações em que participava o nó defeituoso, e o disparo entre regras pôde ser realizado em cadeia.

O experimento foi conduzido com o apoio de uma ferramenta, denominada ADAPT-TOOL, construída para apoiar a automação dos critérios baseados na interação entre regras. A ferramenta concentra-se na fase dinâmica do teste e utiliza estruturas de dados do modelo relacional para a construção de uma base de dados de teste. As seguintes atividades são suportadas pela ferramenta: geração de dados de teste; controle de versões defeituosas de um conjunto de regras; aplicação e re-aplicação de casos de teste; oráculo de teste; e avaliação de conjuntos de casos de teste por granularidade. O Apêndice C descreve aspectos funcionais da ferramenta, apresentando algumas interfaces com o testador para a aplicação de casos de teste.

5.3.5 Resultados

Os resultados obtidos são analisados sob três óticas: versões defeituosas do conjunto de regras, associações de interação e casos de teste aplicados.

As Figuras 5.19, 5.20, 5.21 e 5.22 sintetizam os dados apresentados no Apêndice E, nas Tabelas E.1, E.2, E.3 e E.4, respectivamente. Nessas tabelas, os dados foram obtidos considerando-se a granularidade mais precisa em cada medição; por exemplo, se uma cobertura ocorre no nível *tupla* e, conseqüentemente, no nível *relação*, esta é computada como uma cobertura em nível *tupla*. A medição foi realizada observando-se, em cada caso de teste, apenas as associações de interação em que participava o comando SQL defeituoso; em tais associações, o defeito estaria presente no comando representado pelo nó da definição persistente, ou no comando representado pelo nó em cujos arcos de saída estão os usos persistentes.

VERSÕES DEFEITUOSAS DO CONJUNTO DE REGRAS

Foram construídas 26 versões do conjunto R_x, cada qual possuindo um único defeito de manipulação. Em todos os conjuntos de regras defeituosas foi possível construir casos de teste cujas execuções resultaram em manifestação de falha. A partir dos dados coletados, pôde-se observar que a cobertura de associações de interação em níveis de granularidade mais precisos não agrega, em geral, maior capacidade à descoberta de defeitos.

As Figuras 5.19 e 5.20 apresentam, para os casos de teste reveladores e não reveladores de defeito, respectivamente, o resumo do número de versões de R_x para as granularidades *relação*, *tupla*, *atributo* e *valor de atributo*. A abscissa *1* denota o número de versões de R_x em que houve cobertura de associações de interação em cada granularidade. A abscissa *2* denota o número de versões de R_x em que não houve cobertura de associações de interação em cada granularidade. A abscissa *3* denota o número de versões de R_x em que houve maior incidência de cobertura de associações de interação em cada granularidade em relação às demais granularidades. A abscissa *4* denota o número de versões de R_x em que houve cobertura exclusiva de associações de interação em cada granularidade.

Nos casos de teste reveladores de defeito, conforme especificado na Figura 5.19, a cobertura no nível *valor de atributo* não foi exclusiva em quaisquer das versões defeituosas do conjunto de regras. As coberturas nas precisões *atributo* e *tupla* ocorreram em 71,1% (19 de 26) e 19,2% (5 de 26) das versões de R_x, respectivamente, mas em todas as versões defeituosas também não houve exclusividade para revelar defeitos. Um resultado marcante à redução do custo de teste é que a cobertura em nível *relação* ocorreu em todas as versões de R_x, diferentemente dos demais níveis de granularidade; em adição, houve exclusividade de cobertura em 5 versões de R_x e maior incidência de cobertura nesta granularidade em todas as versões de R_x.

Conforme especificado na Figura 5.20, para os casos de teste não reveladores de defeito, em 15,4% (4 de 26) das versões de R_x a cobertura na precisão *valor de atributo* foi a de maior incidência em relação às demais, sendo exclusiva em 3,9% (1 de 26) das versões de R_x; em adição, em 61,5% (16 de 26) das versões de R_x foi possível cobrir associações de interação nesta granularidade. Nos casos de teste em que ocorreu tal situação, foram observados dois cenários: *(i)* a associação de interação é caracterizada entre 2 comandos de mudança de estado (por exemplo, *update-update*), e a execução do segundo comando corrige o erro decorrente do defeito presente no primeiro comando; e *(ii)* o disparo entre regras leva à correção do erro decorrente do defeito.

A cobertura nas precisões *atributo* e *tupla* também foi observada em 65,4% (17 de 26) e 26,9% (7 de 26) das versões de R_x para os casos de teste não reveladores de defeito, respectivamente, de onde conclui-se que a adequação a granularidades mais precisas pode não resultar na descoberta de defeitos.

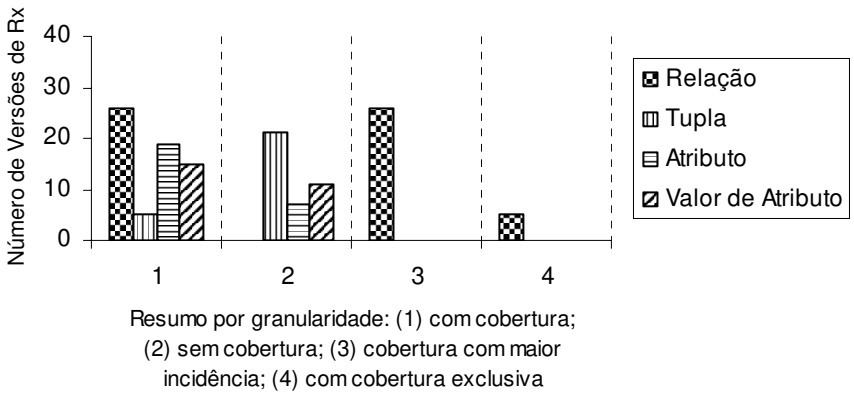


Figura 5.19 – Resumo do número de versões de R_x por granularidade, nos casos de teste reveladores de defeito.

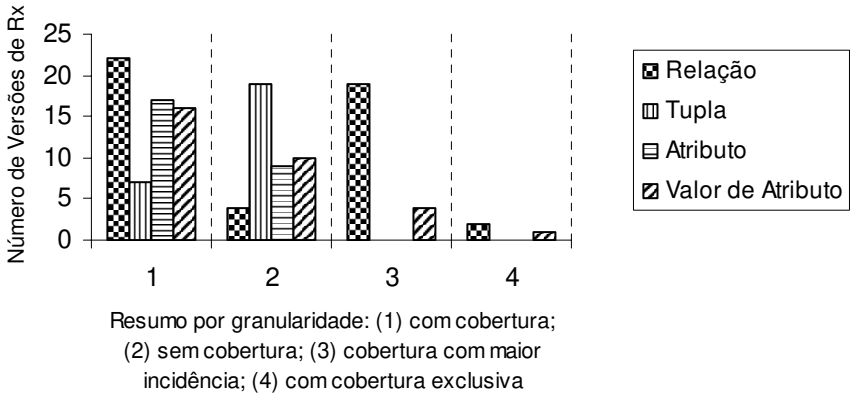


Figura 5.20 – Resumo do número de versões de R_x por granularidade, nos casos de teste não reveladores de defeito.

ASSOCIAÇÕES DE INTERAÇÃO

O conjunto de regras R_x possui 74 associações de interação, conforme as interações entre regras ilustradas nos grafos da Figura 5.18. As não exercitáveis em todas as granularidades correspondem a 35,1% (26 de 74), e estão sempre relacionadas ao uso de dados persistentes em arcos atribuídos a alguma exceção na execução de comandos de manipulação. A análise da não executabilidade de associações por granularidade é uma atividade muito custosa visto que, em geral, exige essencialmente a intervenção humana. Em 4,2% (2 de 48) das associações exercitáveis, não foi possível elaborar casos de teste reveladores de defeito.

As Figuras 5.21 e 5.22 apresentam, para os casos de teste reveladores e não reveladores de defeito, respectivamente, o resumo das associações de interação para as granularidades *relação*, *tupla*, *atributo* e *valor de atributo*. A abscissa *1* denota o número de associações de interação cobertas em cada granularidade. A abscissa *2* denota o número de associações de interação não cobertas em cada granularidade. A abscissa *3* denota o número de associações de interação em que houve maior incidência de cobertura em cada granularidade em relação às demais granularidades. A abscissa *4* denota o número de associações de interação em que houve cobertura exclusiva em cada granularidade.

Sobre os casos de teste reveladores de defeito, conforme especificado na Figura 5.21, 85,4% (41 de 48) das associações exercitáveis são cobertas em nível *relação*; neste nível de granularidade, observa-se também o menor número de associações não cobertas com respeito aos casos de teste reveladores de defeito em comparação com as demais granularidades (33 contra 63, 51 e 58), o que novamente demonstra, relativamente, a eficiência da precisão *relação* na manifestação de falhas. Em adição, somente em 6,3% (3 de 48) da associações exercitáveis, o defeito é revelado apenas nas demais precisões de fluxo de dados (duas associações para a precisão *tupla* e uma associação para a precisão *atributo*).

Sobre os casos de teste não reveladores de defeito, conforme especificado na Figura 5.22, a cobertura de algumas associações de interação em determinado nível de granularidade é sempre não reveladora de defeito; este fato foi observado em 10,4% (5 de 48) das associações para a precisão *relação*, e em 12,5 (6 de 48) das associações para as demais granularidades. Outro aspecto para análise é a fração dos números de associações cobertas entre os casos de teste reveladores de defeito e os casos de teste não reveladores de defeito (comparação entre os valores da abscissa *1* nas Figuras 5.21 e 5.22). Os valores desta fração são 41/22 (1,86), 11/13 (0,85),

23/16 (1,44) e 16/19 (0,84) para as granularidades *relação*, *tupla*, *atributo* e *valor de atributo*, respectivamente, demonstrando que a cobertura na precisão *relação* possui uma melhor resposta para a descoberta de defeito relativo às coberturas nas demais granularidades.

Conclui-se que os domínios de entrada próprios de cada precisão de fluxo de dados não demonstraram comportamento suplementar à descoberta de defeitos. As observações até este ponto apresentadas refutam a hipótese *(h1)*, pois não se observa atuação suplementar para a descoberta de defeitos na cobertura de associações de interação em diversas granularidades: a cobertura em nível menos preciso (nível *relação*) ocorreu nos casos de teste reveladores de defeito para todos os defeitos existentes.

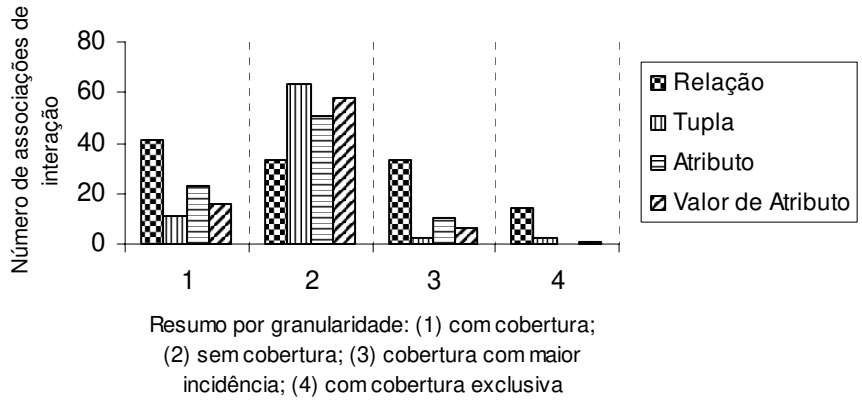


Figura 5.21 – Resumo do número de associações de interação por granularidade, nos casos de teste reveladores de defeito.

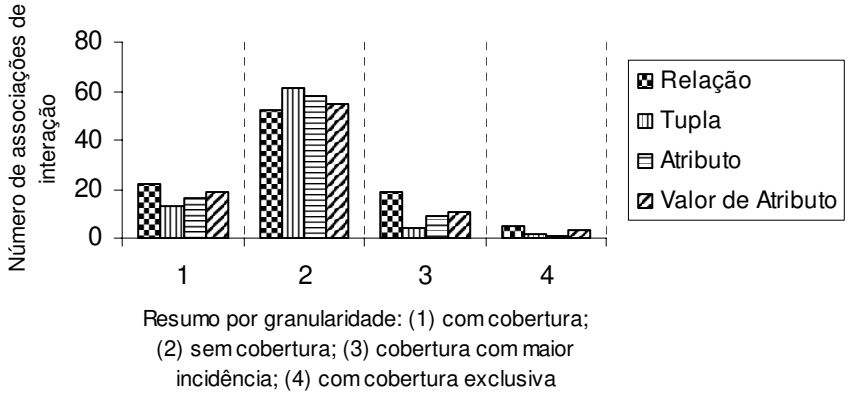


Figura 5.22 – Resumo do número de associações de interação por granularidade, nos casos de teste não reveladores de defeito.

CASOS DE TESTE

A Figura 5.23 apresenta um resumo dos 632 casos de teste aplicados. Os dados se referem à quantidade de casos de teste, apresentando valores para a cobertura de associações de interação em cada granularidade e para as exceções ocorridas. A figura é estruturada em linhas e colunas. As colunas *(I)* e *(II)* organizam os casos de teste em reveladores e não reveladores de defeito, respectivamente; os dados de todos os casos de teste, somatório das colunas *(I)* e *(II)*, é posto na coluna *(III)*. A linha *(a)* refere-se aos casos de teste que não exercitaram o nó defeituoso. Os casos de teste que exercitaram o nó defeituoso, mas não cobriram qualquer associação de interação, são contabilizados na linha *(b)*. As linhas *(c)*, *(d)*, *(e)* e *(f)* sintetizam os casos de teste cobertos nas granularidades *relação*, *tupla*, *atributo* e *valor de atributo*, respectivamente. Para entender como os valores foram computados para estas linhas, considere o item *(I:d)*, coluna *(I)* e linha *(d)*; este item exprime a quantidade de casos de teste reveladores de defeito em nível *tupla*, ou seja, refere-se àqueles casos de teste em que o nível *tupla* foi a maior precisão alcançada de fluxo de dados, em relação a todas as associações de interação cobertas pelos casos de teste. As exceções ocorridas são tratadas nas colunas *(h)*, *(i)* e *(j)*. Dois tipos de exceção foram monitorados: exceções do usuário, provocadas pelo programador (comando *raise*); e exceções do sistema, ocorridas na execução de comandos de manipulação e que não foram previstas pelo programador. Os casos de teste que não tiveram qualquer exceção são listados na linha *(h)*. As linhas *(i)* e *(j)* totalizam os casos de teste em que houve exceção do usuário e exceção do sistema, respectivamente.

Os resultados por caso de teste dão uma dimensão do efeito da aplicação dos critérios propostos. Vale ressaltar que a adequação aos critérios é traduzida pela cobertura das associações de interação e, nesse sentido, os casos de teste elaborados no experimento cobriram todos os requisitos *(critério, granularidade)* julgados como exercitáveis. A busca para se alcançar cobertura de associações em níveis mais precisos requereu maior atenção por parte do testador, visto que foi preciso observar o conteúdo da base de dados gerada para, então, elaborar comandos para o disparo de regras. O experimento não objetivou avaliar o custo associado à adequação as diversas granularidades; contudo, o esforço do testador é claramente maior à medida em que se procura alcançar granularidades mais precisas.

	(I) Casos de Teste Reveladores de Defeito		(II) Casos de Teste Não Reveladores de Defeito		(III) Todos os Casos de Teste
(a)	Defeito Não Exercitado...	0	Defeito Não Exercitado...	8	8
(b)	Nenhuma Cobertura...	20	Nenhuma Cobertura...	2	22
(c)	Relação...	213	Relação...	89	302
(d)	Tupla...	46	Tupla...	20	66
(e)	Atributo...	81	Atributo...	37	118
(f)	Valor de Atributo...	64	Valor de Atributo...	52	116
(g)	**TOTAL...**	**424**	**TOTAL...**	**208**	**632**
(h)	Nenhuma Exceção...	281	Nenhuma Exceção...	198	479
(i)	Exceção do Usuário...	112	Exceção do Usuário...	10	122
(j)	Exceção do Sistema...	31	Exceção do Sistema...	0	31
(k)	**TOTAL...**	**424**	**TOTAL...**	**208**	**632**

Figura 5.23 – Resumo dos casos de testes aplicados reveladores e não reveladores de defeito, por cobertura de associação de interação e por exceção ocorrida.

Para obter indícios da contribuição dos critérios na descoberta de defeitos, utilizou-se uma medida de eficácia, definida como o número de casos de teste reveladores de defeito no conjunto adequado, a qual é dada pela expressão:

[(I:c) + (I:d) + (I:e) + (I:f)] / [(III:c) + (III:d) + (III:e) + (III:f)].

Esta medida é utilizada porque todos os critérios de fluxo de dados, utilizando qualquer modelo, derivavam conjuntos adequados que incluíam pelos menos um caso de teste revelador de defeito (Chaim, 2001). O valor *(0,6711)* foi obtido do cálculo da expressão acima, denotando que 2/3 do conjunto adequado é revelador de defeito. Este resultado apóia a verdade da hipótese *(h2)*, que investiga se a abordagem de teste proposta ampara a descoberta de defeitos na interação entre regras escritas em SQL. Se este cálculo for realizado por precisão de fluxo de dados, obter-se-ão os valores *(0,7053)*, *(0,6970)*, *(0,6864)* e *(0,5517)* para as precisões *relação*, *tupla*, *atributo* e *valor de atributo*, respectivamente, alcançando índices de eficácia ainda mais favoráveis na direção de granularidades menos precisas.

Um aspecto interessante é ocorrência de exceções na execução dos conjuntos de regras para os casos de teste reveladores de defeito. Desses casos de teste aplicados, 33,7% (143 de 424) resultaram em exceções ocorridas, em sua maioria provocadas pela execução de comandos do tipo *raise*. A instrumentação de situações de exceção requereu mecanismos especiais, pois

ocorrências de exceção causam em geral o cancelamento de transações de banco de dados, eliminando também os registros de instrumentação realizados, pois as estruturas de banco de dados para instrumentação participam da mesma transação que inclui a operação de disparo de regras.

A primeira hipótese é refutada quando se alcançam cenários nos quais defeitos são revelados em diversas precisões de fluxo de dados, sem que seja caracterizada uma atuação suplementar entre granularidades na descoberta de defeitos. A prova da segunda hipótese envolve muitos fatores. Neste caso, o que se está investigando é uma tendência, não se busca alcançar uma verdade absoluta, pois empiricamente isso não é possível: em geral, os programas, seus domínios de entrada e seus possíveis defeitos constituem uma combinação infinita. Apesar da aparente eficácia dos critérios propostos na descoberta de defeitos, vários fatores estão envolvidos nos resultados do estudo, tais como: técnica adotada para a geração de dados de teste; regras ativas utilizadas no experimento; dados de entrada (dados de teste) gerados a partir da técnica adotada; os defeitos e tipos de defeito nas regras ativas; requisitos de teste.

5.3.6 Ameaças à Validade

Nesta subseção, são apresentadas ameaças à validade do experimento, de forma similar às limitações apontadas por Chaim (2001) em um experimento para o teste de fluxo de dados de programas com ponteiros e registros. A validade externa dos experimentos relaciona-se à generalização dos resultados às regras em uso no mundo real. Os resultados apresentados foram obtidos considerando-se algumas aproximações da realidade; por isso, eles devem ser analisados tendo em vista as limitações dessas aproximações.

Embora os defeitos introduzidos não representem de fato defeitos coletados da prática, eles são representativos dos diversos tipos de defeitos reais, conforme publicado em (Leitão et al., 2005). Mesmo assim, é pertinente replicar no futuro esse estudo para outras regras utilizadas na prática com defeitos reais identificados. Uma indagação seria se os resultados obtidos estariam sendo influenciados pela abordagem de geração de dados de teste, pois qualquer abordagem distinta da aleatória poderá introduzir alguma parcialidade a experimentos de engenharia de software (Kitchenham et al., 2002); conforme descrito na Subseção 5.3.2, buscou-se introduzir aleatoriedade na geração de bases de dados para o teste de regras. A adequação aos critérios foi

apreciada na ótica de cobertura dos elementos requeridos, para então quantificar a descoberta da presença de defeitos. As medidas realizadas buscaram estabelecer uma relação quantitativa entre adequação e descoberta de defeitos para cada granularidade de fluxo de dados.

Uma ameaça é o fato do experimento estar baseado em um único conjunto de regras, embora várias versões diferentes tenham sido utilizadas. Todavia, esse conjunto possui muitos pontos de interação, tornando complexo o seu comportamento em tempo de execução. Tais pontos de interação, sobretudo o disparo entre regras, resultaram, em alguns casos de teste, no exercício de mais de mil nós, elevando as chances de conduta inesperada ou incorreta dos conjuntos de regras.

Outra limitação é que os defeitos foram semeados artificialmente e são únicos em cada versão, apesar da seleção aleatória de ocorrências de manipulação e dos tipos de defeito. Na prática, observa-se que os programas em geral possuem vários defeitos. Deve-se ressaltar, entretanto, que o objeto do experimento é avaliar o desempenho das diferentes granularidades de fluxo de dados na detecção de defeitos em ocorrências de manipulação ligadas à interação entre regras, e obter indícios de que os critérios baseados na interação entre regras apóiam a descoberta desses defeitos. Nesse sentido, o isolamento de defeitos é interessante, pois permite rastrear o relacionamento entre a existência de tais defeitos e as hipóteses investigadas.

Os resultados devem ainda ser analisados levando em consideração as características dos dados de teste. A base de dados de entrada foi determinada pela combinação aleatória de dados para cada atributo da base de dados, os quais foram fornecidos pelo testador. Os comandos de disparo de regras buscaram cobrir as associações de interação nas diversas granularidades, em contextos reveladores e não reveladores de defeito. Os conjuntos selecionados, por sua vez, são *aproximadamente* adequados, pois as associações não executáveis em cada precisão de fluxo de dados foram determinadas pela avaliação subjetiva do testador. Dessa forma, a eficácia e o tamanho dos conjuntos adequados podem estar subestimados.

5.4 Considerações Finais

Um conjunto de critérios de adequação foi proposto neste capítulo, os quais são baseados no modelo de interação entre regras ativas discutido no Capítulo 4. A aplicação dos critérios visa a descoberta de defeitos que normalmente não são revelados no teste individual de regras. A essência dos critérios é explorar as relações de fluxo de dados persistentes existentes na interação entre regras, representando uma extensão ao critério *todos os usos*, pela dependência exclusiva de fluxo de dados e pela dependência resultante da transferência de controle oriunda do disparo entre regras.

A análise de inclusão entre os critérios propostos e alguns dos critérios baseados em fluxo de dados demonstrou que eles são incomparáveis. Abordagens de teste que endereçam unicamente variáveis de programa em geral não alcançam as relações de fluxo de dados persistentes. Observou-se que mesmo a abordagem de Spoto (2000) não cobre a interação entre regras, pois a definição de dados persistentes deve ser confirmada (execução implícita ou explícita do comando *commit*) antes de alcançar o uso de dados, condição esta não prevista pelos critérios propostos. Conclui-se que os critérios baseados na interação entre regras alcançam um contexto ainda não tratado por outras abordagens – interação entre regras escritas em SQL para o teste baseado em fluxo de dados persistentes –, o qual representa fonte de defeitos em aplicações de banco de dados ativos.

Mostrou-se, também, que o par *(critério, granularidade)* determina um novo requisito de teste, pois existe uma relação de inclusão entre os domínios de entrada próprios para cobertura de um particular critério em cada granularidade. Essa relação de inclusão é válida mesmo na presença de não executabilidade. O requisito *(critério, granularidade)* influencia a complexidade dos critérios propostos: *(i)* a redefinição de dados persistentes é sensível aos dados manipulados em cada ocorrência de manipulação; e *(ii)* as estruturas de fluxo de controle não dão origem ao cenário de pior caso. Foi observado que os critérios baseados na interação entre regras possuem complexidade polinomial.

Sobre o experimento aplicado, observou-se que as precisões de fluxo de dados – *relação*, *tupla*, *atributo* e *valor de atributo* – não possuem papel suplementar na descoberta de defeitos; foram identificados cenários reveladores e não reveladores de defeito para cada precisão, onde a

descoberta de defeito ocorre de forma não sistemática em qualquer das granularidades: casos de teste reveladores de defeito estão relacionados à cobertura de associações de interação nas diversas precisões de fluxo de dados. A eficácia da cobertura de associações de interação alcançou 2/3 do conjunto adequado, obtendo índices de eficácia ainda mais favoráveis na direção de granularidades menos precisas.

Este estudo empírico apresentou índices propícios ao apoio dos critérios propostos à descoberta de defeitos na interação entre regras. Apesar dessa tendência, é pertinente a realização de novos estudos empíricos, visto que vários fatores estão envolvidos nos resultados do estudo, tais como: técnica adotada para a geração de dados de teste; regras ativas utilizadas no experimento; dados de entrada (dados de teste) gerados a partir da técnica adotada; defeitos e tipos de defeito nas regras ativas.

Capítulo 6

Implementação dos Critérios Baseados na Interação entre Regras

Neste capítulo são discutidos aspectos que norteiam a implementação dos Critérios Baseados na Interação entre Regras. A avaliação à adequação aos critérios requer atividades de análise estática e dinâmica, tais como: determinação dos elementos requeridos, na forma de associações de interação exigidas pelos critérios; instrumentação de fluxo de controle, para registrar os caminhos de fluxo resultantes da aplicação de casos de teste, incluindo a transferência de controle entre regras; instrumentação de fluxo de dados, visando a capturar os dados persistentes definidos e usados ao longo dos caminhos exercitados; decisão sobre a manifestação de falhas, em resposta à aplicação de casos de teste; e avaliação de cobertura nas várias precisões de fluxo de dados.

A automação da aplicação dos critérios propostos apóia-se em modelos de implementação para regras escritas em SQL, visando atender à abordagem de interação entre regras descrita no Capítulo 4. O Modelo de Fluxo de Controle é destinado à abstração das transferências de controle para o fluxo intra-regra e para o disparo entre regras. O Modelo de Fluxo de Dados é útil à determinação das ocorrências de definição e de uso de dados persistentes ao longo dos caminhos exercitados. O Modelo de Instrumentação instrui como extrair os elementos dos fluxos de controle e de dados exercitados na execução de casos de teste. O Modelo de Avaliação habilita a análise de cobertura dos critérios em cada uma das precisões de fluxo de dados.

A organização deste capítulo é apresentada a seguir. A Seção 6.1 discute elementos sobre os modelos de implementação dos critérios propostos: fluxo de controle, fluxo de dados,

instrumentação e avaliação. A Seção 6.2 descreve a automação dos critérios baseados na interação entre regras realizada por uma ferramenta, denominada ADAPT-TOOL, cujos aspectos funcionais são apresentados no Apêndice C.

6.1 Modelos de Implementação

Os modelos de implementação explorados nesta seção foram originalmente propostos em (Leitão et al., 2002). As Subseções 6.1.1 e 6.1.2 apresentam, respectivamente, os Modelos de Fluxo de Controle e de Fluxo de Dados, em conjunto com o Modelo de Instrumentação para a coleta de informação de fluxo de controle e de fluxo de dados oriunda da execução de regras ativas.

A instrumentação é normalmente realizada pela inserção de comandos no código fonte, cuja execução grava dados utilizando estruturas de dados próprias para este fim. Após a execução de casos de teste, tais estruturas de dados estarão disponíveis à avaliação sobre os requisitos de teste. Para a instrumentação de fluxo de controle, utilizar-se-á um pseudoprocedimento destinado ao registro dos nós exercitados, denominado *ponta-de-prova(n)*, onde *n* representa um número de nó; conforme será descrito na Seção 6.2, a implementação deste procedimento requer outros parâmetros de entrada, tais como identificação do caso de teste e identificação da regra ativa. A instrumentação de fluxo de dados considera a granularidade da análise de fluxo de dados e requer a utilização de estruturas de dados que denotem as *tuplas* e os atributos definidos e usados nas operações de manipulação de dados persistentes.

6.1.1 Modelo de Fluxo de Controle

Para ilustrar a definição de regras ativas, considere a Figura 6.1 que possui um modelo simplificado de regra escrita em SQL. A sintaxe indica uma separação entre as partes evento, condição e ação. Do ponto de vista de construção de grafo de fluxo de controle, o evento e a condição de regra podem ser representados por nós específicos. A ação de regra é representada por um subgrafo.

Em regras ativas escritas em SQL, um bloco *b* é definido como *b = (dcl, sta, exp)*; *dcl* é o conjunto de variáveis declaradas e de escopo local ao bloco; *sta* representa os comandos que definem a tarefa do bloco; e *exp* consiste nas rotinas de tratamento das exceções ocorridas devido à execução do conjunto *sta* de comandos. O subgrafo atribuído à ação de regra é obtido pela extensão do bloco associado. Essa definição de bloco é similar à definição de Daou et al. (2001) 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 entre os módulos e os dados gerados por um comando, e que são usados por outros comandos no mesmo módulo, ou outros módulos, criando relações de fluxo de dados.

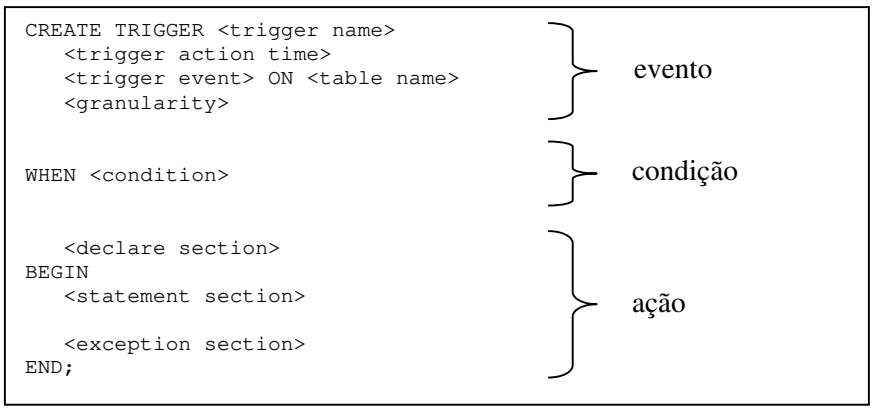


Figura 6.1 – Exemplo da estrutura de regra escrita em SQL.

No modelo de fluxo de controle adotado, uma regra *r* é representada por um grafo dirigido, dado por *G(r) = (N, A, e, x)*, conforme ressaltado no Capítulo 2, onde: *N* representa o conjunto de nós; *A* denota o conjunto de arcos; o nó de entrada (evento da regra) é identificado por *e*; e *x* representa o nó de saída. A estrutura de controle de uma regra ativa segue o desenho apresentado na Figura 6.2; nessa figura são ressaltados: o disparo de regra e o controle interno de uma regra ativa. Seja *P* um processo que possui uma operação θ capaz de provocar eventos de disparo de regras; *P* pode ser uma aplicação do usuário, uma rotina gravada no banco de dados ou, mais especificamente, uma regra ativa da base de dados. Considere que r_i e r_j sejam regras sem a presença de condição e com a presença de condição, respectivamente, 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 aplica-se à regra r_j. Considerando a estrutura de controle de r_j, o nó e_j é o nó de entrada da regra, e representa o evento associado ao disparo da regra; o nó c_j denota a condição da regra, e significa uma seleção entre o subgrafo a_j e o nó x_j. O subgrafo a_j está associado à ação da regra. O nó x_j é o nó de saída da regra r_j.

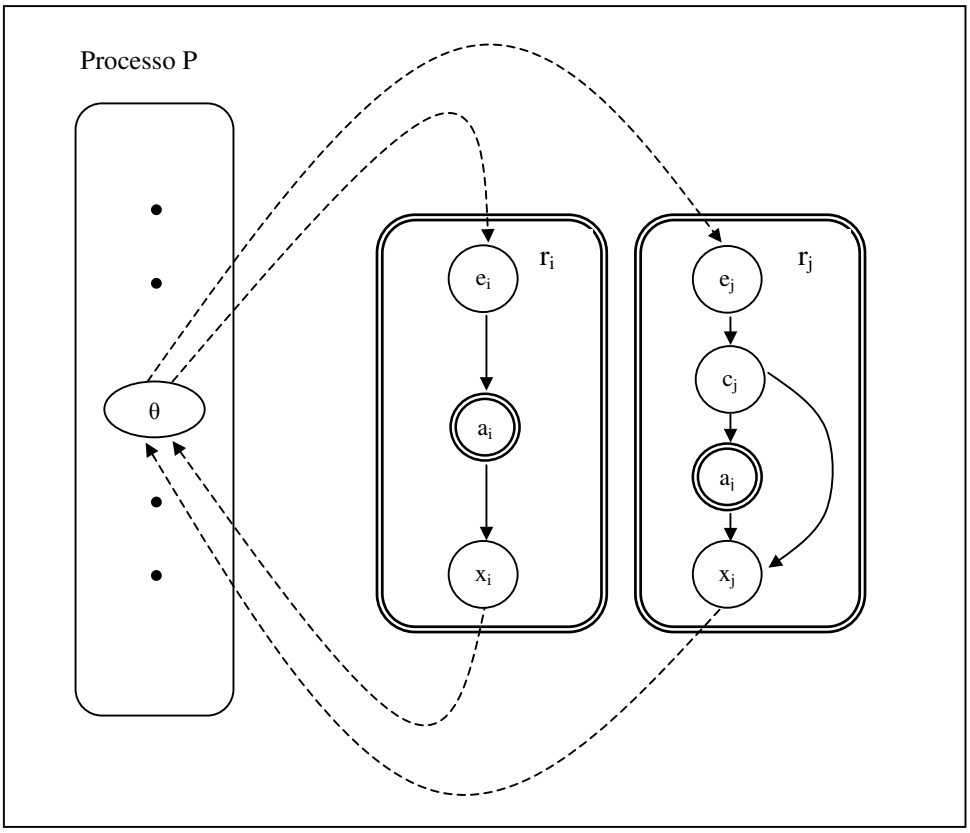


Figura 6.2 – Grafo de fluxo de controle para regras ativas com e sem condição, respectivamente representadas pelas regras r_i e r_j.

Cada ocorrência de comando de manipulação é representada por um nó dedicado no grafo de fluxo de controle. Abordagem similar foi usada em (Spoto, 2000; Daou et al., 2001); particularmente, Spoto (2000) altera 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. No contexto de regras ativas, a adoção desse modelo permite explorar as relações de fluxos de dados entre os comandos de manipulação em uma mesma regra ativa e em regras distintas.

Conforme descrito no Capítulo 4, o modelo de fluxo de controle inter-regra que foi adotado no presente trabalho – política de ciclo recursiva e modos de acoplamento imediatos para

evento-condição e condição-ação – prevê que a execução de uma regra é interrompida quando o evento de disparo de alguma regra é provocado e o controle é transferido para esta regra. A Figura 6.3 ilustra o fluxo de controle para comandos de manipulação, com e sem disparo entre regras. Toda ocorrência de comando de manipulação ocasiona pelo menos duas alternativas de fluxo, representadas pelos arcos *(s, n_k)* e *(s, n_j)* para a execução normal do comando e para a execução que provoca exceção, respectivamente. O nó *s* mapeia uma ocorrência de manipulação de dados persistentes; o nó n_j representa o nó de saída da regra ou um subgrafo de uma rotina de tratamento de exceção; e o nó n_k representa um sucessor do nó *s* para execuções normais (sem provocar exceção) do comando de manipulação. Em 6.3(b), o subgrafo *r* denota uma regra ativa que pode ser disparada a partir da execução do comando mapeado pelo nó *s*; observe que o disparo entre regras não ocorre quando exceções são provocadas.

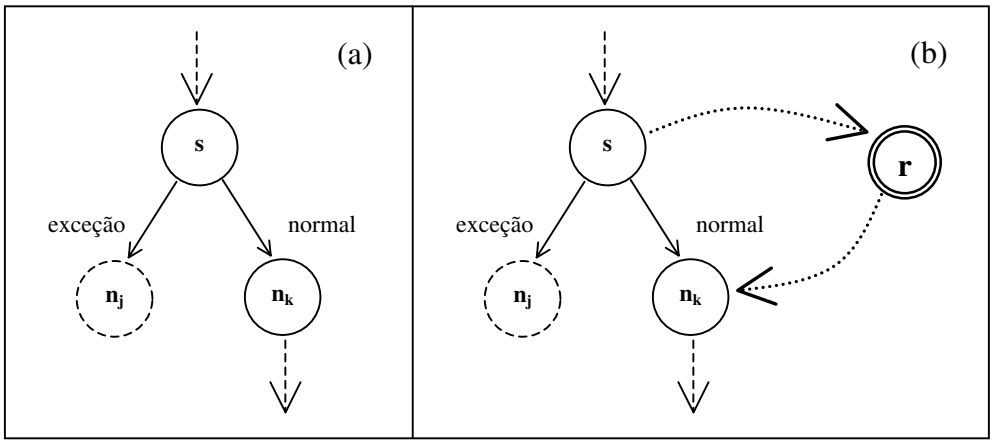


Figura 6.3 – Fluxo de controle para comandos de manipulação de dados:
(a) sem disparo entre regras; (b) com disparo entre regras.

No contexto de regras escritas em SQL, as extensões da linguagem relacionadas com construções estruturadas são similares àquelas usadas em linguagens do tipo *Algol*. Em geral, além dos comandos de manipulação de dados persistentes, encontram-se comandos associados a estruturas de controle como seleção e laço; neste sentido, está o modelo para a condição de regra, que considera uma estrutura de seleção simples entre o subgrafo que representa a ação da regra e o nó de saída da regra. Os modelos de fluxo de controle e de instrumentação para tais estruturas foram explorados pela literatura (Chaim, 1991; Maldonado, 1991; Spoto, 2000, Leitão et al.,

2002; Cardoso, 2004), não sendo, portanto, incluídos neste texto. Ênfase será dada às estruturas de tratamento de exceção, que constituem uma contribuição pertinente às regras escritas em SQL.

Segundo Daou et al. (2001), programação de exceção complica as dependências de fluxo de controle, resultando na construção de grafo de fluxo de controle diferente para módulos de banco de dados, em relação a programas convencionais. Cada bloco pode ter uma única área de tratamento de exceção, a qual deverá estar no final do bloco. 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 mais para o bloco em que está inserido; o tratamento de exceção é processado e o controle prossegue no bloco imediatamente mais externo. Várias exceções podem ser tratadas em um único bloco; a cláusula *OTHERS* constitui um predicado genérico para o tratamento de exceções. Um exemplo de tratamento de exceção em blocos aninhados é apresentado na Figura 6.4.

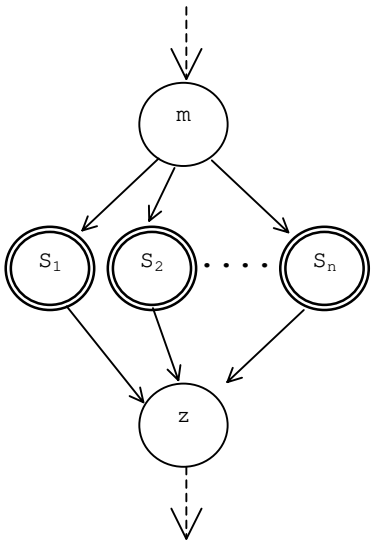
```
DECLARE USER_EXC EXCEPTION;
        ::::::::::
BEGIN
        ::::::::::
        DECLARE
                ::::::::::
        BEGIN
                ::::::::::
        EXCEPTION
                WHEN TOO_MANY_ROWS THEN
                        :::::::::
        END;
        ::::::::::              (*)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

Figura 6.4 – Exemplo de tratamento de exceção em blocos aninhados.

Sempre que uma exceção ocorrer, o seu tratamento é realizado dentro do bloco em que foi provocada; se não houver tratamento específico neste bloco, o controle é desviado para a área de tratamento em blocos mais externos; a presença da cláusula *OTHERS* evita que esse desvio ocorra. Ou seja, ocorrida 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. Na Figura 6.4, uma exceção ocorrida no bloco mais interno que necessite ser tratada no bloco mais externo evita que o controle de fluxo alcance a área (*) localizada na

figura. Quando alguma exceção é provocada nas rotinas de tratamento de exceção, independentemente do nível do bloco, resultará em um erro na execução da regra, e a primeira exceção tratada na regra será retornada ao processo que possui a operação que provocou o evento de disparo da regra.

A Figura 6.5 apresenta o fluxo de controle e de instrumentação para rotinas de tratamento de exceção. O nó *m* refere-se a uma estrutura do tipo seleção múltipla, que utiliza o predicado da exceção ocorrida para a seleção da rotina para tratamento de exceção; os subgrafos S_1, S_2, ..., S_n representam as rotinas para tratamento de exceção, onde os seus respectivos nós de entrada são t_1, t_2, ..., t_n; o nó *z* denota o nó de saída do bloco. Os comandos de instrumentação são marcados com */*i*/* no início de cada linha. A indicação */*1*/* no final de linha indica que este comando de instrumentação é utilizado quando as rotinas de tratamento de exceção referem-se ao bloco mais externo da regra ativa, caso contrário utiliza-se o comando com a indicação */*2*/*.




```
     BEGIN
          ::::::::::::::
     EXCEPTION
          WHEN <p1> THEN
/*i*/           ponta-de-prova (m);
/*i*/           ponta-de-prova (t1);
                ::::::::::::::
/*i*/           ponta-de-prova (z); /*1*/
          ::::::::::::::
          ::::::::::::::
          WHEN <pn> THEN
/*i*/           ponta-de-prova (m);
/*i*/           ponta-de-prova (tn);
                ::::::::::::::
/*i*/           ponta-de-prova (z); /*1*/
     END;
/*i*/     ponta-de-prova (z);      /*2*/
```

Figura 6.5 – Fluxo de controle e de instrumentação para rotinas de tratamento de exceção.

Se o bloco mais externo não possuir rotina para tratamento de exceção para o predicado *OTHERS*, esta é inserida visando a registrar o desvio para o nó de saída da regra. Especificamente em ocorrências de manipulação de dados persistentes, a ausência de rotinas para tratamento de exceção é suprida pela inserção de rotina dedicada a cada comando; esse enfoque é

adotado para registrar o comando que provocou a exceção, em adição ao desvio para o nó de saída da regra, conforme demonstrado na Figura 6.6. Nesta figura, o nó *n* representa a ocorrência do comando *update*; o pseudoprocedimento *contexto-excecao* objetiva registrar a posição relativa do nó *n* em relação à seqüência de nós exercitados pela aplicação de casos de teste; o nó de saída da regra é mapeado para o nó *x*. Uma implementação para *contexto-excecao* é apresentada na Subseção 6.2.3, permitindo, para um particular caso de teste, comparar o contexto de exceções ocorridas entre as versões correta e defeituosa de um conjunto de regras.

```
          .......
/*i*/    begin
/*i*/       ponta-de-prova (n);
            UPDATE ......;
/*i*/    exception
/*i*/    when others then
/*i*/       contexto-excecao ( ... );
/*i*/       ponta-de-prova (x);
/*i*/       return;
/*i*/    end;
          .......
```

Figura 6.6 – Instrumentação de fluxo de controle para comandos de manipulação desprovidos de rotinas de tratamento de exceção.

6.1.2 Modelo de Fluxo de Dados

Associações de interação exploram as relações entre regras, focando no fluxo de dados persistentes. A Seção 4.1 apresentou o modelo de fluxo de dados para a manipulação de dados persistentes, onde são realçados os atributos definidos e usados em cada comando de manipulação, efetuando uma análise por precisão de fluxo de dados. A identificação dos atributos que podem ser manipulados é realizada estaticamente, pois independe do estado da base de dados. Contudo, a determinação das *tuplas* manipuladas é alcançada pela análise dinâmica dos dados, para então se avaliar a cobertura de associações de interação para as granularidades *tupla* e *valor de atributo*. O conteúdo desta subseção complementa o modelo de fluxo de dados, focando, especificamente, na identificação de *tuplas* afetadas por comandos de manipulação.

A essência da identificação de *tuplas* é a utilização do predicado de seleção presente no comando de manipulação. O modelo utiliza a relação *fdados-tupla*, cujo esquema é *fdados-tupla*

(contexto, rel, chave, fdados), a qual é destinada a registrar a chave primária das *tuplas* definidas e usadas em cada comando de manipulação: o atributo *contexto* denota o cenário da execução do comando, caracterizado por dados tais como caso de teste, nó exercitado, posição relativa do nó exercitado, etc.; a relação sendo manipulada é identificada pelo atributo *rel*; *chave* é o atributo que recebe a identificação de *tuplas*; e o atributo *fdados* representa o tipo de ocorrência para fluxo de dados.

O modelo será apresentado para diversas construções de comandos de manipulação por meio de exemplos. Para tal, considere as relações *tab001*, *tab002* e *tab003*, descritas pelos esquemas *tab001(key001, att011, att012)*, *tab002(key002, att021, att022)* e *tab003(key003, att031, att032)*, respectivamente. A Figura 6.7 introduz um comando com seleção simples; as linhas iniciadas por */*i*/* referem-se à instrumentação para a identificação de *tuplas*; e a variável *v_contexto* possui o contexto em que o nó está sendo exercitado. É pertinente ressaltar que nas linhas de 1 até 5 está o comando para a inserção de dados na relação *fdados-tupla*, utilizando uma subconsulta que seleciona as mesmas *tuplas* do comando sendo instrumentado. O valor *'duso'* para o fluxo de dados é aplicado a comandos do tipo *select*.

A Figura 6.8 explora a seleção com função agregada e com agrupamento de dados, os quais constituem recursos de sumarização dos dados consultados. Independentemente da complexidade do comando, o modelo adotado é similar à seleção simples, pois os recursos de sumarização não interferem nos predicados de seleção de dados. Quando são utilizadas várias relações em um comando *select*, o modelo prevê a identificação de *tuplas* para cada uma das relações envolvidas. No caso da operação de junção, conforme exibido na Figura 6.9, a lista de nomes de tabelas usadas como fonte de dados e o predicado de seleção presente no comando de junção são repetidos nas subconsultas existentes nos comandos de instrumentação; o mesmo enfoque é aplicado às variações da operação de junção, tal como *outer join*.

```
/*i*/  insert into fdados-tupla
/*i*/      ( contexto, rel, chave, fdados )
/*i*/      select v_contexto, 'tab001', key001, 'duso'
/*i*/      from tab001
/*i*/      where att011 > 1200;
       select att011, att012
       from tab001
       where att011 > 1200;
```

Figura 6.7 – Identificação de *tuplas* em seleção simples.

```
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'duso'
/*i*/     from tab001
/*i*/     where att011 > 1200;
       select att011, count(*)
       from tab001
       where att011 > 1200
       group by att011
       having count(*) > 1;
```

Figura 6.8 – Identificação de *tuplas* em seleção com função agregada e agrupamento de dados.

```
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'duso'
/*i*/     from from tab001, tab002
/*i*/     where key001 = key002
/*i*/     and att011 > 1200;
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab002', key002, 'duso'
/*i*/     from from tab001, tab002
/*i*/     where key001 = key002
/*i*/     and att011 > 1200;
       select att011, att022
       from tab001, tab002
       where key001 = key002
       and att011 > 1200;
```

Figura 6.9 – Identificação de *tuplas* em seleção com junção de dados.

Subconsultas constituem comandos de consulta que estão embutidos em outros comandos de manipulação. O uso de subconsultas possui duas abordagens; em ambas o modelo prevê a identificação de *tuplas* para a subconsulta e para o comando em que ela está inserida, o qual é denominado *comando principal*. Na primeira, exibida na Figura 6.10, a execução da subconsulta é independente do comando que a contém; a instrumentação da subconsulta não referencia o comando principal que, por sua vez, possui a subconsulta em sua instrumentação. O segundo caso é mais complexo, conhecido como subconsulta correlata, cuja execução requer dados do comando principal. No exemplo da Figura 6.11, a instrumentação do comando principal repete a seleção de *tuplas* que referencia a subconsulta; contudo, é necessária uma análise semântica do

comando para construir a instrumentação da subconsulta. A abordagem descrita para subconsultas é extensiva a comandos de mudança de estado. Subconsultas correlatas estão ligadas somente a comandos de consulta; se uma subconsulta correlata estiver embutida em um comando de mudança de estado, existirá um outro comando de consulta, que será o comando principal da subconsulta. O modelo para estruturas do tipo *cursor* segue as mesmas diretrizes acima descritas para comandos de consulta, onde o comando de inserção na relação *fdados-tupla* é executado imediatamente antes da abertura do *cursor*.

```
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'duso'
/*i*/     from from tab001
/*i*/     where att011 > (select avg(att021) from tab002);
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab002', key002, 'duso'
/*i*/     from from tab002
        select att011, att012
        from tab001
        where att011 > (select avg(att021) from tab002);
```

Figura 6.10 – Identificação de *tuplas* em seleção com subconsulta.

```
 /*i*/  insert into fdados-tupla
 /*i*/     ( contexto, rel, chave, fdados )
 /*i*/     select v_contexto, 'tab001', key001, 'duso'
 /*i*/     from from tab001
 /*i*/     where att011 > (select avg(att012) from tab002
 /*i*/                      where att011 = att022);
 /*i*/  insert into fdados-tupla
 /*i*/     ( contexto, rel, chave, fdados )
 /*i*/     select v_contexto, 'tab002', key002, 'duso'
 /*i*/     from from tab002
 /*i*/     where exists (select 1 from tab001
 /*i*/                    where att011 = att022);
        select att011, att021
        from tab001
        where att011 > (select avg(att012) from tab002
                        where att011 = att022);
```

Figura 6.11 – Identificação de *tuplas* em seleção com subconsulta correlata.

Em operações de mudança de estado, o modelo é específico para cada tipo de comando. A instrução de escrita para o comando *insert* é posta imediatamente após o comando, pois requer

que os dados sendo inseridos já estejam na base de dados. A estrutura sintática do comando *insert* permite explicitar ou não a lista ordenada de atributos da relação: o conjunto de atributos e sua seqüência. A Figura 6.12 ilustra o modelo para a inserção de dados sem lista ordenada de atributos. A ausência da lista de atributos leva à necessidade de se obter a posição relativa da chave primária da relação a partir do esquema da base de dados – meta-dados; no exemplo, a chave primária da relação *tab001* posiciona-se no primeiro atributo: o valor *202* na *tupla (202, 3277, 4422)*. A instrumentação utiliza uma subconsulta à *tupla* recém-inserida, para então realizar a inserção de dados em *fdados-tupla*. O atributo *fdados* recebe o valor *'ddef'*, caracterizando a definição de dados persistentes. Na Figura 6.13 está o modelo para a inserção de dados com lista ordenada de atributos. A posição relativa dos atributos da chave primária na lista ordenada de atributos é a mesma do valor existente na lista ordenada de valores; na lista *(att011, att012, key001)*, a chave primária refere-se ao terceiro atributo. Se a relação possuir chave primária composta, é preciso que o predicado de seleção referencie cada atributo que compõe a chave primária. O conteúdo da Figura 6.14 relata a inserção de dados sem lista ordenada de atributos e com subconsulta; neste caso, é preciso conhecer os meta-dados de ambas as relações: a relação sendo instrumentada e a relação utilizada na subconsulta. No exemplo, a posição da chave primária das relações *tab001* e *tab002* é a mesma, e os valores para o atributo *chave* das *tuplas* inseridas na relação *fdados-tupla* foram obtidos diretamente do conteúdo da relação *tab002*.

```
       insert into tab001
       values (202, 3277, 4422);
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'ddef'
/*i*/     from tab001
/*i*/     where key001 = 202;
```

Figura 6.12 – Identificação de *tuplas* em inserção de dados sem lista ordenada de atributos.

```
       insert into tab001
         (att011, att012, key001)
       values (3277, 4422, 202);
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'ddef'
/*i*/     from from tab001
/*i*/     where key001 = 202;
```

Figura 6.13 – Identificação de *tuplas* em inserção de dados com lista ordenada de atributos.

```
       insert into tab001
       select * from tab002;
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key002, 'ddef'
/*i*/     from tab002;
```

Figura 6.14 – Identificação de *tuplas* em inserção de dados sem lista ordenada de atributos e com subconsulta.

As Figuras 6.15 e 6.16 exploram o fluxo de dados para os comandos *delete* e *update*, respectivamente. Segundo o modelo introduzido no Capítulo 4, definição e uso de dados persistentes podem existir em cada ocorrência desses comandos. Na Figura 6.15, as instruções de escrita para ambos os tipos de ocorrência – definição e uso – são postas imediatamente antes do comando, visto que a exclusão de dados impede que os dados manipulados sejam obtidos após a execução da operação. Note que a instrução de escrita para a definição de dados na Figura 6.16 situa-se logo após o comando, pois a execução normal do comando *update* (execução sem exceção) é requerida para a definição de dados; esta abordagem não pôde ser adotada para o comando *delete*, devido ao motivo acima mencionado.

Em geral, a instrumentação para o uso de dados persistentes deve ocorrer imediatamente antes do comando de manipulação, pois são assinaladas ocorrências de uso nos arcos de saída do nó que representa o comando, os quais caracterizam a execução normal e a execução com exceção. A existência de subconsulta implícita em qualquer dos comandos de manipulação é tratada pelo modelo da mesma maneira que subconsultas explícitas: é utilizada uma instrução de escrita na relação *fdados-tupla* para cada relação utilizada como fonte de dados.

```
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'duso'
/*i*/     from tab001
/*i*/     where att011 > 1200;
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'ddef'
/*i*/     from tab001
/*i*/     where att011 > 1200;
       delete from tab001
       where att011 > 1200;
```

Figura 6.15 – Identificação de *tuplas* em exclusão de dados.

```
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'duso'
/*i*/     from tab001
/*i*/     where att011 > 1200;
       update tab001
       set att012 = att012 * 1.1
       where att011 > 1200;
/*i*/  insert into fdados-tupla
/*i*/     ( contexto, rel, chave, fdados )
/*i*/     select v_contexto, 'tab001', key001, 'ddef'
/*i*/     from tab001
/*i*/     where att011 > 1200;
```

Figura 6.16 – Identificação de *tuplas* em modificação de dados.

6.1.3 Modelo de Avaliação

O modelo de avaliação prevê a utilização dos critérios baseados na interação entre regras na forma de critérios de adequação. A precisão de fluxo de dados persistentes afeta a avaliação de adequação a um critério de teste. Abordagens focadas unicamente no exercício de caminhos não atendem à determinação dos dados persistentes definidos e usados nos nós e nos arcos exercitados. Para aplicar a avaliação de adequação por granularidade, foi utilizado um enfoque baseado na agregação de informação de fluxo de controle e de fluxo de dados.

No contexto de associações de interação, pode-se especificar genericamente um elemento requerido como *[v, <r', n_1>, < r'', n_2>, < r'', n_3>, < r', n_4>, < r'', n_5>]*, onde: *v* representa a variável persistente; e cada par *<r, n>* denota o nó *n* da regra *r*. Sobre os pares *<r, n>* presentes nesta sêxtupla, é pertinente comentar: o primeiro par representa a regra e o nó relativos à definição de dados persistentes; o segundo e terceiro pares denotam a regra e o arco relativos ao uso de dados persistentes; e o quarto e quinto pares denotam um arco que deve ser exercitado entre a definição e o uso de dados persistentes; vale ressaltar que este arco não existe na representação de fluxo de controle de cada regra individual, pois o mesmo denota a transferência de controle devido ao disparo entre regras ou a execução seqüencial de regras. Para exemplificar a representação genérica de elementos requeridos, as associações de interação *(48)*, *(50)*, *(54)*, *(60)*, *(83)* e *(86)*, que são apresentadas nas Figuras 5.7, 5.8, 5.9, 5.10, 5.11 e 5.12, respectivamente, possuem as representações *[tab004, <trg002,7>, <trg002,7>, <trg002,9>,*

<trg002,9>, <trg002,1>], [tab003, <trg001,2>, <trg005,2>, <trg005,4>, <trg001,25>, <trg005,1>], [tab005, <trg001,12>, <trg005,1>, <trg005,2>, <trg001,12>, <trg005,1>], [tab003, <trg001,2>, <trg002,2>, <trg002,9>, <trg001,13>, <trg002,1>], [tab004, <trg002,7>, <trg001,23>, <trg001,24>, <trg002,9>, <trg001,24>] e *[tab003, <trg001,2>, <trg005,2>, <trg005,4>, <trg001,12>, <trg005,1>]*, respectivamente.

Dados o conjunto de casos de teste Λ e a especificação *[v, <r', n_1>, < r'', n_2>, < r'', n_3>, < r', n_4>, < r'', n_5>]* para a associação de interação α, o modelo de avaliação examina a dupla $< \Pi, \Gamma >$ resultante da aplicação de Λ, tal que Π é o conjunto de caminhos completos resultantes, onde: $\Pi = \{ \pi_1, ..., \pi_p \}, p > 0$; e Γ determina os dados persistentes definidos e usados ao longo de cada caminho completo de Π. A análise de cobertura da associação α na precisão de fluxo de dados β busca em $< \Pi, \Gamma >$ por pelo menos um subcaminho $p' . p''$ em algum caminho completo $\pi_i \in \Pi$, tal que: *(i)* p' é um subcaminho no grafo de fluxo de controle da regra r' descrito por $p' = (n_1, ... , n_4)$; *(ii)* p'' é um subcaminho no grafo de fluxo de controle da regra r'' descrito por $p'' = (n_5 , ... n_2 , n_3)$; *(iii)* $(n_2 , n_3) \in ddu(v, n_1)$; *(iv)* $ddef(n_1) \cap duso(n_2 , n_3) \neq \varnothing$ na precisão β; e *(v)* $[(ddef(n_1) \cap duso(n_2 , n_3)) - (ddef(succ(n_1)) \cup ... \cup ddef(prev(n_2))] \neq \varnothing$, onde *succ(m)* é o nó sucessor de *m* em π_i e *prev(m)* é o nó antecessor de *m* em π_i.

6.2 Automação dos Critérios de Teste

Esta seção descreve a automação dos critérios baseados na interação entre regras realizada por uma ferramenta, pela introdução de rotinas escritas em SQL para a fase dinâmica do teste. A referida ferramenta denomina-se ADAPT-TOOL; a sua descrição funcional sucinta é apresentada no Apêndice C. É proposto um esquema conceitual de base de dados para entidades ligadas à atividade de teste, tais como: conjunto de regras ativas em teste; tipos de defeito; regras defeituosas; dados de teste; caso de teste; oráculo, fluxo de dados e fluxo de controle.

No modelo relacional, as operações de dados persistentes manipulam conjuntos de dados, conhecidos como *relações* (Codd, 1970). O volume de dados manipulados em cada operação é afetado pelas relações de entrada e pela operação propriamente dita, sendo, portanto, pertinente utilizar estruturas de dados que suportem demandas de volume elevado. Em adição, os comandos

de instrumentação são executados no contexto transacional das operações de disparo de regras e o modelo de instrumentação deve ser capaz de conservar os registros de fluxos de controle e de dados mesmo na ocorrência de situações de exceção.

As observações acima motivaram a definição de uma base de dados para a produção da atividade de teste, utilizando as mesmas estruturas de dados do modelo relacional, a qual inclui os registros de instrumentação. A representação gráfica do esquema conceitual simplificado para a base de dados é apresentada na Figura 6.17, inspirando-se na nomenclatura adotada em (Elmasri e Navathe, 2003). Os retângulos representam tipos de entidades e os losangos denotam tipos de relacionamento; os demais elementos da notação podem ser entendidos a partir da breve descrição do esquema da base de dados apresentada abaixo:

> "Dados de teste são compostos por comandos SQL responsáveis por disparos de regras, os quais são representativos de aplicações de banco de dados, e por bases de dados de entrada; reaplicar um caso de teste requer a utilização de entidades de ambos os tipos. Toda regra defeituosa possui um único tipo de defeito e um mesmo tipo de defeito pode estar ligado a várias regras defeituosas. O conjunto de regras para qualquer caso de teste pode ser composto por várias regras, das quais somente uma única regra é defeituosa. Qualquer entidade dos tipos fluxo de dados e fluxo de controle está relacionada a um único caso de teste; contudo, um caso de teste pode possuir várias entidades de tais tipos; a mesma observação estende-se às entidades do tipo exceção ocorrida e mensagem emitida. Todo caso de teste possui oráculo próprio, significando que os dados para a decisão sobre a manifestação de falha são específicos para qualquer caso de teste".

Do ponto de vista semântico, a introdução de comandos de instrumentação não alteraria o comportamento de regras em tempo de execução, visto que a escrita de dados ocorrerá somente na base de instrumentação. Os tipos de entidade da base de dados para a produção da atividade de teste, que estão relacionados com a instrumentação de fluxo de dados e de fluxo de controle, são descritos nas subseções seguintes, na forma de comandos de definição de relações da base de dados (por exemplo, *create table*). Métodos (procedimentos e funções escritos em SQL) foram especificados para a gravação e leitura de dados de instrumentação, constituindo um recurso à automação dos critérios de interação entre regras.

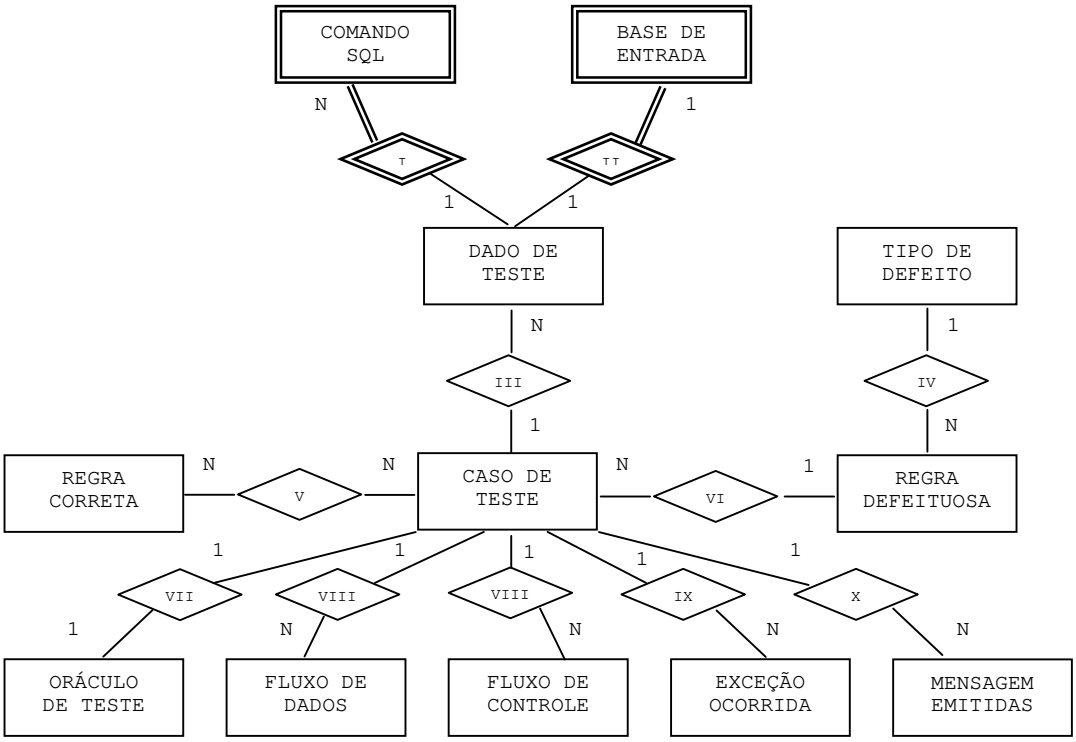


Figura 6.17 – Esquema conceitual simplificado da base de dados para a produção da atividade de teste.

6.2.1 Instrumentação de Fluxo de Controle

A instrumentação de fluxo de controle é registrada na relação *T_NODES*, cujo esquema é apresentado na Figura 6.18. Esta relação possui atributos que descrevem cada nó exercitado: o atributo *CASE_ID* identifica o caso de teste; o atributo *LINE_ID* constitui a seqüência cronológica relativa dos nós exercitados por cada caso de teste; a regra ativa é identificada pelo atributo *RULE_ID*; *NODE_ID* é o atributo que identifica o nó exercitado; e o atributo *COMMENTS* pode incluir observações sobre o nó. Na Figura 6.19 está um exemplo da relação *T_NODES*, que é resultado da aplicação do caso de teste *583*, onde foram exercitados 22 nós das regras *TRG_INVENTORY_QUANTITY* e *TRG_PRODUCT_PENDING*; ressalta-se que a execução do comando *update* nó *6* da regra *TRG_INVENTORY_QUANTITY* provocou o disparo da regra *TRG_PRODUCT_PENDING*.

A instrumentação é realizada pela inserção de chamadas à função *t_f_setnode*, cujo código fonte é apresentado na Figura 6.20, a qual é responsável pela gravação de *tuplas* na relação *T_NODES*. Esta função requer os parâmetros de entrada: identificação do caso de teste, identificação da regra ativa, número do nó exercitado e comentários sobre o nó exercitado, e sua execução retorna a seqüência cronológica relativa ao nó exercitado. O atributo retornado será utilizado pelas instrumentações de fluxo de dados, de exceções ocorridas e de mensagens emitidas, com o intuito de identificar o contexto em que o nó foi exercitado.

```
CREATE TABLE T_NODES (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,RULE_ID             VARCHAR2(50)       NOT NULL
   ,NODE_ID             INTEGER            NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_NODES__PK
                        PRIMARY KEY(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_NODES__FK_CASES
                        FOREIGN KEY(CASE_ID) REFERENCES T_CASES(CASE_ID)
   ,CONSTRAINT          T_NODES__FK_RULES
                        FOREIGN KEY(RULE_ID) REFERENCES T_RULES(RULE_ID)
   );
```

Figura 6.18 – Definição de relação *T_NODES*, destinada à instrumentação de fluxo de controle.

CASE_ID	LINE_ID	RULE_ID	NODE_ID	COMMENTS
583	1	TRG_INVENTORY_QUANTITY	1	EVENT NODE
583	2	TRG_INVENTORY_QUANTITY	2	FIRST ACTION NODE
583	3	TRG_INVENTORY_QUANTITY	4	
583	4	TRG_INVENTORY_QUANTITY	5	
583	5	TRG_INVENTORY_QUANTITY	6	UPDATE COMMAND
583	6	TRG_PRODUCT_PENDING	1	EVENT NODE
583	7	TRG_PRODUCT_PENDING	2	FIRST ACTION NODE
583	8	TRG_PRODUCT_PENDING	4	
583	9	TRG_PRODUCT_PENDING	5	
583	10	TRG_PRODUCT_PENDING	7	BEGIN BLOCK
583	11	TRG_PRODUCT_PENDING	8	SELECT COMMAND
583	12	TRG_PRODUCT_PENDING	9	
583	13	TRG_PRODUCT_PENDING	10	
583	14	TRG_PRODUCT_PENDING	12	
583	15	TRG_PRODUCT_PENDING	17	
583	16	TRG_PRODUCT_PENDING	21	END BLOCK
583	17	TRG_PRODUCT_PENDING	22	
583	18	TRG_PRODUCT_PENDING	23	EXIT NODE
583	19	TRG_INVENTORY_QUANTITY	7	
583	20	TRG_INVENTORY_QUANTITY	8	
583	21	TRG_INVENTORY_QUANTITY	10	
583	22	TRG_INVENTORY_QUANTITY	11	EXIT NODE

Figura 6.19 – Exemplo da relação *T_NODES* para a instrumentação de fluxo de controle.

```
CREATE OR REPLACE FUNCTION t_f_setnode
      ( t_v_case_id  T_NODES.CASE_ID%TYPE
       ,t_v_rule_id  T_NODES.RULE_ID%TYPE
       ,t_v_node_id  T_NODES.NODE_ID%TYPE
       ,t_v_comments T_NODES.COMMENTS%TYPE )
      RETURN NUMBER
IS
      t_v_line_id integer;
BEGIN
    SELECT (NVL(MAX(LINE_ID),0)+1) INTO t_v_line_id
    FROM T_NODES
    WHERE CASE_ID = t_v_case_id;
    INSERT INTO T_NODES
      (CASE_ID, LINE_ID, RULE_ID, NODE_ID, COMMENTS)
    VALUES
      (t_v_case_id, t_v_line_id, t_v_rule_id, t_v_node_id, t_v_comments);
    RETURN t_v_line_id;
EXCEPTION
    WHEN OTHERS THEN
       raise_application_error(-20001,'[FUNCTION t_f_setnode] ' || SQLCODE
|| ' -ERROR- ' || SQLERRM);
END t_f_setnode;
```

Figura 6.20 – A função *t_f_setnode*, que registra o exercício de fluxo de controle.

6.2.2 Instrumentação de Fluxo de Dados

A instrumentação de fluxo de dados é registrada na relação *T_DATA*, cujo esquema é apresentado na Figura 6.21. Essa atividade ocorre para toda definição e para todo uso ocorridos em quaisquer das relações da base de dados da aplicação. Cada *tupla* na relação *T_DATA* registra a definição e/ou o uso de um atributo em alguma *tupla* manipulada da relação sendo instrumentada. Diferentemente da relação *T_NODES* que possui uma única *tupla* para cada nó exercitado, a relação *T_DATA* pode possuir várias *tuplas* resultantes do exercício de algum nó que mapeia ocorrências de manipulação. Esta relação é descrita pelos atributos seguintes: o atributo *CASE_ID* identifica o caso de teste; o atributo *LINE_ID* constitui a seqüência cronológica relativa do nó exercitado; a relação da aplicação (variável persistente) é identificada pelo atributo *TABLE_ID*; *KEY_ID* é o atributo que se refere a uma chave primária da relação identificada pelo atributo *TABLE_ID*; *COLUMN_ID* identifica um atributo da relação estabelecida em *TABLE_ID*; e os atributos *DDEF* e *DUSE* determinam, respectivamente, se houve definição e uso de dados persistentes no atributo identificado por *COLUMN_ID* e na *tupla* identificada *KEY_ID* da relação identificada por *TABLE_ID*.

```
CREATE TABLE T_DATA (
    CASE_ID     INTEGER            NOT NULL
   ,LINE_ID     INTEGER            NOT NULL
   ,TABLE_ID    VARCHAR2(50)       NOT NULL
   ,KEY_ID      VARCHAR2(50)       NOT NULL
   ,COLUMN_ID   VARCHAR2(50)       NOT NULL
   ,DDEF        NUMBER(1)              NULL
   ,DUSE        NUMBER(1)              NULL
   ,CONSTRAINT  T_DATA__PK
                PRIMARY KEY(CASE_ID, LINE_ID, TABLE_ID, KEY_ID, COLUMN_ID)
   ,CONSTRAINT  T_DATA__FK_NODES
                FOREIGN KEY(CASE_ID, LINE_ID) REFERENCES T_NODES(CASE_ID, LINE_ID)
   ,CONSTRAINT  T_DATA__FK_COLUMNS
                FOREIGN KEY(TABLE_ID, COLUMN_ID) REFERENCES T_COLUMNS(TABLE_ID, COLUMN_ID)
   );
```

Figura 6.21 – Definição de relação *T_DATA*, destinada à instrumentação de fluxo de dados.

Na Figura 6.22 está um exemplo da relação *T_DATA*, que é resultado da aplicação do caso de teste *607*. Para ilustrar, no *LINE_ID 23* é registrada a manipulação da *tupla* identificada por *1729* da relação *INVENTORY_PENDING*, onde são definidos os atributos *DATE_PENDING* e *QUANTITY_PENDING* e usado o atributo que referente à chave primária (*T_KEY*). Note que não existe referência à regra nem ao nó exercitado; tais dados podem ser obtidos a partir da restrição *T_DATA__FK_NODES*, descrita na Figura 6.21: o par *(CASE_ID, LINE_ID)* é uma chave estrangeira que referencia a relação *T_NODES*.

A instrumentação é realizada pela inserção de *tuplas* na relação *T_DATA_WORK*, a qual possui um regra ativa, denominada *T_TRG_DATA_WORK*, que é disparada a cada inserção ou atualização de dados nesta relação; a Figura 6.23 apresenta o esquema da relação e a definição da referida regra ativa. A função dessa regra ativa é interpretar cada *tupla* inserida na relação *T_DATA_WORK* para então inserir *tuplas* na relação *T_DATA* referentes à instrumentação de fluxo de dados. Note que se está utilizando o recurso de regras ativas na aplicação de produção da atividade de teste.

A Figura 6.24 exemplifica a instrumentação de fluxo de dados e de fluxo de controle para uma ocorrência comando *update*, presente no nó *16* de uma regra ativa. A nomenclatura */*node*/* no início de linha de código indica que a instrução desta linha refere-se à instrumentação de fluxo de controle; a nomenclatura */*data*/* denota a instrumentação de fluxo de dados. A primeira instrução registra o exercício do nó *16*; nesta instrução, a chamada da função *t_f_setnode* atribui

valor à variável *t_v_line_id*, destinada à seqüência cronológica relativa ao exercício do nó. As variáveis *t_v_case_id* e *t_v_rule_id* possuem a identificação do caso de teste e da regra ativa, respectivamente. Existem duas inserções de dados na relação *T_DATA_WORK*; a primeira objetiva a instrumentação do uso de dados persistentes para o atributo *QUANTITY_COMMIT* e para os atributos da chave primária da relação *ORDER_ITEMS*; a segunda tenciona a instrumentação da definição de dados persistentes para o atributo *QUANTITY_COMMIT* desta relação. Em ambas as inserções, as *tuplas* manipuladas são obtidas a partir do mesmo predicado de seleção de *tuplas* presente no comando *update* (*ORDER_ID = v_order_id_to_apply AND PRODUCT_ID = v_product_id_to_apply*).

CASE_ID	LINE_ID	TABLE_ID	KEY_ID	COLUMN_ID	DDEF	DUSE
607	14	PRODUCT_STATUS	1729	QUANTITY_ON_HAND	1	1
607	14	PRODUCT_STATUS	1729	T_KEY		1
607	20	INVENTORY_PENDING	1729	T_KEY		1
607	23	INVENTORY_PENDING	1729	DATE_PENDING	1	
607	23	INVENTORY_PENDING	1729	QUANTITY_PENDING	1	
607	23	INVENTORY_PENDING	1729	T_KEY		1
607	31	PRODUCT_STATUS	1729	QUANTITY_ON_HAND	1	1
607	31	PRODUCT_STATUS	1729	T_KEY		1
607	37	INVENTORY_PENDING	1729	T_KEY		1
607	40	INVENTORY_PENDING	1729	DATE_PENDING	1	
607	40	INVENTORY_PENDING	1729	QUANTITY_PENDING	1	
607	40	INVENTORY_PENDING	1729	T_KEY		1

Figura 6.22 – Exemplo da relação *T_DATA* para a instrumentação de fluxo de dados.

```
CREATE TABLE T_DATA_WORK (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,TABLE_ID            VARCHAR2(50)       NOT NULL
   ,KEY_ID              VARCHAR2(50)       NOT NULL
   ,DDEF                NUMBER(1)              NULL
   ,DUSE                NUMBER(1)              NULL
   ,COLUMNS             VARCHAR2(200)          NULL
);

CREATE OR REPLACE TRIGGER T_TRG_DATA_WORK
    AFTER INSERT OR UPDATE ON T_DATA_WORK
    FOR EACH ROW
DECLARE
      t_v_instance    INTEGER;
      t_v_column_id   VARCHAR2(50);
BEGIN
    IF UPDATING THEN
       raise_application_error(-20001,'[TRIGGER T_TRG_DATA_WORK] Updating on T_DATA_WORK is
not allowed.');
    END IF;
    t_v_instance := 1;
    t_v_column_id := t_f_getcmp (:NEW.COLUMNS, t_v_instance, ';');
    WHILE (t_v_column_id IS NOT NULL) LOOP
       BEGIN
          INSERT INTO T_DATA
             (CASE_ID, LINE_ID, TABLE_ID, KEY_ID, COLUMN_ID, DDEF, DUSE )
          VALUES
             (:NEW.CASE_ID, :NEW.LINE_ID, :NEW.TABLE_ID, :NEW.KEY_ID, t_v_column_id,
:NEW.DDEF, :NEW.DUSE );
       EXCEPTION
          WHEN OTHERS THEN
             IF :NEW.DDEF IS NOT NULL THEN
                UPDATE T_DATA
                SET    DDEF      = :NEW.DDEF
                WHERE  CASE_ID   = :NEW.CASE_ID
                AND    LINE_ID   = :NEW.LINE_ID
                AND    TABLE_ID  = :NEW.TABLE_ID
                AND    KEY_ID    = :NEW.KEY_ID
                AND    COLUMN_ID = t_v_column_id;
             END IF;
             IF :NEW.DUSE IS NOT NULL THEN
                UPDATE T_DATA
                SET    DUSE      = :NEW.DUSE
                WHERE  CASE_ID   = :NEW.CASE_ID
                AND    LINE_ID   = :NEW.LINE_ID
                AND    TABLE_ID  = :NEW.TABLE_ID
                AND    KEY_ID    = :NEW.KEY_ID
                AND    COLUMN_ID = t_v_column_id;
             END IF;
       END;
       t_v_instance := t_v_instance + 1;
       t_v_column_id := t_f_getcmp (:NEW.COLUMNS, t_v_instance, ';');
    END LOOP;
END T_TRG_DATA_WORK;
```

Figura 6.23 – Definição de relação *T_DATA_WORK* e da regra ativa destinada à instrumentação de fluxo de dados.

```
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 16, 'UPDATE COMMAND');
/*data*/ INSERT INTO T_DATA_WORK
/*data*/    ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/    SELECT t_v_case_id, t_v_line_id, 'ORDER_ITEMS',
/*data*/           CONCAT(ORDER_ID, PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY_COMMIT;'
/*data*/    FROM   ORDER_ITEMS
/*data*/    WHERE  ORDER_ID   = v_order_id_to_apply
/*data*/    AND    PRODUCT_ID = v_product_id_to_apply;
         UPDATE ORDER_ITEMS
         SET    QUANTITY_COMMIT = QUANTITY_COMMIT + v_quantity_to_apply
         WHERE  ORDER_ID   = v_order_id_to_apply
         AND    PRODUCT_ID = v_product_id_to_apply;
/*data*/ INSERT INTO T_DATA_WORK
/*data*/    ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/    SELECT t_v_case_id, t_v_line_id, 'ORDER_ITEMS',
/*data*/           CONCAT(ORDER_ID, PRODUCT_ID), 1, NULL, 'QUANTITY_COMMIT;'
/*data*/    FROM   ORDER_ITEMS
/*data*/    WHERE  ORDER_ID   = v_order_id_to_apply
/*data*/    AND    PRODUCT_ID = v_product_id_to_apply;
```

Figura 6.24 – Exemplo de instrumentação de fluxo de dados e de fluxo de controle para o comando *update*.

6.2.3 Instrumentação de Exceções Ocorridas

A instrumentação de exceções ocorridas possui dois objetivos: *(i)* registrar o contexto da ocorrência de exceção, o qual será utilizado pelo oráculo de teste na determinação da manifestação de falha; e *(ii)* em alguns casos, evitar que a ocorrência de exceção cancele a transação da execução de regras ativas e, conseqüentemente, elimine as *tuplas* de instrumentação gravadas. Esta instrumentação é registrada na relação *T_CASE_EXCEPTION*, cujo esquema é apresentado na Figura 6.25. Cada *tupla* nesta relação refere-se a: exceções não tratadas ocorridas em comando de manipulação de dados; quaisquer exceções tratadas em blocos *begin-exception-end*; e exceções provocadas por comandos *raise*. Esta relação é descrita pelos atributos seguintes: o atributo *CASE_ID* identifica o caso de teste; os atributos *RULE_ID* e *NODE_ID* denotam, respectivamente, a regra ativa e o nó exercitado; observações sobre a exceção são postas no atributo *COMMENTS*.

A instrumentação é realizada pela inserção de chamadas ao procedimento *t_p_setcaseexception*, cujo código fonte é apresentado na Figura 6.26, o qual é responsável pela gravação de *tuplas* na relação *T_CASE_EXCEPTION*. Este procedimento requer os parâmetros de entrada: identificação do caso de teste, seqüência cronológica relativa ao nó exercitado e observações sobre a exceção ocorrida.

```
CREATE TABLE T_CASE_EXCEPTION (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,RULE_ID             VARCHAR2(50)       NOT NULL
   ,NODE_ID             INTEGER            NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_CASE_EXCEPTION__PK
                        PRIMARY KEY(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_EXCEPTION__FK_NODES
                        FOREIGN KEY(CASE_ID, LINE_ID) REFERENCES T_NODES(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_EXCEPTION__FK_RULES
                        FOREIGN KEY(RULE_ID) REFERENCES T_RULES(RULE_ID)
   );
```

Figura 6.25 – Definição de relação *T_CASE_EXCEPTION*, destinada à instrumentação de exceções ocorridas.

```
CREATE OR REPLACE PROCEDURE t_p_setcaseexception
      ( t_v_case_id      T_NODES.CASE_ID%TYPE
       ,t_v_line_id      T_NODES.LINE_ID%TYPE
       ,t_v_comments     T_CASE_EXCEPTION.COMMENTS%TYPE )
IS
        t_v_rule_id      T_NODES.RULE_ID%TYPE;
        t_v_node_id      T_NODES.NODE_ID%TYPE;
BEGIN
   SELECT RULE_ID, NODE_ID INTO t_v_rule_id, t_v_node_id
   FROM   T_NODES
   WHERE  CASE_ID = t_v_case_id
   AND    LINE_ID = t_v_line_id;
   INSERT INTO T_CASE_EXCEPTION
       ( CASE_ID, LINE_ID, RULE_ID, NODE_ID, COMMENTS )
   VALUES
       ( t_v_case_id, t_v_line_id, t_v_rule_id, t_v_node_id, t_v_comments
);
EXCEPTION
      WHEN OTHERS THEN
           RAISE_APPLICATION_ERROR (-20001,'[PROCEDURE ' ||
'T_P_SETCASEEXCEPTION] ' || SQLCODE || ' -ERROR- ' || SQLERRM);
END t_p_setcaseexception;
```

Figura 6.26 – O procedimento *t_p_setcaseexception*, que registra exceções ocorridas.

6.2.4 Instrumentação de Mensagens Emitidas

A instrumentação de mensagens objetiva registrar a emissão de mensagens, para apoiar o oráculo de teste na determinação da manifestação de falha. A instrumentação é registrada na relação *T_CASE_MESSAGE*, cujo esquema é apresentado na Figura 6.27. Esta relação é descrita pelos atributos seguintes: o atributo *CASE_ID* identifica o caso de teste; os atributos *RULE_ID* e *NODE_ID* denotam, respectivamente, a regra ativa e o nó exercitado; e as mensagens propriamente ditas são postas no atributo *COMMENTS*.

A instrumentação de mensagens é realizada pela inserção de chamadas ao procedimento *t_p_setcasemessage*, cujo código fonte é apresentado na Figura 6.28, o qual é responsável pela gravação de *tuplas* na relação *T_CASE_MESSAGE*. Este procedimento requer os parâmetros de entrada: identificação do caso de teste, seqüência cronológica relativa ao nó exercitado e mensagem emitida.

6.2.5 Oráculo de Teste

O oráculo de teste determina se houve manifestação de falha após a aplicação de cada caso de teste. Essa tomada de decisão considera as seguintes categorias de dados: estado final da base de dados; mensagens emitidas; e exceções ocorridas. A primeira e segunda categorias são utilizadas por motivos óbvios, conforme a noção de *caso de teste* definida no Capítulo 4. A última categoria captura o contexto de cada exceção ocorrida, para determinar o estado de execução no momento da exceção, em relação aos conjuntos de regras corretas e defeituosas; vale lembrar que a instrumentação de exceções ocorridas habilita que a transação de banco de dados não seja cancelada, permitindo que os registros de fluxo de controle e de fluxo de dados não sejam perdidos.

Em cada caso de teste, os dados de teste são aplicados aos conjuntos de regras corretas e defeituosas. As três categorias de dados acima citadas são coletadas em ambas as execuções e comparadas entre si para a tomada de decisão do oráculo. A partir dos dados coletados, é calculado um valor numérico (*tag*) que retrata qualquer execução do conjunto de regras. A automação do cálculo da *tag* determina a automação do trabalho do oráculo.

```
CREATE TABLE T_CASE_MESSAGE (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,RULE_ID             VARCHAR2(50)       NOT NULL
   ,NODE_ID             INTEGER            NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_CASE_MESSAGE__PK
                        PRIMARY KEY(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_MESSAGE__FK_NODES
                        FOREIGN KEY(CASE_ID, LINE_ID) REFERENCES T_NODES(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_MESSAGE__FK_RULES
                        FOREIGN KEY(RULE_ID) REFERENCES T_RULES(RULE_ID)
   );
```

Figura 6.27 – Definição de relação *T_CASE_MESSAGE*, destinada à instrumentação de mensagens emitidas.

```
CREATE OR REPLACE PROCEDURE t_p_setcasemessage
      ( t_v_case_id      T_NODES.CASE_ID%TYPE
       ,t_v_line_id      T_NODES.LINE_ID%TYPE
       ,t_v_comments     T_CASE_EXCEPTION.COMMENTS%TYPE )
IS
        t_v_rule_id      T_NODES.RULE_ID%TYPE;
        t_v_node_id      T_NODES.NODE_ID%TYPE;
BEGIN
   SELECT RULE_ID, NODE_ID INTO t_v_rule_id, t_v_node_id
   FROM   T_NODES
   WHERE  CASE_ID = t_v_case_id
   AND    LINE_ID = t_v_line_id;
   INSERT INTO T_CASE_MESSAGE
        ( CASE_ID, LINE_ID, RULE_ID, NODE_ID, COMMENTS )
   VALUES
        ( t_v_case_id, t_v_line_id, t_v_rule_id, t_v_node_id, t_v_comments
);
EXCEPTION
      WHEN OTHERS THEN
           RAISE_APPLICATION_ERROR (-20001,'[PROCEDURE ' ||
'T_P_SETCASEMESSAGE] ' || SQLCODE || ' -ERROR- ' || SQLERRM);
END t_p_setcasemessage;
```

Figura 6.28 – O procedimento *t_p_setcasemessage*, que registra mensagens emitidas.

```
CREATE OR REPLACE FUNCTION t_f_gettabletag
      ( t_v_table_id       T_TABLES.TABLE_ID%TYPE )
      RETURN T_TABLES.TABLETAG%TYPE
IS
      t_v_column_name      SYS.ALL_TAB_COLUMNS.COLUMN_NAME%TYPE;
      t_v_tabletag         T_TABLES.TABLETAG%TYPE;
      t_v_sql_command      VARCHAR2(5000);
      CURSOR t_c_columns
             IS
             SELECT COLUMN_NAME
             FROM   SYS.ALL_TAB_COLUMNS
             WHERE  ( OWNER = SYS_CONTEXT ('USERENV', 'CURRENT_USER') )
             AND    ( TABLE_NAME = t_v_table_id )
             AND    ( DATA_TYPE NOT LIKE 'TIMESTAMP%' );
BEGIN
      UPDATE T_TABLES
      SET    TABLETAG = NULL
      WHERE  TABLE_ID = t_v_table_id;
      OPEN t_c_columns;
      LOOP
         FETCH t_c_columns INTO t_v_column_name;
         IF t_c_columns%NOTFOUND THEN
            CLOSE t_c_columns;
            EXIT;
         END IF;
         t_v_sql_command :=
                    'DECLARE t_v_tabletag       T_TABLES.TABLETAG%TYPE; '          ||
                            't_v_column_value   VARCHAR2(2000); '                  ||
                            't_v_index          INTEGER; '                         ||
                            'CURSOR t_c_column_value '                             ||
                            'IS '                                                  ||
                            'SELECT TO_CHAR(' || t_v_column_name || ') '           ||
                            'FROM ' || t_v_table_id || '; '                        ||
                    'BEGIN '                                                       ||
                          't_v_tabletag := NULL; '                                 ||
                          'OPEN t_c_column_value; '                                ||
                          'LOOP '                                                  ||
                          '   FETCH t_c_column_value INTO t_v_column_value; '      ||
                          '   IF t_c_column_value%NOTFOUND THEN '                  ||
                          '      CLOSE t_c_column_value; '                         ||
                          '      EXIT; '                                           ||
                          '   END IF; '                                            ||
                          '   t_v_index := 1; '                                    ||
                          '   WHILE (t_v_column_value IS NOT NULL) AND '           ||
                          '         (LENGTH(t_v_column_value) >= t_v_index) LOOP ' ||
                          '     t_v_tabletag := NVL(t_v_tabletag, 0) + '          ||
                          '       ASCII(SUBSTR(t_v_column_value, t_v_index, 1)); ' ||
                          '     t_v_index := t_v_index + 1; '                      ||
                          '   END LOOP; '                                          ||
                          'END LOOP; '                                             ||
                          'UPDATE T_TABLES '                                       ||
                          'SET    TABLETAG = NVL(TABLETAG, 0) + '                  ||
                          '                  NVL(t_v_tabletag, 0) '               ||
                          'WHERE  TABLE_ID = ''' || t_v_table_id || '''; '         ||
                    'END; ';
        BEGIN
           EXECUTE IMMEDIATE t_v_sql_command;
        EXCEPTION
           WHEN OTHERS THEN
                RAISE_APPLICATION_ERROR (-20001,'[PROCEDURE T_P_SETTABLETAG] ' ||
                'computing tag table: ' || t_v_table_id || ' [' ||
                t_v_column_name || '] ' || SQLCODE || ' -ERROR- ' || SQLERRM);
        END;
      END LOOP;
      SELECT TABLETAG INTO t_v_tabletag
      FROM   T_TABLES
      WHERE  TABLE_ID = t_v_table_id;
      RETURN t_v_tabletag;
END t_f_gettabletag;
```

Figura 6.29 – A função *t_f_gettabletag*, que efetua o calcula da *tag* da uma dada relação.

A função *t_f_gettabletag*, ilustrada na Figura 6.29, realiza o cálculo da *tag* para uma dada relação. O nome da relação é o parâmetro de entrada da função, e o valor de retorno é a *tag* calculada a partir do estado corrente da relação. Uma breve descrição do algoritmo é apresentada abaixo:

1. iniciar a *tag* da relação de entrada r_x;
2. para cada atributo a_x da relação r_x:
   2.1. construir um comando que consulta o atributo a_x na relação r_x (*select* a_x *from* r_x):
   2.2. executar o comando de consulta; para cada *tupla* t_x retornada pelo comando:
      2.2.1. para cada *byte* b_x do atributo a_x na *tupla* t_x:
         2.2.1.1. acumular o equivalente decimal de b_x na *tag* de r_x;
3. retornar a *tag* da relação de entrada r_x.

A *tag* da base de dados é a soma das *tags* das relações da base de dados. A automação da tarefa do oráculo utiliza a *tag* resultante da aplicação de um caso de teste λ, a qual é dada pela expressão $tag(\lambda) = t^{bd} + t^{m} + t^{e}$, onde: t^{bd} representa a *tag* da base de dados; t^{m} e t^{e} são as *tags* das relações *T_CASE_MESSAGE* e *T_CASE_EXCEPTION*, respectivamente, considerando somente as *tuplas* referentes ao caso de teste λ.

6.2.6 Avaliação por Granularidade

A avaliação por precisão de fluxo de dados é realizada pelo uso de funções dedicadas a cada granularidade. Para ilustrar esse aspecto considere o comando SQL da Figura 6.30, o qual está presente na função que retorna se o caso de teste λ cobre ou não a associação de interação α na granularidade *valor de atributo*.

```
/* L01 */       SELECT COUNT(*) INTO t_v_cover
/* L02 */       FROM   T_DATA D1, T_DATA D2,
/* L03 */              T_NODES N1, T_NODES N2, T_NODES N3, T_NODES N4, T_NODES N5
/* L04 */       WHERE  N1.CASE_ID   = t_v_case_id
/* L05 */       AND    N1.RULE_ID   = t_v_rule_id_1
/* L06 */       AND    N1.NODE_ID   = t_v_node_id_1
/* L07 */       AND    N2.CASE_ID   = t_v_case_id
/* L08 */       AND    N2.RULE_ID   = t_v_rule_id_2
/* L09 */       AND    N2.NODE_ID   = t_v_node_id_2
/* L10 */       AND    N3.CASE_ID   = t_v_case_id
/* L11 */       AND    N3.RULE_ID   = t_v_rule_id_3
/* L12 */       AND    N3.NODE_ID   = t_v_node_id_3
/* L13 */       AND    N4.CASE_ID   = t_v_case_id
/* L14 */       AND    N4.RULE_ID   = t_v_rule_id_4
/* L15 */       AND    N4.NODE_ID   = t_v_node_id_4
/* L16 */       AND    N5.CASE_ID   = t_v_case_id
/* L17 */       AND    N5.RULE_ID   = t_v_rule_id_5
/* L18 */       AND    N5.NODE_ID   = t_v_node_id_5
/* L19 */       AND    D1.CASE_ID   = t_v_case_id
/* L20 */       AND    D1.TABLE_ID  = t_v_table_id
/* L21 */       AND    D1.LINE_ID   = N1.LINE_ID
/* L22 */       AND    D1.DDEF      = 1
/* L23 */       AND    D2.CASE_ID   = t_v_case_id
/* L24 */       AND    D2.TABLE_ID  = t_v_table_id
/* L25 */       AND    D2.LINE_ID   = N2.LINE_ID
/* L26 */       AND    D2.DUSE      = 1
/* L27 */       AND    N1.LINE_ID   < N2.LINE_ID
/* L28 */       AND    ( ( N3.LINE_ID = N2.LINE_ID + 1 ) OR
/* L29 */                ( N3.LINE_ID = ( SELECT NVL(MIN(X.LINE_ID), 0) FROM T_NODES X
/* L30 */                                 WHERE  ( X.CASE_ID = N3.CASE_ID )
/* L31 */                                 AND    ( X.RULE_ID = N3.RULE_ID )
/* L32 */                                 AND    ( X.LINE_ID > N2.LINE_ID ) ) ) )
/* L33 */       AND    N1.LINE_ID   < N4.LINE_ID
/* L34 */       AND    N2.LINE_ID   > N4.LINE_ID
/* L35 */       AND    N5.LINE_ID   = N4.LINE_ID + 1
/* L36 */       AND    N5.LINE_ID   < N2.LINE_ID
/* L37 */       AND    D2.COLUMN_ID = D1.COLUMN_ID
/* L38 */       AND    D2.KEY_ID    = D1.KEY_ID
/* L39 */       AND    NOT EXISTS ( SELECT 1
/* L40 */                           FROM   T_DATA X
/* L41 */                           WHERE  X.CASE_ID   = t_v_case_id
/* L42 */                           AND    X.TABLE_ID  = t_v_table_id
/* L43 */                           AND    X.LINE_ID   > N1.LINE_ID
/* L44 */                           AND    X.LINE_ID   < N2.LINE_ID
/* L45 */                           AND    X.DDEF      = 1
/* L46 */                           AND    X.COLUMN_ID = D1.COLUMN_ID
/* L47 */                           AND    X.KEY_ID    = D1.KEY_ID );
```

Figura 6.30 – Comando SQL para a avaliação de associações de interação na granularidade *valor de atributo*.

Com respeito à Figura 6.30, no início de cada linha do comando está um identificador da linha segundo o formato *Lnn*, onde *nn* é um seqüencial da posição relativa da linha. A variável *t_v_case_id* identifica o caso de teste λ; a variável persistente é representada em *t_v_table_id*; *t_v_rule_id_m* e *t_v_node_id_m* denotam, respectivamente, os componentes dos pares $<r, n>$ da sêxtupla que descreve a associação α, conforme modelo apresentado na Subseção 6.1.3, onde *m*, $1 \leq m \leq 5$, é o identificador do par. Ao ser executado, o comando retorna um valor numérico à

variável *t_v_cover*; se este valor for superior a zero, então a aplicação de λ cobre α na precisão *valor de atributo*. A linha *L02* declara duas instâncias da relação *T_DATA*, para a definição e para o uso de dados persistentes, respectivamente. A linha *L03* declara cinco instâncias para a relação *T_NODES*, para cada um dos pares *<r, n>*. As linhas *L04* a *L18* selecionam cada uma das instâncias da relação *T_NODES*, pela atribuição de valor aos atributos *CASE_ID*, *RULE_ID* e *NODE_ID*. As linhas *L19* a *L26* selecionam as instâncias da relação *T_DATA*, pela atribuição de valor aos atributos *CASE_ID*, *TABLE_ID* e *LINE_ID*. As demais linhas complementam a operação de seleção de dados. As linhas *L27* a *L36* restringem a seqüência de nós nos caminhos executados. As linhas *L37* e *L38* representam os predicados para que a associação seja coberta na precisão *valor de atributo*. A redefinição de dados persistentes na precisão *valor de atributo* é prevista nos predicados das linhas *L39* a *L47*. A avaliação da adequação de associações de interação as demais granularidades é realizada por variações dos predicados presentes nas linhas *L37* a *L47*.

6.3 Considerações Finais

Neste capítulo, foram apresentados aspectos de implementação dos Critérios Baseados na Interação entre Regras Ativas. Apresentaram-se os modelos de fluxo de controle, de fluxo de dados, de instrumentação e de avaliação.

Uma contribuição pertinente é a abordagem para a instrumentação de fluxo de dados, que visa à avaliação da cobertura de associações de interação em diversas precisões de fluxo de dados. A avaliação de cobertura em várias granularidades é um dos aspectos investigados nesta pesquisa, conforme resultados apresentados no experimento descrito no Capítulo 5, requerendo, portanto, um suporte automatizado visando a sua aplicação.

A instrumentação para os fluxos de controle e de dados é realizada de forma separada, cada qual possuindo comandos de escrita próprios. Exceções na execução de comandos de manipulação inspiraram uma abordagem dedicada, dando-se atenção ao aninhamento de blocos, ao contexto de sua ocorrência e à prevenção para que os registros de instrumentação não sejam perdidos.

Outra característica interessante é que os aspectos de implementação consideram o apoio de uma base de dados de teste. Esta base utiliza as mesmas estruturas de dados do modelo relacional, habilitando o uso direto da SQL no desenvolvimento das funções de uma ferramenta de teste. Rotinas escritas em SQL foram apresentadas para a implementação dos modelos propostos.

Os tópicos explorados neste capítulo foram utilizados no desenvolvimento da ferramenta ADAPT-TOOL, cuja funcionalidade é mostrada no Apêndice C, que automatiza a fase dinâmica atribuída à aplicação dos critérios propostos, incluindo as funções: geração de dados de teste; registro dos casos de teste aplicados, aplicação e re-aplicação de casos de teste; avaliação da cobertura de associações de interação por granularidade; e decisão se a presença de defeitos foi revelada.

Capítulo 7

Conclusões e Trabalhos Futuros

Apesar de amplamente utilizada na indústria e na academia, a tecnologia de bancos de dados e a persistência de dados ainda são um campo aberto a pesquisas focadas na busca da qualidade de software. Técnicas de teste para programas convencionais têm sido propostas, implementadas e avaliadas, mas relativamente pouco esforço tem sido dedicado ao desenvolvimento de técnicas sistemáticas na direção de corretude de aplicações de banco de dados (Chan and Cheung, 1999; Chays et al., 2000; Kapfhammer and Soffa, 2003; Zhang et al., 2001). Particularmente sobre regras ativas escritas em SQL, sabemos de uma única pesquisa que aborda o teste de aplicações de bancos de dados ativos, a qual aplica os critérios Potenciais Usos (Maldonado, 1991; Maldonado et al., 1992) ao teste de regras individuais (Cardoso, 2004).

Neste trabalho foi investigado o uso de informação de fluxo de dados persistentes no contexto de teste de regras ativas escritas em SQL. Especificamente, buscou-se estabelecer uma estratégia de teste para a descoberta de defeitos existentes em um conjunto de regras, que podem não ter sido revelados durante o teste de regras isoladas. Em adição, a interação entre regras tem sido citada pela literatura como fonte de defeitos e de comportamento não determinístico (Vaduva, 1999), cuja complexidade dificulta seu entendimento e gerenciamento (Widow e Ceri, 1996).

O objetivo principal foi contribuir com a qualidade de aplicações de bancos de dados ativos, tendo em vista a descoberta da presença de defeitos a um custo aplicável, especificamente para: propor requisitos de teste de adequação para tais aplicações; focar em aspectos de interação entre regras escritas em SQL; e explorar as relações de fluxo de dados persistentes no contexto de teste estrutural de software.

7.1 Síntese do Trabalho

No âmbito da SQL, estudou-se o suporte previsto por padrões de programação sobre os recursos disponíveis às aplicações de bancos de dados ativos. Constatou-se que, em relação aos modelos descritivos de sistemas de bancos de dados ativos – modelo de conhecimento e de execução –, a SQL possui dimensões próprias para o desenvolvimento e uso de regras ativas, em sintonia com a constatação de que cada sistema de banco de dados ativo, protótipo ou produto, adota um subconjunto reduzido dos modelos de descrição (Chakravarthy, 1993). Esse aspecto influenciou as representações adotadas nesta pesquisa, especificamente para delinear os aspectos de interação entre regras que basearam a estratégia de teste proposta e para estabelecer modelos de implementação aos critérios de teste.

Visando à compreensão do relacionamento entre defeitos e falhas na programação SQL, estabeleceu-se uma enumeração de defeitos em comandos de manipulação, constituindo um elemento pertinente na direção de se propor uma abordagem sistemática de teste para aplicações de banco de dados. As construções básicas de comandos SQL de manipulação foram examinadas e organizadas em itens estruturais, onde uma lista de tipos de defeitos foi apresentada para cada item estrutural. O conceito de falha de manipulação foi estudado, caracterizando-se dimensões de falha em nível de comandos individuais, permitindo o entendimento de como defeitos de manipulação são propagados para a saída de execução de comandos. Um mapeamento entre tipos de defeitos e categorias de falha de manipulação foi construído, aumentando o conhecimento sobre defeitos e falhas de manipulação para a atividade de teste.

Foi enfatizado o conceito de associação persistente, fundamentada nos fluxos de dados oriundos da presença dos comandos de manipulação da SQL. Cada comando foi analisado para determinar as ocorrências de definição e de uso de dados persistentes, onde se ressaltaram as diferenças desse modelo de fluxo de dados em relação a outras iniciativas da literatura. Adotou-se uma abordagem inspirada em Spoto (2000), onde a definição persistente ocorre em nós de comandos de manipulação e o uso persistente existe nos arcos de saída de tais nós, em decorrência do potencial de exceções provocadas.

Uma característica essencial evidenciada no contexto de associações persistentes é a precisão de fluxo de dados, também denominada granularidade, que é adotada para a cobertura dessas associações. A granularidade impacta o teste, visto que demanda dados de teste

específicos para cada precisão de fluxo de dados. Foram compiladas as definições das granularidades encontradas na literatura: *relação*, *tupla*, *atributo* e *valor de atributo*.

Foi delineado o modelo de interação entre regras escritas em SQL adotado por esta pesquisa. O modelo adotado – política de ciclo recursiva e modos de acoplamento imediatos para evento-condição e condição-ação – determina que o disparo entre regras ocasiona a imediata transferência de controle entre regras, estabelecendo novas oportunidades de fluxo de dados persistentes. Foram tratadas as diversas formas de interação entre regras, onde os componentes de uma regra interagem com os componentes de outra regra, ou da mesma regra, tal que são estabelecidas relações de fluxo de dados entre componentes de regras não necessariamente distintas.

Associações de interação foram introduzidas, explorando o fluxo de dados persistentes entre regras. As associações são derivadas do disparo entre regras e da manipulação de entidades comuns da base de dados por duas regras não necessariamente distintas. Em adição, regras são executadas em uma certa ordem, sem que ocorra disparo entre regras, obedecendo-se uma seqüência estabelecida pela definição e uso persistentes. As associações *associação-ddef-duso-AC* e *associação-ddef-duso-AA* não prevêem o disparo de regras em sua cobertura; as associações *associação-ddef-duso-AE*, *associação-ddef-duso-AC de chamada*, *associação-ddef-duso-AA de chamada* e *associação-ddef-duso-AA de retorno* utilizam o disparo entre regras e conseqüente transferência de controle, dando oportunidade a novas relações de fluxo de dados persistentes resultantes da concatenação entre subcaminhos das regras envolvidas.

Um conjunto de critérios de adequação foi proposto – Critérios Baseados na Interação entre Regras –, visando à descoberta de defeitos que porventura não tenham sido revelados no teste individual de regras. A essência dos critérios é explorar as relações de fluxo de dados persistentes existentes na interação entre regras, representando uma extensão ao critério *todos usos*. Especificamente, seus requisitos buscam a cobertura das associações de interação abstraídas para um conjunto de regras. A análise de inclusão entre os critérios propostos e alguns dos critérios baseados em fluxo de dados, incluindo aqueles focados em aplicações de banco de dados, demonstrou que eles são incomparáveis.

Mostrou-se também que o par *(critério, granularidade)* determina um novo requisito de teste, pois existe uma relação de inclusão entre os domínios de entrada próprios para cobertura de um particular critério em cada granularidade. Essa relação de inclusão é válida mesmo na

presença de não executabilidade. O requisito *(critério, granularidade)* afeta a complexidade dos critérios propostos: *(i)* a redefinição de dados persistentes é sensível aos dados manipulados em cada ocorrência de manipulação; e *(ii)* as estruturas de fluxo de controle não indexam o cenário de pior caso. Foi observado que os Critérios Baseados na Interação entre Regras possuem complexidade polinomial.

A partir de um experimento aplicado, observou-se que as precisões de fluxo de dados não possuem papel suplementar na descoberta de defeitos. Foram identificados cenários reveladores e não reveladores de defeito para cada precisão, nos quais a descoberta de defeitos ocorre de forma não sistemática em qualquer das granularidades: casos de teste reveladores de defeito estão relacionados à cobertura de associações de interação nas diversas precisões de fluxo de dados. A eficácia da cobertura de associações de interação alcançou 2/3 do conjunto adequado, obtendo índices de eficácia ainda mais favoráveis na direção de granularidades menos precisas. Este estudo empírico apresentou índices propícios ao apoio dos critérios propostos à descoberta de defeitos na interação entre regras. Apesar dessa tendência, é pertinente a realização de novos estudos empíricos, visto que vários fatores estão envolvidos nos resultados do estudo, tais como: técnica adotada para a geração de dados de teste; regras ativas utilizadas no experimento; dados de entrada (dados de teste) gerados a partir da técnica adotada; defeitos e tipos de defeito nas regras ativas.

Foram apresentados aspectos de implementação para os critérios propostos, pela introdução dos modelos de fluxo de controle, de fluxo de dados, de instrumentação e de avaliação. A abordagem para a instrumentação de fluxo de dados visa à análise da cobertura de associações de interação em diversas precisões de fluxo de dados. Os fluxos de controle e de dados foram instrumentados de forma separada, cada qual possuindo comandos de escrita próprios. Exceções na execução de comandos de manipulação inspiraram uma abordagem dedicada, dando-se atenção ao aninhamento de blocos, ao contexto de sua ocorrência e à prevenção para que os registros de instrumentação não sejam perdidos.

Uma ferramenta foi construída para automatizar a fase dinâmica atribuída à aplicação dos critérios propostos, denominada ADAPT-TOOL, incluindo funções tais como: geração de dados de teste; registro dos casos de teste aplicados, aplicação e re-aplicação de casos de teste; avaliação da cobertura de associações de interação por granularidade; e decisão se a presença de defeitos foi revelada. A ferramenta manipula uma base de dados de teste, a qual utiliza as

estruturas de dados do modelo relacional, habilitando o uso direto da SQL no desenvolvimento de suas funções, constituindo-se, em si, uma aplicação de banco de dados.

Conclui-se que os Critérios Baseados na Interação entre Regras alcançam um contexto ainda não tratado por outras abordagens, e constituem a primeira iniciativa que explora a interação entre regras escritas em SQL para o teste baseado em fluxo de dados persistentes.

7.2 Contribuições

Este trabalho introduziu uma abordagem pioneira ao teste de regras ativas escritas em SQL, representando um recurso valioso à melhoria de qualidade de aplicações de bancos de dados ativos. Sua análise empírica demonstrou uma eficácia promissora à descoberta de defeitos e sua complexidade é polinomial. A abordagem pode encorajar o emprego de teste sistemático no contexto em que se insere.

As principais contribuições deste trabalho, descritas na seção anterior, são sintetizadas abaixo:

1. Categorização de iniciativas de pesquisa sobre teste de bases de dados, realçando a carência de trabalhos metódicos focados na qualidade de aplicações de banco de dados (Seções 2.4 e 2.5).
2. Construção de um mapeamento entre defeitos e falhas de manipulação, pela investigação empírica do relacionamento entre tipos de defeito e dimensões de falha enumeradas para comandos de manipulação da SQL (Seções 3.1, 3.2 e 3.3).
3. Abstração empírica de propriedades potencialmente reveladoras de defeito, no que diz respeito a bases de dados de entrada, abrindo uma linha de investigação para a seleção de dados de teste (Seção 3.4).
4. Composição de um modelo para a interação de regras escritas em SQL e definição de associações de fluxo de dados persistentes que exploram tal modelo, denominadas de associações de interação, em cujas coberturas podem basear-se requisitos de teste (Seções 4.3 e 4.4).

5. Definição de um conjunto de critérios de adequação – Critérios Baseados na Interação entre Regras Ativas – que apóia a busca pela qualidade de conjuntos de regras escritas em SQL, inserindo uma estratégia sistemática ao teste pela cobertura de associações de interação (Seções 5.1 e 5.2).
6. Proposição de modelos de implementação visando à análise de cobertura de associações de interação, considerando aspectos de fluxo de controle, de fluxo de dados, de instrumentação e de avaliação, focados na granularidade de fluxo de dados para as precisões *relação*, *atributo*, *tupla* e *valor de atributo* (Seção 6.1).
7. Desenvolvimento de uma ferramenta, denominada ADAPT-TOOL, para a aplicação dos Critérios Baseados na Interação entre Regras Ativas, automatizando funções dinâmicas da atividade de teste estrutural, desde a geração de dados de teste até o oráculo de teste e a avaliação de cobertura de requisitos por precisão de fluxo de dados (Seção 6.2 e Apêndice C).
8. Investigação empírica de requisitos do tipo *(critério, granularidade)* no contexto de associações de interação, sobre sua habilidade suplementar à detecção de defeitos (Seção 5.3).
9. Experimentação dos Critérios Baseados na Interação entre Regras Ativas, constituindo-se em uma estratégia para avaliar empiricamente conjuntos de regras escritas em SQL (Seção 5.3).

7.3 Trabalhos Futuros

Alguns trabalhos podem ser conduzidos para agregar valor às contribuições desta pesquisa, como a seguir:

- Estender o modelo de interação entre regras descrito no Capítulo 4 e, conseqüentemente, os critérios de teste propostos, a associações que envolvam mais de duas regras, buscando alcançar dependências de fluxo de dados persistentes que envolvam o disparo em cadeia entre regras.
- Conduzir experimentos para avaliar se as propriedades *diversidade*, *nulidade* e *cardinalidade*, tratadas no Capítulo 3, representam recursos reveladores de defeitos relacionados a bases de dados. Propor medidas eficazes buscando investigar se existem

evidências quantitativas da presença dessas propriedades à propensão à descoberta de defeitos.

- Experimentar conjuntos adicionais de regras ativas, buscando ensaiar amostras reais de defeitos e reduzir suas ameaças à validade. Contrapor os novos resultados aos já obtidos para confirmar ou refutar suas conclusões, o que também poderia contribuir à argumentação para a introdução de outras pesquisas sobre o teste de aplicações de bancos de dados ativos.
- Utilizar amostras reais de defeitos para comparar empiricamente o teste individual de regras com o teste baseado na interação entre regras. Investigar se os critérios propostos revelam os defeitos que normalmente são descobertos pela aplicação dos critérios Potenciais Usos (Maldonado, 1991) ao teste individual de regras escritas em SQL (Cardoso, 2004).
- Estender a ferramenta ADAPT-TOOL pela incorporação de funções próprias da fase estática, tais como instrumentação de código fonte e abstração de elementos requeridos pelos Critérios Baseados na Interação entre Regras Ativas.
- Modelar máquinas de estado para a definição e uso persistentes atribuídos a regras ativas. Investigar se máquinas de estados finitos são eficazes para formalizar a interação entre regras no contexto de teste de regras escritas em SQL.
- Aplicar técnicas de teste baseadas em defeitos a regras ativas escritas em SQL. Propor e investigar empiricamente operadores de mutação para interface de regra (evento de regra e condição de regra), para comandos de manipulação de dados persistentes e para as rotinas de tratamento de exceção.

Referências Bibliográficas

B. **Adelberg**, B. Kao and H. Garcia-Molina. Database Support for Efficiently Maintaining Derived Data. *Proc. of the 5th International Conference on Extending Database Technology (EDBT'96)*, Avignon, France, March 1996.

A. **Aiken**, J. M. Hellerstein and J. Widom. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM Transactions on Database Systems*, 20(1): 3-41, March 1995.

P. **Ammann** and A. J. Outt. Using Formal Methods to Derive Test Frames in Category-partition Testing. *Proc. of the 9th Annual Conference on Computer Assurance (COMPASS 94)*, pages 69-80, Gaithersburg, MD, June 1994.

M. C. L. F. M. **Aranha**, N. C. Mendes, M. Jino and C. M. T. Toledo. RDBTool: Uma Ferramenta de Apoio ao Teste de Bases de Dados Relacionais, *XI CITS: Conferência Internacional de Tecnologia de Software*, Curitiba, Paraná, Agosto 2000.

M. C. L. F. M. **Aranha**. Sistematização e Uso de Critérios de Teste em Bancos de Dados Relacionais. Tese de doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, 2005, em andamento.

E. **Baralis**. Rule Analysis. Active Rules in Database Systems, Norman W. Paton, editor, pages 51-67, Springer-Verlag, 1998.

E. **Baralis**, S. Ceri and S. Paraboschi. Compile-Time and Runtime Analysis of Active Behaviors. *IEEE Transactions on Knowledge and Data Engineering*, 10(3): 353-370, May 1998.

V. R. **Basili** and B. T. Perricone. Software Errors and Complexity: an Empirical Study. *Communications of the ACM*, 27(1): 42-52, January 1984.

B. **Beizer**. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

J. **Bowman**, Sandra Emerson, and Marcy Damovsky. *The Practical SQL Handbook*. Addison-Wesley, 1997.

V. M. **Cardoso**. Uma Ferramenta para Teste Estrutural de Regras Ativas. Dissertação de mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Fevereiro 2004.

M. J. **Carey**, D. J. DeWitt and J. F. Naughton. The 007 Benchmark. *Proc. of the 1993 ACM SIGMOD Intl. Conf. On Management of Data*, pages 12-21, ACM Press, New York, NY, May 1993.

S. **Ceri** and J. Widow. Deriving Production Rules for Constraint Maintenance. *Proc of the 16th Very Large Database (VLDB) Conference*, pages 566-577, Brisbane, Australia, August 1990.

S. **Ceri** and J. Widow. Deriving Production Rules for Incremental View Maintenance. *Proc of the 17th Very Large Database (VLDB) Conference*, pages 577-589, Barcelona, Spain, September 1991.

S. **Ceri** and J. Widow. Managing Semantic Heterogeneity with Production Rules and Persistent Queries. *Proc of the 19th Very Large Database (VLDB) Conference*, pages 108-119, Dublin, Ireland, August 1993.

S. **Ceri**, P. Fraternali, S. Paraboschi and L. Tanca. Automatic Generation of Production Rules for Integrity Maintenance. *ACM Transactions on Database Systems*, 19(3): 367-422, September 1994.

S. **Ceri**, P. Fraternali, S. Paraboschi and L. Tanca. Active Rule Management in Chimera. In Jennifer Widom and Stefano Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, pages 151-175, Morgan Kaufmann Publishers, 1996.

S. **Ceri** and J. Widow. Applications of Active Databases. In Jennifer Widom and Stefano Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, pages 259-291, Morgan Kaufmann Publishers, 1996.

S. **Ceri**, R. Cochrane and J. Widom. Practical Applications of Triggers and Constraints: Success and Lingering Issues. *Proc of the 26th Very Large Database (VLDB) Conference*, pages 254-262, Cairo, Egypt, 2000.

M. L. **Chaim**. 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 e de Computação, UNICAMP, Abril 1991.

M. L. **Chaim**. Depuração de Programas Baseada em Informação de teste Estrutural. Tese de doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Novembro 2001.

S. **Chakravarthy**. A Comparative Evaluation of Active Relational Databases. Technical report UF-CIS-TR-93-002. Computer and Information Sciences, University of Florida, 1993.

H. W. R. **Chan**, S. W. Dietrich and S. U. Urban. On Control Flow Testing of Active Rules in a Declarative Object-Oriented Framework. *Proc. of the 3rd Intl. Workshop on Rules in Database Systems (RIDS'97)*, pages 165-180, Skoevde, Sweden, June 1997.

M. **Chan** and S. Cheung. Testing Database Applications with SQL Semantics. *Proc. of the 2nd Intl. Symp. on Cooperative Database Systems for Advanced Applications (CODAS'99)*, pages 364-375, March 1999.

R. **Chandra** and A. Segev. Active Databases for Financial Applications. *Proc. of the 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS 94)*, Houston, February 1994.

D. **Chays**, S. Dan, P. G. Frankl, F. I. Vokolos and E. J. Weyuker. A Framework for Testing Database Applications. *Proc. of the 2000 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis*, pages 147-157, Portland, Oregon, 2000.

D. **Chays**, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos and E. J. Weyuker. AGENDA: A Test Generator for Relational Database Applications. Technical report, Department of Computer Science, Polytechnic University, Brooklyn, Long Island, Westchester, August 2002.

D. **Chays** and Y. Deng. Demonstration of AGENDA Tool Set for Testing Relational Database Applications. *Proc. of the 25th Intl. Conference on Software Engineering*, Portland, Oregon, 2003.

P. P. **Chen**, The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1): 9-36, March 1976.

R. **Chillarege**, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray and M. Wong. Orthogonal Defect Classification – A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, 18(11): 943-956, November 1992.

L. A. **Clarke**, A. Podgurski, D. J. Richardson, and S. J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, 15(11): 1318-1332, November 1989.

E. F. **Codd**. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6): 377-387, June 1970.

P. B. **Crosby**. *Quality is Free*. McGraw Hill, 1979.

C. J. **Date** and H. Darwen. *A Guide to The SQL Standard*. Addison-Wesley, 1997.

B. **Daou**, R. A. Haraty and N. Mansour. Regression Testing of Database Applications. *Proc. of the 2001 ACM Symposium on Applied Computing*, Las Vegas, Nevada, 2001.

R. A. **Davies**, R. J. A. Beynon and B. F. Jones. Automating the Testing of Databases. *Proc. of the 1st Intl. Workshop on Automated Program Analysis, Testing and Verification*, June 2000.

U. **Dayal**. Active Database Management Systems. *Proc. of the 3rd Intl. Conf. On Data and Knowledge Bases*, pages 150-169, Jerusalem, Israel, June 1988.

DB2 Information Management Software. Página na Internet, IBM Products, 2005. `http://www-306.ibm.com/software/data`.

M. E. **Delamaro**, J. C. Maldonado and ª P. Mathur. Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering*, 27(3): 228 – 247, March 2001.

R. A. **DeMillo**, R. J. Lipton and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4): 34-43, 1978.

M. S. **Deutsch**. *Software Verification and Validation*. Prentice-Hall, 1982.

K. R. **Dittrich** and D. Jonscher. Current Trends in Database Technology and Their Impact on Security Concepts. In J. Biskup, M. Morgenstern, and C.E. Landwehr, editors, *Database Security, VIII (A-60)*, Elsevier Science B.V. (North-Holland), pages 11-33, 1994.

R. **Elmasri** and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2003.

S. M. **Embury** and P. M. D. Gray. Database Internal Applications. Active Rules in Database Systems, Norman W. Paton, editor, pages 339-366, Springer-Verlag, 1998.

P. J. **Fortier**. *Implementing the SQL Foundation Standard*. McGraw-Hill, 1999.

P. G. **Frankl** and E. J. Weyuker. Data Flow Testing in the Presence of Unexecutable Paths. *Workshop on Software Testing*, pages 4-13, Banff, Canada, July1986.

P. G. **Frankl** and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10): 1483-1498, October 1988.

D. **Galin**. *Software Quality Assurance*. Addison Wesley, 2004.

S. **Gatziu**, A. Geppert and K. R. Dittrich. The SAMOS Active DBMS Prototype. *Proc. ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

J. **Goodenough** and S. L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2): 156-173, June 1975.

J. **Hammer**, H. Garcia-Molina, J. Widow, W. Labio and Y. Zhuge. The Stanford Data Warehousing Project. *Data Engineering Bulletin*, 18(2): 40-47, June 1995.

E. N. **Hanson**. The Design and Implementation of the Ariel Active Database Rule System. *IEEE Transactions On Knowledge and Data Engineering*, (8)1: 157-172, February 1996.

M. J. **Harrold** and M. L. Soffa. Selecting and Using Data for Integration Testing. *IEEE Software*, 8(2): 58-65, March 1991.

A. **Hartman**. Is ISSTA Research Relevant to Industry? *Intl. Symposium on Software Testing and Analysis*, Industry Panel, ACM SIGSOFT Software Engineering Notes, 2002.

P. M. **Herman**. A Data Flow Analysis Approach to Program Testing. *The Australian Computer Journal*, 8(3): 92-96, November 1976.

J. R. **Horgan** and S. London. Data Flow Coverage and the C Language. *Proc. of the Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 87-97, Victoria, B.C., Canada, October 1991.

W. E. **Howden**. Methodology for the Generation of Program Test Data. *IEEE Transactions on Computer*, 24(5): 554-559, 1975.

W. E. **Howden**. *Functional Program Testing and Analysis*. McGraw-Hill, 1987.

Informix Database Server. Página na Internet, IBM Products, 2005. `http://www-306.ibm.com/software/data/informix`.

G. W. **Jones**. *Software Engineering*. John Wiley & Sons, 1990.

D. **Jonscher**. Extending Access Control with Duties Realised by Active Mechanism, *Proc. of the 6th Working Conference on Database Security*, Vancouver, August 1992.

G. M. **Kapfhammer** and M. L. Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2003*, Helsinki, Finland, September 2003.

D. **Kelly** and T. Shepard. A Case Study in the Use of Defect Classification in Inspections. *Proc. of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, November 2001.

B. A. **Kitchenham**, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8): 721-733, 2002.

G. **Knolmayer**, H. Herbst and M. Schlesinger. Enforcing Business Rules by the Application of Trigger Concepts. *Proc. of the Priority Programme Informatics Research, Information Conference*, Swiss National Science Foundation, November 1994.

B. **Korel** and J. Laski. A Tool for Data Flow Oriented Program Testing. *Proc. of the 2nd ACM Conference on Software Development Tools, Techniques, and Alternatives*, 34-37, December 1985.

A. **Kotz-Dittrich** and E. Simon. Active Database Systems: Expectations, Commercial Experience, and Beyond. Active Rules in Database Systems, Norman W. Paton, editor, pages 367-404, Springer-Verlag, 1998.

K. **Kulkarni**, N. Mattos and R. Cochrane. Active Database Features In SQL3. Active Rules in Database Systems, Norman W. Paton, editor, pages 197-219, Springer-Verlag, 1998.

J. **Laski** and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, 9(3): 347-354, May 1983.

P. S. J. **Leitão**, V. M. Cardoso and M. Jino. Aspectos de Fluxo de Controle, de Fluxo de Dados e de Instrumentação no Contexto de Banco de Dados Ativos. Relatório Técnico, Universidade Estadual de Campinas, UNICAMP, 2002.

P. S. J. **Leitão**, P. R. S. Vilela and M. Jino. Mapping Faults to Failures in SQL Manipulation Commands. *Proc. of the 3rd ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-05)*, Egypt, Cairo, January 2005.

H. K. N. **Leung** and L. White. A Study of Integration Testing and Software Regression at the Integration Level. *Proc. of the International Conference on Software Maintenance*, pages 290-300, November 1990.

N. R. **Lyons**. An Automatic Data Generating System for Data Base Simulation and Testing. *Database*, 8(4):10-13, 1977.

P. **Madiraju** and R. Sunderraman. A Mobile Agent Approach for Global Database Constraint Checking. *ACM Symposium on Applied Computing (SAC'04)*, pages 679–683, Nicosia, Cyprus, 2004.

J. C. **Maldonado**. Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. Tese de doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Julho 1991.

J. C. **Maldonado**, M. L. Chaim and M. Jino. Potential Uses Testing Criteria: a Step Towards Bridging the Gap in the Presence of Infeasible Paths. *Proc. of the 12th Intl. Conference of the Chilean Computer Society*, Santiago, Chile, 1992.

J. C. **Maldonado** and S. C. P. F. Fabbri. Verificação e Validação de Software. In A. R. C. Rocha, J. C. Maldonado and K. C. Weber, editors, *Qualidade de Software: Teoria e Prática*, pages 66-73, Prentice-Hall, São Paulo, 2001.

H. **Mannila** and K. Räihä. Small Amstrong Relations for Database Design. *Proc. of the 4th ACM SIGACT-SIGMOD Conf. on Principles of Database Systems*, pages 145-250, March 1985.

H. **Mannila** and K. Räihä. Automatic Generation of Test Data for Relational Queries. *Journal of Computer and System Sciences*, 38(2):240-258, 1989.

M. **Marré** and A. Bertolino. Unconstrained Duas and Their Use in Achieving All-uses Coverage. *Proc. Of the Intl. Symposium on Software Testing and Analysis*, pp. 147–157, January 1996.

D. I. S. **Marx** and P. G. Frankl. The Path-wise Approach to Data Flow Testing with Pointer Variables. *Proc. of the Intl. Symposium on Software Testing and Analysis*, pp. 135-146, San Diego, New York, 1996.

J. **Melton** and A. R. Simon. *Understanding the New SQL*. Morgan Kaufmann, 1993.

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

H. **Noble**. The Automatic Generation of Test Data for a Relational Database. *Information Systems*, 8(2):79-86, 1983.

S. C. **Ntafos**. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, (14)6: 868-874, June 1988.

Oracle Documentation. Página na Internet, Oracle Tecnology Network, 2005. `http://www.oracle.com/technology/documentation/index.html`.

V. **Okun**, P. E. Black and Y. Yesha. Testing with Model Checker: Insuring Fault Visibility. *Proc. 2002 WSEAS International Conference on System Science*, Applied Mathematics and Computer Science, Copacabana, Rio de Janeiro, October 2002.

T. J. **Ostrand** and M. J. Balcer. The Category-partition Method for Specifying and Generating Functional Testing, *Communications of the ACM*, 31(6): 676-686, June 1988.

T. J. **Ostrand** and E. J. Weyuker. Data Flow-based Testing Adequacy Analysis for Languages with Pointers. *Proc. of the 4th Symposium on Software Testing, Analysis and Verification*, pages 74-86, New York, October 1991.

N. W. **Paton** and O. Díaz. Active Database Systems. *ACM Computing Surveys*, 31(1): 63-103, March 1999.

R. S. **Pressman**. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, *6th* edition, 2004.

R. **Ramakrishnan**. *Database Management Systems*, McGraw-Hill, 1998.

S. **Rapps** and E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367-375, April 1985.

M. A. **Robbert** and Fred J. Maryanski. Automated Test Plan Generator for Database Application Systems. *Proc. of the ACM SIGSAMLL/PC Symposium on Small Systems*, pages 100-106, Toronto, Ontario, 1991.

D. **Slutz**. Massive Stochastic Testing of SQL. *Proc. 24th Int. Conf. Very Large Data Bases (VLDB)*, pages 618-622, 1998.

I. **Sommerville**. *Software Engineering*. Addison Wesley, 2001.

E. S. **Spoto**, M. Jino and J. C. Maldonado. Teste Estrutural de Software: Uma Abordagem para Aplicações de Banco de Dados Relacional. *XIV SBES: Simpósio Brasileiro de Engenharia de Software*, João Pessoa, PB, Outubro 2000.

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

E. S. **Spoto**, P. S. J. Leitão, M. Jino and J. C. Maldonado. Teste Estrutural de Integração de Programas de Aplicação de Banco de Dados Relacional. *IV Simpósio Brasileiro de Qualidade de Software (SBQS 2005)*, Porto Alegre, Junho 2005.

P. **Stocks** and D. Carrington. A Framework for Specification-based Testing. *IEEE Transactions on Software Engineering*, 22(11): 777-793, November 1996.

M. **Stonebraker**, A. Jhingran, J. Goh and S. Potamianos. On Rules, Procedures, Caching and Views in Database Systems. *Proc. of the ACM SIGMOD Intl. Conf. on Management of data*, Atrantic City, USA, May 1990.

M. **Stonebraker** and G. Kemnitz. The POSTGRES Next-generation Database Management System. *Communications of the ACM*, 34(10): 78-92, October 1991.

SQL Server. Página na Internet, Microsoft SQL Server, 2005. `http://www.microsoft.com/sql/default.mspx`.

M. **Stonebraker** and L. A. Rowe. The Design of Postgres. *Proc. ACM-SIGMOD International Conference on the Management of Data*, ACM Press, Washington D. C., June 1986.

M. J. **Suárez-Cabal** and J. Tuya. Using a SQL Coverage Measurement for Testing Database Applications. *Proc. of the 12th Intl. Symp. on the Foundations of Software Engineering*, Newport Beach, California, November 2004.

TPC. Transaction Processing Performance Council. TPC Benchmark C Standard Specification Revision 5.4. April 2005. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`.

J. D. **Ullman** and J. Widow. *A First Course in Database System*. Prentice Hall, 1997.

S. D. **Urban**, A. P. Karadimce, S. W. Dietrich, T. B. Abdellatif and H. W. R. Chan. CDOL: A Comprehensive Declarative Object Language. *Data & Knowledge Engineering*, 22(1): 67-111, March 1997.

A. **Vaduva**. Rule Development for Active Database Systems. Ph.D. Thesis, University of Zurich, April 1999.

P. R. S. **Vilela**, J. C. Maldonado and M. Jino. Data Flow Based Testing of Programs with Pointers: A Strategy based on Potential Uses. *Proc. of the 10th Intl. Software Quality Week*, San Francisco, CA, May 1997.

P. R. S. **Vilela**. Critérios Potenciais Usos de Integração: Definição e Análise. Tese de doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Abril 1998.

J. **Widow**. The Starburst Active Database Rule System. *IEEE Transactions On Knowledge and Data Engineering*, (8)4: 583-595, August 1996.

J. **Widom** and S. Ceri, Introduction to Active Database Systems. In Jennifer Widom and Stefano Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, pages 1-41, Morgan Kaufmann Publishers, 1996.

M. R. **Woodward**, D. Heddley and M. A. Hennel. Experience with Path Analysis and Testing of Programs. *IEEE Transactions on Software Engineering*, 6(3): 278 - 286, May 1980.

C. **Zaniolo**, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian and R. Zicari. *Advanced Database System*, Morgan Kaufmann Publishers, 1997.

J. **Zhang**. Specification Analysis and Test Data Generation by Solving Boolean Combinations of Numeric Constraints. *Proc. of the 1st Asia-Pacific Conference on Quality Software*, *(APAQS)*, pages 267-274, October 2000.

J. **Zhang**, C. Xu and S. Cheung. Automatic Generation of Database Instances for White-box Testing. *Proc. of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, Chicago, Illinois, 2001.

P. **Zoio**. Testing 1, 2, 3. *Oracle Magazine*, 19(4): 49-52, July/August 2005.

Apêndice A

Instrumentação de um Conjunto de Regras Ativas Escritas em SQL

Este apêndice exibe a versão instrumentada de um conjunto de regras ativas escritas em SQL, o qual foi utilizado no experimento descrito no Capítulo 5. As regras, identificadas por *TRG_COMMIT_ITEMS*, *TRG_INVENTORY_QUANTITY*, *TRG_PRODUCT_PENDING*, *TRG_PRODUCT_QUANTITY*, manipulam dados de um banco de dados cujo esquema conceitual é apresentado no Apêndice B. Os códigos fonte são apresentados na forma de comandos de definição de dados, tal como *create trigger*; as linhas iniciadas pelos prefixos */*node*/*, */*data*/* e */*----*/* referem-se, respectivamente, à instrumentação de fluxo de controle, à instrumentação de fluxo de dados e à instrumentação de exceções ocorridas.

```
CREATE OR REPLACE TRIGGER TRG_COMMIT_ITEMS
    AFTER INSERT OR DELETE ON ORDER_ITEMS_COMMIT
    FOR EACH ROW
DECLARE  v_quantity             ORDER_ITEMS.QUANTITY%TYPE;
         v_quantity_commit      ORDER_ITEMS.QUANTITY%TYPE;
         v_quantity_to_apply    ORDER_ITEMS.QUANTITY%TYPE;
         v_order_id_to_apply    ORDER_ITEMS.ORDER_ID%TYPE;
         v_product_id_to_apply  ORDER_ITEMS.ORDER_ID%TYPE;
/*node*/ t_v_case_id            T_NODES.CASE_ID%TYPE;
/*node*/ t_v_line_id            T_NODES.LINE_ID%TYPE;
/*node*/ t_v_rule_id            T_NODES.RULE_ID%TYPE;
BEGIN
/*node*/ t_v_rule_id := 'TRG_COMMIT_ITEMS';
/*node*/ t_v_case_id := t_f_getcase;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 1, 'EVENT NODE');
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 2, 'FIRST ACTION NODE');
         IF INSERTING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 3, 'SELECT COMMAND');
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'ORDER_ITEMS',
/*data*/              CONCAT(ORDER_ID, PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY;QUANTITY_COMMIT;'
/*data*/       FROM   ORDER_ITEMS
/*data*/       WHERE  ORDER_ID   = :NEW.ORDER_ID
/*data*/       AND    PRODUCT_ID = :NEW.PRODUCT_ID;
/*----*/    begin
```

```
            SELECT SUM(QUANTITY), SUM(QUANTITY_COMMIT) INTO v_quantity, v_quantity_commit
            FROM   ORDER_ITEMS
            WHERE  ORDER_ID   = :NEW.ORDER_ID
            AND    PRODUCT_ID = :NEW.PRODUCT_ID;
/*----*/    exception
/*----*/    when others then
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'SELECT');
/*node*/         t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, 'EXIT NODE - ERROR -
SELECT');
/*----*/       return;
/*----*/    end;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 4, NULL);
            v_order_id_to_apply    := :NEW.ORDER_ID;
            v_product_id_to_apply  := :NEW.PRODUCT_ID;
            IF (:NEW.QUANTITY_COMMIT > (NVL(v_quantity, 0) - NVL(v_quantity_commit, 0))) THEN
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 5, 'RAISE COMMAND');
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, 'EXIT NODE - RAISE');
/*----*/       return;
               RAISE_APPLICATION_ERROR (-20011,'[TRIGGER TRG_COMMIT_ITEMS] Error inserting commit
items');
            ELSE
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 6, NULL);
               v_quantity_to_apply := :NEW.QUANTITY_COMMIT;
            END IF;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 7, NULL);
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 8, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 9, NULL);
         IF DELETING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 10, 'SELECT COMMAND');
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'ORDER_ITEMS',
/*data*/              CONCAT(ORDER_ID, PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY_COMMIT;'
/*data*/       FROM   ORDER_ITEMS
/*data*/       WHERE  ORDER_ID   = :OLD.ORDER_ID
/*data*/       AND    PRODUCT_ID = :OLD.PRODUCT_ID;
/*----*/    begin
            SELECT SUM(QUANTITY_COMMIT) INTO v_quantity_commit
            FROM   ORDER_ITEMS
            WHERE  ORDER_ID   = :OLD.ORDER_ID
            AND    PRODUCT_ID = :OLD.PRODUCT_ID;
/*----*/    exception
/*----*/    when others then
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'SELECT');
/*node*/         t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, 'EXIT NODE - ERROR -
SELECT');
/*----*/       return;
/*----*/    end;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 11, NULL);
            v_order_id_to_apply    := :OLD.ORDER_ID;
            v_product_id_to_apply  := :OLD.PRODUCT_ID;
            IF (v_quantity_commit IS NULL) OR
               (:OLD.QUANTITY_COMMIT > v_quantity_commit) THEN
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 12, 'RAISE COMMAND');
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, 'EXIT NODE - RAISE');
/*----*/       return;
               RAISE_APPLICATION_ERROR  (-20012,'[TRIGGER  COMMIT_ITEMS]  Error  deleting  commit
items');
            ELSE
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 13, NULL);
               v_quantity_to_apply := :OLD.QUANTITY_COMMIT * (-1);
            END IF;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 14, NULL);
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 15, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 16, 'UPDATE COMMAND');
```

```
/*data*/ INSERT INTO T_DATA_WORK
/*data*/    ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/    SELECT t_v_case_id, t_v_line_id, 'ORDER_ITEMS',
/*data*/           CONCAT(ORDER_ID, PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY_COMMIT;'
/*data*/    FROM   ORDER_ITEMS
/*data*/    WHERE  ORDER_ID   = v_order_id_to_apply
/*data*/    AND    PRODUCT_ID = v_product_id_to_apply;
/*----*/ begin
         UPDATE ORDER_ITEMS
         SET    QUANTITY_COMMIT = QUANTITY_COMMIT + v_quantity_to_apply
         WHERE  ORDER_ID   = v_order_id_to_apply
         AND    PRODUCT_ID = v_product_id_to_apply;
/*----*/ exception
/*----*/ when others then
/*----*/    t_p_setcaseexception (t_v_case_id, t_v_line_id, 'UPDATE');
/*node*/      t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, 'EXIT NODE - ERROR -
UPDATE');
/*----*/    return;
/*----*/ end;
/*data*/ INSERT INTO T_DATA_WORK
/*data*/    ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/    SELECT t_v_case_id, t_v_line_id, 'ORDER_ITEMS',
/*data*/           CONCAT(ORDER_ID, PRODUCT_ID), 1, NULL, 'QUANTITY_COMMIT;'
/*data*/    FROM   ORDER_ITEMS
/*data*/    WHERE  ORDER_ID   = v_order_id_to_apply
/*data*/    AND    PRODUCT_ID = v_product_id_to_apply;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, 'EXIT NODE');
END TRG_COMMIT_ITEMS;



CREATE OR REPLACE TRIGGER TRG_INVENTORY_QUANTITY
    AFTER INSERT OR DELETE OR UPDATE OF QUANTITY_ON_HAND ON INVENTORIES
    FOR EACH ROW
DECLARE
/*node*/    t_v_case_id         T_NODES.CASE_ID%TYPE;
/*node*/    t_v_line_id         T_NODES.LINE_ID%TYPE;
/*node*/    t_v_rule_id         T_NODES.RULE_ID%TYPE;
BEGIN
/*node*/ t_v_rule_id := 'TRG_INVENTORY_QUANTITY';
/*node*/ t_v_case_id := t_f_getcase;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 1, 'EVENT NODE');
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 2, 'FIRST ACTION NODE');
         IF (:OLD.PRODUCT_ID <> :NEW.PRODUCT_ID) THEN
/*node*/   t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 3, 'RAISE COMMAND');
/*----*/   t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/   t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 11, 'EXIT NODE - RAISE');
/*----*/   return;
           RAISE_APPLICATION_ERROR     (-20031,'[TRG     INVENTORY_QUANTITY]     Error     updating
inventories');
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 4, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 5, NULL);
         IF DELETING OR UPDATING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 6, 'UPDATE COMMAND');
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/              TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY_ON_HAND;'
/*data*/       FROM   PRODUCT_STATUS
/*data*/       WHERE  PRODUCT_ID = :OLD.PRODUCT_ID;
/*----*/    begin
           UPDATE PRODUCT_STATUS
           SET QUANTITY_ON_HAND  = NVL(QUANTITY_ON_HAND, 0) - NVL(:OLD.QUANTITY_ON_HAND, 0)
           WHERE PRODUCT_ID = :OLD.PRODUCT_ID;
/*----*/    exception
/*----*/    when others then
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'UPDATE');
```

```
/*node*/         t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 11, 'EXIT NODE - ERROR -
UPDATE');
/*----*/       return;
/*----*/    end;
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/              TO_CHAR(PRODUCT_ID), 1, NULL, 'QUANTITY_ON_HAND;'
/*data*/       FROM   PRODUCT_STATUS
/*data*/       WHERE  PRODUCT_ID = :OLD.PRODUCT_ID;
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 7, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 8, NULL);
         IF INSERTING OR UPDATING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 9, 'UPDATE COMMAND');
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/              TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY_ON_HAND;'
/*data*/       FROM   PRODUCT_STATUS
/*data*/       WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
/*----*/    begin
            UPDATE PRODUCT_STATUS
            SET QUANTITY_ON_HAND  = NVL(QUANTITY_ON_HAND, 0) + NVL(:NEW.QUANTITY_ON_HAND, 0)
            WHERE PRODUCT_ID = :NEW.PRODUCT_ID;
/*----*/    exception
/*----*/    when others then
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'UPDATE');
/*node*/         t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 11, 'EXIT NODE - ERROR -
UPDATE');
/*----*/       return;
/*----*/    end;
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/              TO_CHAR(PRODUCT_ID), 1, NULL, 'QUANTITY_ON_HAND;'
/*data*/       FROM   PRODUCT_STATUS
/*data*/       WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 10, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 11, 'EXIT NODE');
END TRG_INVENTORY_QUANTITY;



CREATE OR REPLACE TRIGGER TRG_PRODUCT_PENDING
    AFTER INSERT OR DELETE OR UPDATE ON PRODUCT_STATUS
    FOR EACH ROW
DECLARE v_exists INTEGER;
/*node*/    t_v_case_id         T_NODES.CASE_ID%TYPE;
/*node*/    t_v_line_id         T_NODES.LINE_ID%TYPE;
/*node*/    t_v_rule_id         T_NODES.RULE_ID%TYPE;
BEGIN
/*node*/ t_v_rule_id := 'TRG_PRODUCT_PENDING';
/*node*/ t_v_case_id := t_f_getcase;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 1, 'EVENT NODE');
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 2, 'FIRST ACTION NODE');
         IF UPDATING AND (:OLD.PRODUCT_ID <> :NEW.PRODUCT_ID) THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 3, 'RAISE COMMAND');
/*----*/    t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 23, 'EXIT NODE - RAISE');
/*----*/    return;
            RAISE_APPLICATION_ERROR   (-20041,'[TRG   PRODUCT_PENDING]   Error   updating   product
status');
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 4, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 5, NULL);
         IF DELETING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 6, 'DELETE COMMAND');
```

```
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/              TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;'
/*data*/       FROM   INVENTORY_PENDING
/*data*/       WHERE  PRODUCT_ID = :OLD.PRODUCT_ID;
/*data*/    INSERT INTO T_DATA_WORK
/*data*/       ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/       SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/              TO_CHAR(PRODUCT_ID), 1, NULL, 'T_KEY;DATE_PENDING;QUANTITY_PENDING;'
/*data*/       FROM   INVENTORY_PENDING
/*data*/       WHERE  PRODUCT_ID = :OLD.PRODUCT_ID;
/*----*/    begin
            DELETE FROM INVENTORY_PENDING
            WHERE PRODUCT_ID = :OLD.PRODUCT_ID;
/*----*/    exception
/*----*/    when others then
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'DELETE');
/*node*/         t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 23, 'EXIT NODE - ERROR -
DELETE');
/*----*/       return;
/*----*/    end;
          ELSE
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 7, 'BEGIN BLOCK');
            BEGIN
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 8, 'SELECT COMMAND');
/*data*/       INSERT INTO T_DATA_WORK
/*data*/          ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/          SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/                 TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;'
/*data*/          FROM   INVENTORY_PENDING
/*data*/          WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
               SELECT COUNT(*) INTO v_exists
               FROM INVENTORY_PENDING
               WHERE PRODUCT_ID = :NEW.PRODUCT_ID;
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 9, NULL);
               IF   NVL(:NEW.QUANTITY_ON_HAND,   0)   >=   (NVL(:NEW.QUANTITY_ORDERED,   0)   -
NVL(:NEW.QUANTITY_COMMIT, 0)) THEN
/*node*/          t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 10, NULL);
                  IF v_exists > 0 THEN
/*node*/             t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 11, 'DELETE COMMAND');
/*data*/             INSERT INTO T_DATA_WORK
/*data*/                ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/                SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/                       TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;'
/*data*/                FROM   INVENTORY_PENDING
/*data*/                WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
/*data*/             INSERT INTO T_DATA_WORK
/*data*/                ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/                SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/                                                   TO_CHAR(PRODUCT_ID),   1,   NULL,
'T_KEY;DATE_PENDING;QUANTITY_PENDING;'
/*data*/                FROM   INVENTORY_PENDING
/*data*/                WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
                     DELETE FROM INVENTORY_PENDING
                     WHERE PRODUCT_ID = :NEW.PRODUCT_ID;
                  END IF;
/*node*/          t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 12, NULL);
               ELSE
/*node*/          t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 13, NULL);
                  IF v_exists > 0 THEN
/*node*/             t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 14, 'UPDATE COMMAND');
/*data*/             INSERT INTO T_DATA_WORK
/*data*/                ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/                SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/                       TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;'
/*data*/                FROM   INVENTORY_PENDING
/*data*/                WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
                     UPDATE INVENTORY_PENDING
```

```
                    SET     QUANTITY_PENDING     =     NVL(:NEW.QUANTITY_ORDERED,     0)     -
NVL(:NEW.QUANTITY_COMMIT, 0) - NVL(:NEW.QUANTITY_ON_HAND, 0),
                        DATE_PENDING = (SELECT SYSDATE FROM DUAL)
                    WHERE PRODUCT_ID = :NEW.PRODUCT_ID;
/*data*/            INSERT INTO T_DATA_WORK
/*data*/               ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/               SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/                      TO_CHAR(PRODUCT_ID), 1, NULL, 'DATE_PENDING;QUANTITY_PENDING;'
/*data*/               FROM   INVENTORY_PENDING
/*data*/               WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
                 ELSE
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 15, 'INSERT COMMAND');
                    INSERT INTO INVENTORY_PENDING
                       (PRODUCT_ID, QUANTITY_PENDING)
                    VALUES
                       (:NEW.PRODUCT_ID,         NVL(:NEW.QUANTITY_ORDERED,         0)         -
NVL(:NEW.QUANTITY_COMMIT, 0) - NVL(:NEW.QUANTITY_ON_HAND, 0));
/*data*/            INSERT INTO T_DATA_WORK
/*data*/               ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/               SELECT t_v_case_id, t_v_line_id, 'INVENTORY_PENDING',
/*data*/                                                      TO_CHAR(PRODUCT_ID),   1,   NULL,
'T_KEY;DATE_PENDING;QUANTITY_PENDING;'
/*data*/               FROM   INVENTORY_PENDING
/*data*/               WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
                 END IF;
/*node*/          t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 16, NULL);
              END IF;
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, NULL);
            EXCEPTION
               WHEN OTHERS THEN
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 18, 'EXCEPTION');
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 19, 'OTHERS');
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 20, 'RAISE COMMAND');
/*----*/            t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/               t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 23, 'EXIT NODE -
RAISE');
/*node*/            return;
                    RAISE_APPLICATION_ERROR       (-20042,'[TRG       PRODUCT_PENDING]       Error
inserting/updating inventory pending: ' || sqlerrm);
            END;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 21, 'END BLOCK');
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 22, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 23, 'EXIT NODE');
END TRG_PRODUCT_PENDING;



CREATE OR REPLACE TRIGGER TRG_PRODUCT_QUANTITY
    AFTER INSERT OR UPDATE OR DELETE ON ORDER_ITEMS
    FOR EACH ROW
DECLARE
/*node*/    t_v_case_id         T_NODES.CASE_ID%TYPE;
/*node*/    t_v_line_id         T_NODES.LINE_ID%TYPE;
/*node*/    t_v_rule_id         T_NODES.RULE_ID%TYPE;
BEGIN
/*node*/ t_v_rule_id := 'TRG_PRODUCT_QUANTITY';
/*node*/ t_v_case_id := t_f_getcase;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 1, 'EVENT NODE');
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 2, 'FIRST ACTION NODE');
         IF DELETING AND (:OLD.QUANTITY_COMMIT > 0) THEN
/*node*/   t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 3, 'RAISE COMMAND');
/*----*/   t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/   t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 29, 'EXIT NODE - RAISE');
/*----*/   return;
           RAISE_APPLICATION_ERROR  (-20021,'[TRIGGER  TRG_PRODUCT_QUANTITY]  Error  deleting  order
items');
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 4, NULL);
```

```
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 5, NULL);
         IF UPDATING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 6, NULL);
            IF (:OLD.PRODUCT_ID <> :NEW.PRODUCT_ID) THEN
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 7, 'RAISE COMMAND');
/*----*/       t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 29, 'EXIT NODE - RAISE');
/*----*/       return;
               RAISE_APPLICATION_ERROR  (-20022,'[TRIGGER  TRG_PRODUCT_QUANTITY]  Error  updating
order items');
            END IF;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 8, NULL);
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 9, NULL);
            IF (:OLD.QUANTITY = :NEW.QUANTITY) AND
               (:OLD.QUANTITY_COMMIT = :NEW.QUANTITY_COMMIT) THEN
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 10, 'RETURN COMMAND');
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 29, 'EXIT NODE - RETURN');
               RETURN;
            END IF;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 11, NULL);
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 12, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 13, NULL);
         IF DELETING OR UPDATING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 14, 'BEGIN BLOCK');
            BEGIN
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 15, 'UPDATE COMMAND');
/*data*/       INSERT INTO T_DATA_WORK
/*data*/          ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/          SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/                 TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY_ORDERED;QUANTITY_COMMIT;'
/*data*/          FROM   PRODUCT_STATUS
/*data*/          WHERE  PRODUCT_ID = :OLD.PRODUCT_ID;
               UPDATE PRODUCT_STATUS
               SET QUANTITY_ORDERED = NVL(QUANTITY_ORDERED, 0) - NVL(:OLD.QUANTITY, 0),
                   QUANTITY_COMMIT  = NVL(QUANTITY_COMMIT, 0)  - NVL(:OLD.QUANTITY_COMMIT, 0)
               WHERE PRODUCT_ID = :OLD.PRODUCT_ID;
/*data*/       INSERT INTO T_DATA_WORK
/*data*/          ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/          SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/                 TO_CHAR(PRODUCT_ID), 1, NULL, 'QUANTITY_ORDERED;QUANTITY_COMMIT;'
/*data*/          FROM   PRODUCT_STATUS
/*data*/          WHERE  PRODUCT_ID = :OLD.PRODUCT_ID;
            EXCEPTION
               WHEN OTHERS THEN
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 16, 'EXCEPTION');
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 17, 'OTHERS');
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 18, 'RAISE COMMAND');
/*----*/            t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/                t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 29, 'EXIT NODE -
RAISE');
/*----*/            return;
                    RAISE_APPLICATION_ERROR    (-20023,'[TRIGGER    TRG_PRODUCT_QUANTITY]    Error
deleting/updating pending inventories: ' || sqlerrm);
            END;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 19, 'END BLOCK');
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 20, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 21, NULL);
         IF INSERTING OR UPDATING THEN
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 22, 'BEGIN BLOCK');
            BEGIN
/*node*/       t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 23, 'UPDATE COMMAND');
/*data*/       INSERT INTO T_DATA_WORK
/*data*/          ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/          SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/                 TO_CHAR(PRODUCT_ID), NULL, 1, 'T_KEY;QUANTITY_ORDERED;QUANTITY_COMMIT;'
/*data*/          FROM   PRODUCT_STATUS
/*data*/          WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
```

```
               UPDATE PRODUCT_STATUS
               SET QUANTITY_ORDERED = NVL(QUANTITY_ORDERED, 0) + NVL(:NEW.QUANTITY, 0),
                   QUANTITY_COMMIT  = NVL(QUANTITY_COMMIT, 0)  + NVL(:NEW.QUANTITY_COMMIT, 0)
               WHERE PRODUCT_ID = :NEW.PRODUCT_ID;
/*data*/       INSERT INTO T_DATA_WORK
/*data*/          ( CASE_ID, LINE_ID, TABLE_ID, KEY_ID, DDEF, DUSE, COLUMNS )
/*data*/          SELECT t_v_case_id, t_v_line_id, 'PRODUCT_STATUS',
/*data*/                 TO_CHAR(PRODUCT_ID), 1, NULL, 'QUANTITY_ORDERED;QUANTITY_COMMIT;'
/*data*/          FROM   PRODUCT_STATUS
/*data*/          WHERE  PRODUCT_ID = :NEW.PRODUCT_ID;
            EXCEPTION
               WHEN OTHERS THEN
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 24, 'EXCEPTION');
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 25, 'OTHERS');
/*node*/            t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 26, 'RAISE COMMAND');
/*----*/            t_p_setcaseexception (t_v_case_id, t_v_line_id, 'RAISE');
/*node*/                t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 29, 'EXIT NODE -
RAISE');
/*----*/            return;
                    RAISE_APPLICATION_ERROR    (-20024,'[TRIGGER    TRG_PRODUCT_QUANTITY]    Error
inserting/updating pending inventories: ' || sqlerrm);
            END;
/*node*/    t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 27, 'END BLOCK');
         END IF;
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 28, NULL);
/*node*/ t_v_line_id := t_f_setnode (t_v_case_id, t_v_rule_id, 29, 'EXIT NODE');
END TRG_PRODUCT_QUANTITY;
```

Apêndice B

Esquema Conceitual da Base de Dados para um Conjunto de Regras Ativas Escritas em SQL

Este apêndice exibe o modelo conceitual simplificado da base de dados manipulada por um conjunto de regras ativas escritas em SQL, o qual foi utilizado no experimento descrito no Capítulo 5; o código fonte instrumentado atribuído a tais regras localiza-se no Apêndice A. A Figura B.1 possui uma representação gráfica desse modelo, inspirada na nomenclatura adotada em (Elmasri e Navathe, 2003): os retângulos representam tipos de entidades e os losangos denotam tipos de relacionamento; os demais elementos da notação podem ser entendidos a partir da descrição do esquema da base de dados apresentada abaixo, onde as denominações entre parêntesis denotam o tipo de relacionamento mencionado:

> "Cada departamento possui uma localização (*IV*), que por sua vez situa-se em um país (*II*) de uma particular região (*I*). Uma localização é atribuída a cada depósito (*III*), o qual possui inventários de produtos (*X*): um mesmo produto pode pertencer a vários inventários (*XIII*). Empregados estão associados a cargos (*VII*) e estão alocados a departamentos (*V*), os quais são administrados por empregados com função de gerência (*VI*). Existe ainda uma relação de supervisão entre empregados (*VIII*). Ordens de compra são solicitadas por clientes (*XI*) a empregados (*IX*). Tais ordens são compostas por itens (*XII*), que representam a intenção de adquirir produtos (*XIV*): uma intenção é materializada pela entrega de unidades de produto (*XV*). Todo produto está associado a uma situação que descreve quantitativos de estoque, ordens de compra e entrega (*XVII*) e pode possuir uma solicitação de reposição de estoque (*XVI*)".

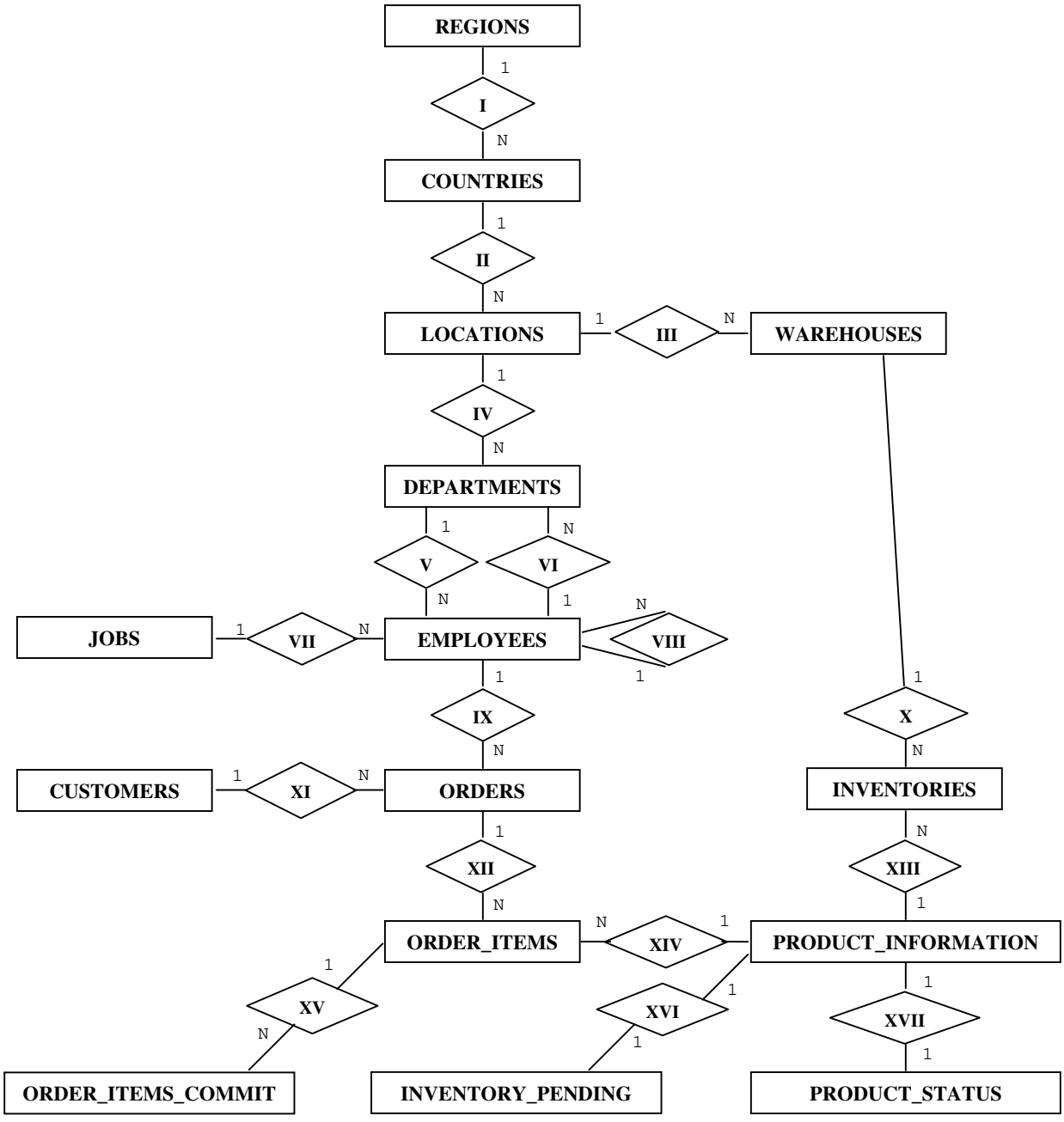


Figura B.1: Representação gráfica do modelo conceitual simplificado do banco de dados utilizado no experimento descrito no Capítulo 5.

Apêndice C

ADAPT-TOOL – Uma Ferramenta de Suporte à Aplicação dos Critérios Baseados na Interação entre Regras Escritas em SQL

De forma similar a outros critérios estruturais, os critérios baseados na interação entre regras ativas demandam por uma ferramenta para seu emprego efetivo, pois a aplicação de critérios baseados em análise de fluxo de dados sem o apoio de uma ferramenta automatizada é limitada a programas muito simples (Korel e Laski, 1985). Neste apêndice são realçados os principais elementos funcionais de uma ferramenta de suporte à aplicação dos Critérios Baseados na Interação entre Regras Ativas escritas em SQL, denominada ADAPT-TOOL (***A****ctive* ***D****atabase* ***AP****plication* ***T****esting* ***TOOL*** *for active rules written in SQL*); aspectos e modelos de implementação dos critérios foram descritos no Capítulo 6.

Com o desenvolvimento da ferramenta ADAPT-TOOL pretende-se viabilizar, preliminarmente, o uso prático dos Critérios Baseados na Interação entre Regras, bem como permitir a comparação entre granularidades de fluxo de dados, no que diz respeito a sua eficácia para a descoberta de defeitos. Representa um recurso para a análise de fluxo de dados persistentes em diversas precisões, permitindo a apreciação de conjuntos de regras defeituosas escritas em SQL, pelo confronto de diversos tipos de defeito em comandos de manipulação.

O objetivo não é ser "a" ferramenta de teste baseado na interação entre regras, mas buscar a automação de muitos dos encargos atribuídos à atividade de teste. Nesse sentido, algumas tarefas de custo elevado no âmbito de aplicações de banco de dados, tais como geração de bases de entrada e oráculo de teste, são implementadas, encorajando o testador à aplicação da abordagem de teste proposta.

Um aspecto pertinente é a construção de uma base dos dados atribuídos ao teste. Neste recurso são armazenados dados tais como: versões instrumentadas das regras em teste; associações persistentes; versões defeituosas de conjuntos de regras; dados de entrada de cada caso de teste, incluindo base de dados e comandos de disparo; fluxo de controle e fluxo de dados por granularidade oriundos da aplicação de cada caso de teste; avaliação do oráculo; etc.

A versão inicial da ferramenta utiliza o sistema gerenciador de banco de dados *Oracle*, mas pode ser estendida a outros sistemas que suportam o modelo relacional. Sua arquitetura é simples, fundamentando-se em um conjunto de rotinas escritas em SQL (procedimentos e funções), que estão gravadas no banco de dados (exemplos de tais rotinas são demonstrados no Capítulo 6), as quais são invocadas pelo testador utilizando uma *interface* de fácil manuseio.

A *interface* foi desenvolvida utilizando o recurso *Delphi*, apoiando-se em uma extensão da linguagem *Pascal* orientada a objetos. Janelas são utilizadas para as diversas funcionalidades implementadas, baseando-se nas facilidades de botões e de teclas de atalho para iniciar tarefas específicas em cada etapa do teste. Algumas dessas *interfaces* são exemplificadas nos módulos da ferramenta abaixo descritos. Os termos *base de dados de teste* e *base de teste* referem-se, indistintamente, aos dados dedicados ao controle da atividade de teste, visando à avaliação dos testes aplicados; outras referências a bases de dados destinam-se a bancos de dados da aplicação.

C.1 Módulos da Ferramenta

Distinguem-se duas etapas na automação de critérios de teste estrutural: estática e dinâmica. A primeira em geral concentra-se na análise do código fonte, para a abstração de elementos requeridos e para sua instrumentação, a qual constrói uma versão modificada do código fonte pela inserção de comandos que, em tempo de execução, extraem informação útil à fase dinâmica. A segunda suporta a aplicação de casos de teste e inclui atividades desde a geração de dados de teste até avaliação do teste. A ferramenta ADAPT-TOOL concentra-se na fase dinâmica e utiliza estruturas de dados do modelo relacional para a construção de uma base de dados de teste. O Capítulo 6 (Seção 6.2) trata o modelo conceitual da base de dados de teste, e o Apêndice D apresenta a implementação desse modelo.

As seguintes atividades são suportadas pela ferramenta: geração de dados de teste; controle de versões defeituosas de um conjunto de regras; aplicação e re-aplicação de casos de teste; oráculo de teste; e avaliação de conjuntos de casos de teste por granularidade.

C.1.1 Geração de Dados de Teste

Conforme definido no Capítulo 4, um caso de teste λ para uma regra ativa r é dado pela quádrupla $\langle \theta, \Delta_0, \Delta_1, M \rangle$, tal que: θ é a operação que provocou o evento de disparo de r; Δ_0 é o estado da(s) base(s) de dado(s) antes da execução de θ (e de r); Δ_1 é o estado da(s) base(s) de dado(s) após a execução de r; e M é a seqüência $\langle m_1, m_2, ..., m_k \rangle$, $k \geq 0$, de mensagens emitidas durante a execução de r; nesse sentido, os dados de teste de uma regra ativa são dados por $\langle \theta, \Delta_0 \rangle$.

A geração de bases de dados segue a abordagem descrita em (Ostrand e Balcer, 1988; Chays et al., 2000), onde são atribuídos valores para cada atributo (ver a Figura C.1, onde é associado um conjunto de valores ao atributo *PRODUCT_ID* da relação *ORDER_ITEMS*), que serão combinados para compor relações de entrada, garantindo que restrições de integridade dessas relações sejam satisfeitas, ou seja, são produzidos estados válidos do banco de dados. As *tuplas* geradas são armazenadas na forma de comandos do tipo *insert*, permitindo a construção de bases de dados dedicadas ao teste de regras. Assim, casos de teste podem ser aplicados e re-aplicados a qualquer tempo, evitando que bancos de dados reais sejam corrompidos pela atividade de teste, problema este realçado em (Chays et al., 2000) para aplicações de banco de dados.

Ao contrário da construção de bases de dados de entrada, a elaboração dos comandos de manipulação que provocam o disparo de regras é realizada de maneira semi-automatizada. A partir da seleção da variável persistente e do tipo de comando de mudança de estado, a ferramenta gera um comando para o disparo de regras, o qual pode ser editado pelo testador. A *interface* da Figura C.2 ilustra a geração de um comando *update* para a relação *ORDER_ITEMS*, utilizando os dados de uma *tupla* existente no banco de dados.

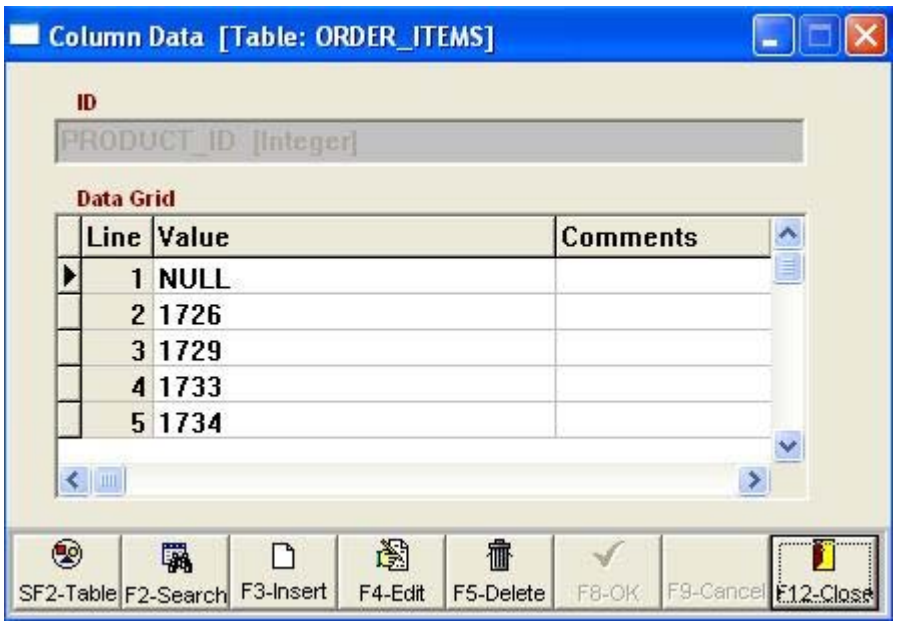


Figura C.1: *Interface* para a associação de conjuntos de valores a atributos da base de dados.

Figura C.2: *Interface* para a elaboração de comandos de disparo de regras.

C.1.2 Controle de Versões Defeituosas

O controle de versões implementado pela ferramenta apóia o experimento descrito no Capítulo 5, onde foram construídas várias versões de um conjunto de regras, onde cada qual possui uma única regra defeituosa pela inserção de defeitos em comandos de manipulação. As regras do conjunto em teste são catalogadas na base de dados de teste, incluindo seu código fonte sem defeitos. Em adição, a regra defeituosa de cada versão é descrita, determinando-se o tipo de defeito existente e o código fonte com defeito. A idéia é que o testador possa descrever os objetos em teste, deixando para a ferramenta o papel de selecionar os objetos devidos para cada caso de teste.

A Figura C.3 ilustra a *interface* para versões defeituosas de um conjunto de regras. São determinados: a regra ativa; o nó defeituoso; o tipo de defeito existente; os comandos correto e defeituoso; e se a presença do defeito modifica a instrumentação de fluxo de dados em relação à versão sem defeitos. O código fonte da regra defeituosa é editado pela inserção do comando defeituoso, conforme exibido na Figura C.4.

Cada caso de teste refere-se a uma particular versão do conjunto de regras. Em tempo de execução, as regras são criadas na base de meta-dados da aplicação, utilizando-se seus códigos fonte sem defeitos, como também a regra defeituosa atribuída ao caso de teste. Assim, a ferramenta disponibiliza automaticamente ao teste uma das versões do conjunto de regras, habilitando o disparo das regras dessa versão e permitindo a apreciação dos resultados de teste por tipo de defeito e por precisão de fluxo de dados, sem a intervenção do testador.

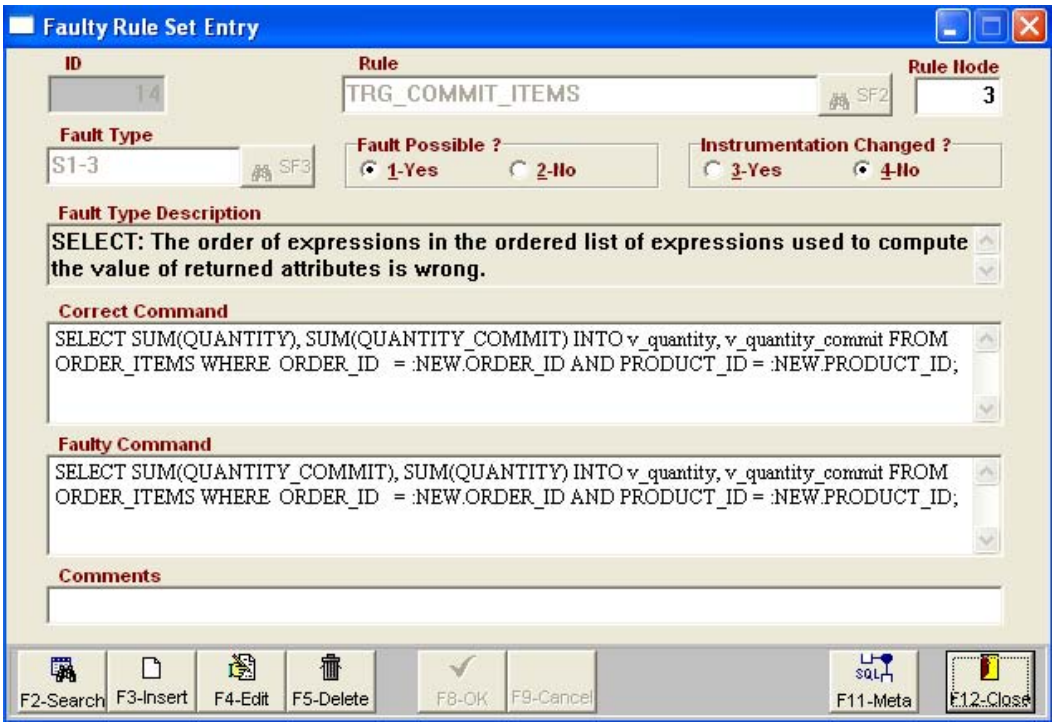


Figura C.3: *Interface* para versões defeituosas de um conjunto de regras.

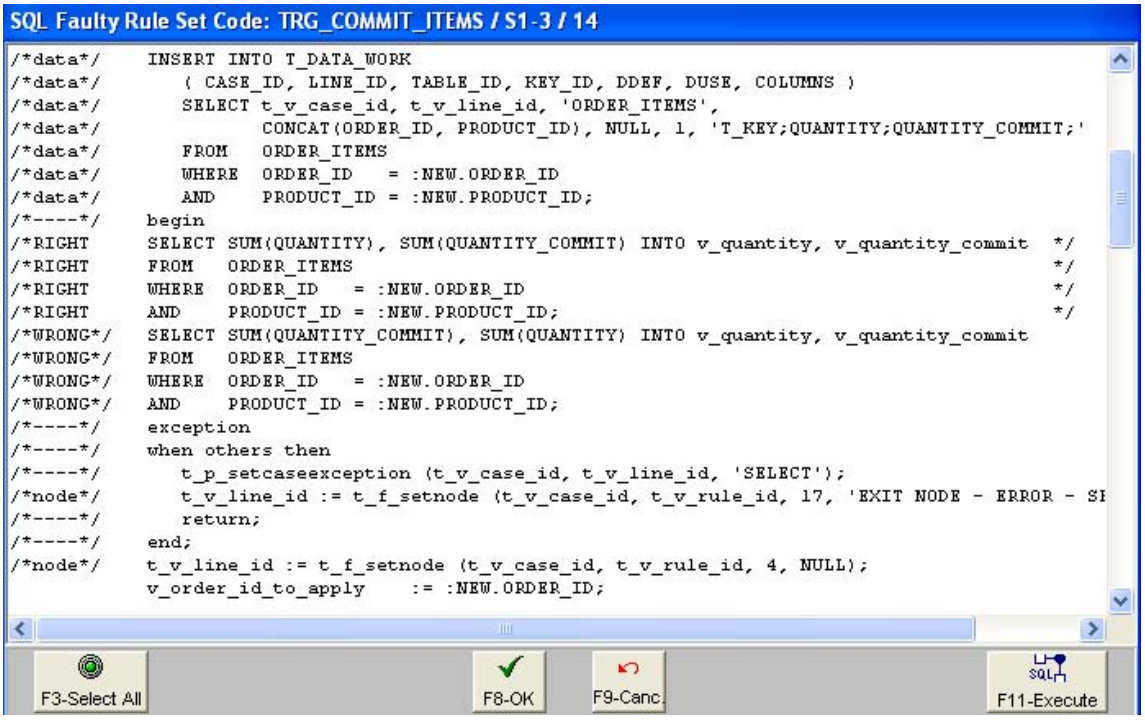


Figura C.4: *Interface* para edição de código fonte de regras defeituosas.

C.1.3 Oráculo de Teste

A introdução de persistência de dados na descrição de casos de teste aumenta a complexidade da tarefa do oráculo, visto que bases de dados ocupam os espaços de entrada e de saída do teste. O modelo descrito na Seção 6.2 permitiu a implementação integral da tarefa do oráculo pela ferramenta. Para alcançar tal objetivo, cada caso de teste é aplicado às versões correta e defeituosa do conjunto de regras, o que viabiliza a comparação das saídas esperada e obtida. A saída resultante da aplicação de qualquer caso de teste é computada no âmbito da base de teste, gerando-se um valor numérico representativo, denominado *tag de teste*. Dessa forma, obtém-se duas *tags* para cada caso de teste aplicado, que, quando forem distintas, detectará a presença do defeito na versão defeituosa do conjunto de regras.

C.1.4 Aplicação (e Re-aplicação) de Casos de Teste

A *interface* para a aplicação de casos de teste é apresentada na Figura C.5. Os dados dos casos de teste são armazenados na base de teste, permitindo que cada caso de teste possa ser re-aplicado a qualquer tempo. São exibidos dados descritivos de um particular caso de teste, tais como:

- *id*: identificador do caso de teste;
- *create timestamp*: data e tempo de criação do caso de teste;
- *set database timestamp*: data e tempo de seleção dados para a base de dados de entrada;
- *set application timestamp*: data e tempo de elaboração dos comandos de manipulação para o disparo de regras, responsáveis por provocar eventos externos;
- *apply timestamp*: data e tempo da aplicação do caso de teste;
- *result*: se o defeito foi revelado pela aplicação do caso de teste;
- *rule fault*: versão defeituosas do conjunto de regras;
- *application input*: comandos de manipulação para o disparo de regras;
- *application error message*: mensagem que descreve algum erro detectado pelo sistema gerenciador, quando for o caso.

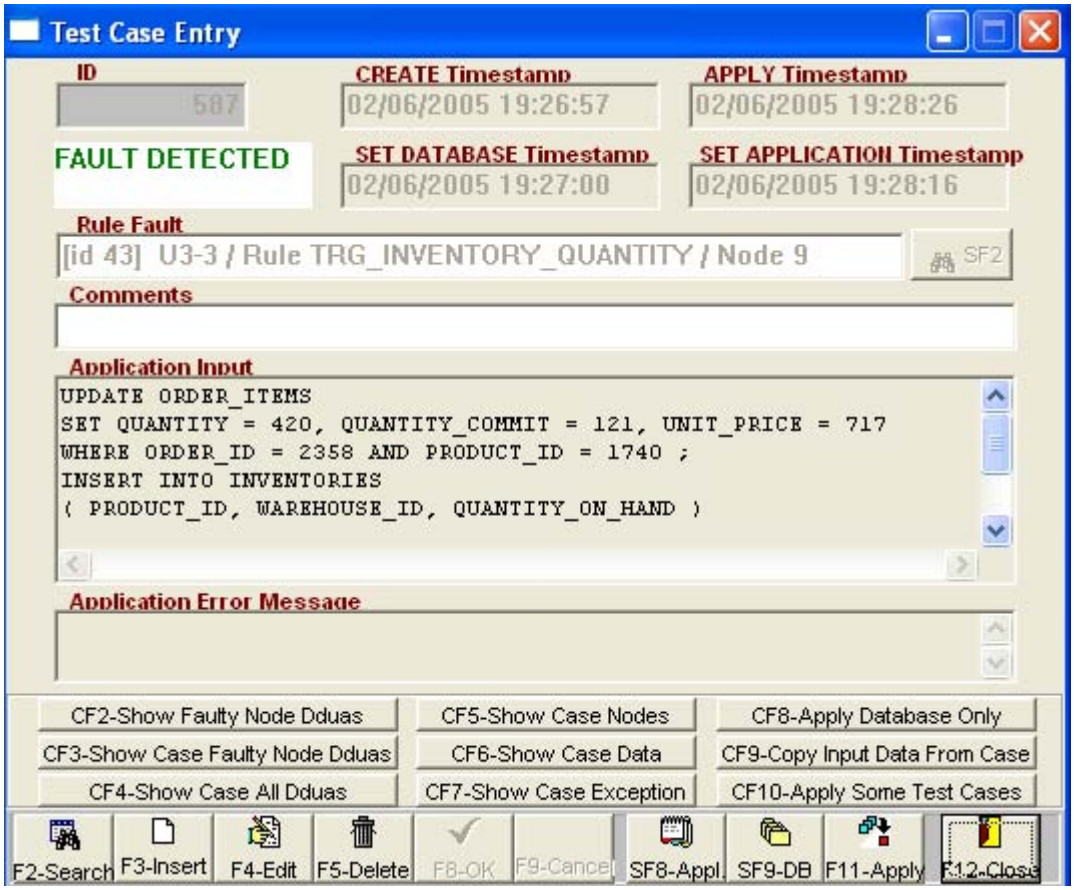


Figura C.5: *Interface* para a aplicação (e re-aplicação) de casos de teste.

Um conjunto de tarefas pode ser iniciado a partir da *interface* presente na Figura C.5, desde consultar as associações cobertas por granularidade, até aplicar um conjunto de casos de teste, conforme descrito abaixo:

- *CF2 – show faulty node dduas*: exibir as associações que participam do nó defeituoso atribuído à versão do conjunto de regras associado ao caso de teste, apresentando resultados de cobertura por precisão de fluxo de dados de todos os casos de teste aplicados;
- *CF3 – show case faulty node dduas*: exibir as associações que participam do nó defeituoso atribuído à versão do conjunto de regras associado ao caso de teste, apresentando resultados de cobertura por precisão de fluxo de dados deste caso de teste;
- *CF4 – show case all dduas*: exibir todas associações, apresentando resultados de cobertura por precisão de fluxo de dados deste caso de teste;

- *CF5 – show case nodes*: exibir a seqüência de nós exercitados pelo caso de teste;
- *CF6 – show case data*: exibir os dados persistentes definidos e usados pela aplicação do caso de teste, distinguindo atributos e *tuplas* para cada nó exercitado;
- *CF7 – show case exception*: exibir as exceções ocorridas em comandos manipulação pela aplicação do caso de teste, distinguindo o contexto em que foram provocadas;
- *CF8 – apply database only*: criar o banco de dados da aplicação, a partir dos comandos de inserção de *tuplas* gerados para as relações para a base de dados de entrada;
- *CF9 – copy input data from case*: importar os dados de teste de um outro caso de teste;
- *CF10 – apply some test cases*: selecionar e aplicar um conjunto de casos de teste, dentre os presentes na base de teste;
- *SF8 – appl*: editar os comandos de manipulação para o disparo de regras;
- *SF9 – db*: gerar comandos de inserção de *tuplas* (*insert*), gerados para as relações para a base de dados de entrada;
- *F11 – apply*: aplicar o caso de teste, pela criação do banco de dados da aplicação, a partir dos comandos de inserção de *tuplas* gerados para as relações para a base de dados de entrada, e pela execução dos comandos de manipulação para o disparo de regras.

C.1.5 Avaliação por Granularidade

A avaliação de conjuntos de casos de teste é realizada por precisão de fluxo de dados, conforme implementação descrita no Capítulo 6, suportando que estudos comparativos sobre granularidades sejam conduzidos. A cobertura de associações de interação é computada por granularidade, distinguindo-se entre casos de teste reveladores e não reveladores de defeito. A ferramenta analisa os resultados por associação de interação e por versão de conjuntos de regras, e consolida todos os casos de teste aplicados.

C.2 Relatórios da Ferramenta

A versão atual da ferramenta ADAPT-TOOL produz uma série de relatórios para a fase dinâmica do teste, conforme descrito abaixo:

- *rules*: exibir a identificação o tipo de operação de disparo do conjunto de regras em teste;
- *dduas*: apresentar todas as associações de interação do conjunto de regras em teste, apontando seu elementos descritivos (tipo de associação, variável persistente, nó de definição e arco de uso persistentes, etc.), mostrando o número de casos de teste por granularidade que cobre cada associação;
- *dduas per rule fault set*: para uma particular versão defeituosa do conjunto de regras, abstrair as associações de interação cujo nó defeituoso participa da definição ou do uso persistentes, apontando os elementos descritivos de tais associações, e mostrando em que granularidades foram cobertas pelos casos de teste aplicados;
- *cases per rule fault set*: exibir os casos de teste aplicados para uma particular versão defeituosa, descrevendo-os pelos dados: tipo dos comandos da aplicação utilizados para o disparo de regras; se o nó defeituoso foi exercitado; número de exceções ocorridas em comandos de manipulação; número de associações de interação cobertas em cada granularidade, distinguindo entre todas as associações e as associações cujo nó defeituoso participa da definição ou do uso persistentes; e se a presença do defeito foi detectada;
- *cases and dduas per faulty rule set*: exibir os casos de teste aplicados para uma particular versão defeituosa, descrevendo-os pelos dados: tipo dos comandos da aplicação utilizados para o disparo de regras; se o nó defeituoso foi exercitado; número de exceções ocorridas em comandos de manipulação; número de associações de interação cobertas em cada granularidade, considerando apenas as associações cujo nó defeituoso participa da definição ou do uso persistentes; os elementos descritivos de tais associações, apontando a granularidade mais precisa em que foram cobertas; e se a presença do defeito foi detectada;
- *cases per ddua – summary*: para uma particular associação de interação, exibir os casos de teste atribuídos às versões defeituosas do conjunto de regras, cujo nó defeituoso participa da definição ou do uso persistentes da associação, descrevendo-os pelos dados: versão

defeituosa do conjunto de regras; tipo de defeito; se a presença do defeito foi detectada; número de exceções ocorridas em comandos de manipulação; número de vezes em que o nó defeituoso foi exercitado; e granularidade mais precisa em que cobriu a associação;

- *results per faulty rule set*: apresentar as versões defeituosas do conjunto de regras em teste, indicando para cada versão: o tipo de defeito associado; o número de associações de interação cobertas por granularidade, com distinção entre casos de teste reveladores e não reveladores de defeitos;
- *results per fault type*: apresentar os tipos de defeito de manipulação, exibindo, para cada tipo, o número de associações de interação cobertas por granularidade, com distinção entre casos de teste reveladores e não reveladores de defeitos;
- *results per ddua*: apresentar as associações de interação exercitadas, excetuando-se aquelas ainda não cobertas em qualquer granularidade, exibindo, para cada associação, o número de casos de teste por granularidade, com distinção entre casos de teste reveladores e não reveladores de defeitos;
- *testing summary*: sintetizar a aplicação de um conjunto de casos de teste, computando o número de casos de teste reveladores e não reveladores de defeito, estratificando em: cobertura por granularidade (relação, atributo, *tupla* e valor de atributo) e tipo de exceção (provocada pelo programador e provocada pelo sistema).

Em síntese, a ferramenta ADAPT-TOOL representa a primeira iniciativa para apoiar o emprego e para apreciar a aplicabilidade dos Critérios Baseados na Interação entre Regras, e constituiu um recurso valioso para a condução e para a análise de resultados do experimento descrito no Capítulo 5.

Apêndice D

Esquema de uma Base de Dados de Teste para a Ferramenta ADAPT-TOOL

Este apêndice introduz o esquema da base de dados de teste adotado pela ferramenta ADAPT-TOOL, a qual foi descrita no Apêndice C. A referida base de dados utiliza estruturas de dados do modelo relacional em sua implementação, cujo modelo conceitual sucinto foi tratado no Capítulo 6 (Seção 6.2). O esquema é apresentado abaixo na forma de comandos de definição da base de dados, tais como *create table* e *create trigger*.

```
CREATE TABLE T_TABLES (
    TABLE_ID             VARCHAR2(50)       NOT NULL
   ,SET_CASEINPUT        CHAR(1)            NOT NULL
   ,SET_TABLETAG         CHAR(1)            NOT NULL
   ,SET_TABLEPROPERTIES  CHAR(1)            NOT NULL
   ,TABLETAG             NUMBER(12)             NULL
   ,DELETE_SEQUENCE      NUMBER                 NULL
   ,COMMENTS             VARCHAR2(200)          NULL
   ,CONSTRAINT           T_TABLES__PK
                         PRIMARY KEY(TABLE_ID)
   ,CONSTRAINT           T_TABLES__CK_SET_CASEINPUT
                         CHECK (SET_CASEINPUT IN ('Y', 'N'))
   ,CONSTRAINT           T_TABLES__CK_SET_TABLETAG
                         CHECK (SET_TABLETAG IN ('Y', 'N'))
   ,CONSTRAINT           T_TABLES__CK_SET_TABPROPERTIES
                         CHECK (SET_TABLEPROPERTIES IN ('Y', 'N'))
   );

CREATE TABLE T_COLUMNS (
    TABLE_ID             VARCHAR2(50)       NOT NULL
   ,COLUMN_ID            VARCHAR2(50)       NOT NULL
   ,COLUMN_TYPE          CHAR(9)            NOT NULL
   ,DATA_TYPE            CHAR(1)            NOT NULL
   ,ALLOW_DISPERSAL      CHAR(1)            NOT NULL
   ,ALLOW_CONCENTRATION  CHAR(1)            NOT NULL
   ,ALLOW_NULITY         CHAR(1)            NOT NULL
   ,COMMENTS             VARCHAR2(200)          NULL
   ,CONSTRAINT           T_COLUMNS__PK
                         PRIMARY KEY(TABLE_ID, COLUMN_ID)
   ,CONSTRAINT           T_COLUMNS__FK_TABLES
                         FOREIGN KEY(TABLE_ID) REFERENCES T_TABLES(TABLE_ID)
```

```
   ,CONSTRAINT          T_COLUMNS__CK_DATA_TYPE
                        CHECK (DATA_TYPE IN ('D', 'S', 'I', 'N'))
   ,CONSTRAINT          T_COLUMNS__CK_COLUMN_TYPE
                        CHECK (COLUMN_TYPE IN ('KEYNAME', 'KEYDATA', 'ATTRIBUTE'))
   ,CONSTRAINT          T_COLUMNS__CK_DISPERSAL
                        CHECK (ALLOW_DISPERSAL IN ('Y', 'N'))
   ,CONSTRAINT          T_COLUMNS__CK_CONCENTRATION
                        CHECK (ALLOW_CONCENTRATION IN ('Y', 'N'))
   ,CONSTRAINT          T_COLUMNS__CK_NULITY
                        CHECK (ALLOW_NULITY IN ('Y', 'N'))
   );

CREATE TABLE T_COLUMN_DATA (
    TABLE_ID            VARCHAR2(50)       NOT NULL
   ,COLUMN_ID           VARCHAR2(50)       NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,DATA_VALUE          VARCHAR2(100)      NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_COLUMN_DATA__PK
                        PRIMARY KEY(TABLE_ID, COLUMN_ID, LINE_ID)
   ,CONSTRAINT          T_COLUMN_DATA__FK_COLUMNS
                        FOREIGN KEY(TABLE_ID, COLUMN_ID) REFERENCES T_COLUMNS(TABLE_ID,
COLUMN_ID)
);

CREATE OR REPLACE TRIGGER T_TRG_COLUMN_DATA
    BEFORE INSERT ON T_COLUMN_DATA
    FOR EACH ROW
    DECLARE
       t_v_line_id   T_COLUMN_DATA.LINE_ID%TYPE;
BEGIN
    SELECT (NVL(MAX(LINE_ID),0)+1) INTO t_v_line_id
    FROM   T_COLUMN_DATA
    WHERE  TABLE_ID = :NEW.TABLE_ID
    AND    COLUMN_ID = :NEW.COLUMN_ID;
    :NEW.LINE_ID := t_v_line_id;
END T_TRG_COLUMN_DATA;

CREATE TABLE T_OBJECT_CODE (
    OBJECT_ID           VARCHAR2(50)       NOT NULL
   ,OBJECT_TYPE         VARCHAR2(30)       NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,SQL_CODE            VARCHAR2(500)          NULL
   ,CONSTRAINT          T_OBJECT_CODE__PK
                        PRIMARY KEY(OBJECT_ID, OBJECT_TYPE, LINE_ID)
   ,CONSTRAINT          T_OBJECT_CODE__CK_OBJECT_TYPE
                        CHECK (OBJECT_TYPE IN ('RULE', 'RULE_FAULT', 'METHOD'));

   );

CREATE OR REPLACE TRIGGER T_TRG_OBJECT_CODE
    BEFORE INSERT ON T_OBJECT_CODE
    FOR EACH ROW
    DECLARE
       t_v_line_id   T_OBJECT_CODE.LINE_ID%TYPE;
BEGIN
    SELECT (NVL(MAX(LINE_ID),0)+1) INTO t_v_line_id
    FROM   T_OBJECT_CODE
    WHERE  OBJECT_ID   = :NEW.OBJECT_ID
    AND    OBJECT_TYPE = :NEW.OBJECT_TYPE;
    :NEW.LINE_ID := t_v_line_id;
END T_TRG_OBJECT_CODE;

CREATE TABLE T_PARAMETERS (
    PARAMETER_ID        VARCHAR2(50)       NOT NULL
   ,PARAMETER_VALUE     VARCHAR2(200)          NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_PARAMETERS__PK
                        PRIMARY KEY(PARAMETER_ID)
   );
```

```
CREATE TABLE T_FAULT_TYPES (
    FAULT_TYPE_ID       VARCHAR2(10)       NOT NULL
   ,TARGET_COMMAND      VARCHAR2(10)       NOT NULL
   ,DESCRIPTION         VARCHAR2(500)          NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_FAULT_TYPES__PK
                        PRIMARY KEY(FAULT_TYPE_ID)
   ,CONSTRAINT          T_FAULT_TYPES__CK_COMMAND
                        CHECK (TARGET_COMMAND IN ('SELECT','INSERT', 'UPDATE','DELETE'))
   );

CREATE TABLE T_RULES (
    RULE_ID             VARCHAR2(50)       NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_RULES__PK
                        PRIMARY KEY(RULE_ID)
   );

CREATE TABLE T_RULE_FAULTS (
    RULE_FAULT_ID       INTEGER            NOT NULL
   ,RULE_ID             VARCHAR2(50)       NOT NULL
   ,FAULT_TYPE_ID       VARCHAR2(10)       NOT NULL
   ,NODE_ID             INTEGER            NOT NULL
   ,COMMAND_CORRECT     VARCHAR2(500)          NULL
   ,COMMAND_FAULTY      VARCHAR2(500)          NULL
   ,FAULT_POSSIBLE      CHAR(1)            NOT NULL
   ,INSTR_CHANGED       CHAR(1)            NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_RULE_FAULTS__PK
                        PRIMARY KEY(RULE_FAULT_ID)
   ,CONSTRAINT          T_RULE_FAULTS__FK_RULES
                        FOREIGN KEY(RULE_ID) REFERENCES T_RULES(RULE_ID)
   ,CONSTRAINT          T_RULE_FAULTS__FK_FAULT_TYPES
                        FOREIGN KEY(FAULT_TYPE_ID) REFERENCES T_FAULT_TYPES(FAULT_TYPE_ID)
   ,CONSTRAINT          T_RULE_FAULTS__CK_FPOSSIBLE
                        CHECK (FAULT_POSSIBLE IN ('Y','N'))
   ,CONSTRAINT          T_RULE_FAULTS__CK_ICHANGED
                        CHECK (INSTR_CHANGED IN ('Y','N'))
   );

CREATE OR REPLACE TRIGGER T_TRG_RULE_FAULTS
    BEFORE INSERT ON T_RULE_FAULTS
    FOR EACH ROW
    DECLARE
       t_v_rule_fault_id   T_RULE_FAULTS.RULE_FAULT_ID%TYPE;
BEGIN
    SELECT (NVL(MAX(RULE_FAULT_ID),0)+1) INTO t_v_rule_fault_id
    FROM   T_RULE_FAULTS;
    :NEW.RULE_FAULT_ID  := t_v_rule_fault_id;
END T_TRG_RULE_FAULTS;

CREATE TABLE T_CASES (
    CASE_ID                     INTEGER            NOT NULL
   ,CREATE_TIMESTAMP            TIMESTAMP          NOT NULL
   ,RULE_FAULT_ID               INTEGER            NOT NULL
   ,APPLY_TIMESTAMP             TIMESTAMP              NULL
   ,SET_DATABASE_TIMESTAMP      TIMESTAMP              NULL
   ,SET_APPLICATION_TIMESTAMP   TIMESTAMP              NULL
   ,APPLICATION_ERROR           VARCHAR2(2000)         NULL
   ,COMMENTS                    VARCHAR2(200)          NULL
   ,DATABASETAG_CORRECT         NUMBER(12)             NULL
   ,DATABASETAG_FAULT           NUMBER(12)             NULL
   ,CONSTRAINT                  T_CASES__PK
                                PRIMARY KEY(CASE_ID)
   ,CONSTRAINT                  T_CASES__FK_RULE_FAULTS
                                FOREIGN KEY(RULE_FAULT_ID) REFERENCES T_RULE_FAULTS(RULE_FAULT_ID)
   );

CREATE OR REPLACE TRIGGER T_TRG_CASES
    BEFORE INSERT ON T_CASES
    FOR EACH ROW
```

```
    DECLARE
       t_v_case_id   T_CASES.CASE_ID%TYPE;
BEGIN
    SELECT (NVL(MAX(CASE_ID),0)+1) INTO t_v_case_id
    FROM   T_CASES;
    :NEW.CASE_ID         := t_v_case_id;
    SELECT SYSTIMESTAMP INTO :NEW.CREATE_TIMESTAMP FROM DUAL;
END T_TRG_CASES;

CREATE TABLE T_CASE_INPUT (
    CASE_ID             INTEGER            NOT NULL
   ,INPUT_TYPE          CHAR(11)           NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,SQL_COMMAND         VARCHAR2(2000)         NULL
   ,SQL_COMMAND_ERROR   VARCHAR2(2000)         NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_CASE_INPUT__PK
                        PRIMARY KEY(CASE_ID, INPUT_TYPE, LINE_ID)
   ,CONSTRAINT          T_CASE_INPUT__FK_CASES
                        FOREIGN KEY(CASE_ID) REFERENCES T_CASES(CASE_ID)
   ,CONSTRAINT          T_CASE_INPUT__CK_INPUT_TYPE
                        CHECK (INPUT_TYPE IN ('DATABASE','APPLICATION'))
   );

CREATE OR REPLACE TRIGGER T_TRG_CASE_INPUT
    BEFORE INSERT ON T_CASE_INPUT
    FOR EACH ROW
    DECLARE
       t_v_line_id   T_CASE_INPUT.LINE_ID%TYPE;
BEGIN
    SELECT (NVL(MAX(LINE_ID),0)+1) INTO t_v_line_id
    FROM   T_CASE_INPUT
    WHERE  CASE_ID = :NEW.CASE_ID
    AND    INPUT_TYPE = :NEW.INPUT_TYPE;
    :NEW.LINE_ID := t_v_line_id;
END T_TRG_CASE_INPUT;

CREATE TABLE T_NODES (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,RULE_ID             VARCHAR2(50)       NOT NULL
   ,NODE_ID             INTEGER            NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_NODES__PK
                        PRIMARY KEY(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_NODES__FK_CASES
                        FOREIGN KEY(CASE_ID) REFERENCES T_CASES(CASE_ID)
   ,CONSTRAINT          T_NODES__FK_RULES
                        FOREIGN KEY(RULE_ID) REFERENCES T_RULES(RULE_ID)
   );

CREATE TABLE T_CASE_EXCEPTION (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,RULE_ID             VARCHAR2(50)       NOT NULL
   ,NODE_ID             INTEGER            NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_CASE_EXCEPTION__PK
                        PRIMARY KEY(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_EXCEPTION__FK_NODES
                        FOREIGN KEY(CASE_ID, LINE_ID) REFERENCES T_NODES(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_EXCEPTION__FK_RULES
                        FOREIGN KEY(RULE_ID) REFERENCES T_RULES(RULE_ID)
   );

CREATE TABLE T_CASE_MESSAGE (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,RULE_ID             VARCHAR2(50)       NOT NULL
   ,NODE_ID             INTEGER            NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
```

```
   ,CONSTRAINT          T_CASE_MESSAGE__PK
                        PRIMARY KEY(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_MESSAGE__FK_NODES
                        FOREIGN KEY(CASE_ID, LINE_ID) REFERENCES T_NODES(CASE_ID, LINE_ID)
   ,CONSTRAINT          T_CASE_MESSAGE__FK_RULES
                        FOREIGN KEY(RULE_ID) REFERENCES T_RULES(RULE_ID)
   );

CREATE TABLE T_DATA (
    CASE_ID     INTEGER            NOT NULL
   ,LINE_ID     INTEGER            NOT NULL
   ,TABLE_ID    VARCHAR2(50)       NOT NULL
   ,KEY_ID      VARCHAR2(50)       NOT NULL
   ,COLUMN_ID   VARCHAR2(50)       NOT NULL
   ,DDEF        NUMBER(1)              NULL
   ,DUSE        NUMBER(1)              NULL
   ,CONSTRAINT  T_DATA__PK
                PRIMARY KEY(CASE_ID, LINE_ID, TABLE_ID, KEY_ID, COLUMN_ID)
   ,CONSTRAINT  T_DATA__FK_NODES
                FOREIGN KEY(CASE_ID, LINE_ID) REFERENCES T_NODES(CASE_ID, LINE_ID)
   ,CONSTRAINT  T_DATA__FK_COLUMNS
                FOREIGN KEY(TABLE_ID, COLUMN_ID) REFERENCES T_COLUMNS(TABLE_ID, COLUMN_ID)
   );

CREATE TABLE T_DATA_WORK (
    CASE_ID             INTEGER            NOT NULL
   ,LINE_ID             INTEGER            NOT NULL
   ,TABLE_ID            VARCHAR2(50)       NOT NULL
   ,KEY_ID              VARCHAR2(50)       NOT NULL
   ,DDEF                NUMBER(1)              NULL
   ,DUSE                NUMBER(1)              NULL
   ,COLUMNS             VARCHAR2(200)          NULL
);

CREATE OR REPLACE TRIGGER T_TRG_DATA_WORK
    AFTER INSERT OR UPDATE ON T_DATA_WORK
    FOR EACH ROW
DECLARE
      t_v_instance    INTEGER;
      t_v_column_id   VARCHAR2(50);
BEGIN
    IF UPDATING THEN
       raise_application_error(-20001,'[TRIGGER T_TRG_DATA_WORK] Updating on T_DATA_WORK is not
allowed.');
    END IF;
    t_v_instance := 1;
    t_v_column_id := t_f_getcmp (:NEW.COLUMNS, t_v_instance, ';');
    WHILE (t_v_column_id IS NOT NULL) LOOP
       BEGIN
          INSERT INTO T_DATA
             (CASE_ID, LINE_ID, TABLE_ID, KEY_ID, COLUMN_ID, DDEF, DUSE )
          VALUES
             (:NEW.CASE_ID, :NEW.LINE_ID, :NEW.TABLE_ID, :NEW.KEY_ID, t_v_column_id, :NEW.DDEF,
:NEW.DUSE );
       EXCEPTION
          WHEN OTHERS THEN
             IF :NEW.DDEF IS NOT NULL THEN
                UPDATE T_DATA
                SET    DDEF      = :NEW.DDEF
                WHERE  CASE_ID   = :NEW.CASE_ID
                AND    LINE_ID   = :NEW.LINE_ID
                AND    TABLE_ID  = :NEW.TABLE_ID
                AND    KEY_ID    = :NEW.KEY_ID
                AND    COLUMN_ID = t_v_column_id;
             END IF;
             IF :NEW.DUSE IS NOT NULL THEN
                UPDATE T_DATA
                SET    DUSE      = :NEW.DUSE
                WHERE  CASE_ID   = :NEW.CASE_ID
                AND    LINE_ID   = :NEW.LINE_ID
                AND    TABLE_ID  = :NEW.TABLE_ID
```

```
                AND    KEY_ID    = :NEW.KEY_ID
                AND    COLUMN_ID = t_v_column_id;
             END IF;
       END;
       t_v_instance := t_v_instance + 1;
       t_v_column_id := t_f_getcmp (:NEW.COLUMNS, t_v_instance, ';');
    END LOOP;
END T_TRG_DATA_WORK;

CREATE TABLE T_ASSOCIATIONS (
    ASSO_ID             VARCHAR2(20)       NOT NULL
   ,ASSO_TYPE           VARCHAR2(20)       NOT NULL
   ,RULE_ID_DDEF        VARCHAR2(50)       NOT NULL
   ,NODE_ID_DDEF        INTEGER            NOT NULL
   ,RULE_ID_DUSE_1      VARCHAR2(50)       NOT NULL
   ,NODE_ID_DUSE_1      INTEGER            NOT NULL
   ,RULE_ID_DUSE_2      VARCHAR2(50)       NOT NULL
   ,NODE_ID_DUSE_2      INTEGER            NOT NULL
   ,RULE_ID_THRU_1      VARCHAR2(50)           NULL
   ,NODE_ID_THRU_1      INTEGER                NULL
   ,RULE_ID_THRU_2      VARCHAR2(50)           NULL
   ,NODE_ID_THRU_2      INTEGER                NULL
   ,COMMAND_DDEF        VARCHAR2(10)           NULL
   ,COMMAND_DUSE        VARCHAR2(10)           NULL
   ,TABLE_ID            VARCHAR2(50)       NOT NULL
   ,COMMENTS            VARCHAR2(200)          NULL
   ,CONSTRAINT          T_ASSOCIATIONS__PK
                        PRIMARY KEY(ASSO_ID)
   ,CONSTRAINT          T_ASSOCIATIONS__CK_ASSO_TYPE
                        CHECK(ASSO_TYPE IN ( 'DDEF-DUSE-AE'     , 'DDEF-DUSE-AC'     ,
                                             'DDEF-DUSE-AA'     , 'DDEF-DUSE-AC-CAL' ,
                                             'DDEF-DUSE-AA-CAL' , 'DDEF-DUSE-AA-RET' ))
   );

CREATE TABLE T_CASE_ASSOCIATIONS (
    CASE_ID               INTEGER            NOT NULL
   ,ASSO_ID               VARCHAR2(20)       NOT NULL
   ,COVER_RELATION        NUMBER(1)              NULL
   ,COVER_TUPLE           NUMBER(1)              NULL
   ,COVER_ATTRIBUTE       NUMBER(1)              NULL
   ,COVER_ATTRIBUTEVALUE  NUMBER(1)              NULL
   ,LAST_TIMESTAMP        TIMESTAMP          NOT NULL
   ,COMMENTS              VARCHAR2(200)          NULL
   ,CONSTRAINT            T_CASE_ASSOCIATIONS__PK
                          PRIMARY KEY(CASE_ID, ASSO_ID)
   ,CONSTRAINT            T_CASE_ASSOCIATIONS__FK_CASES
                          FOREIGN KEY(CASE_ID) REFERENCES T_CASES(CASE_ID)
   ,CONSTRAINT            T_CASE_ASSOCIATIONS__FK_ASSO
                          FOREIGN KEY(ASSO_ID) REFERENCES T_ASSOCIATIONS(ASSO_ID)
   );
```

Apêndice E

Dados da Aplicação dos Critérios Baseados na Interação entre Regras Ativas

Este apêndice apresenta alguns dados obtidos na aplicação dos Critérios Baseados na Interação entre Regras Ativas. O emprego dos critérios refere-se ao experimento descrito detalhadamente na Seção 5.3, que foi suportado pela ferramenta ADAPT-TOOL.

Foram derivadas 26 versões defeituosas de um conjunto de regras denominado R_x, o qual é composto por quatro regras ativas escritas em SQL; o conjunto R_x possui 74 associações de interação. Em cada versão defeituosa de R_x existe um único defeito em comando de manipulação da SQL. Foram aplicados 632 casos de teste visando à descoberta da presença desses defeitos.

As Tabelas E.1, E.2, E.3 e E.4 apresentam os dados resultantes do experimento; a análise desses dados é explorada na Subseção 5.3.5. Os dados denotam números de coberturas das associações de interação nas granularidades *relação*, *tupla*, *atributo* e *valor de atributo*. É considerada a granularidade mais precisa em cada medição; por exemplo, se uma cobertura ocorre no nível *tupla* e, conseqüentemente, no nível *relação*, esta é computada como uma cobertura em nível *tupla*. A medição foi realizada observando-se, em cada caso de teste, apenas as associações de interação em que participavam o comando SQL defeituoso; em tais associações, o defeito estaria presente no comando representado pelo nó da definição persistente, ou no comando representado pelo nó em cujos arcos de saída estão os usos persistentes.

As Tabelas E.1 e E.2 apresentam, nos casos de teste reveladores e não reveladores de defeito, respectivamente, o número de coberturas de associações de interação por granularidade em cada uma das 26 versões de R_x. A coluna *Ident* refere-se à identificação da versão de R_x e as colunas *Rel*, *Tup*, *Atr* e *VAtr* denotam o número de coberturas nas granularidades *relação*, *tupla*,

atributo e *valor de atributo*, respectivamente. Estas tabelas buscam evidenciar a influência da precisão da análise de fluxo de dados na descoberta de defeitos, considerando-se que em cada versão defeituosa de R_x existe um defeito de manipulação.

Uma visão da cobertura por granularidade de cada associação de interação é apresentada nas Tabelas E.3 e E.4 para os casos de teste reveladores e não reveladores de defeito, respectivamente. A coluna *Ident* refere-se à identificação da associação de interação e as colunas *Rel*, *Tup*, *Atr* e *VAtr* denotam o número de coberturas nas granularidades *relação*, *tupla*, *atributo* e *valor de atributo*, respectivamente. As associações de interação que possuem valor zero para todas as granularidades em ambas as tabelas são ditas não executáveis. Estas tabelas buscam realçar a cobertura dos requisitos dos Critérios Baseados na Interação entre Regras nas diversas granularidades e sua influência na descoberta de defeitos.

Tabela E.1 – Número de associações de interação cobertas por granularidade para as versões do conjunto R_x, nos casos de teste reveladores de defeito.

Ident.	Rel	Tup	Atr	VAtr	Ident.	Rel	Tup	Atr	VAtr
1	51	0	4	12	**14**	8	0	0	2
2	12	0	6	0	**15**	24	0	2	7
3	13	0	5	0	**16**	12	0	1	0
4	25	0	8	0	**17**	19	0	0	0
5	26	0	8	0	**18**	173	32	18	15
6	23	0	8	0	**19**	36	0	10	10
7	29	0	0	0	**20**	36	0	12	9
8	13	2	1	2	**21**	115	15	23	23
9	14	0	3	3	**22**	83	16	0	4
10	15	0	2	3	**23**	66	0	0	0
11	15	0	2	2	**24**	115	24	16	15
12	23	0	3	2	**25**	68	0	0	0
13	7	0	0	0	**26**	13	0	10	2

Tabela E.2 – Número de associações de interação cobertas por granularidade para as versões do conjunto R_x, nos casos de teste não reveladores de defeito.

Ident.	Rel	Tup	Atr	VAtr	Ident.	Rel	Tup	Atr	VAtr
1	10	0	0	0	**14**	21	0	4	10
2	5	0	0	0	**15**	15	0	4	6
3	8	0	4	0	**16**	5	1	3	1
4	3	0	0	14	**17**	0	0	0	0
5	3	0	1	15	**18**	65	10	9	6
6	3	0	0	14	**19**	0	0	0	0
7	0	0	0	1	**20**	0	0	0	0
8	23	3	3	4	**21**	23	0	1	0
9	21	0	7	5	**22**	23	0	1	0
10	12	0	2	1	**23**	91	16	9	8
11	18	0	8	4	**24**	27	4	0	1
12	21	0	6	4	**25**	85	12	7	8
13	3	0	2	0	**26**	10	5	1	0

Tabela E.3 – Ocorrências de cobertura de associações de interação em todas as versões de R_x, nos casos de teste reveladores de defeito.

Ident.	Rel	Tup	Atr	VAtr	Ident.	R	T	A	VAtr
1	16	2	13	11	**38**	3	0	0	0
2	0	0	0	0	**39**	0	0	0	0
3	8	0	10	17	**40**	7	0	0	0
4	1	0	0	0	**41**	1	0	0	0
5	14	0	16	24	**42**	0	0	0	0
6	0	0	4	4	**43**	0	0	0	0
7	0	0	0	0	**44**	3	0	0	0
8	4	0	4	4	**45**	0	0	0	0
9	0	0	0	0	**46**	9	0	3	0
10	0	8	0	0	**47**	0	0	0	0
11	10	9	0	0	**48**	28	0	11	1
12	0	0	0	0	**49**	0	0	0	0
13	15	10	0	0	**50**	2	0	6	0
14	4	0	5	4	**51**	0	0	0	0
15	0	0	0	0	**52**	1	0	5	1
16	6	0	3	3	**53**	0	0	0	0
17	0	0	0	0	**54**	14	6	4	4
18	0	8	0	0	**55**	0	0	0	0
19	12	9	0	0	**56**	16	5	4	4
20	0	0	0	0	**57**	0	0	0	0
21	18	9	0	0	**58**	7	0	0	0
22	23	0	4	0	**59**	0	0	7	6
23	0	0	0	0	**60**	0	0	0	0
24	61	0	7	0	**61**	8	0	9	8
25	7	0	0	0	**62**	16	12	0	0
26	9	0	4	0	**63**	0	0	0	0
27	0	0	0	0	**64**	19	11	0	0
28	14	0	2	0	**65**	0	0	0	0
29	0	0	0	0	**66**	0	0	0	4
30	4	0	0	0	**67**	4	0	9	8
31	0	0	0	0	**68**	0	0	0	0
32	11	0	2	0	**69**	6	0	8	8
33	1	0	0	0	**70**	67	0	0	0
34	0	0	0	0	**71**	110	0	0	0
35	0	0	0	0	**72**	104	0	0	0
36	2	0	2	0	**73**	185	0	0	0
37	0	0	0	0	**74**	184	0	0	0

Tabela E.4 – Ocorrências de cobertura de associações de interação em todas as versões de R_x, nos casos de teste não reveladores de defeito.

Ident.	Rel	Tup	Atr	Vatr	Ident.	Rel	Tup	Atr	VAtr
1	2	3	10	8	**38**	1	1	0	0
2	0	0	0	0	**39**	0	0	0	0
3	0	0	7	11	**40**	0	2	0	1
4	0	0	0	0	**41**	0	0	0	0
5	0	0	19	15	**42**	0	2	0	0
6	0	0	4	3	**43**	0	0	0	0
7	0	0	0	0	**44**	0	1	0	0
8	0	0	1	2	**45**	0	0	0	0
9	0	0	0	0	**46**	2	0	0	8
10	0	0	0	0	**47**	0	0	0	0
11	10	5	0	0	**48**	2	0	0	12
12	0	0	0	0	**49**	0	0	0	0
13	10	6	0	0	**50**	0	0	0	6
14	0	0	2	2	**51**	0	0	0	0
15	0	0	0	0	**52**	0	0	0	6
16	0	0	0	2	**53**	0	0	0	0
17	0	0	0	0	**54**	10	6	0	0
18	0	0	0	0	**55**	0	0	0	0
19	10	4	0	0	**56**	12	5	0	0
20	0	0	0	0	**57**	0	0	0	0
21	10	5	0	0	**58**	0	0	0	0
22	12	0	4	0	**59**	0	0	8	4
23	0	0	0	0	**60**	0	0	0	0
24	24	0	1	6	**61**	0	0	4	3
25	0	0	0	0	**62**	10	6	0	0
26	3	0	0	3	**63**	0	0	0	0
27	0	0	0	0	**64**	11	5	0	0
28	0	0	1	3	**65**	0	0	0	0
29	0	0	0	0	**66**	0	0	0	0
30	1	0	1	0	**67**	0	0	4	5
31	0	0	0	0	**68**	0	0	0	0
32	2	0	1	0	**69**	0	0	4	2
33	0	0	0	0	**70**	44	0	0	0
34	0	0	1	0	**71**	56	0	0	0
35	0	0	0	0	**72**	54	0	0	0
36	0	0	0	0	**73**	105	0	0	0
37	0	0	0	0	**74**	104	0	0	0