Teste estrutural de integração de programas orientados a aspectos: Uma abordagem baseada em conjuntos de junção para AspectJ

Otávio Augusto Lazzarini Lemos

Orientador: Prof. Dr. Paulo Cesar Masiero

São Carlos/SP
Fevereiro/2009

A Ana Luísa, Ana Luíza, Marco, Julio, Mariana e Gabriel.

Agradecimentos

Agradeço aos meus pais e família pelo carinho, amor e formação que me dedicaram e continuam dedicando; à Ana Luísa por estar sempre ao meu lado, com muito amor e carinho; aos pais da Ana, Walter e Virginia, por todo o carinho e apoio dados; e ao meu orientador Paulo Cesar Masiero pela dedicada atenção e perseverante auxílio durante os projetos de mestrado e doutorado.

Aos amigos antigos e novos do LabES, professores, alunos e agregados: Sandro, Fabiano, Marcelo Eler, Marco Aurélio, André Endo, André Domingues, Marllos, Rodrigo Gondim, Jorge, Paulo, Rodolfo, Erika, Rafael Messias, Valter, Aretha, Alessandra, Leandro, Vanessa, André Abe, Paulo Gabriel, Reginaldo, Katia, Vânia, Ivan, Rogério, Renata, Maldonado, Simone e Paulo, Adenilso, Auri, Elisa, Rosângela, Ellen, Rosana e Delamaro.

Aos amigos de sempre: Leonardo Cyrino, Wilson, Ricardo Menotti, Emílio, Rodrigo Pedra, Fernando Molento, Luciano, Luis Sanhermelando, Maurício, Luis Fernando, José Luiz, Alexandre, João, José, Claudinei, Fábio, Guilherme, Edson, Paulo Henrique, Tadeu, Rafael, Renato, Ícaro, Vinícius, Leopoldo, Guto Costa, Bruno Freitas, Bruno Feres, Rodrigo Calvo, Débora e Renan; e a todos os outros que esqueci no momento mas que são não menos importantes.

Aos amigos da UCI: Cristina Lopes, Sushil Bajracharya, Joel Ossher, Wiwat, Pierre Baldi, e Ricardo Morla.

Agradeço também a todos os funcionários e professores do ICMC, e a todas as pessoas que de alguma maneira contribuíram para a realização deste trabalho.

Agradeço à FAPESP pelo auxílio financeiro.

*O que o homem precisa de verdade não é um estado sem tensões
mas sim a luta e o empenho por um objetivo digno dele.*
(V. Frankl)

Sumário

Lista de Figuras	v
Lista de Tabelas	ix
Resumo	xi
Abstract	xiii
1 Introdução	1
1.1 Contextualização e Motivação	1
1.2 Objetivo	2
1.3 Organização	3
2 Fundamentos da Programação Orientada a Aspectos	5
2.1 Considerações Iniciais	5
2.2 Visão Histórica, Terminologia e Conceitos Básicos	6
2.3 A Linguagem AspectJ	8
2.3.1 Combinação de Adendos	11
2.4 Considerações Finais	14
3 Fundamentos do Teste de Software	15
3.1 Considerações Iniciais	15
3.2 Terminologia e Conceitos Básicos	15
3.3 Fases de Teste	18
3.4 Técnicas e Critérios de Teste	19
3.4.1 Teste Funcional (Caixa Preta)	19
3.4.2 Teste Estrutural (Caixa Branca)	20
3.4.3 Teste Baseado em Defeitos	24
3.4.4 Teste Baseado em Modelos	26
3.5 Ferramentas de Teste	28
3.5.1 Teste Estrutural	28
3.5.2 Teste de Mutação	30

3.6	Estudos de Custo e Eficácia de Critérios de Teste	30
3.6.1	Estudos Teóricos	31
3.6.2	Estudos Experimentais	33
3.7	Considerações Finais	36
4	Teste de Programas Orientados a Objetos	37
4.1	Considerações Iniciais	37
4.2	Fases do Teste de Programas OO	38
4.3	Abordagens de Teste de Programas OO	39
4.3.1	Teste Baseado em Especificação	39
4.3.2	Teste Estrutural	40
4.3.3	Teste Baseado em Defeitos	52
4.4	Ferramentas de Teste para Programas OO	52
4.4.1	Ferramenta JaBUTi – <i>Java Bytecode Understanding and Testing</i>	54
4.5	Considerações Finais	55
5	Teste de Programas Orientados a Aspectos	57
5.1	Considerações Iniciais	57
5.2	Fases do Teste de Programas OA	58
5.3	Abordagens de Teste de Programas OA	60
5.3.1	Teste Estrutural	60
5.3.2	Teste Baseado em Defeitos	70
5.3.3	Teste Baseado em Modelos	71
5.3.4	Outras abordagens	73
5.4	Considerações Finais	75
6	Teste Estrutural de Programas OA Baseado em Conjuntos de Junção	77
6.1	Considerações Iniciais	77
6.2	Fundamentos do Teste Baseado em Conjuntos de Junção	78
6.2.1	Exemplos	78
6.3	Grafo de Fluxo Baseado em Conjuntos de Junção	81
6.4	Critérios de Teste	86
6.4.1	Critérios de Fluxo de Controle	86
6.4.2	Critério de Fluxo de Dados	88
6.4.3	Relação de Inclusão entre os Critérios	99
6.5	Estratégia Básica de Teste	102
6.6	Limitações	103
6.7	Considerações Finais	104
7	Automação do Teste Estrutural Baseado em Conjuntos de Junção	105
7.1	Considerações Iniciais	105
7.2	Extensão da JaBUTi para Apoio ao Teste Estrutural Baseado em Conjuntos de Junção	106
7.2.1	Identificação de Pontos de Junção para Adendos Específicos	106
7.2.2	Geração do Grafo <i>PCDU</i>	107

7.2.3	Implementação dos Critérios Baseados em Conjuntos de Junção	107
7.2.4	Implementação do Ambiente de Teste Baseado em Conjuntos de Junção	108
7.2.5	Esforço de Implementação	108
7.3	Exemplos de Uso da Ferramenta JaBUTi/PC-AJ	109
7.4	Limitações	118
7.5	Considerações Finais	119
8	Avaliação da Abordagem de Teste Estrutural Baseado em Conjuntos de Junção	121
8.1	Considerações Iniciais	121
8.2	Avaliação do Custo de Aplicação dos Critérios Baseados em Conjuntos de Junção	122
8.2.1	Estudo Teórico sobre o Custo dos Critérios de Fluxo de Controle	122
8.2.2	Estudo Experimental sobre o Custo dos Critérios de Fluxo de Controle	125
8.2.3	Estudo Teórico sobre o Custo do Critério de Fluxo de Dados	128
8.3	Avaliação da Eficácia dos Critérios Baseados em Conjuntos de Junção	131
8.3.1	Caracterização do Estudo	133
8.3.2	Resultados	135
8.3.3	Limitações do Estudo	136
8.4	Considerações Finais	136
9	Conclusão	137
9.1	Considerações Finais	137
9.2	Contribuições	138
9.2.1	Grafo <i>PCDU</i>	138
9.2.2	Família de Critérios Baseada em Conjuntos de Junção	138
9.2.3	A Ferramenta JaBUTi/PC-AJ	139
9.2.4	Estudos de Avaliação da Abordagem Proposta	139
9.3	Trabalhos Futuros	139
	Referências Bibliográficas	141

Lista de Figuras

2.1	Interesse transversal de atualização da tela (adaptado do trabalho de Elrad <i>et al.</i> (2001)).	7
2.2	Exemplo de código em AspectJ.	9
2.3	Exemplo de aspecto escrito em AspectJ com introduções, DCJs e adendos. . . .	12
2.4	Exemplo do <i>bytecode</i> gerado para o adendo posterior que afeta o DCJ <i>mu-</i> <i>dancaEstado</i>	12
2.5	Método <i>move</i> da classe <i>Linha</i> antes e depois do processo de combinação. . . .	13
3.1	Grafo Def-Uso do método para validação de identificadores.	21
3.2	Exemplo de diagrama para uma MEF (extraído do trabalho de Chow (1978)). .	27
3.3	Hierarquia dos critérios de Rapps e Weyuker (1982) e Potenciais-Usos (Maldo- nado, 1991).	32
3.4	Formato de grafo que eleva o custo de aplicação dos critérios de fluxo de dados ao pior caso (adaptado do trabalho de Weyuker (1984)).	34
3.5	Outro formato de grafo (adaptado do trabalho de Weyuker (1984)).	34
4.1	Modelo FREE para uma classe de contas bancárias (adaptado do trabalho de Xu <i>et al.</i> (2005)).	40
4.2	Implementação parcial da classe <i>SymbolTable</i>	43
4.3	Grafo de chamadas de classe para a classe <i>SymbolTable</i> (adaptado do traba- lho de Harrold e Rothermel (1994)).	43
4.4	Grafo de chamadas de classe para a classe <i>SymbolTable</i> englobado por <i>frame</i> (adaptado do trabalho de Harrold e Rothermel (1994)).	44
4.5	Grafo de fluxo de controle de classe para a classe <i>SymbolTable</i> (adaptado do trabalho de Harrold e Rothermel (1994)).	45
4.6	Tarefas realizadas pela JaBUTi (adaptado da tese de Vincenzi (2004)).	54
5.1	Exemplo de um programa orientado a aspectos escrito em AspectJ.	64
5.2	Parte do <i>bytecode</i> do método <i>affectedMethod</i>	65
5.3	<i>AODU</i> do método <i>affectedMethod</i> com a presença do aspecto <i>AnAspect</i>	65
5.4	<i>PWDU</i> do método <i>pmf</i> sendo afetado pelo adendo <i>printOperands</i>	67
5.5	Classe de defeitos em DCJs.	69

5.6	Combinação de modelos de estados no trabalho de Xu e Xu (2006).	72
5.7	Exemplo de grafo gerado a partir da abordagem proposta por Xu e Rountev (2008).	75
6.1	Parte de uma implementação em AspectJ do sistema de pagamento.	80
6.2	Parte de uma implementação em AspectJ do sistema de envio de mensagens. . .	81
6.3	<i>AODUs</i> de duas unidades do sistema de pagamento.	83
6.4	<i>PCDU</i> construído a partir do adendo anterior <code>controleSuperOperacoes</code> e conjunto de junção <code>superOperacoes</code>	84
6.5	Exemplo de um programa simples com adendo de contorno e o <i>PCDU</i> correspondente.	85
6.6	Exemplo de um programa simples com adendo posterior <i>after throwing</i> e o <i>PCDU</i> correspondente.	86
6.7	Exemplo de um programa simples com adendo anterior que afeta métodos polimórficos e o <i>PCDU</i> correspondente.	87
6.8	Exemplo que motiva a utilização de critério de fluxo de dados.	89
6.9	<i>PCDU</i> construído a partir do adendo anterior <code>registra</code> e conjunto de junção <code>envio</code>	98
6.10	Grafos <i>PCDU</i> utilizados como contraexemplos para as provas.	102
7.1	Modelo conceitual simplificado de programas OA adotado nesta tese.	107
7.2	Conjunto de casos de teste adequado para os critérios de teste de unidade, para o sistema de pagamento.	111
7.3	Cobertura do sistema de pagamento para os critérios de unidade implementados na JaBUTi, para o conjunto de testes funcionais.	112
7.4	Requisitos e cobertura do adendo <code>controleSuperOperacoes</code> e conjunto de junção correspondente para o critério todos-nós-CJ.	112
7.5	<i>PCDU</i> do adendo <code>controleSuperOperacoes</code> e conjunto de junção correspondente após a execução de casos de teste de unidade.	113
7.6	Caso de teste para cobrir requisito do critério todos-nós-CJ, e que revela o defeito do conjunto de junção.	114
7.7	Requisitos e cobertura do adendo <code>controleSuperOperacoes</code> e conjunto de junção correspondente para o critério todos-nós-CJ, após a execução do caso de teste que revela um defeito.	114
7.8	<i>PCDU</i> do adendo <code>controleSuperOperacoes</code> e conjunto de junção correspondente após a execução do caso de teste que revela um defeito.	115
7.9	Conjunto de casos de teste adequado para os critérios de teste de unidade, para o sistema de envio de mensagens.	116
7.10	Cobertura do adendo <code>registra</code> e conjunto de junção <code>envio</code> , após a execução do conjunto de casos de teste inicial.	116
7.11	<i>PCDU</i> do adendo <code>registra</code> e conjunto de junção <code>envio</code> após a execução de casos de teste iniciais, para o critério todos-usos-CJ.	117
7.12	Casos de teste para cobrir os pares def-uso remanescentes no <i>PCDU</i> do sistema de envio de mensagens que revelam um defeito.	118

8.1	Cobertura de um <i>PCDU</i> e do <i>AODU</i> do adendo correspondente, considerado isoladamente.	123
8.2	Um <i>PCDU</i> genérico construído para um adendo que afeta quatro pontos de junção.	129
8.3	Descrições dos sistemas utilizados no estudo exploratório.	132
8.4	Parte de uma implementação em AspectJ do sistema de música online.	133
8.5	Esquema utilizado no estudo de avaliação de eficácia dos critérios baseados em conjuntos de junção.	134

Lista de Tabelas

2.1	DCJs primitivos da linguagem AspectJ e suas respectivas sintaxes (adaptado do livro de Laddad (2003)).	11
4.1	Diferentes classes de instruções de bytecode e respectivo impacto no fluxo de dados (Vincenzi, 2004).	48
5.1	Resumo dos trabalhos de teste de programas OA.	61
6.1	Requisitos de teste para cada critério de fluxo de controle baseado em conjuntos de junção, para o adendo <code>controleOperacoesSuper</code> e conjunto de junção <code>superOperacoes</code>	88
6.2	Exemplos de sentenças Java genéricas e suas implicações no fluxo de dados. . .	93
6.3	Requisitos de teste do critério todos-usos-CJ para o adendo <code>registra</code> e conjunto de junção <code>envio</code>	100
7.1	Arquivos criados na extensão da JaBUTi/PW-AJ para JaBUTi/PC-AJ.	109
8.1	Dados coletados a partir do teste de nove aplicações OA.	126
8.2	Estimativa de casos de teste para o teste de três aplicações OA de grande porte.	128
8.3	Resultados do estudo de eficácia dos critérios baseados em conjuntos de junção.	135

Resumo

A Programação Orientada a Aspectos (POA) é uma técnica de desenvolvimento que apoia a separação de interesses transversais. Na POA, adendos são aplicados a pontos de junção do sistema por meio de uma construção especial chamada descritor de conjuntos de junção (ou simplesmente conjunto de junção). Esse mecanismo apoia a modularização de comportamentos transversais, entretanto, como as interações adicionadas não ficam explícitas no código-fonte, é difícil assegurar que estão corretas. Para lidar com esse problema, nesta tese é proposta uma abordagem rigorosa de teste estrutural de integração para programas orientados a aspectos. É definido um modelo de fluxo de controle e de dados baseado no *bytecode* Java – chamado Grafo Def-Use baseado em conjuntos de junção (ou *PointCut-based Def-Use graph*, *PCDU*) – que modela as regiões de execução de um programa escrito em AspectJ que são afetadas por um conjunto de junção. Sobre este modelo são definidos três critérios de teste: todos-nós-baseados-em-conjunto-de-junção, todas-arestas-baseadas-em-conjunto-de-junção e todos-usos-baseados-em-conjunto-de-junção, que requerem a cobertura de todos os comandos, condicionais e pares def-uso no contexto de cada ponto de junção selecionado. Para automatizar o uso do modelo e critérios propostos, é implementada uma ferramenta chamada JaBUTi/PC-AJ. Além disso, para validar a abordagem proposta, são conduzidos estudos teóricos e experimentais que procuram avaliar os critérios tanto do ponto de vista do custo de aplicação quanto do ponto de vista da eficácia em encontrar defeitos. Os estudos oferecem indícios da aplicabilidade e da eficácia dos critérios para encontrar defeitos diretamente relacionados com a POA.

Abstract

Aspect-Oriented Programming (AOP) is a promising development technique that supports separation of crosscutting concerns. In AOP, advice is applied to join points in the system through a special construct called pointcut. This mechanism supports the modularization of crosscutting behavior; however, since the added interactions are not explicit in the source code, it is hard to ensure their correctness. To tackle this problem, this thesis proposes a rigorous structural testing approach for aspect-oriented programs. A control and data flow model based on Java bytecode – the PointCut-based Def-Use Graph (*PCDU*) – that models execution regions of a program affected by a pointcut is proposed. On top of this model three testing criteria are defined: all-pointcut-based-advice-nodes, all-pointcut-based-advice-edges, and all-pointcut-based-advice-uses, that require exercising all statements, branches and def-use pairs of each advice at the context of each affected join point. To automate the use of the proposed model and criteria, a tool named JaBUTi/PC-AJ is implemented. Moreover, to validate the proposed approach, both theoretical and experimental studies are presented. These studies provide evidence of the applicability and efficacy of the proposed criteria.

Introdução

1.1 Contextualização e Motivação

Em geral, as técnicas de desenvolvimento de software que obtêm sucesso levam em torno de vinte anos ou mais para passarem do estado da arte para o estado da prática (vide os paradigmas de programação orientadas a procedimentos – POP – e a objetos – POO). Apesar de ser uma técnica intelectualmente controversa, a realidade parece ser um pouco diferente com a Programação Orientada a Aspectos (POA). Segundo Kiczales (2006), com pouco mais de dez anos desde sua concepção, empresas já adotam na prática alguns de seus mecanismos no desenvolvimento de software. Isso demonstra que, apesar de não ser uma ‘bala de prata’ – expressão utilizada por Brooks (1987) indicando que não existem soluções fáceis para resolver problemas essenciais em Engenharia de Software – alguns problemas obtêm soluções elegantes a partir de sua utilização.

Pode-se afirmar que é relativamente fácil desenvolver software; difícil é desenvolver software que funcione (Armour, 2000). As técnicas de desenvolvimento auxiliam, mas não impedem que defeitos sejam introduzidos ao longo do desenvolvimento. De fato, novas abordagens de desenvolvimento podem introduzir novos tipos de defeitos, já que mecanismos que apoiam um maior poder de expressividade podem ser mal utilizados, fazendo com que a técnica se torne um obstáculo ao invés de um auxílio. Um exemplo é o mecanismo de polimorfismo apoiado pela POO: apesar de ser um conceito poderoso que pode auxiliar no desenvolvimento de pro-

gramas mais flexíveis e limpos, pode também dar origem a diferentes tipos de defeitos (Offutt *et al.*, 2001). Para contornar esse problema devem ser utilizadas práticas de garantia de qualidade de software, entre elas o teste de software, que é importante pois, em geral, as outras atividades são insuficientes para descobrir os defeitos introduzidos ao longo do processo de desenvolvimento (Pressman, 2000).

Sendo assim, para que a POA se solidifique como uma técnica de desenvolvimento aplicável, são necessárias abordagens de teste de software efetivas para esse contexto. De fato, como enfatizado por Alexander (2003), o entendimento e teste adequado de programas orientados a aspectos (OA) é um dos principais meios para tornar essa técnica menos custosa na prática. Outros pesquisadores têm chamado a atenção para a dificuldade de entendimento e subsequente teste de programas OA, e que tal problema deve ser explorado antes que a técnica possa ser efetivamente adotada na prática (Bodkin, 2003; Huang, 2000). Além disso, percebe-se o crescente interesse nesse assunto, com trabalhos publicados recentemente (Alexander *et al.*, 2004; Ferrari *et al.*, 2008; Xie *et al.*, 2005; Xu e Xu, 2006; Zhou *et al.*, 2004), e também com a realização de três workshops sobre o tema – o WTAOP (*Workshop on Testing Aspect Oriented Programs*) 2005, 2006 e 2007 – realizados em conjunto com o AOSD 2005 e 2007 (*Aspect Oriented Software Development Conference*) e com o ISSTA 2006 (*International Symposium on Software Testing and Analysis*).

Com isso, levando em conta o fato dessa área não ter sido completamente explorada e a continuação do trabalho de mestrado realizado (Lemos, 2005), abre-se um nicho de pesquisa interessante e importante a ser explorado para a adoção efetiva da POA. De fato, uma abordagem de teste estrutural de integração para programas OA pode possibilitar o aumento da confiança em programas desse tipo e também auxiliar o entendimento das novas construções e do comportamento envolvidos nesses programas.

1.2 Objetivo

O objetivo desta tese é propor uma abordagem de teste estrutural de integração para programas OA com a definição de um modelo e critérios específicos, a automatização de sua aplicação e a avaliação dos critérios propostos. A abordagem é estrutural e foca no mecanismo de descrição de conjuntos de junção da POA e auxilia a aumentar a confiança de que os comportamentos transversais implementados como adendos interagem corretamente com o resto do sistema. Assim, enfatiza-se o teste de integração, já que em trabalhos anteriores tratou-se com mais propriedade o teste de unidade (Lemos *et al.*, 2007).

Os objetivos específicos desta tese podem ser divididos nas seguintes linhas de pesquisa em teste de software:

1. Estudos teóricos: investigação e avaliação de uma abordagem para o teste de programas OA baseada em conjuntos de junção, para aumentar a confiança na integração dos adendos ao resto do sistema. Como objetivos nesta linha, destacam-se:
 - A definição de um modelo de representação dos adendos integrados ao sistema a partir dos conjuntos de junção, para fins de teste de integração de programas OA;
 - A definição de uma família de critérios com base no modelo definido; e
 - O estudo do custo de aplicação dos critérios com base em suas propriedades.
2. Automatização: desenvolvimento de mecanismos de apoio à aplicação do teste estrutural de programas OA baseado em conjuntos de junção. Como objetivo nesta linha, destaca-se:
 - O desenvolvimento de uma ferramenta de apoio à aplicação de critérios de teste de integração para programas OA e que apoie a utilização do modelo adjacente definido.
3. Estudos experimentais: definição e condução de estudos que visem a fornecer evidências da aplicabilidade dos resultados teóricos alcançados. Como objetivos nesta linha, destacam-se:
 - A avaliação do custo de aplicação dos critérios de teste estrutural baseado em conjuntos de junção para programas OA; e
 - A avaliação da eficácia dos critérios de teste estrutural baseado em conjuntos de junção para programas OA.

1.3 Organização

No Capítulo 2 e no Capítulo 3 são apresentados os fundamentos teóricos da POA e do teste de software necessários para o entendimento desta tese. No Capítulo 4 e no Capítulo 5 são apresentados conceitos fundamentais e revisões bibliográficas sobre o teste de programas OO e o teste de programas OA, dando ênfase aos trabalhos mais diretamente relacionados com esta tese. No Capítulo 6 é apresentada uma abordagem de teste estrutural de integração baseada em conjuntos de junção para programas OA e no Capítulo 7 é apresentada a implementação da

abordagem em uma ferramenta de teste. No Capítulo 8 são apresentados estudos de avaliação da abordagem proposta, tanto do ponto de vista de custo de aplicação quanto do ponto de vista da eficácia; e, por fim, no Capítulo 9 são apresentadas as conclusões, as principais contribuições deste trabalho e os trabalhos futuros relacionados com esta tese.

Fundamentos da Programação Orientada a Aspectos

2.1 Considerações Iniciais

Para a proposta de uma abordagem de teste estrutural efetiva para a POA, é necessário ter um conhecimento sedimentado dessa técnica para poder investigar: (1) os tipos de defeitos mais frequentes que se pode inserir no software a partir do seu (mau) uso; (2) como adaptar os critérios de teste tradicionais para explorar esses tipos de defeitos; e (3) novas abordagens para auxiliar a encontrar esses tipos de defeitos, caso as tradicionais não sejam suficientes.

Neste capítulo são apresentados os fundamentos da POA relevantes para esta tese. Na Seção 2.2 é apresentada uma visão histórica juntamente com os conceitos básicos da POA e das linguagens que a apoiam. Na Seção 2.3 são definidos os principais mecanismos da linguagem AspectJ – que estende a linguagem Java para apoiar a POA e é utilizada como base neste trabalho – e, por fim, na Seção 2.4 são apresentadas as considerações finais do capítulo.

2.2 Visão Histórica, Terminologia e Conceitos Básicos

O desenvolvimento sistemático de software requer a condução de várias atividades de engenharia, incluindo especificação, projeto, implementação, teste e manutenção. Essencial para a realização dessas três últimas é a seleção de uma linguagem de programação como veículo de implementação, o que mostra a forte dependência entre a Engenharia de Software (ES) e as linguagens de programação (LP). De fato, as comunidades de ES e LP tem gozado de uma relação simbiótica com impacto visível desde os anos 1970, época em que essas áreas começaram a se distinguir uma da outra (Ryder *et al.*, 2005). Uma ilustração da influência da ES no projeto das LP, por exemplo, é a maneira como as modernas linguagens têm dado suporte ao tratamento de exceções que, segundo Ryder e Soffa (2003), é altamente influenciada pela ênfase dada pela ES à confiabilidade dos sistemas.

Nesse mesmo contexto, pesquisas na Xerox Palo Alto Research Center (PARC) em meados dos anos 1990 iniciavam as investigações que deram início à técnica de desenvolvimento de software orientada a aspectos. Naquela época os pesquisadores não tinham os conceitos e definições bem formados, que foram sendo sedimentados com o tempo, contando com contribuições das duas comunidades de ES e LP. De fato, em determinado momento os pesquisadores fizeram um esforço consciente por colocar o assunto em discussão por essas comunidades, pois se constatou que as pesquisas tangenciavam ambas (Lopes, 2005).

A POA surgiu inicialmente da observação de que certos interesses dos sistemas de software – ou seja, características ou *aspectos* importantes que podem ser analisados separadamente (Dijkstra, 1976) – não se encaixam bem em módulos de programação individuais. Esses tipos de interesse tendem a se espalhar por vários módulos do software e, por isso, têm sido chamados de transversais (*crosscutting*). Exemplos de possíveis interesses transversais são: políticas de sincronização, mecanismos de tolerância a falhas, funcionalidades de qualidade de serviços (*QoS*), controle de acesso, rastreamento da execução de programas e persistência (Elrad *et al.*, 2001). Um interesse transversal que tem sido utilizado extensamente a título de ilustração é apresentado na Figura 2.1. O diagrama de classes representa um sistema para a edição de figuras, no qual existem duas classes concretas de elementos de figura: Ponto e Linha. Quando as coordenadas de um ponto ou os pontos de uma linha são alterados, é necessário que a tela seja atualizada. Dessa forma, o interesse de atualização da tela não pertence a nenhuma das duas classes, mas *entrecorta* ambas e, por isso, pode ser entendido como transversal a essas classes.

Com o tempo foi-se descobrindo que os interesses transversais são parte inerente dos sistemas de software e que devem ser tratados não somente na implementação, mas também em outras fases do desenvolvimento de software. Por exemplo, Clarke e Baniassad (2005) propõem

Figura 2.1: Interesse transversal de atualização da tela (adaptado do trabalho de Elrad *et al.* (2001)).

o desenvolvimento de software orientado a aspectos por meio da utilização de Temas, que inclui também as fases de análise de requisitos e projeto, antes da implementação propriamente dita. Outras abordagens de análise e projeto de software OA tem sido propostas, porém, ainda não existe nenhuma que seja considerada suficientemente madura.

No que diz respeito à implementação – onde realmente se debruçaram as pesquisas iniciais –, foram sendo diferenciados dois tipos de linguagens OA: linguagens de domínio específico e linguagens de propósito geral. As de domínio específico tratam somente de alguns tipos de interesses transversais de um determinado domínio. Alguns exemplos são: COOL (Lopes, 1997) – interesse de *locking*/exclusão mútua; RIDL (Lopes, 1997) – interesse de invocação remota; D²AL (Becker *et al.*, 1998) – computação distribuída; e QuO (Karr *et al.*, 2001) – política para o transporte em rede.

As linguagens de propósito geral oferecem mecanismos linguísticos que supostamente permitiriam implementar qualquer tipo de interesse transversal em módulos separados. Para isso, essas linguagens devem dar suporte a mecanismos para a definição de pontos do sistema nos quais os aspectos – módulos que implementam os interesses transversais – possam alterar a semântica do sistema. Esses pontos em que o sistema é alterado são chamados de pontos de junção e podem ser tanto eventos dinâmicos da execução de um programa quanto regiões estáticas do código, dependendo da linguagem. Assim, uma linguagem orientada a aspectos deve definir modelos de pontos de junção que caracterizam: 1) o que são os pontos de junção para essa linguagem; 2) mecanismos para identificar os pontos de junção; e 3) mecanismos para afetar os pontos de junção. Em AspectJ (Kiczales *et al.*, 2001), por exemplo, existem dois modelos de pontos de junção: um dinâmico, pelo qual se utiliza estruturas similares a métodos para se alterar a semântica nos pontos de junção dinâmicos do sistema; e um estático, pelo qual se introduz estaticamente membros em outros módulos para o propósito de um dado aspecto.

Após a codificação de um programa OA, é necessária a realização de um processo chamado combinação (*weaving*), que une os aspectos com o restante do sistema¹ em um programa executável. Esse processo pode ser realizado tanto em tempo de compilação – também chamada de combinação estática – quanto em tempo de execução – também chamada de combinação dinâmica – de acordo com a abordagem adotada pela linguagem.

Lopes *et al.* (2003) afirmam que os pesquisadores na área de ES e LP estão constantemente buscando formas de expressar programas mais próximos da maneira como os programadores pensam. Como os seres humanos descrevem alguns tipos de ideias de maneira transversal, as pesquisas em POA apenas fizeram com que essa característica natural fosse levada ao desenvolvimento de software, tornando os programas mais expressivos nesse sentido. Por exemplo, em POA pode-se programar frases do tipo “para todos os métodos, chame `Trace.in` antes que eles iniciem a execução, e `Trace.out` depois que eles terminarem a execução”. Sem utilizar a POA, o programador é forçado a quebrar essa sentença em “no método A, chame `Trace.in`; ... chame `Trace.out`; retorne. No método B, etc”. Utilizando a POA, esse tipo de ideia pode ser expressada de forma mais natural, sem que se quebre a sentença. Em síntese, ainda, segundo Lopes *et al.* (2003), o poder da POA está em apoiar um conjunto rico de mecanismos de referência estrutural (‘para todos os métodos’) e temporal (‘antes que...’, ‘depois que...’) que dá suporte ao que se tem nas linguagens naturais.

Na próxima seção é discutida a linguagem de propósito geral AspectJ, que apoia os mecanismos de referência estrutural (por meio dos descritores de conjuntos de junção) e temporal (por meio dos adendos) comentados anteriormente, possibilitando a modularização de interesses transversais. Apesar de existirem outras linguagens OA, inclusive para aplicação em outros contextos – como o da Web (Lemos *et al.*, 2006b) – AspectJ tem sido a mais difundida e utilizada.

2.3 A Linguagem AspectJ

A linguagem AspectJ é uma extensão de Java criada para permitir a POA de maneira genérica, no contexto dessa linguagem. Foi inicialmente desenvolvida no centro de pesquisas da Xerox em Palo Alto, CA, EUA, e posteriormente tornou-se um projeto da comunidade de software livre *Eclipse*. Basicamente, as novas construções de AspectJ consistem em: descritores de conjuntos de junção (DCJs), que identificam conjuntos de (pontos de) junção; adendos, que

¹Esse *restante* do sistema é comumente chamado de *programa* (código ou aplicação) *base*. Programa base nesta tese refere-se à porção do sistema que não pertence ao aspecto sendo tratado, na qual o comportamento transversal pode ser aplicado. Note que o programa base no que diz respeito a um determinado aspecto pode conter tanto classes quanto outros aspectos, nos quais o seu comportamento transversal pode ser aplicado.

alteram a semântica antes, depois ou em lugar de um dado conjunto de junção; construções para afetar estaticamente a estrutura dos módulos básicos do programa (declarações intertipos – ou introduções –) e que alteram a hierarquia de classes; e aspectos, que encapsulam as construções novas e as tradicionais de uma classe Java.

Na Figura 2.2 é mostrada uma implementação em AspectJ para o exemplo de interesse transversal apresentado na seção anterior. O aspecto *AtualizacaoDaTela* se encarrega de atualizar a visualização todas as vezes que alguma figura é alterada. Como as figuras são alteradas a partir dos métodos que alteram as coordenadas dos pontos ou os pontos das retas, os adendos são definidos nas chamadas a esses métodos.

```

1 public class Ponto implements ElementoDeFigura {
2     private int x = 0, y = 0;
3     int getX() { return x; }
4     int getY() { return y; }
5     void setX(int x) { this.x = x; }
6     void setY(int y) { this.y = y; }
7     public void move(int dx, int dy) { x += dx; y += dy; }
8 }
9
10 public class Linha implements ElementoDeFigura {
11     private Ponto p1, p2;
12     Ponto getP1() { return p1; }
13     Ponto getP2() { return p2; }
14     void setP1(Ponto p1) { this.p1 = p1; }
15     void setP2(Ponto p2) { this.p2 = p2; }
16     public void move(int dx, int dy) {
17         p1.move(dx, dy);
18         p2.move(dx, dy);
19     }
20 }
21
22 public aspect AtualizacaoDaTela {
23     public pointcut mudancaEstado() :
24         call(void ElementoDeFigura.move(int,int)) ||
25         call(* Ponto.set*(*)) || call(* Linha.set*(*));
26
27     before(): mudancaEstado() {
28         System.out.println("O estado vai mudar...");
29     }
30
31     after() returning : mudancaEstado() {
32         Tela.atualiza();
33     }
34 }

```

Figura 2.2: Exemplo de código em AspectJ.

Os DCJs identificam os conjuntos de junção por meio de referências estruturais, a partir de alguma propriedade comum aos pontos de junção relevantes para os comportamentos transversais. Por exemplo, dado um comportamento transversal que deve ser aplicado antes de serem feitas chamadas a determinados métodos (como no exemplo), pode ser escrito um DCJ que captura essa propriedade. Os DCJs primitivos capturam propriedades do tipo: acesso de modificação/leitura de atributos (*get* e *set*), execução/chamada a métodos (*call* e *execution*),

etc. A Tabela 2.1 apresenta os DCJs primitivos apoiados pela linguagem AspectJ. Um exemplo de DCJ é o `mudancaEstado` localizado nas linhas 23–25 do código apresentado na Figura 2.2. Esse DCJ identifica todas as chamadas aos métodos `move` de qualquer elemento de figura que recebem dois inteiros como parâmetros e todas as chamadas aos métodos cujos nomes iniciam com ‘set’ das classes `Ponto` e `Linha` e que recebem um parâmetro de nome qualquer. Note que o caracter coringa ‘*’ identifica qualquer sequência de caracteres, com exceção do separador de pacotes ‘.’. A expressão ‘..’ pode ser utilizada para identificar qualquer sequência de caracteres incluindo o separador ‘.’. Além disso, ‘..’ indica também qualquer número de parâmetros (por exemplo, `m1 ( . . )` identifica os métodos com nome `m1` que recebem qualquer número de parâmetros).

Existem também outros tipos de DCJs que são compostos aos primitivos para filtrar pontos de junção de acordo com propriedades do tipo: fluxo de controle em um dado conjunto de junção (`cflow` e `cflowbelow`), checagem de alguma condição dinâmica ou estática (`if`), tipo do objeto corrente no ponto de junção (`this`), tipo do objeto alvo da chamada no ponto de junção (`target`), argumentos passados para o método no qual o ponto de junção ocorreu (`args`), etc. Esses três últimos também podem ser utilizados para acessar essas informações do contexto do ponto de junção (objeto corrente, objeto alvo da chamada e argumentos do método do ponto de junção, respectivamente). Os DCJs primitivos e não primitivos podem ser compostos utilizando os operadores `||` ou `&&` e também podem ser negados utilizando o operador unário `!` (utilizado para não selecionar os pontos de junção capturados por um dado DCJ). Note-se que os DCJs *cortam* novas interfaces transversais no sistema sobre as quais se pode programar por meio dos adendos (Kiczales e Mezini, 2005).

Outro conceito importante é o de sombra estática (SE) de pontos de junção, que se refere às partes do código que representam possíveis pontos de junção dinâmicos. Por exemplo, para um ponto de junção que é uma chamada a um determinado método em determinado ponto da execução, deve existir uma sentença no código que se refere a essa chamada. Essa sentença é a SE do ponto de junção. As SEs podem existir tanto no código-fonte quanto no código-objeto (nesse caso o *bytecode* Java).

Os adendos em AspectJ são blocos de código similares a métodos que podem executar antes (adendo anterior – `before`), depois (adendo posterior – `after`) ou no lugar (adendo de contorno – `around`) dos pontos de junção identificados por um dado DCJ. Dessa forma, todo adendo deve estar atrelado a um DCJ que indica quando ele será executado e, assim, diferentemente dos métodos que são chamados explicitamente, os adendos executam implicitamente nos pontos de junção. Os adendos posteriores podem ser do tipo `after returning`, que executam somente depois do retorno normal dos pontos de junção; `after throwing`, que executam somente quando é lançada uma exceção de determinado tipo a partir dos pontos de

Tabela 2.1: DCJs primitivos da linguagem AspectJ e suas respectivas sintaxes (adaptado do livro de Laddad (2003)).

Tipo	Sintaxe
Execução de método	execution(AssinaturaDeMétodo)
Chamada a método	call(AssinaturaDeMétodo)
Execução de construtor	execution(AssinaturaDeConstrutor)
Chamada a construtor	call(AssinaturaDeConstrutor)
Iniciação de classe	staticinicialization(AssinaturaDeTipo)
Acesso de leitura de atributo	get(AssinaturaDeAtributo)
Acesso de modificação de atributo	set(AssinaturaDeAtributo)
Execução de tratador de exceção	handler(AssinaturaDeTipo)
Iniciação de objeto	initialization(AssinaturaDeConstrutor)
Pré-iniciação de objeto	preinitialization(AssinaturaDeConstrutor)
Execução de adendos	adviceexecution()

junção; e apenas `after`, que executam depois dos pontos de junção independentemente do retorno normal ou excepcional. O nome *advice* – para adendo no Inglês – vem do mecanismo de mesmo nome das linguagens Lisp e MacLisp, pois evitou-se a criação de novos nomes na medida do possível, preferindo-se utilizar conceitos existentes (Lemos, 2004).

Os aspectos podem declarar membros (atributos, métodos ou construtores) como parte de outros tipos (classes ou interfaces), para os propósitos do interesse implementado no aspecto. Além disso, pode-se declarar que determinados tipos implementam novas interfaces ou estendem novas classes. Esses tipos de declarações são chamadas de declarações intertipos – ou introduções – e são utilizadas para implementar comportamentos transversais de maneira estática. Por exemplo, suponha-se que, para a classe que implementa pontos apresentada anteriormente, se quer assegurar que as coordenadas x e y estejam sempre entre 0 e 100. Nesse caso poder-se-ia implementar esse interesse em um aspecto que introduz dois métodos para checar se os valores de x e y estão no intervalo permitido. Na Figura 2.3 é listado o código do aspecto. As introduções de métodos estão localizadas nas linhas 2–7. Esse código demonstra também exemplos de outros DCJs e adendos (linhas 8–16).

2.3.1 Combinação de Adendos

Para ser compatível com qualquer máquina virtual Java, o compilador da linguagem AspectJ gera, a partir do código-fonte ou *bytecode* das classes e código-fonte dos aspectos, *bytecode* Java comum, obtido por meio da combinação de classes e aspectos. Para que isso seja possí-

```

1  aspect PointAssertions {
2      private boolean Ponto.assertX(int x) {
3          return (x <= 100 && x >= 0);
4      }
5      private boolean Ponto.assertY(int y) {
6          return (y <= 100 && y >= 0);
7      }
8      before(Ponto p, int x): target(p) && args(x) && call(void setX(int)) {
9          if (!p.assertX(x)) {
10             System.out.println("Illegal value for x"); return;
11         }
12     }
13     before(Ponto p, int y): target(p) && args(y) && call(void setY(int)) {
14         if (!p.assertY(y)) {
15             System.out.println("Illegal value for y"); return;
16         }
17     }
18 }

```

Figura 2.3: Exemplo de aspecto escrito em AspectJ com introduções, DCJs e adendos.

vel, em *bytecode* os aspectos são transformados em classes e os adendos são transformados em métodos comuns. Os parâmetros passados para esses métodos são os mesmos parâmetros dos adendos, possivelmente acrescentados das variáveis que contêm informações reflexivas (por exemplo, o `thisJoinPoint`). O corpo do método correspondente é o mesmo do adendo, com exceção do tratamento especial que é dado para os adendos de contorno, quando é encontrada a palavra-chave `proceed()`, utilizada para se prosseguir com o ponto de junção (Hilsdale e Hugunin, 2004).

Na Figura 2.4 é mostrado o exemplo de adendo posterior que atualiza a tela, apresentado no início da seção, e sua respectiva compilação em *bytecode*. O compilador da linguagem AspectJ cria a classe `AtualizacaoDaTela` referente ao aspecto e o método `ajc$afterReturning$AtualizacaoDaTela$2$c177150e` dentro da classe, referente ao adendo posterior.

```

after() returning : mudancaEstado() {
    Tela.atualiza();
}

public void ajc$afterReturning$AtualizacaoDaTela$2$c177150e ()
0: invokestatic #38 <Method void Tela.atualiza()>
3: return

```

Figura 2.4: Exemplo do *bytecode* gerado para o adendo posterior que afeta o DCJ `mudancaEstado`.

A implementação da combinação dos adendos é feita da seguinte maneira: primeiramente o compilador identifica as SEs de possíveis pontos de junção no *bytecode* e, de maneira geral,

insere uma chamada ao método correspondente ao adendo antes, depois ou em vez da SE, de acordo com o tipo do adendo, se é anterior, posterior ou de contorno. Além disso, como alguns pontos de junção só podem ser resolvidos em tempo de execução, para esses casos são adicionados alguns resíduos no *bytecode* para realizar as checagens dinâmicas necessárias. Um exemplo de resíduo pode ser uma instrução condicional *if* que checa o tipo de um determinado parâmetro, colocada antes da chamada ao método correspondente a um adendo, quando a execução do adendo está condicionada ao tipo do parâmetro recebido (Hilsdale e Hugunin, 2004). Na Figura 2.5 é apresentado o *bytecode* do método `move` da classe `Linha` (linhas 16–19 na Figura 2.2) antes e depois da combinação (as operações inseridas pelo compilador/combinador AspectJ estão deslocadas na listagem). Como os adendos do aspecto `AtualizacaoDaTela` afetam as chamadas ao método `move` da classe `Ponto`, observa-se que o combinador insere chamadas aos métodos correspondentes aos adendos anterior e posterior antes e depois dessas chamadas (identificadas na listagem por SE – sombra estática – 1 e 2). Dessa forma, quando o programa executa, os comportamentos transversais são executados nos pontos de junção referentes a essas SEs. As chamadas ao método `aspectOf()` são feitas para se obter a instância do aspecto correspondente para que se possa fazer a chamada ao método referente ao adendo.

```

void move(II)V
0:   aload_0
1:   getfield      Linha.p1 LPonto;
4:   iload_1
5:   iload_2
6:   invokevirtual Ponto.move (II)V // SE 1
9:   aload_0
10:  getfield      Linha.p2 LPonto;
13:  iload_1
14:  iload_2
15:  invokevirtual Ponto.move (II)V // SE 2
18:  return

void move(II)V
0:   aload_0
1:   getfield      Linha.p1 LPonto;
4:   iload_1
5:   iload_2
6:   invokestatic   AtualizacaoDaTela.aspectOf()LAtualizacaoDaTela;
9:   invokevirtual   AtualizacaoDaTela.ajc$before$aspects_AtualizacaoDaTela$1$c177150e()V
12:  invokevirtual   Ponto.move (II)V // SE 1
15:  invokestatic   AtualizacaoDaTela.aspectOf()LAtualizacaoDaTela;
18:  invokevirtual   AtualizacaoDaTela.ajc$afterReturning$aspects_AtualizacaoDaTela$2$c177150e()V
21:  nop
22:  aload_0
23:  getfield      Linha.p2 LPonto;
26:  iload_1
27:  iload_2
28:  invokestatic   AtualizacaoDaTela.aspectOf()LAtualizacaoDaTela;
31:  invokevirtual   AtualizacaoDaTela.ajc$before$aspects_AtualizacaoDaTela$1$c177150e()V
34:  invokevirtual   Ponto.move (II)V // SE 2
37:  invokestatic   AtualizacaoDaTela.aspectOf()LAtualizacaoDaTela;
40:  invokevirtual   AtualizacaoDaTela.ajc$afterReturning$aspects_AtualizacaoDaTela$2$c177150e()V
43:  nop
44:  return

```

Figura 2.5: Método `move` da classe `Linha` antes e depois do processo de combinação.

É importante observar que essa é apenas uma das possíveis soluções para a implementação do modelo da linguagem AspectJ (uma outra opção, por exemplo, seria alterar a máquina vir-

tual Java para detectar os pontos de junção em tempo de execução, e invocar os adendos nos momentos apropriados). Note-se que não se pode confundir a solução de implementação com o conceito utilizado por trás. Uma ideia errada comumente concebida é a de que os aspectos ‘se inserem no código base’ quando, na verdade, esse é apenas um tipo de solução de implementação. De fato, os interesses transversais implementados nos aspectos devem ser vistos como parte integrante do sistema, e não como algo que o afeta ‘externamente’. A diferença é que quando esses interesses são implementados utilizando a POA, eles ficam isolados no código dos aspectos, ao passo que quando implementados sem a POA, eles ficam espalhados no código dos outros módulos.

2.4 Considerações Finais

A POA tem atraído muita atenção das comunidades de ES e LP, com diversas publicações sobre o assunto nos últimos anos e algumas evidências de utilização na indústria. Porém, é necessário observar que existem algumas limitações no seu uso: por exemplo, alguns pesquisadores discutem que os aspectos dificultam o entendimento do comportamento global de um sistema. Essa dificuldade se deve ao fato de que os interesses implementados em aspectos podem afetar diversas partes do sistema, e que isso não fica claro observando o sistema apenas localmente².

Outro problema encontrado é que, muitas vezes, existem interesses transversais que não podem ser implementados separadamente nem mesmo com o uso da POA. Por exemplo, Sullivan *et al.* (2002) apresentam um exemplo de interesse transversal que não é implementável em um aspecto, utilizando a linguagem AspectJ.

Em suma, é importante considerar que a quantidade de discussões que o surgimento da POA tem gerado na comunidade científica deve, no mínimo, contribuir para o melhor entendimento de questões importantes na ES. Além disso, é evidente que o uso da POA possibilita o desenvolvimento de soluções elegantes em diversos casos.

Entretanto, mesmo para os casos em que a POA apresenta boas soluções, continua sendo necessário aumentar a confiança de que o software funciona, pois o simples uso dessa técnica não evita que enganos sejam cometidos pelos desenvolvedores. Isso significa que a atividade de teste de software continua sendo importante.

No próximo capítulo são discutidos os fundamentos do teste de software, necessários para o entendimento da abordagem de teste para programas OA apresentada no Capítulo 6.

²Apesar disso, é interessante notar que Kiczales e Mezini (2005) mostram evidências de que, mesmo sem a utilização da POA, esse tipo de entendimento global do sistema é sempre necessário na presença de interesses transversais. Além disso, os autores também mostram que, nesses casos, a POA facilita o entendimento do sistema em comparação com a implementação OO.

Fundamentos do Teste de Software

3.1 Considerações Iniciais

Neste capítulo são apresentados os principais conceitos de teste de software relevantes para esta tese. Na Seção 3.2 são apresentadas a terminologia e os conceitos básicos de teste de software, definindo como os principais termos são utilizados neste texto. Na Seção 3.3 são abordadas as fases de teste, e na Seção 3.4 são abordadas as principais técnicas de teste, com ênfase na técnica estrutural. Na Seção 3.5 são discutidas algumas ferramentas de teste que implementam critérios das diferentes técnicas apresentadas, e na Seção 3.6 são discutidos os tipos de estudos de avaliação de critérios de teste. Por fim, na Seção 3.7 são apresentadas as considerações finais deste capítulo.

3.2 Terminologia e Conceitos Básicos

Segundo Myers *et al.* (2004), para uma atividade efetiva de teste de software, a adoção de adequadas noções de psicologia (e também economia) aplicadas nesse contexto parece contribuir mais do que considerações tecnológicas. Nesse sentido, de importância vital é a própria definição do termo ‘teste de software’, já que, segundo esse autor, uma das causas primárias desse processo ser frequentemente realizado com pouca qualidade é o fato de muitos iniciarem

com definições psicologicamente equivocadas do termo. Em geral, definições do tipo ‘teste é o processo de demonstrar que erros não estão presentes no programa’ ou ‘o propósito do teste é demonstrar que o programa realiza suas funções corretamente’, contribuem pouco para uma atividade efetiva pois, para aumentar a qualidade do software, é mais interessante que se procure encontrar os defeitos que ele possui do que se começar pela premissa de que eles não estão presentes (tendo-se em conta que um software livre de defeitos é algo praticamente irreal). Sendo assim, uma definição mais adequada é a de teste de software como *o processo de execução de um programa com a intenção de revelar (ou descobrir) defeitos*. Nesse sentido, Armour (2005) afirma que o teste de software é uma atividade essencialmente de *descoberta*, provavelmente mais do que qualquer outra atividade do desenvolvimento de software.

Neste texto, para o restante dos termos, procura-se seguir o padrão IEEE 610.12-1990 (IEEE, 1990) (um glossário padrão de ES) com algumas extensões, quando necessárias. *Defeito* é definido como uma divergência entre o produto de software desenvolvido e o produto supostamente correto; *erro* é definido como um estado interno do produto de software que diverge do estado correto, gerado pela execução de um defeito presente nesse produto; e *falha* é definida como a manifestação externa de um erro quando esse interfere nos comandos que produzem a saída do produto em execução. Seguindo o mesmo padrão, tem-se a definição de *unidade* como um elemento lógico do software separadamente testável; ou ainda, uma parte do software que não pode ser subdividida em outras partes. Os termos módulo, unidade e componente são frequentemente utilizados de forma intercambiável, ou como sub-partes uns dos outros; módulo é utilizado como uma parte do software que engloba as unidades. Por exemplo, em POO, um método pode ser considerado uma unidade, e uma classe um módulo (de fato, nesta tese, *unidade* refere-se aos métodos e adendos; e *módulo* refere-se às classes e aos aspectos). Componente é utilizado como uma parte do software que engloba outros módulos (pode ser tanto a definição de componente utilizada em desenvolvimento de software baseado em componentes (Cheesman e Daniels, 2000) como simplesmente um conjunto de módulos – ou *cluster* – do software). Assim, tem-se três tipos de divisões: unidades como sendo as partes menores e indivisíveis do software; módulos como sendo partes que englobam unidades do software; e componentes ou *clusters* como partes que englobam módulos do software.

Outros termos importantes são módulos ou unidades pseudo-controladoras (*drivers*) e módulos ou unidades pseudo-controladas (*stubs*). Os *drivers* são implementações de software que invocam e, em alguns casos, controlam as unidades em teste, passando os dados necessários. Os *stubs*, por sua vez, consistem em esqueletos ou implementações simplificadas que substituem partes do software que interagem com a unidade em teste¹.

¹Nesta tese decidiu-se por utilizar os termos termos driver e stub ao invés de pseudo-controladores e pseudo-controlados, por serem mais comumente encontrados nos textos da área escritos em Português.

Um caso de teste é um conjunto de entradas, condições de execução, e saídas esperadas de um determinado programa ou unidade, desenvolvido com um objetivo específico (como executar uma determinada parte do software ou verificar a conformidade com um requisito específico). Um caso de teste pode ser definido como um par ordenado $(d, S(d))$ tal que d é um elemento de um determinado domínio D de um programa P ($d \in D(P)$) e $S(d)$ é a saída esperada para uma dada função da especificação, quando d é utilizado como entrada. Uma verificação completa de um determinado programa P poderia ser obtida testando-se P com um conjunto de (casos de) teste T que inclui todos os elementos do domínio. Entretanto, como geralmente o conjunto de elementos do domínio é infinito ou muito grande, torna-se necessária a obtenção de subconjuntos desses casos de teste (ou elementos dos *subdomínios de teste*). É nesse momento que entra a economia do teste de software, como abordada por Myers *et al.* (2004): já que, em geral, é impossível gerar todos os possíveis casos de teste para um dado programa – o tempo e os recursos são limitados –, são necessários *critérios de teste* que auxiliem o testador fornecendo uma maneira sistemática de escolher subconjuntos dos casos de teste.

Essencialmente, existem duas formas de se selecionar elementos dos subdomínios de teste: (1) selecionar casos de teste ‘aleatoriamente’, o chamado ‘teste aleatório’; e (2) estabelecer partições do domínio de teste e selecionar alguns casos de teste dentro de cada subdomínio, o chamado ‘teste de partição’. Cada uma destas estratégias têm suas vantagens (Delamaro *et al.*, 2007b), porém, nesta tese foca-se em critérios dentro da segunda estratégia.

Os critérios de teste podem então serem vistos como *regras* por meio das quais identifica-se quando dados de teste devem ou não estar no mesmo subdomínio de teste. De forma geral são definidos *requisitos de teste* como, por exemplo, executar uma determinada estrutura do programa (*teste estrutural*). A partir daí, os dados que satisfazem esse requisito são considerados como pertencentes ao mesmo subdomínio (Delamaro *et al.*, 2007b).

Os critérios de teste servem também como um método de avaliação dos conjuntos de teste, auxiliando a evidenciar a suficiência da atividade de teste (Frankl e Weyuker, 2000). As técnicas pelas quais os critérios foram concebidos diferem entre si pelo artefato utilizado como base para o método de avaliação e geração de conjuntos de teste.

Limitações

Segundo as definições anteriores sobre critérios de teste, pode-se dizer que existe apenas um critério ideal: aquele que estabelece que cada subdomínio de $D(P)$ é unitário, ou seja, cada dado de teste compõe isoladamente o próprio domínio. Esse é o chamado *teste exaustivo* que, como comentado anteriormente, é impraticável (Delamaro *et al.*, 2007b). Isso é o mesmo que dizer que, na prática, somado a outras limitações teóricas do teste de software, é impossível

mostrar que um programa está correto por meio de testes. É por esse motivo que a definição mais adequada de teste de software é a ‘execução de um programa com a intenção de revelar defeitos’.

Das limitações teóricas que tornam impossível a demonstração de corretude de um programa por meio do teste de software, podem ser citadas as seguintes (principalmente relacionadas com os critérios que levam em conta a estrutura do código dos programas):

- É impossível determinar por meio de algum procedimento se uma determinada sequência de comandos em um programa é executável ou não; e
- é também indecidível se duas rotinas ou dois caminhos do programa computam a mesma função.

Apesar dessas limitações, é importante notar a importância do teste de software. Fica claro que um programa testado a partir de estratégias sistemáticas oferece maior confiança em sua qualidade do que um programa testado de forma não-sistemática ou do que um programa não testado.

3.3 Fases de Teste

Tradicionalmente, a atividade de teste de software é dividida nas seguintes fases:

- Teste de unidade, que visa a identificar defeitos na lógica e na implementação de cada unidade do software, isoladamente;
- Teste de integração, que visa a descobrir defeitos nas interfaces entre as unidades, módulos e componentes durante a integração da estrutura do programa. O teste de integração poderia ser subdividido nos seguintes tipos:
 - Teste de módulo, que visa a encontrar defeitos nas interfaces e interações entre as unidades, dentro de um único módulo;
 - Teste de componente, que visa a encontrar defeitos nas interfaces entre os módulos, dentro de um componente (conjunto de módulos).
- Teste de sistema, que visa a identificar defeitos nas funções e características de desempenho de todo o sistema. Além disso, procura-se verificar se todos os elementos do sistema (por exemplo, hardware, pessoal e bases de dados) combinam adequadamente e se a função/desempenho global do sistema é alcançada.

Essas definições são gerais e podem ser estendidas com peculiaridades em cada contexto. Neste texto, quando necessários, detalhes específicos de cada fase são definidos para cada contexto.

3.4 Técnicas e Critérios de Teste

Nesta seção são apresentadas as principais técnicas de teste existentes, com ênfase na técnica estrutural, base desta tese. Uma técnica de teste é definida como uma abordagem que guia a criação dos casos de teste. Como comentado anteriormente, as diferentes técnicas se diferem pelo artefato utilizado como base para o método de avaliação e geração dos conjuntos de teste. Howden (1986) ainda classifica as técnicas de teste em *teste baseado em especificação* e *teste baseado em programa*, de acordo com a abstração a partir da qual os critérios de teste são derivados. De acordo com esta classificação, tem-se o teste funcional e o teste baseado em modelos como baseados em especificação, e o teste estrutural e o teste baseado em erros como baseados em programa.

3.4.1 Teste Funcional (Caixa Preta)

O principal objetivo da utilização da técnica funcional é encontrar discrepâncias entre o programa e sua especificação externa (Myers *et al.*, 2004) e, para isso, procura-se derivar requisitos de teste que exercitem completamente as funções especificadas (Pressman, 2000). A técnica funcional é chamada também de teste caixa preta, pelo fato de tratar o software como uma caixa com conteúdo desconhecido e da qual só se visualiza o lado externo (dados de entrada e saída). Alguns exemplos de critérios da técnica funcional são:

- *Teste exaustivo*: Como comentado anteriormente, este é o critério de teste funcional mais direto e completo, e consiste no uso de cada condição de entrada como um caso de teste (Adrion *et al.*, 1982; Myers *et al.*, 2004). Conforme observado, o teste exaustivo é geralmente impraticável (considere, por exemplo, o teste de um compilador para o qual o conjunto de entradas é infinito).
- *Particionamento em classes de equivalência*: Este critério consiste em particionar o conjunto de elementos do domínio em um número finito de classes de equivalência, de maneira que um caso de teste utilizando um elemento representativo de uma classe seria equivalente a outro caso teste utilizando qualquer outro elemento da mesma classe. A ideia básica é que se um caso de teste para um dado elemento de uma classe revela um

erro, todos os outros casos de teste para os outros elementos da mesma classe revelariam o mesmo erro. Equivalentemente, se um caso de teste não revela nenhum erro, espera-se também que nenhum outro caso de teste na mesma classe o revelaria (Myers *et al.*, 2004).

- *Análise de valor limite*: Este critério complementa o particionamento em classes de equivalência, exigindo que existam casos de teste para os limites de cada classe de equivalência. A análise de valor limite é muito útil pois geralmente os erros ocorrem nos limites dos domínios de entrada (Pressman, 2000).
- *Grafo causa-efeito*: Este critério explora a fraqueza dos outros critérios funcionais que não combinam as circunstâncias de entrada e apenas exigem o teste com um subconjunto delas (Myers *et al.*, 2004). Para isso verifica o efeito combinado de dados de entrada, identificando as causas (condições de entrada) e os efeitos (ações), unindo-os em um grafo. A partir desse grafo é montada uma tabela de decisão que é utilizada na geração dos casos de teste (Rocha *et al.*, 2001).

3.4.2 Teste Estrutural (Caixa Branca)

A técnica estrutural é vista como complementar à técnica funcional e baseia-se na estrutura interna de um programa para a derivação dos requisitos de teste. Em geral, os critérios dessa técnica utilizam o grafo de fluxo de controle (ou grafo de programa), que representa o fluxo de controle lógico de um programa, utilizando uma notação padrão (Pressman, 2000).

Considerando um programa P , constituído de vários comandos, um bloco consiste em um conjunto de comandos no qual a execução do primeiro comando do bloco acarreta a execução de todos os outros comandos do mesmo bloco, na sequência determinada, sem que haja instruções de desvio para dentro do bloco. Dessa forma, cada comando de um bloco, possivelmente com exceção do primeiro, tem um único predecessor e exatamente um sucessor, exceto possivelmente o último comando. Um grafo de fluxo de controle consiste em uma correspondência entre nós e blocos de um programa, que indica possíveis fluxos de controle entre esses blocos por meio de arestas (Zhu *et al.*, 1997).

No que diz respeito ao grafo de fluxo de controle, alguns conceitos são importantes para o seu uso na técnica estrutural. Dado um grafo de fluxo de controle $G = (N, E, s)$ em que N é o conjunto de nós, E o conjunto de arestas e s o nó de entrada, um *caminho* corresponde a uma sequência finita de nós $(n_1, n_2, \dots, n_k)$, para $k \geq 2$, tal que existam arestas de n_i para n_{i+1} para $i = 1, 2, \dots, k - 1$. Um caminho é dito *simples* se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, forem distintos. Se todos forem distintos, incluindo o primeiro e o último, o caminho é dito *livre de laço*. Os *nós de entrada* e *nós de*

saída correspondem, respectivamente, aos nós que possuem a primeira instrução do programa e aos nós que não possuem nenhum sucessor. Em outras palavras, os nós de saída não possuem nenhuma aresta de saída (Zhu *et al.*, 1997). Um caminho *completo* é um caminho que vai do nó de entrada a um nó de saída do grafo de fluxo de controle.

Outro termo importante no teste estrutural refere-se aos caminhos *não executáveis*. Um caminho não executável é um caminho do grafo de fluxo de controle impossível de ser coberto para qualquer elemento do domínio de entrada. Isso acontece quando são contraditórias as condições lógicas que deveriam ser satisfeitas para que a sequência de nós do caminho fosse executada (Howden, 1986).

A partir do grafo de fluxo de controle, podem ser escolhidos os caminhos a serem executados, com o apoio dos diferentes critérios da técnica estrutural. Na Figura 3.1 é apresentado o código Java de um método para validação de identificadores com o respectivo grafo de fluxo de controle, adicionado também de informações de fluxo de dados.

Figura 3.1: Grafo Def-Use do método para validação de identificadores.

Os critérios de teste estrutural baseiam-se na ideia de que não se pode confiar em um programa P se existem certos caminhos em P que ainda não foram percorridos por nenhum caso de teste. Esses critérios geralmente associam um conjunto T de casos de teste (que contém um subconjunto das entradas do programa) com um conjunto Π de caminhos no grafo de fluxo de

controle de P , que são percorridos quando esses casos de teste são executados. T satisfaz o critério C para P se e somente se todo caminho requerido por C é um subcaminho de um dos caminhos de Π (Rapps e Weyuker, 1982). Nesse caso, diz-se que T é *C-adequado para o teste de P* .

Critérios baseados na complexidade

Os critérios baseados na complexidade utilizam informação da complexidade dos programas para estabelecer requisitos de teste (Barbosa *et al.*, 2007). O critério mais conhecido dessa classe é o critério de McCabe (1976), que utiliza a complexidade ciclomática do grafo de fluxo de controle para derivar os requisitos de teste.

A complexidade ciclomática é uma métrica que estabelece uma medida quantitativa da complexidade lógica dos programas. O valor dessa métrica mede diretamente o número de caminhos linearmente independentes do código-fonte de um programa, oferecendo uma medida de pior caso para o número de casos de teste necessários para a cobertura dos critérios de fluxo de controle. A complexidade ciclomática M pode ser computada de três maneiras:

1. O número de regiões de um CFG, que é computado contando-se todas as áreas delimitadas pelas arestas e nós do grafo e a área não delimitada fora do grafo; ou
2. $M = E - N + 2$, tal que E é o número de arestas do grafo e N é o número de nós. Se o programa tem múltiplos pontos de retorno, a fórmula é modificada como $M = E - N + 1 + R$, onde R é o número de retornos do programa; ou
3. $M = P + 1$, tal que P é o número de nós predicativos contidos no CFG, ou seja, nós que contêm instruções condicionais.

Segundo Pressman (2000), a métrica de McCabe oferece uma medida quantitativa do esforço necessário para conduzir uma atividade de teste (principalmente estrutural), já que os valores para essa métrica aumentam de acordo com o número de caminhos que envolvem decisões e laços dentro de um programa.

Critérios de fluxo de controle

Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou condicionais, para determinar as estruturas que precisam ser cobertas pelos casos de teste. Dos critérios baseados no fluxo de controle do programa, os mais conhecidos são os seguintes:

- *todos-nós*: Este critério exige que cada comando do programa seja executado ao menos uma vez, ou seja, que cada nó do grafo de fluxo de controle seja coberto. Em geral esse é um critério tido como fraco pois, na prática, não revela nem mesmo a existência de defeitos mais simples (Myers *et al.*, 2004).
- *todas-arestas*: Este critério requer que cada desvio do fluxo de controle do programa seja exercitado pelo menos uma vez, ou seja, que cada aresta do grafo seja coberta. No grafo de fluxo de controle/dados para o método de validação de identificadores apresentado na Figura 3.1, é fácil ver que, para este caso, a cobertura de todos os nós do grafo não implica na cobertura de todas as arestas. Por exemplo, pode-se ter casos de teste que passem pelos caminhos (1, 2), (1, 3, 4), (1, 3, 5, 6, 7, 8, 9, 6, 10, 11) e (1, 3, 5, 6, 10, 12) e que cobrem todos os nós, porém não exercitem a aresta 7 – 9.
- *todos-caminhos*: Este critério requer que todos os caminhos possíveis do grafo de fluxo de controle sejam executados. Ele é o critério mais exigente do teste estrutural e o número de requisitos de teste gerados para ele, mesmo para um programa simples, pode ser muito grande (possivelmente infinito) (Myers *et al.*, 2004). Na Figura 3.1, por exemplo, em decorrência do laço (6, 7, 8, 9, 6), tem-se infinitos caminhos no grafo.

Critérios de fluxo de dados

Além dos critérios baseados em fluxo de controle, existem ainda os critérios baseados na análise de fluxo de dados. Esses critérios baseiam-se na ideia de que não se pode supor que um valor atribuído a uma variável está correto, se não existe nenhum caso de teste que execute um caminho da atribuição (ou *definição*) até um *uso* desse valor (Frankl e Weyuker, 1988). Uma outra motivação para esses critérios foi a evidência de que os critérios de fluxo de controle não eram eficazes nem para revelarem a presença de defeitos simples e triviais, mesmo para programas pequenos (Barbosa *et al.*, 2007). Nesse sentido, os critérios de fluxo de dados oferecem uma hierarquia de critérios mais rigorosos, que se colocam entre o critério todos-caminhos – que em geral é impraticável, e o critério todas-arestas – que pode ser considerado fraco.

Para os critérios baseados no fluxo de dados é necessário a utilização de um grafo Definição-Uso (ou Def-Uso), construído a partir do grafo de fluxo de controle, estendido com informações adicionais sobre as definições e subseqüentes usos das variáveis contidas em um programa (como o grafo apresentado na Figura 3.1). Uma definição de variável ocorre toda vez que um valor é atribuído a uma variável. O uso de variáveis, por sua vez, pode ser de dois tipos: *uso computacional* (c-uso), em que o valor da variável é utilizado em uma computação; e *uso predicativo* (p-uso), em que o valor da variável é utilizado em uma estrutura condicional ou de repetição. A partir daí, se existe alguma sentença em um bloco contendo um c-uso ou

definição de uma variável, adiciona-se essa informação ao nó referente no grafo. Já os p-usos são associados às arestas do grafo, visto que o valor das variáveis afeta a sequência de execução do programa. Um *caminho livre de definição* para uma variável x dos nós i a j , é um caminho $(i, n_1, \dots, n_m, j)$, $m \geq 1$, no qual não há definições de x nos nós $n_1, \dots, n_m$ (Rapps e Weyuker, 1982). Outro conceito importante é o de *par def-uso*, que se refere a um par de definição e subsequente c-uso ou p-uso de uma variável.

O critério mais básico da família de critérios definidos por Rapps e Weyuker (1985) é o critério todas-definições. Entre os critérios dessa família, o critério todos-usos tem sido um dos mais utilizados e investigados. Para o propósito deste texto, podem ser destacados os seguintes critérios baseados no fluxo de dados:

- todas-definições: Requer que, para cada definição de uma variável existente no programa, exista pelo menos um caso de teste que execute um caminho livre de definição para a variável da definição até um c-uso ou p-uso da variável.
- todos-usos: Requer que, para cada associação definição e uso (c-uso ou p-uso) de uma variável, exista pelo menos um caso de teste que execute um caminho livre de definição para a variável, da definição até o uso da variável.
- todos-potenciais-usos (Maldonado, 1991): Requer que, para cada associação definição e potencial uso (c-uso ou p-uso) de uma variável – ou seja, nós e arestas que possam ser alcançados através de pelo menos um caminho livre de definição para a variável relacionada –, exista pelo menos um caso de teste que execute um caminho livre de definição para a variável, da definição até o potencial uso.

3.4.3 Teste Baseado em Defeitos

A técnica de teste baseada em defeitos procura mostrar que determinados defeitos prescritos não estão presentes no programa. É assumido que um programa pode estar incorreto apenas de uma maneira limitada, pela associação de expressões alternativas com expressões do próprio programa. Pela substituição de expressões do programa por expressões alternativas, pode-se gerar diversos programas alternativos, que são potencialmente incorretos. O objetivo do teste baseado em defeitos é produzir um conjunto de testes que diferencia o programa original de cada um de seus alternativos (Morell, 1990).

Um dos critérios mais investigados da técnica baseada em defeitos é o teste baseado em Análise de Mutantes, ou teste de mutação. Esse critério tem um forte relacionamento com o modelo de teste de falha única de circuitos digitais, para a detecção de defeitos lógicos (Acree,

1980; Friedman, 1975). A ideia básica é gerar várias versões do programa original ligeiramente modificado, com o objetivo de revelar os defeitos mais comuns introduzidos nos programas pelos programadores (DeMillo, 1978). Dessa forma, o conjunto de casos de teste também é avaliado quanto à sua capacidade de revelar esses defeitos. Se as saídas obtidas na execução de um mutante são iguais às saídas do programa original, cabe ao testador determinar se o mutante é equivalente ao programa original ou se existe a necessidade de se projetar novos casos de teste que resultem em uma saída diferente para o mutante em questão.

A geração dos mutantes tem como base a *hipótese do programador competente* e a *hipótese do efeito de acoplamento* (DeMillo, 1978). A primeira assume que programas a serem testados estão corretos ou próximos do correto. A segunda, por sua vez, assume que casos de teste capazes de revelar defeitos mais simples também revelam defeitos mais complexos. Um defeito simples está associado a uma alteração sintática simples, enquanto que um defeito complexo pode ser considerado como uma composição de defeitos simples.

Para a aplicação do teste de mutação, dados um programa P e um conjunto casos de teste T , quatro passos são necessários:

1. Geração do conjunto M de mutantes: *Operadores de mutação* são aplicados em P para gerar um conjunto de mutantes M . Os operadores implementam as regras de alteração que são aplicadas no programa original. Ressalta-se que a definição dos operadores de mutação depende da linguagem de programação ou de especificação em que os artefatos a serem testados estão escritos.
2. Execução de P com T : A execução de P com T é realizada e verifica-se se o resultado é o esperado. O processo é encerrado caso alguma saída seja incorreta. Caso isso não ocorra, o próximo passo é realizado. Geralmente cabe ao próprio testador decidir se a saída é ou não a esperada para cada caso de teste, fazendo portanto o papel de oráculo (Weyuker, 1982).
3. Execução dos mutantes de M com T : Cada um dos mutantes em M é executado com os casos de teste em T . Se um mutante m apresenta uma saída diferente de P executado com o mesmo caso de teste, conclui-se que o conjunto de casos de testes é sensível e consegue expor a diferença entre m e P , e m é considerado *morto*. Caso contrário, duas possibilidades são consideradas: ou T é de baixa qualidade, não conseguindo revelar a diferença entre m e P , ou m é *equivalente* a P . Encerrada a execução dos mutantes, o escore (*score*) de mutação pode ser calculado pela seguinte fórmula:

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)}$$

Sendo:

- $ms(P,T)$: escore da mutação (*mutation score*);
- $DM(P,T)$: número de mutantes mortos por T ;
- $M(P)$: número total de mutantes gerados;
- $EM(P)$: número de mutantes equivalentes identificados.

O resultado desse cálculo é sempre um valor no intervalo $[0, 1]$, fornecendo uma medida de adequação de T , sendo que quanto mais próximo de 1 for o resultado, mais adequado é T .

4. Análise dos mutantes: Este é o passo que requer mais intervenção humana. Inicialmente, decide-se se o teste deve ou não continuar, dependendo do escore da mutação. Resultados próximos a 1 sugerem um conjunto de casos de teste T suficientemente bom e pode-se optar por encerrar o teste. Caso decida-se por continuar o teste, cada um dos mutantes m “vivos” é analisado para saber se é ou não equivalente a P .

O problema de determinar a equivalência entre dois programas, nesse caso P e um mutante m , é geralmente indecível. Nesse contexto, embora a automatização completa dessa atividade não seja possível, alguns métodos e heurísticas têm sido propostos para a determinação da equivalência entre P e m (Budd, 1981; Simão e Maldonado, 2000; Vincenzi *et al.*, 2002).

Embora o teste de mutação seja voltado principalmente para o teste de unidade, é possível adaptá-lo para outras fases do teste de software. No trabalho de Delamaro (1997), por exemplo, procura-se modelar defeitos relacionados a erros de integração entre as unidades do programa. Como esses erros estão geralmente relacionados com a interface de uma unidade, o critério foca nessas estruturas e, assim, é chamado de mutação de interface. De acordo com o modelo de defeitos apresentado nesse trabalho, erros de integração são caracterizados por valores incorretos trocados por meio das conexões entre as unidades. Assim, a mutação de interface procura, por meio da injeção de defeitos simples, introduzir perturbações somente em construções relacionadas com a interação entre as unidades (Delamaro *et al.*, 2007a).

3.4.4 Teste Baseado em Modelos

As especificações dos sistemas de software podem ser descritas por meio de modelos formais como Máquinas de Estados Finitos (MEFs). Esse tipo de modelagem pode permitir que o conhecimento sobre o sistema seja capturado e reutilizado durante diversas fases do desenvolvimento. Esses modelos – principalmente os que se pode executar através de simulação – podem

servir como oráculos para o testador, definindo o que seria correto para o programa a partir de uma dada entrada. Além de servir como oráculos, os modelos também podem ser usados para a geração automática de scripts e cenários de teste (Simão, 2007).

As sequências de ações que podem ocorrer durante o uso do sistema especificado são definidas no modelo. Em geral, existe mais de uma opção de ‘próximas ações’ que podem ocorrer em um ponto específico da execução. Algumas técnicas permitem descrever quais são as próximas ações em um grafo, no qual os nós representam as configurações (ou estados) do sistema e as arestas representam as transições entre configurações do sistema. Essas técnicas são denominadas de Máquinas de Transição de Estados (MTEs). Algumas técnicas permitem que os modelos sejam decompostos hierarquicamente, fazendo com que comportamentos complexos sejam decompostos em comportamentos de mais baixo nível e mais simples. Capacidades adicionais desses modelos incluem variáveis e condicionais, fazendo com que as transições possam depender de valores das variáveis ou de pré-condições sobre o estado atual do sistema (Simão, 2007).

O modelo mais simples de MTE são as MEFs, que são utilizadas para exemplificar o teste baseado em modelos nesta tese. Em uma MEF, as transições são representadas explicitamente, pela ligação entre um estado *a* e um estado *b* (Simão, 2007). A cada instante a máquina pode estar em apenas um de seus estados; e em resposta a um evento de entrada, a máquina gera um evento de saída e muda de estado. Tanto o evento de saída quanto o novo estado são gerados unicamente em função do estado atual e do evento de entrada. Uma MEF pode ser representada tanto como um diagrama de transição de estados quanto por uma tabela de transição. Em um diagrama de transição de estados, os estados são representados por nós e as transições são representadas por arestas direcionadas entre estados. Cada aresta é rotulada com a entrada que gera a transição e a saída que é produzida, usando o formato *entrada:saída*. A Figura 3.2 apresenta um exemplo de MEF, extraído do trabalho de Chow (1978), que especifica um extrator de comentários. O conjunto de entrada consiste em uma sequência de símbolos ‘*’, ‘/’, ‘v’, sendo que ‘v’ representa qualquer caractere diferente de ‘*’ e ‘/’. A entrada é uma cadeia de caracteres e apenas os comentários são impressos (usando a sintaxe da linguagem C).

Figura 3.2: Exemplo de diagrama para uma MEF (extraído do trabalho de Chow (1978)).

Como pela própria definição de MEF o conjunto de estados atingíveis é finito, é possível responder quase todas as questões sobre o modelo, o que permite a automatização do processo de validação (Simão, 2007). Entretanto, é importante notar que apenas uma classe limitada de sistemas pode ser modelada a partir de uma MEF; e mesmo para alguns sistemas que podem ser modelados, o modelo resultante pode ser excessivamente grande. Por exemplo, no caso de sistemas concorrentes, a impossibilidade de representação implícita dessa característica leva a explosão combinatória do número de estados da MEF. Para aumentar o poder de modelagem de sistemas, outros modelos que estendem as MEFs foram propostos, como Statecharts (Harel, 1987) e Estelle (Budkowski e Dembinski, 1987).

3.5 Ferramentas de Teste

A aplicação de critérios de teste pode ser considerada impraticável se realizada sem o apoio de ferramentas automatizadas. Isso motiva o desenvolvimento de ferramentas para auxiliar na condução de atividades efetivas de teste de software. Existem diversas ferramentas de teste disponíveis, inclusive ferramentas de código aberto. Por exemplo, no sítio opensourcetesting.org (2009) pode ser encontrada uma lista com dezenas de ferramentas de código aberto para o teste funcional (apesar de usar o termo teste funcional, nem sempre essas ferramentas apoiam a aplicação de critérios de teste funcionais). Para dar uma ideia geral, nesta seção são discutidas algumas ferramentas de teste que foram desenvolvidas, com ênfase nas que apoiam o teste estrutural (Barbosa *et al.*, 2007).

3.5.1 Teste Estrutural

Em geral, as ferramentas de apoio ao teste estrutural possibilitam a análise de cobertura de conjuntos de casos de teste segundo algum critério selecionado; permitem a geração de relatórios de cobertura; e apoiam a visualização dos grafos de fluxo. Algumas dessas ferramentas apoiam também a geração automática das entradas dos casos de teste para satisfazer um determinado requisito de teste do programa. Em geral, esse suporte é baseado na execução simbólica do programa em teste. Apesar da maioria das ferramentas não apoiarem diretamente a geração desses dados de teste, grande parte delas apresenta ao usuário os requisitos de teste exigidos para que os critérios sejam satisfeitos e, assim, auxiliam o usuário na elaboração dos casos de teste (Maldonado, 1991).

As primeiras ferramentas comerciais de teste estrutural desenvolvidas foram a RXVP80 (Deutsch, 1982) e a TCAT (Reisman, 1990), as quais apoiavam a aplicação de critérios estruturais baseados em fluxo de controle. A ferramenta RXVP80, distribuída pela *General Research*

Corporation, realiza a análise de cobertura baseada no critério todas-arestas em programas escritos em Fortran. A ferramenta TCAT (*Test-Coverage Analysis Tool*), distribuída pela *Software Research Corporation*, realiza o teste de unidade segundo o critério de teste de condicionais lógicos. Esse tipo de critério divide os predicados encontrados em uma condição em vários “if’s”, no qual cada “if” contém apenas um dos predicados (Barbosa *et al.*, 2007).

Nos anos 1980, com a introdução dos critérios baseados em análise de fluxo de dados, começaram a surgir ferramentas de apoio à aplicação desses critérios. As ferramentas desenvolvidas eram protótipos de pesquisa construídos em universidades (Frankl, 1987; Herman, 1976; Korel e Laski, 1985; White e Sahay, 1985), e não estão disponíveis comercialmente. De modo geral, a diferença entre essas ferramentas estava na abordagem utilizada para determinação dos requisitos de teste; nas características da interface com o usuário; na maneira como os dados de entrada eram informados; no tratamento de diferentes linguagens; e no gerenciamento das sessões de teste (Barbosa *et al.*, 2007).

A ferramenta ASSET (*A System to Select and Evaluate Tests*) (Frankl e Weyuker, 1985), desenvolvida no contexto da Universidade de Nova York, foi uma das primeiras iniciativas de automatização da aplicação dos critérios de teste estruturais. A ferramenta apoia a aplicação dos critérios baseados em fluxo de dados definidos por Rapps e Weyuker (1982, 1985) no teste de programas escritos na linguagem Pascal (Barbosa *et al.*, 2007).

Outra ferramenta que implementa os critérios baseados em fluxo de dados e também os critérios baseados em fluxo de controle é a Proteste (Price e Zorzo, 1990). Essa ferramenta foi desenvolvida na Universidade Federal do Rio Grande do Sul e consiste em um protótipo de apoio ao teste estrutural de programas escritos em Pascal. Além de Pascal, o ambiente pode ser configurado para o teste de programas escritos em outras linguagens por meio da utilização de uma ferramenta que gera analisadores de código-fonte específicos para cada linguagem (Barbosa *et al.*, 2007).

A PokeTool (*Potential Uses Criteria Tool for Program Testing*) (Chaim, 1991; Maldonado *et al.*, 1989) é uma ferramenta que apoia a aplicação dos critérios Potenciais-Usos definidos por (Maldonado, 1991). Ela foi desenvolvida na FEEC/UNICAMP em colaboração com o ICMC/USP. Inicialmente a ferramenta foi desenvolvida para o teste de unidade de programas escritos em C (Chaim, 1991) e, posteriormente, também foram propostas configurações para o teste de programas escritos em Cobol e Fortran (Fonseca, 1993; Leitão, 1992).

Mais recentemente, várias ferramentas para o teste estrutural de programas OO têm sido desenvolvidas, principalmente para o teste de programas Java e C++. Algumas delas serão discutidas na Seção 4.4.

3.5.2 Teste de Mutação

Se no caso da técnica de teste estrutural a atividade de teste seria dificilmente realizada manualmente, no caso do teste de mutação essa atividade se tornaria praticamente impossível. Isso porque, em geral, a aplicação dos operadores de mutação causa a geração de inúmeros programas mutantes, que devem ser executados um a um para a verificação da cobertura dos casos de teste. De maneira geral, as ferramentas de teste de mutação devem apoiar a aplicação de conjuntos de operadores de mutação; a geração dos mutantes a partir da aplicação desses operadores; a execução dos casos de teste nos mutantes; e a análise de cobertura de conjuntos de casos de teste, a partir da constatação do número de mutantes mortos e vivos (ou seja, os que produzem resultados diferentes ou iguais ao programa original).

A primeira ferramenta desenvolvida para o teste de mutação de que se tem conhecimento a partir da literatura é a FMS.1 (*Fortran Mutation System* - versão 1), desenvolvida na Universidade de Yale. A ferramenta trata de um subconjunto de programas Fortran, mais especificamente de programas com apenas uma rotina que utiliza aritmética de inteiros e sem comandos de entrada e saída. O projeto no qual a FMS.1 foi desenvolvida motivou a implementação de outras ferramentas de teste de mutação mais elaboradas, como a FMS.2 e o CMS.1 (*Cobol Mutation System* - versão 1) (Delamaro *et al.*, 2007a).

Em meados dos anos 1980, no Instituto de Tecnologia da Georgia, foi desenvolvida a ferramenta de teste de mutação Mothra. A ferramenta foi desenvolvida na linguagem C e para o teste de programas Fortran 77 (Offut, 2009).

Nos anos 1990, no contexto do ICMC-USP, foi desenvolvida a ferramenta Proteum (Delamaro, 1993), primeiramente para o teste de programas escritos em C. Mais tarde, a ferramenta foi estendida para o teste de MEFs (Fabbri *et al.*, 1999a), Redes de Petri (Fabbri *et al.*, 1995) e Statecharts (Fabbri *et al.*, 1999b).

Mais recentemente algumas ferramentas foram propostas para o teste de mutação de programas OO. Algumas delas serão discutidas na Seção 4.4.

3.6 Estudos de Custo e Eficácia de Critérios de Teste

Dada a quantidade diversa de critérios disponíveis para o teste de programas, a avaliação de custo de aplicação e eficácia desses critérios e a comparação entre eles sempre foi desejável. Entretanto, esse tipo de estudo é complicado porque cada critério é definido com base em um modelo de programas diferente, e de acordo com diferentes fundamentos teóricos. Assim, os resultados são muitas vezes controversos (Zhu *et al.*, 1997).

Nessas avaliações, é sempre importante definir bem o que se entende por um critério ser *melhor* do que outro, por exemplo. Os três tipos de medida encontrados na literatura são: habilidade na detecção de erros (definido como *eficácia* neste texto), confiabilidade do software e custo de aplicação. Em geral, os seguintes três tipos de métodos são utilizados para esses estudos: análise formal (ou estudos teóricos), experimentação estatística (ou estudos experimentais) e simulação (Zhu *et al.*, 1997). Este texto foca os dois primeiros tipos.

3.6.1 Estudos Teóricos

Em estudos teóricos geralmente se define alguma relação entre os critérios, provando para cada par se a relação é verdadeira ou não. A relação de *inclusão*, por exemplo, evidencia o poder relativo entre os diferentes critérios de teste (Ntafos, 1988). Um critério C_1 inclui um critério C_2 se, para qualquer programa P , qualquer conjunto de casos de teste C_1 -adequado for também C_2 -adequado. Um critério C_1 inclui *estritamente* um critério C_2 , denotado por $C_1 \Rightarrow C_2$, se C_1 inclui C_2 e existe um programa P e um conjunto de casos de teste C_2 -adequado que não é C_1 -adequado. Além disso, se C_1 não inclui C_2 e C_2 não inclui C_1 , C_1 e C_2 são ditos *incomparáveis*. Além da relação de inclusão, outro tipo de estudo aplicado aos critérios são aqueles que avaliam a sua *complexidade*. A complexidade de um critério C é definida como o número máximo de casos de teste requeridos para a adequação ao critério, no pior caso (Souza *et al.*, 2007).

Rapps e Weyuker (1982) mostraram que seus critérios de fluxo de dados formam uma hierarquia de acordo com a relação de inclusão, por essa relação estabelecer uma ordem parcial². O mesmo foi feito por Maldonado (1991), incluindo nessa hierarquia a família de critérios Potenciais Usos. Na Figura 3.3(a) é mostrada a hierarquia formada pelos critérios de Rapps e Weyuker (1982) e de Maldonado (1991).

Geralmente, algumas restrições precisam ser impostas aos programas para que se possa demonstrar a hierarquia de inclusão dos critérios. No caso da família de critérios de Rapps e Weyuker (1982), a análise de inclusão se aplica para programas P que satisfaçam as seguintes propriedades:

- **NSUP – No-Syntactic-Undefined-P-Use Property:** Para cada p-uso de uma variável x em uma aresta (i, j) , em P , existe algum caminho do nó de entrada para a aresta (i, j) , o qual contém uma definição global³ de x .

²Uma relação de ordem parcial é uma relação que possui as propriedades de reflexividade, antissimetria e transitividade.

³Um nó i possui uma definição global de uma variável x se ocorre uma definição de x no nó i e existe um caminho livre de definição de i para algum nó $j \neq i$ ou alguma aresta que contém um c-uso ou um p-uso, respectivamente, da variável x .

(a) Desconsiderando caminhos não executáveis

(b) Considerando caminhos não executáveis

Figura 3.3: Hierarquia dos critérios de Rapps e Weyuker (1982) e Potenciais-Usos (Maldonado, 1991).

- **NSL – No-Straight-Line Property:** P contém ao menos um comando condicional ou de repetição:
 - Pelo menos um nó do grafo de fluxo de controle de P tem mais de um sucessor;
 - Pelo menos uma variável tem um p-uso em P .

Frankl e Weyuker (1988) observaram que, na realidade, na presença de caminhos não executáveis, nenhum dos critérios de fluxo de dados de Rapps e Weyuker (1982) inclui o critério todas-arestas. Já os critérios Potenciais Usos exigem menos restrições aos programas para que a análise de inclusão se aplique. Estes critérios exigem apenas que a propriedade **LDEN – at-Least-one-Definition-in-the-Entry-Node** (ao menos uma definição de variável no nó de entrada) – seja satisfeita. Na Figura 3.3(b) é mostrada a hierarquia de inclusão dos critérios de Rapps e Weyuker (1982) e de Maldonado (1991) na presença de caminhos não executáveis.

No que diz respeito à complexidade, Weyuker (1984) avaliou o custo de aplicação dos critérios de fluxo de controle e de dados propostos por Rapps e Weyuker (1982). Considere P um

programa com n variáveis, m atribuições, i comandos de entrada (como, por exemplo, o comando `cin` da linguagem C++, que lê valores de entrada do teclado e os atribui a variáveis) e t transferências condicionais (ou apenas *condicionais*). Como as definições das variáveis podem ser feitas tanto por meio das atribuições quanto por meio dos comandos de entrada (segundo as definições utilizadas por Weyuker (1984)), o número de definições d do programa pode ser definido como $m + i \leq d \leq m + (i \times n)$ (já que cada comando de entrada pode gerar definições de uma a n variáveis).

Segundo Weyuker (1984), é fácil ver que os critérios todos-nós e todas-arestas requerem, no máximo, $t + 1$ casos de teste. A compreensão desse cálculo é facilitada visualizando o grafo de pior caso apresentado na Figura 3.4 (esse resultado é relacionado à complexidade de McCabe (1976) comentada anteriormente – quanto maior o número de condicionais, maior o número de caminhos requeridos). O critério todos-usos, por sua vez, requer $\frac{1}{4}(t^2 + 4t + 3)$ no pior caso (é portanto, quadrático⁴ – $O(t^2)$). Para compreender essa fórmula, considere o grafo apresentado na Figura 3.4, que maximiza o número de pares def-uso e consequentemente o número de caminhos requeridos para a adequação ao critério. O número total de caminhos nesse grafo é $\frac{1}{2}(t + 1) \times \frac{1}{2}(t + 3)$. Para compreender este número, repare que existem $\frac{1}{2}(t + 1)$ nós na última ‘linha de condicionais’ do grafo, representando no máximo $t + 1$ usos distintos para cada definição no nó ‘raiz’. Desses usos, $\frac{1}{2}(t + 1)$ estão em nós de saída, enquanto $\frac{1}{2}(t + 1)$ estão em nós que podem conter definições, cada uma com usos correspondentes em cada nó na outra ‘metade’ do grafo. Assim, tem-se um total de $\frac{1}{2}(t + 1) \times \frac{1}{2}(t + 1) + \frac{1}{2}(t + 1) = \frac{1}{4}(t^2 + 4t + 3)$ caminhos distintos requeridos pelos pares def-uso.

É importante notar que esse custo é alcançado somente quando se considera o grafo de pior caso mostrado na Figura 3.4 – que é bastante incomum – com definições e usos estrategicamente localizados no grafo (Weyuker, 1984). Quando, por exemplo, se considera o grafo apresentado na Figura 3.5, o custo do critério todos-usos diminui para, no pior caso, $t + 1$ (e note que esse grafo é ainda bastante complexo e incomum). Isso mostra a importância de se considerar a aplicação dos critérios em cada caso específico e da realização de estudos experimentais para avaliar o custo de aplicação dos critérios na prática.

3.6.2 Estudos Experimentais

A realização de estudos experimentais no contexto de critérios de teste é motivado, por exemplo, pelas seguintes questões (Souza *et al.*, 2007):

⁴Maldonado (1991) discute o grafo de pior caso definido por Weyuker (1984), apresentando outro formato de grafo no qual a complexidade do critério todos-usos torna-se exponencial (2^t). Entretanto, é importante notar que ambos casos são incomuns, especialmente no contexto da POO e POA.

Figura 3.4: Formato de grafo que eleva o custo de aplicação dos critérios de fluxo de dados ao pior caso (adaptado do trabalho de Weyuker (1984)).

Figura 3.5: Outro formato de grafo (adaptado do trabalho de Weyuker (1984)).

1. Tendo-se dois critérios de teste (ou duas famílias de critérios de teste), qual é mais difícil de satisfazer?
2. Qual é o custo de aplicação de um critério (ou família de critérios) de teste?
3. Quão bom é um critério (ou família de critérios) para revelar defeitos em um programa?
4. Como diminuir o custo de aplicação de um critério sem diminuir sua eficácia?

Mathur e Wong (1994) ilustram uma situação interessante para motivar o uso de estudos experimentais para a avaliação de critérios de teste. Considere a necessidade de se testar um programa P que é parte de um sistema crítico. Obviamente, o funcionamento correto do sistema depende de P . Assim, P deverá ser testado o quanto for possível, para aumentar a confiança de que está correto. Para isso, o testador decide utilizar vários critérios de teste com o objetivo de avaliar os casos de teste desenvolvidos. Inicialmente, os casos de teste são desenvolvidos para satisfazer um critério C_1 , e, a partir daí, surge a seguinte questão: *tendo obtido um conjunto de casos de teste C_1 -adequado, é possível melhorar a qualidade dos casos de teste utilizando o critério C_2 ?* Essa é uma questão prática que surge diante da dificuldade em se decidir se um programa foi suficientemente testado ou não. Os estudos experimentais podem, dessa forma, auxiliar a responder questões desse gênero.

No contexto de teste de software, a condução adequada de experimentos requer a realização das seguintes atividades (Souza *et al.*, 2007):

- seleção e preparação dos programas a serem utilizados;
- seleção dos participantes do experimento;
- seleção das ferramentas de teste a serem utilizadas;
- geração de conjuntos de casos de teste adequados;
- execução dos programas com os casos de teste gerados; e
- análise dos resultados do experimento.

Como descrito anteriormente, os critérios de teste baseados em fluxo de dados têm complexidade exponencial, o que motiva a realização de estudos experimentais para determinar o custo de aplicação desses critérios do ponto de vista prático (Souza *et al.*, 2007). Diversos estudos experimentais desse tipo foram realizados; por exemplo, Weyuker (1990) caracterizou um *benchmark* para a avaliação de critérios de fluxo de dados, e esse mesmo *benchmark* foi utilizado para uma avaliação experimental dos critérios Potenciais-Usos (Maldonado, 1991).

O teste de mutação também tem sido alvo de experimentos para avaliar o seu custo e habilidade de encontrar defeitos. Por exemplo, Mathur e Wong (1993) compararam dois critérios de mutação alternativos: mutação aleatória (seleciona 10% de mutantes para cada operador de mutação) e mutação restrita (seleciona um subconjunto de operadores de mutação). Segundo os autores, por meio do experimento foi possível obter indícios de que os critérios são igualmente eficazes e geram um número reduzido de mutantes a serem analisados (em comparação ao critério original).

3.7 Considerações Finais

Neste capítulo foram apresentados os fundamentos do teste de software, as ferramentas e estudos de avaliação de critérios, abordando as técnicas tradicionais desenvolvidas com base em diferentes estratégias. Esse embasamento é importante para a compreensão dos próximos dois capítulos, que abordam os temas de teste de programas OO e teste de programas OA e também para o entendimento da abordagem de teste apresentada no Capítulo 6.

No próximo capítulo são apresentadas as principais questões relacionadas com o teste de programas OO, delineando as abordagens adaptadas para esse contexto que são relacionadas com esta tese.

Teste de Programas Orientados a Objetos

4.1 Considerações Iniciais

A análise da aplicação das técnicas de teste comentadas no Capítulo 3 para os paradigmas procedimental e OO é interessante para avaliar o que pode ser utilizado diretamente no contexto deste trabalho, e em que momentos são necessárias adaptações. Neste capítulo são apresentadas as principais questões envolvendo o teste de programas OO e alguns trabalhos que propõem abordagens de teste estrutural para esse contexto. Este assunto é importante pois as linguagens OA em geral são construídas sobre o paradigma OO. Na Seção 4.2 são discutidas as fases de teste aplicadas ao teste de programas OO, e na Seção 4.3 são discutidas algumas abordagens de teste de programas OO, dando ênfase naquelas mais importantes para esta tese. Na Seção 4.4 são apresentadas algumas ferramentas de teste para programas OO e, por fim, na Seção 4.5 são apresentadas as considerações finais do capítulo.

4.2 Fases do Teste de Programas OO

Neste texto, seguindo o trabalho de Vincenzi (2004), considera-se que as unidades de programas OO são os métodos, pois podem ser vistos como as partes indivisíveis do software OO. Além disso, por definição, uma classe engloba um conjunto de atributos e métodos que manipulam esses atributos e, assim, considerando uma única classe já é possível pensar em teste de integração: métodos da mesma classe podem interagir entre si para desempenhar funções específicas, caracterizando uma integração que deve ser testada.

Portanto, com essas considerações e seguindo as definições contidas no capítulo anterior, a atividade de teste de software OO poderia ser dividida nas seguintes fases:

1. Teste de unidade, chamado de intramétodo, que visa a identificar defeitos na lógica e na implementação de cada método.

Uma questão importante no teste intramétodo é a execução dos métodos isoladamente. Sobre esse problema, segundo Harrold e Rothermel (1994), a classe à qual o método pertence pode ser considerada o driver do método, pois sem ela não é possível executá-lo;

2. Teste de integração, que pode ser subdividido nos seguintes tipos:

- Teste de classe, que visa a identificar defeitos na lógica e na implementação de cada classe. O teste de uma classe pode ser dividido nos seguintes tipos:
 - (a) Intermétodos, que consiste no teste de um método juntamente com outros que ele chama direta ou indiretamente dentro da mesma classe (chamadas indiretas são aquelas que ocorrem dentro de métodos chamados por outros em qualquer nível de profundidade);
 - (b) Sequencial¹ (chamado de intraclasses por Harrold e Rothermel (1994)), que consiste no teste das interações dos métodos públicos de uma mesma classe quando são chamados em diferentes sequências. Auxilia a aumentar a confiança de que as sequências de chamadas a métodos de uma mesma classe interagem de maneira apropriada;
- Teste de componentes, que visa a identificar defeitos na integração entre conjuntos de classes. O teste de componentes pode ser dividido nos seguintes tipos:

¹Resolveu-se adotar o nome sequencial em vez de intraclasses pois entende-se que o teste intermétodo também pode ser visto como intraclasses, já que os métodos estão 'dentro' (intra) de uma classe. Além disso quando se trata de aspectos (como no próximo capítulo), também poder-se-ia considerar o teste de sequências de execução de unidades (métodos e adendos), o que torna o nome sequencial mais genérico, já que pode ser utilizado para outros tipos de módulos.

- (a) Intermétodos, que consiste no teste de integração de métodos de diferentes classes que interagem por meio de chamadas diretas ou indiretas;
- (b) Sequencial, que consiste no teste das interações dos métodos acessíveis de diferentes classes quando são chamados em diferentes sequências. Auxilia a aumentar a confiança de que as sequências de chamadas de classes diferentes interagem de maneira apropriada;

4.3 Abordagens de Teste de Programas OO

Nesta seção são apresentadas algumas abordagens de teste de programas OA encontradas na literatura. A seleção dos trabalhos foi feita com base na literatura mais recente e no trabalho de Vincenzi *et al.* (2007). Em particular, é dada ênfase para as abordagens de teste estrutural, base desta tese. Alguns conceitos sobre abordagens funcionais e baseadas em modelo são descritos brevemente na Seção 4.3.1. Depois, na Seção 4.3.2, são apresentadas as principais abordagens de teste estrutural de programas OO relacionadas com esta tese. Na Seção 4.3.3 são discutidos alguns conceitos e abordagens sobre o teste baseado em defeitos de programas OO.

4.3.1 Teste Baseado em Especificação

De acordo com a classificação das técnicas de teste definida por Howden (1986), o teste de software pode ser baseado em especificação ou em programa. Os critérios de teste funcional discutidos no Capítulo 3, que são baseados em especificação, podem ser aplicados diretamente no contexto da POO, pois são independentes do paradigma de programação. Além disso, pelo caráter dinâmico dos objetos e de sua capacidade de armazenar estados, foram propostas diversas abordagens que utilizam modelos de estados (como as MEFs discutidas na Seção 3.4.4) para representar o comportamento dos objetos e derivar requisitos de teste (Binder, 1999; Chaim *et al.*, 2003; Hoffman e Strooper, 1997; Kung *et al.*, 1996; McGregor, 1994; Offutt e Irvine, 1995; Turner e Robson, 1993). Dessa forma, tanto os critérios funcionais quanto os baseados em modelos podem ser aplicados no contexto da POO.

Dos trabalhos de teste baseado em modelos para programas OO, pode ser destacada a abordagem descrita no livro de Binder (1999). Nessa abordagem é proposto o modelo FREE (*Flat-tened Regular Expression*) para representar o comportamento de uma classe. Como existe uma equivalência entre máquinas de estado e expressões regulares, uma máquina que modela o comportamento de uma classe pode também ser escrita como uma expressão regular (daí o nome do modelo proposto). A ideia básica do FREE é representar os estados dos objetos – ou seja, os valores de seus atributos – após a ativação e desativação de métodos, com a possibilidade

de adicionar condições às transições. A partir desse modelo, casos de teste podem ser gerados para cobrir as diferentes partes do modelo (por exemplo, para alcançar determinados estados a partir de outros estados, e seguindo determinadas condições). Para dar uma ideia do modelo, na Figura 4.1 é mostrado o FREE em forma de máquina de estados para uma classe que modela contas bancárias.

Figura 4.1: Modelo FREE para uma classe de contas bancárias (adaptado do trabalho de Xu *et al.* (2005)).

Mais recentemente, destacam-se abordagens de teste de programas OO que utilizam outros tipos de modelos para a geração de casos de teste (Vincenzi *et al.*, 2007). Por exemplo, Chaim *et al.* (2003) definiram uma família de critérios de teste de cobertura em especificações UML, mais especificamente baseada em diagramas de casos de uso. A ideia dos critérios propostos é assegurar, por exemplo, que as interações entre os próprios casos de usos e entre atores e casos de uso sejam exercitadas, contribuindo para uma especificação mais precisa do problema modelado (Vincenzi *et al.*, 2007). Além desse trabalho, nessa mesma linha ainda podem ser citados os trabalhos de Offutt e Abdurazik (1999), Heumann (2001) e Beckman e Grupta (2002).

4.3.2 Teste Estrutural

Os métodos são entidades base de programas OO similares aos procedimentos da programação procedimental. Assim, muitos dos trabalhos propostos para o teste estrutural de programas procedimentais podem ser adaptados para o teste de programas OO, no que diz respeito aos métodos. Os grafos de fluxo de dados podem ser construídos para representar a estrutura dos métodos e, assim, os critérios são utilizados quase que diretamente. Os trabalhos de Harrold e

Rothermel (1994), Vincenzi *et al.* (2003) e Buy *et al.* (2000) seguem essa linha, incorporando as propriedades do teste procedimental no teste OO e, indiretamente, tratando algumas características da POO. Por outro lado, existem trabalhos que incorporam diretamente as características específicas da POO na abordagem de teste (Alexander e Offutt, 1999; Chen e Kao, 1999; Souter e Pollock, 2003; Souter *et al.*, 1999). Nesta seção são analisadas as propostas de Harrold e Rothermel (1994) e Vincenzi (2004) – da primeira estratégia – e a abordagem de Souter e Pollock (2003) – da segunda estratégia – por terem relação mais direta com a abordagem proposta nesta tese.

Abordagem de Harrold & Rothermel

Harrold e Rothermel (1994) adaptaram as abordagens tradicionais de teste baseado em fluxo de dados de programas procedimentais para o teste de programas OO. Assim como são testados os procedimentos de forma isolada (teste intraprocedimental) e a interação entre eles (teste interprocedimental), a mesma ideia pode ser aplicada aos métodos isolados de uma classe (teste intramétodo), e aos métodos de uma classe que interagem entre si (teste intermétodo). Porém, no paradigma OO devem ser consideradas também as interações de fluxo de dados que acontecem quando usuários de uma classe invocam sequências de métodos de maneira arbitrária, fazendo com que o estado da classe mude. Por exemplo, em uma classe *C*, com atributo *a* e métodos m_1, m_2, m_3 , deve ser analisado se as diferentes sequências de chamadas (por exemplo: m_2, m_1, m_3, m_2) não provocam nenhum estado inconsistente na classe *C* com relação ao atributo *a*.

A partir desse problema, Harrold e Rothermel (1994) propuseram o teste de fluxo de dados de classe, que considera não somente os testes intra e intermétodo, mas também o teste intraclasse (ou sequencial) que leva em conta a interação entre métodos públicos de uma classe quando chamados em diferentes sequências. Esses autores consideram a classe como a menor unidade de um programa OO e, assim, diferentemente deste texto, consideram três níveis para o teste de unidade:

- Intramétodo, que testa os métodos individualmente (equivalente ao teste de unidade de programas procedimentais);
- Intermétodo, que testa os métodos em conjunto com outros métodos da mesma classe (equivalente ao teste de integração de programas procedimentais);
- Intraclasse, que testa a interação entre métodos públicos quando chamados em diferentes sequências (denominado de teste sequencial em outras partes deste texto).

Com isso, devem ser considerados três tipos de pares def-uso, correspondentes a cada nível de teste. Considerando uma classe C em teste, d uma sentença definindo uma variável e u uma sentença contendo um uso de uma variável, definem-se:

- Pares def-uso intramétodo: Seja M um método de C . Se d e u estão em M e existe um programa P que chama M , tal que (d, u) é um par def-uso exercitado durante uma simples invocação de M , então (d, u) é um par def-uso intramétodo.
- Pares def-uso intermétodo: Seja M_0 um método público de C e seja $\{M_1, M_2, \dots, M_n\}$ o conjunto de métodos chamados direta ou indiretamente quando M_0 é invocado. Suponha que d está em M_i e que u está em M_j , sendo que tanto M_i quanto M_j estão em $\{M_1, M_2, \dots, M_n\}$. Se existe um programa P que chama M_0 tal que, em P , (d, u) é um par def-uso exercitado durante uma simples invocação de M_0 por P , e $M_i \neq M_j$ ou M_i e M_j são invocações separadas do mesmo método, então (d, u) é um par def-uso intermétodo.
- Pares def-uso intraclasses: Seja M_0 um método público de C e seja $\{M_1, M_2, \dots, M_n\}$ o conjunto de métodos chamados direta ou indiretamente quando M_0 é invocado. Seja N_0 um método público de C e seja $\{N_1, N_2, \dots, N_n\}$ o conjunto de métodos chamados direta ou indiretamente quando N_0 é invocado. Suponha que d está em algum dos métodos em $\{M_1, M_2, \dots, M_n\}$ e u em algum dos métodos em $\{N_1, N_2, \dots, N_n\}$. Se existe um programa P que chama M_0 e N_0 , tal que em P (d, u) é um par def-uso e que a chamada a M_0 é feita após d ter sido executado e M_0 encerra sua execução antes que u seja executado, então (d, u) é um par def-uso intraclasses.

Os autores argumentam que para os testes inter e intramétodo pode ser utilizado o algoritmo de Pande, Landi e Ryder (PLR) (Pande *et al.*, 1994), concebido para computar os pares def-uso intra e interprocedimentais para a linguagem C. Porém, para a computação dos pares def-uso intraclasses, o algoritmo não pode ser utilizado diretamente e, para isso, é gerado o *grafo de fluxo de controle de classe* (CCFG - *Class Control Flow Graph*), que *conecta* todos os métodos de uma classe (Harrold e Rothermel, 1994). Na Figura 4.2 é mostrada uma implementação parcial em C++ de uma classe que implementa uma tabela de símbolos (*SymbolTable*), utilizada para exemplificar essa abordagem de teste (Harrold e Rothermel, 1994).

Para construir o CCFG, primeiramente é necessário gerar o *grafo de chamadas de classe*, que representa as chamadas entre métodos e também as chamadas que podem ser feitas de fora da classe – os nós representam cada método e as arestas representam as chamadas. Na Figura 4.3 é mostrado o grafo de chamadas para a classe de tabela de símbolos, as arestas pontilhadas representam as chamadas aos métodos públicos da classe. A partir daí, para poder ser utilizado

```

01 // symboltable.h: definition
02 #include "symbol.h"
03
04 class SymbolTable {
05 private:
06     TableEntry *table;
07     int numentries, tablemax;
08     int *Lookup(char *);
09 public:
10     SymbolTable(int n) {
11         tablemax = n;
12         numentries = 0;
13         table = new TableEntry[tablemax]; };
14     SymbolTable() { delete table; };
15     int AddtoTable(char *symbol, char *syminfo);
16     int GetfromTable(char *symbol, char *syminfo);
17 };
18
19 // symboltable.c: implementation
20 #include "symboltable.h"
21
22 int SymbolTable::Lookup(char *key, int index) {
23     int saveindex;
24     int Hash(char *);
25     saveindex = index = Hash(key);
26     while (strcmp(GetSymbol(index),key) != 0) {
27         index++;
28         if (index == tablemax) /* wrap around */
29             index = 0;
30         if (GetSymbol(index)==0 || index==saveindex)
31             return NOTFOUND;
32     }
33     return FOUND;
34 }
35
36 int SymbolTable::AddtoTable(char *symbol,
37                             char *syminfo) {
38     int index;
39     if (numentries < tablemax) {
40         if (Lookup(symbol,index) == FOUND)
41             return NOTOK;
42         AddSymbol(symbol,index);
43         AddInfo(syminfo,index);
44         numentries++;
45         return OK;
46     }
47     return NOTOK;
48 }
49 int SymbolTable::GetfromTable(char *symbol,
50                               char **syminfo) {
51     int index;
52     if (Lookup(symbol,index) == NOTFOUND)
53         return NOTOK;
54     *syminfo = GetInfo(index);
55     return OK;
56 }
57
58 void SymbolTable::AddInfo(syminfo,index)
59 ...
60 strcpy(table[index].syminfo,syminfo);
61 }
62
63 char *SymbolTable::GetInfo(index)
64 ...
65 return table[index].syminfo;
66 }

```

Figura 4.2: Implementação parcial da classe SymbolTable.

Figura 4.3: Grafo de chamadas de classe para a classe SymbolTable (adaptado do trabalho de Harrold e Rothermel (1994)).

Figura 4.4: Grafo de chamadas de classe para a classe `SymbolTable` englobado por *frame* (adaptado do trabalho de Harrold e Rothermel (1994)).

o algoritmo PLR, é necessário englobar o grafo de chamadas com um *frame* para permitir que sejam feitas as chamadas aos métodos em quaisquer sequências para a computação dos pares def-uso intraclasses. O *frame* funciona como uma espécie de unidade pseudo-controladora da classe² e possui cinco nós: *frame entry*, *frame exit*, que representam a entrada e saída do *frame*; *frame loop*, que facilita o sequenciamento dos métodos; *frame call* e *frame return*, que representam a chamada e o retorno a qualquer método público que possa ser invocado. O *frame* ainda possui quatro arestas: (*frame entry*, *frame loop*), (*frame loop*, *frame call*), (*frame loop*, *frame exit*) e (*frame return*, *frame loop*). A partir daí, cada nó que representa um método no grafo de chamadas é substituído pelo seu grafo de fluxo de controle.

Na Figura 4.4 é mostrado o grafo de chamadas da classe `SymbolTable` englobado pelo *frame*, e na Figura 4.5 o CCFG parcial da classe. Na última figura apenas são expandidos os nós correspondentes aos métodos `AddtoTable`, `GetFromTable` e `Lookup` com os grafos de fluxo de controle de cada um deles, adicionando também arestas referentes às chamadas de

²Harrold e Rothermel (1994) definem o *frame* como uma abstração de um método principal (*main*) *P*, no qual chamadas aos métodos públicos são selecionadas arbitrariamente por um comando de escolha (*switch*) *S*, onde *S* é englobado por um laço *L*.

Figura 4.5: Grafo de fluxo de controle de classe para a classe `SymbolTable` (adaptado do trabalho de Harrold e Rothermel (1994)).

um método a outros métodos da mesma classe. Os *C*s e *R*s contidos na figura correspondem às chamadas e retornos de métodos, respectivamente.

Os três tipos de teste são úteis para o teste de classes. Por exemplo, se for utilizado o critério baseado em fluxo de dados todos-usos, então o par def-uso intramétodo nas linhas 27–28 testa se o método `Lookup` volta ao começo da tabela quando chega ao final do vetor. O par def-uso intermétodo nas linhas 29–41 testa se o usuário pode inserir um símbolo na posição 0 da tabela, enquanto que o par def-uso intraclasses nas linhas 43–38 testa se o método `AddtoTable` age

corretamente quando a tabela está cheia. Finalmente o par def-uso intraclasses nas linhas 73–82 testa se a informação adicionada anteriormente à tabela pode ser encontrada.

Para exercitar esse último par def-uso, por exemplo, poderia-se utilizar um caso de teste que executasse o seguinte caminho do grafo da Figura 4.5: (*frame entry, frame loop, frame call*, 36, 38, C, 22, 25, 26, 27, 28, 30, 31, 34, R, 41, 42, 43, 44, 47, *frame return, frame loop, frame call*, 49, C, 22, 25, 26, 33, 34, R, 52, 55, *frame return, frame loop, frame exit*). A sequência de chamadas poderia ser: `<AddtoTable("B", "bbbbbb"), GetFromTable("B", &info)>`, em que *info* é um ponteiro de carácter.

Uma outra vantagem apontada por Harrold e Rothermel (1994) em seu artigo é que os pares def-uso intraclasses também auxiliam na seleção de sequências de métodos que devem ser testadas. Por exemplo, para exercitar o par def-uso intraclasses das linhas 73–82, é necessário chamar a sequência de métodos `<AddtoTable, GetFromTable>`, como comentado anteriormente. Entretanto, como não existe par def-uso intraclasses cuja definição se encontra no método `GetFromTable` e uso no método `AddtoTable`, tem-se a evidência de que a execução do primeiro método não afeta o segundo e, portanto, a sequência não precisa ser exercitada.

Harrold e Rothermel (1994) consideram ainda o teste de fluxo de dados na integração das classes, que corresponde a um quarto nível de teste, o interclasses. Esse tipo de teste envolve os pares def-uso em que a definição *d* de uma variável se encontra em uma classe e o uso *u* em outra. Para isso deve ser feita a interação entre os diversos CCFGs de cada classe. Harrold e Rothermel (1994) apenas discutem esse tipo de teste, não tratando de detalhes.

Apesar de ser uma abordagem que efetivamente adapta o teste de fluxo de dados primeiramente proposto para programas procedimentais para programas OO, uma limitação essencial pode ser apontada: duas das principais peculiaridades da programação OO – o polimorfismo e o acoplamento dinâmico – não são tratadas. Por exemplo, quando existe uma chamada polimórfica em uma dada classe, em tempo de compilação não se sabe qual dos métodos na hierarquia de classes será chamado. Os autores discutem duas soluções: (1) gerar todas as possíveis integrações, ou seja, gerar os grafos do método que chama integrado com todos os métodos que podem ser chamados (uma abordagem conservativa); e (2) seleccionar apenas algumas classes representativas da hierarquia de classe para gerar os grafos do método chamado com os métodos dessas classes. No primeiro caso teria-se uma abordagem precisa mas possivelmente impraticável e, no segundo caso, o método seria testado juntamente com alguns, mas não todos, os métodos chamados.

A porção desse trabalho que de fato se refere aos programas OO é apenas a parte relacionada com o teste intraclasses. O restante é uma adaptação quase direta do que havia sido proposto para o paradigma procedimental.

Abordagem de Vincenzi et al.

A ideia dos trabalhos de Vincenzi (2004) e Vincenzi *et al.* (2005) é viabilizar o teste estrutural intramétodo de programas Java a partir do *bytecode*, permitindo também, com isso, o teste estrutural de unidade de componentes – definido neste caso como um conjunto de classes com interface bem definida e especificada – para os quais o código-fonte nem sempre se encontra disponível.

Um modelo de fluxo de dados subjacente foi definido, caracterizando as instruções de *bytecode* responsáveis pela definição e/ou uso de variáveis. Essa classificação encontra-se na Tabela 4.1.

De posse do modelo de fluxo de dados, um modelo de representação de programa – o Grafo Definição-Uso ($\mathcal{DU}$) – é construído, considerando os mecanismos de tratamento de exceção, em geral, presentes nas linguagens OO. Desse modo, o grafo $\mathcal{DU}$ é utilizado para representar o fluxo de controle e o fluxo de dados intramétodo, tanto da execução normal quanto da execução excepcional do programa. Uma observação importante é que Vincenzi *et al.* (2003) consideram também os atributos das classes em seu modelo de dados, o que é observado por Souter *et al.* (1999) como uma propriedade interessante do ponto de vista de teste OO, já que explora as associações entre objetos e seus atributos.

Um grafo $\mathcal{DU}$ de uma dada unidade u é definido como um grafo dirigido $\mathcal{DU}(u) = (N, E, s, O)$, tal que:

- N representa o conjunto de nós de um grafo $\mathcal{DU}$: $N = \{n | n \text{ corresponde a uma sequência linear de computações, ou bloco de instruções, de } u\}$;
- $E \subseteq N \times N = E_r \cup E_e$ é o conjunto completo de arestas do $\mathcal{DU}$. Cada aresta $e \in E$ representa a transferência de controle que pode ocorrer entre dois nós, tal que:
 - E_r e E_e correspondem a dois subconjuntos disjuntos de arestas regulares e de exceção, respectivamente:
 - * E_r é o conjunto de arestas regulares definido como $E_r = \{(n_i, n_j) | \text{o bloco de instruções em } n_j \text{ pode ser executado imediatamente após o bloco de instruções em } n_i \text{ e } (n_i, n_j) \notin E_e\}$.
 - * E_e é o conjunto de arestas de exceção definido como $E_e = \{(n_i, n_j) | \text{as instruções de } n_i \text{ estão no escopo de um tratador de exceção que inicia em } n_j\}$;
- $s \in N$ é o nó de entrada de u . s é o nó do grafo que possui a primeira instrução da unidade;

Tabela 4.1: Diferentes classes de instruções de bytecode e respectivo impacto no fluxo de dados (Vincenzi, 2004).

Classe	Instrução de Bytecode	Implicações no Fluxo de Dados
0	athrow, goto, goto_w, if_acmpeq, if_acmpne, if_icmpeq, if_icmpge, if_icmpgt, if_icmple, if_icmplt, if_icmpne, ifeq, ifge, ifgt, ifle, iflt, ifne, ifnonnull, ifnull, lookupswitch, tableswitch, areturn, dreturn, freturn, ireturn, lreturn, return, ret, monitoreenter, monitorexit, pop, pop2, breakpoint, impdep1, impdep2, nop, checkcast, wide, swap	Essas instruções não têm implicações no fluxo de dados da unidade.
1	invokeinterface, invokespecial, invokestatic, invokevirtual, jsr, jsr_w, dadd, ddiv, dmul, dneg, drem, dsub, fadd, fdiv, fmul, fneg, frem, fsub, iadd, iand, idiv, imul, ineg, ior, irem, ishl, ishr, isub, iushr, ixor, ladd, land, ldiv, lmul, lneg, lor, lrem, lshl, lshr, lsub, lushr, lxor, arraylength, instanceof, aconst_null, bipush, dconst, fconst, lconst, sipush, ldc, ldc_w, ldc2_w, d2f, d2i, d2l, f2d, f2i, f2l, i2b, i2c, i2d, i2f, i2l, i2s, i2d, i2f, i2i, new, multianewarray, anewarray, newarray, dcmpl, dcmpl, fcmpl, fcmpl, lcmp	Essas instruções não têm implicações no fluxo de dados da unidade. Além disso, elas deixam no topo da pilha de execução um elemento desconhecido. Acessos a tal elemento não irão caracterizar definição ou uso de nenhuma variável. Por exemplo, a instrução new empilha a referência a um novo objeto no topo da pilha de execução e qualquer uso futuro de tal referência, tal como o acesso de um de seus atributos, não é considerado um uso ou definição desse objeto.
2	aaload, baload, caload, daload, faload, iaload, laload, saload	Carregam no topo da pilha de execução um elemento de um vetor caracterizando um uso do vetor.
3	aastore, bastore, castore, dastore, fastore, iastore, lastore, sastore	Armazenam o valor do topo da pilha de execução em um elemento do vetor caracterizando uma definição do vetor.
4	putfield	Armazena o valor do topo da pilha de execução em um atributo de instância caracterizando a definição desse atributo.
5	putstatic	Armazena o valor do topo da pilha de execução em um atributo de classe caracterizando a definição desse atributo.
6	dup, dup2, dup_x1, dup_x2, dup2_x1, dup2_x2	Essas instruções duplicam o valor do topo da pilha de execução e não têm implicações no fluxo de dados da unidade.
7	aload, dload, fload, iload, lload	Carregam no topo da pilha de execução o valor armazenado em uma variável local caracterizando um uso de tal variável.
8	astore, dstore, fstore, istore, lstore	Armazenam o valor do topo da pilha de execução em uma variável local caracterizando uma definição de tal variável.
9	getfield	Carrega no topo da pilha de execução o valor armazenado em um atributo de instância caracterizando um uso de tal atributo.
10	getstatic	Carrega no topo da pilha de execução o valor armazenado em um atributo de classe caracterizando um uso de tal atributo.
11	iinc	Incrementa o valor armazenado em uma dada variável local caracterizando um uso e uma definição dessa variável.

- $O \subseteq N$ é o conjunto (possivelmente vazio) de nós de saída. Ou seja, cada $o \in O$ não possui nenhuma aresta de saída.

Para cada DU construído a partir do *bytecode* Java, as componentes do grafo são representadas da seguinte forma:

- Nós regulares são representados por círculos com o rótulo contendo a primeira instrução de *bytecode* do bloco;

- Nós de chamada são representados por círculos duplos;
- Nós de saída são representados por nós em negrito;
- Arestas de exceção são representadas por setas tracejadas, representando o fluxo de controle do ponto onde uma exceção é gerada até o primeiro nó correspondente ao tratador daquela exceção.

Uma vez que o grafo $\mathcal{DU}$ de cada método tenha sido obtido, critérios podem ser definidos para derivar diferentes requisitos de teste, os quais podem ser utilizados tanto para avaliar a qualidade de um determinado conjunto de teste quanto para a própria geração de dados de teste. Ao todo, oito critérios de teste estruturais foram definidos. Vincenzi (2004) optou por separar os requisitos de teste em dois conjuntos disjuntos: (1) os que podem ser cobertos durante a execução normal do programa, denominados independentes de exceção; e (2) os que para serem cobertos exigem, obrigatoriamente, que uma exceção tenha sido lançada, denominados dependentes de exceção. Desse modo, foram estabelecidos os seguintes critérios:

- todos-nós:
 - Todos-Nós-Independentes-de-Exceção (Todos-Nós_{ei}): Requer que cada nó $n \in N$ do grafo $\mathcal{DU}$ que é alcançável por pelo menos um caminho livre de exceção, ou seja, caminhos que não incluam arestas de exceção, seja exercitado por algum caso de teste.
 - Todos-Nós-Dependentes-de-Exceção (Todos-Nós_{ed}): Requer que cada nó $n \in N$ do grafo $\mathcal{DU}$ que não é alcançável por pelo menos um caminho livre de exceção seja exercitado por algum caso de teste.
- todas-arestas:
 - Todas-Arestas-Independentes-de-Exceção ($\text{Todas-Arestas}_{ei}$): Requer que cada aresta $e \in E$ do grafo $\mathcal{DU}$ que é alcançável por pelo menos um caminho livre de exceção seja exercitada por algum caso de teste.
 - Todas-Arestas-Dependentes-de-Exceção ($\text{Todas-Arestas}_{ed}$): Requer que cada aresta $e \in E$ do grafo $\mathcal{DU}$ que não é alcançável por pelo menos um caminho livre de exceção seja exercitada por algum caso de teste.

- todos-usos:
 - Todos-Usos-Independentes-de-Exceção (Todos-Usos_{ei}): Requer que seja exercitado pelo menos um caminho livre de exceção e livre de definição para uma variável definida em um nó n para todo nó e toda aresta que possui um uso da mesma variável e que possa ser alcançada a partir de n .
 - Todos-Usos-Dependentes-de-Exceção (Todos-Usos_{ed}): Requer que seja exercitado pelo menos um caminho que não seja livre de exceção mas que seja livre de definição para uma variável definida em um nó n para todo nó e toda aresta que possui um uso da mesma variável e que possa ser alcançada a partir de n .
- todos-potenciais-usos (Maldonado, 1991):
 - Todos-Pot-Usos-Independentes-de-Exceção (Todos-Potenciais-Usos_{ei}): Requer que seja exercitado pelo menos um caminho livre de exceção e livre de definição de uma variável definida em um nó n para todo nó e toda aresta possível de ser alcançada a partir de n .
 - Todos-Pot-Usos-Dependentes-de-Exceção (Todos-Potenciais-Usos_{ed}): Requer que seja exercitado pelo menos um caminho que não seja livre de exceção mas que seja livre de definição para a variável definida em um nó n para todo nó e toda aresta possível de ser alcançada a partir de n .

Os critérios e modelos definidos por Vincenzi (2004) são utilizados apenas para o teste de unidade intramétodo. No trabalho de Franchin (2006) foi investigada a extensão desses modelos e critérios para o teste intermétodo par-a-par. Por tratar também do teste de programas OA, esse trabalho é descrito na Seção 5.3.

Abordagem de Souter & Pollock

Segundo Souter (2002), mesmo que as abordagens de teste procedimentais possam ser entendidas para lidar com as novas propriedades de linguagens OO, a utilidade dessas extensões não está suficientemente clara. A partir dessa motivação, propõem uma técnica para o teste de fluxo de dados de programas OO de forma mais efetiva utilizando pares def-uso contextuais, que agregam informação sobre o contexto de uso dos objetos.

A partir da condução de um estudo empírico realizado em um trabalho anterior com um conjunto de programas grandes escritos em Java, Souter *et al.* (1999) chegaram à conclusão de que projetos de software OO resultam em programas com diversos métodos, cada um com número limitado de sentenças e com fluxo de controle intramétodo bastante simples. Um resumo das conclusões a que chegaram é destacado a seguir:

1. O número de instruções condicionais dentro de um método é muitas vezes 0, e em média variaram entre 0 e 3;
2. Muitas vezes uma pequena porcentagem dos métodos definidos em uma classe servidora é realmente utilizada por uma dada classe cliente;
3. Apenas aproximadamente metade dos métodos utilizados pelas classes clientes de fato alteram o estado do objeto utilizado;
4. Manipulações de variáveis de tipos primitivos tipicamente utilizadas como base no teste de fluxo de dados ocorrem poucas vezes em programas OO; e
5. As computações necessárias são na maioria das vezes alcançadas a partir da manipulação de variáveis de instância de objetos via chamadas a métodos.

Esses resultados sugerem que o teste baseado em fluxo de controle pode não ser a maneira mais efetiva de se revelar os comportamentos de um programa OO. Além disso, Souter e Pollock (2000) argumentam que as abordagens de fluxo de dados como, por exemplo, a de Harrold e Rothermel (1994) discutida anteriormente, não atingem o objetivo principal do teste de fluxo de dados OO que é levar em conta as manipulações de objetos (baseado na última conclusão do resumo).

A partir daí, é proposta uma abordagem de teste baseada em manipulações de objetos utilizando a análise de ponteiros e escape proposta por Whaley e Rinard (1999) cujo objetivo é: 1) caracterizar como variáveis locais e atributos referenciam outros objetos; e 2) determinar como objetos alocados em uma região do programa podem *escapar* e serem acessados por outras regiões. Souter e Pollock (2003) utilizam esse tipo de informação, estendendo o grafo *points-to escape* para um grafo *Annotated Points-to Escape* (APE) para caracterizarem pares def-uso contextuais, agregando contexto às definições e usos dos objetos. As definições e uso de objetos são baseadas nas modificações e acessos aos atributos dos objetos.

Um par def-uso contextual é definido como uma tupla (o, def, uso) para um objeto o em que def e uso são definidos como uma sequência de chamadas $(CS_{om} - CS_1 - \dots - CS_m - L)$ na qual CS_{om} é o local da primeira chamada a método que leva a modificação (referência) do (ao) estado do objeto o . L é o local da instrução de atribuição (leitura) em si do dado, alterando o estado modificado (referenciado) do objeto. Cada CS_i na sequência interna é o local de chamada em uma sequência de chamadas que leva do local da chamada original, CS_{om} , à atribuição (leitura).

Essas sequências de chamadas é que configuram contextos para cada associação def-uso. Enquanto que as associações definidas nas abordagens anteriores apenas consideravam a definição e uso em qualquer lugar, sem considerar as sequências de chamadas a métodos específicas

que podem levar à execução da associação. É por esse fato que cada associação livre de contexto pode dar origem a mais de uma associação contextual. Essa abordagem leva em conta o polimorfismo porque as chamadas polimórficas são tratadas na construção dos grafos APE (em cada ponto de chamada, todos os grafos APE dos métodos potencialmente chamados são juntados ao APE da região em análise).

4.3.3 Teste Baseado em Defeitos

Atualmente, alguns pesquisadores têm trabalhado na definição de operadores de mutação para programas OO (Vincenzi *et al.*, 2007). Entre eles, pode ser destacado o trabalho de Kim *et al.* (1999, 2000), onde é definido um conjunto de operadores de classes (porém não explorando todas as características da POO); de (Bieman *et al.*, 2001a), que definem operadores de mutação para chamadas a bibliotecas Java utilizadas comumente em programas comerciais; e de Ma *et al.* (2002) que definem operadores que incluem os definidos por Kim *et al.* (2000), e mais alguns. É importante notar, entretanto, que todas essas abordagens focam no teste de programas Java e no teste de classes, e não levam em consideração o teste intramétodo e intermétodo de forma similar como é realizado o teste de mutação de programas procedimentais.

Vincenzi (2004), por outro lado, analisou a aplicação mais direta dos operadores de mutação propostos para programas procedimentais no contexto de programas OO. Especificamente, foram avaliados operadores anteriormente propostos para o teste de programas escritos em C, para o teste de programas escritos em Java e C++.

4.4 Ferramentas de Teste para Programas OO

Para dar uma ideia de algumas ferramentas de teste de programas OO que foram desenvolvidas, a seguir é apresentada uma lista baseada no trabalho de Domingues (2002). As ferramentas focam principalmente no teste de programas escritos em C++ e Java.

Em meados dos anos 1990, fruto da reengenharia de uma ferramenta de análise de cobertura para programas C/C++, foi desenvolvida a ferramenta *Coverage Tracker for Java* (CTJ). A CTJ foi adaptada para o teste de *applets* Java por meio da análise de cobertura segundo critérios similares aos critérios todos-nós e todas-arestas (Binns e McGraw, 1996).

Também em meados dos anos 1990, foi desenvolvido na Sun (Staff, 1997) o conjunto de ferramentas SunTest – *Java Testing Tools from Sun*. Esse conjunto incluía as seguintes ferramentas: JavaStar, para o teste de programas Java por meio da interface gráfica (ideia de capturar e re-executar as interações do usuário com o programa); JavaScope, para a análise de cobertura

segundo critérios similares aos critérios todos-nós e todas-arestas, e ainda a realização de *profiling* (por exemplo, a verificação de quantas vezes uma dada linha de código foi executada); e JavaSpec, para auxiliar na geração de especificações de pré e pós condições para o teste de classes. Em 1999 a Sun descontinuou o desenvolvimento desse conjunto de ferramentas.

A ferramenta χ ATAC é uma ferramenta de análise de cobertura segundo os critérios de fluxo de controle todos-nós e todas-arestas (para o teste de programas C/C++); e de fluxo de dados todos-c-usos, todos-p-usos e todos-usos (para o teste de programas C apenas). A χ ATAC foi desenvolvida no contexto do conjunto de ferramentas χ Suds da Telcordia Technologies (Agrawal *et al.*, 1998; Vincenzi *et al.*, 2007).

A Parasoft (Parasoft, 2009) é uma empresa que desenvolve ferramentas de teste para programas escritos em diversas linguagens (como C++, Java) e que utilizam diversas tecnologias (como arquiteturas orientadas a serviço). A Parasoft C++ Test permite a realização de teste de unidade de programas escritos em C/C++ e incorpora critérios estruturais similares aos critérios todos-nós e todas-arestas, além de diversas outras funcionalidades relacionadas com a infraestrutura necessária para o teste. A Jtest é uma versão similar da ferramenta para o teste de programas escritos em Java.

Rational PureCoverage (Rational Software Corporation, 2000) é uma ferramenta de análise de cobertura que também utiliza critérios similares aos critérios todos-nós e todas-arestas para programas escritos em C++ e Java. Essa ferramenta possui ainda diversas opções de apresentação do código ao testador, que auxiliam na compreensão de quais partes foram ou não exercitadas pelos casos de teste.

Algumas ferramentas também foram desenvolvidas para o teste de mutação de programas OO. Alexander *et al.* (2002) propuseram a arquitetura de uma ferramenta denominada *Object Mutation Engine* – OME, que implementa um subconjunto dos operadores de mutação propostos por Bieman *et al.* (2001b) no teste de classes Java. Chevalley (2001) e Ma *et al.* (2002) também apresentam ferramentas de teste de mutação para apoiar a aplicação de conjuntos de operadores desenvolvidos ou estendidos por eles. A característica comum entre essas ferramentas é que elas utilizam o conceito de reflexão computacional para implementar as mutações.

Mais recentemente, Bybro (Bybro, 2003) desenvolveu uma ferramenta de teste denominada *Mutation Testing System*, que implementa parte do conjunto de operadores de mutação proposto por Ma *et al.* (2002). A ferramenta utiliza o framework JUnit para documentar e executar de forma automática os casos de teste, determinando o número de mutantes mortos e o escore de mutação obtido.

4.4.1 Ferramenta JaBUTi – *Java Bytecode Understanding and Testing*

A ferramenta JaBUTi foi desenvolvida com o propósito de apoiar os critérios definidos por Vincenzi (2004) comentados anteriormente, para o teste de programas Java. Ela recebe particular atenção pois é utilizada como base da implementação da abordagem proposta nesta tese. Na Figura 4.6 são apresentadas as principais tarefas realizadas pela ferramenta para apoiar o teste estrutural. A parte superior apresenta a análise do programa em teste, a geração dos requisitos de teste e a análise de cobertura dos casos de teste. A parte inferior apresenta a instrumentação do programa, a execução dos casos de teste e a coleta dos dados de *trace*.

Figura 4.6: Tarefas realizadas pela JaBUTi (adaptado da tese de Vincenzi (2004)).

Para apoiar a construção de casos de teste, a ferramenta atribui pesos aos requisitos de teste utilizando diferentes cores que indicam quais são os requisitos que, quando cobertos, aumentariam a cobertura por cobrir outros requisitos. Esses pesos são atribuídos utilizando os conceitos de super-blocos e dominadores de Agrawal (1994).

Para avaliar a condução da atividade de teste, a ferramenta gera relatórios de teste em diferentes níveis (por projeto, por classes, por métodos e por casos de teste). Esses relatórios auxiliam o testador a decidir quando parar a atividade de teste e mostra quais partes ainda não foram suficientemente cobertas, o que pode ser usado para melhorar o conjunto de teste.

4.5 Considerações Finais

Neste capítulo foram apresentadas algumas questões fundamentais envolvidas no teste de programas OO, juntamente com alguns trabalhos relacionados com o teste estrutural OO. Em particular, enfatizou-se o trabalho de Harrold e Rothermel (1994), que adapta de maneira mais direta o teste estrutural baseado em fluxo de dados para programas OO. Esse trabalho foca na computação de pares def-uso para atributos de instância de classes e não trata com propriedade de interações interclasse, polimorfismo e *aliasing*³.

O trabalho de Vincenzi (2004), que trata do teste estrutural intramétodo para programas Java também foi apresentado. Esse trabalho é utilizado como base para as propostas de Lemos et al. (Lemos, 2005; Lemos *et al.*, 2004, 2007) apresentadas no próximo capítulo. Por último foi discutido o trabalho de Souter e Pollock (2000), que propõem uma abordagem mais avançada de teste estrutural de programas OO baseada na análise de ponteiros e escape de Whaley e Rinard (1999). Em geral, pode-se afirmar que esse trabalho contempla com mais propriedade as particularidades dos programas OO, pois trata das interações interclasse, polimorfismo e *aliasing*.

Devido ao fato dos programas OA em geral herdarem as peculiaridades de programas OO, esses trabalhos são importantes para o embasamento da abordagem apresentada no Capítulo 6. No próximo capítulo são apresentadas algumas especificidades do teste de programas OA, juntamente com alguns trabalhos relacionados com a abordagem apresentada nesta tese. Em geral, procurou-se manter coerência entre os termos utilizados e entre as abordagens de teste OO e OA.

³O efeito de *aliasing* ocorre quando um objeto é alcançável via múltiplos caminhos, geralmente por ser referenciado por outros objetos.

Teste de Programas Orientados a Aspectos

5.1 Considerações Iniciais

As técnicas modernas de desenvolvimento de software podem auxiliar na produção de sistemas com maior qualidade, a partir de projetos constituídos de módulos mais coesos e menos acoplados, com código mais legível e de alto nível. Entretanto, por mais avançadas que sejam, essas técnicas não podem evitar que defeitos sejam introduzidos ao longo do desenvolvimento, já que essa atividade está sujeita a falhas humanas. Na verdade, novas técnicas de desenvolvimento podem inclusive introduzir novos tipos de defeitos, já que mecanismos que apoiam um maior poder de expressividade podem ser mal utilizados, fazendo com que essas técnicas se tornem obstáculo ao invés de auxílio.

Existem evidências de que o cenário não é diferente com a POA. Alexander *et al.* (2004), por exemplo, propõem uma taxonomia de defeitos baseado no mau uso dessa técnica, dividido nas seguintes classes: (1) restrição incorreta em padrões de conjuntos de junção; (2) precedência incorreta de aspectos; (3) defeito na preservação de pós-condições impostas; (4) defeito na preservação de invariantes de estado; (5) foco incorreto no fluxo de controle; e (6) mudanças incorretas em dependências de controle.

A primeira classe de defeitos explora defeitos que podem acontecer na formulação de DCJs, pois o desenvolvedor pode definir conjuntos de junção que são muito restritivos, ou seja, capturam menos pontos de junção do que deviam, ou pouco restritivos, ou seja, capturam mais pontos de junção do que deviam. A segunda classe de defeitos explora enganos que o desenvolvedor pode cometer na definição da precedência entre aspectos que afetam pontos de junção coincidentes. Já a terceira e quarta classe de defeitos exploram defeitos inseridos nos aspectos que alteram dados das unidades que afetam, ferindo pós-condições e invariantes de estado previamente definidas. A quinta classe de defeitos explora defeitos inseridos em DCJs que restringem a seleção de pontos de junção a determinados fluxos de controle (em AspectJ, por exemplo, utilizando o DCJ `cflow` em sua composição). Por fim, a sexta classe de defeitos explora mudanças errôneas nas dependências de fluxo de controle que podem ser cometidas a partir do mau uso de aspectos.

Sendo assim, percebe-se a necessidade de utilização de técnicas de teste também nesse contexto, para aumentar a garantia de qualidade do software. Neste capítulo é abordado o teste de programas OA, apresentando os trabalhos mais importantes nessa linha, com ênfase para a técnica estrutural. Em geral, as propostas apresentadas são adaptações de abordagens já consagradas para o teste OO, já que a maioria das linguagens OA são também OO (como é o caso de AspectJ). Na Seção 5.2 são tratadas as peculiaridades na divisão das fases de teste de programas OA, com base na divisão das fases de teste básica e OO apresentadas nas Seções 3.3 e 4.2. Em seguida, na Seção 5.3 trata-se das abordagens de teste de programas OA que utilizam as técnicas de teste tradicionais. Por fim, na Seção 5.4 são apresentadas as considerações finais do capítulo.

5.2 Fases do Teste de Programas OA

Considera-se neste texto que em programas OA que são extensões de programas OO, como acontece com AspectJ, as menores unidades a serem testadas são os métodos (inclusive os introduzidos por aspectos) e também os adendos.

Por definição, uma classe engloba um conjunto de atributos e métodos que manipulam esses atributos; e um aspecto engloba basicamente conjuntos de atributos, métodos, adendos e DCJs. Portanto, considerando uma única classe ou um único aspecto, já é possível pensar em teste de integração. Métodos da mesma classe ou mesmo aspecto, bem como adendos e métodos de um mesmo aspecto, podem interagir entre si para desempenhar funções específicas, caracterizando uma integração que deve ser testada.

Levando em conta tais considerações, a atividade de teste de programas OA pode ser dividida nas seguintes fases:

1. Teste de Unidade, ou teste intramétodo e intra-adendo, que visa a identificar defeitos na lógica e na implementação de cada método, método introduzido ou adendo.

Uma questão importante é a execução dos adendos isoladamente, já que, de maneira geral e segundo o modelo da POA, eles são executados implicitamente, por meio de um ponto de junção. Assim, da mesma forma que Harrold e Rothermel (1994) consideram a classe como sendo o driver de um método, pode-se considerar o aspecto somado a um ponto de junção que faça com que um adendo seja executado como o driver desse adendo, pois sem eles não é possível executá-lo (a não ser que haja alguma infra-estrutura especial que permita a execução do adendo como se fosse um método, o que descartaria a necessidade do ponto de junção). Além disso, métodos comuns e introduzidos pertencentes a aspectos podem também necessitar de infra-estrutura especial para serem executados isoladamente. Essas questões são dependentes da linguagem OA utilizada e dos mecanismos que ela apoia e serão abordadas quando necessário;

2. Teste de integração, que pode ser subdividido nos seguintes tipos:

- Teste de módulos (classes e aspectos), que visa a identificar defeitos na lógica e na implementação de cada classe ou aspecto. O teste de módulos pode ser dividido nos seguintes tipos:
 - a. Interunidades, que consiste no teste da integração entre unidades (métodos ou adendos) que interagem entre si dentro de um mesmo módulo;
 - b. Sequencial, que consiste no teste das interações das unidades quando são executadas em diferentes sequências alterando o estado do módulo. Auxilia a aumentar a confiança de que as sequências de execuções das unidades de um mesmo módulo interagem de maneira apropriada;
- Teste de componentes, que visa a identificar defeitos na integração entre conjuntos de módulos. O teste de componentes pode ser dividido nos seguintes tipos:
 - a. Interunidades, que consiste no teste da integração entre unidades de módulos diferentes;
 - b. Sequencial, que consiste no teste das interações das unidades acessíveis de diferentes módulos quando são executadas em diferentes sequências. Auxilia a aumentar a confiança de que as sequências de execuções de unidades de módulos diferentes interagem de maneira apropriada;

- c. Teste de conjuntos de junção, que consiste no teste da integração do adendo no contexto de cada ponto de junção que ele afeta.

O teste de programas OA interunidades tanto no nível de módulo (no caso dos aspectos) quanto no nível de componente, quando considerado par-a-par, pode ser classificado em: Inter-método, Método-adendo, Adendo-método e Interadendo, de acordo com os tipos das unidades que interagem entre si.

5.3 Abordagens de Teste de Programas OA

Nesta seção são apresentadas as principais abordagens de teste de programas OA encontradas na literatura. A seleção dos trabalhos foi feita com base na literatura mais recente e na revisão sistemática sobre teste de programas OA conduzida por Ferrari e Maldonado (2007). Para dar uma ideia geral, na Tabela 5.1 são mostrados os trabalhos que são descritos nesta seção, indicando: a técnica na qual a abordagem é baseada (**Técnica**); a fase de teste – segundo as definições utilizadas nesta tese – na qual a abordagem enfatiza (**Fase**); se a abordagem define formalmente algum critério específico para programas OA (**Critério**); e se o trabalho apresenta alguma implementação de ferramenta de apoio (**Ferramenta**).

5.3.1 Teste Estrutural

O primeiro pesquisador a propor uma abordagem de teste estrutural para programas orientados a aspectos foi Zhao (2002, 2003). Em seus artigos, esse pesquisador apresenta uma abordagem baseada em fluxo de dados para o teste de programas orientados a aspectos e propõe a construção de uma ferramenta que implementa a abordagem.

Zhao argumenta, a partir da implementação da linguagem AspectJ, ser impraticável o teste separado de classes e aspectos pelo fato de não haver referências explícitas aos aspectos nas classes afetadas, após a compilação e combinação (o que não se aplica às versões mais recentes de AspectJ, no que diz respeito ao *bytecode*). Por conseguinte, propõe que para se testar corretamente aspectos e classes, deve-se: (1) testar os aspectos juntamente com os métodos cujos comportamentos podem ser afetados pelos adendos (*perspectiva dos aspectos*); e (2) testar as classes juntamente com os adendos que podem afetar o seu comportamento (*perspectiva das classes*). A partir daí são definidas as noções de aspecto e classe combinados (c-aspecto e c-classe), para indicar um aspecto juntamente com as partes das classes que ele afeta e, inversamente, a classe com os adendos que a afetam. Consequentemente, para os métodos que são afetados por adendos, são definidos os c-métodos (que compreendem os métodos juntamente

Tabela 5.1: Resumo dos trabalhos de teste de programas OA.

Técnica	#	Autores	Fase	Critério	Ferramenta
Estrutural	1	(Zhao, 2002)	integração	N	N
	2	(Zhao, 2003)	integração	N	N
	3	(Lemos, 2005)	unidade	S	S
	4	(Lemos <i>et al.</i> , 2007)	unidade	S	S
	5	(Franchin <i>et al.</i> , 2007)	integração	S	S
	6	(Xie e Zhao, 2006)	- -	S	S
	7	(Xu e Rountev, 2008)	unidade (regressão)	N	S
Estrutural e Mutação	8	(Lemos <i>et al.</i> , 2006a)	integração	N	N
	9	(Mortensen e Alexander, 2004)	- -	N	N
	10	(Mortensen e Alexander, 2005)	- -	N	N
Estrutural e Funcional	11	(Deursen <i>et al.</i> , 2005)	integração	S	N
Mutação	12	(Ferrari <i>et al.</i> , 2008)	integração	S	N
Baseada em Modelos	13	(Xu e Xu, 2006)	integração	S	S
	14	(Badri <i>et al.</i> , 2005)	integração	S	S
- -	15	(Bækken, 2006)	- -	N	N
	16	(Ceccato <i>et al.</i> , 2005)	integração	S	N
	17	(Eaddy <i>et al.</i> , 2007)	- -	- -	- -
	18	(Alexander <i>et al.</i> , 2004)	- -	S	N
	19	(Zhou <i>et al.</i> , 2004)	unidade	S	S
	20	(Lesiecki, 2005)	- -	S	S
	21	(McEachen e Alexander, 2005)	- -	N	N

S/N = Sim/Não

- - = Não enfatizam

com os adendos que os afetam), e para os adendos que afetam um conjunto de métodos, os c-adendos (que compreendem os adendos juntamente com os métodos que afetam).

Divergindo das definições utilizadas neste trabalho, na proposta de Zhao consideram-se as c-classes e c-aspectos como as menores unidades do programa a serem testadas. A partir daí, baseado no trabalho de Harrold e Rothermel (1994), comentado no capítulo anterior, são definidos três níveis de teste de unidade: intramódulo, intermódulo e intra-aspecto ou intraclasse. Zhao utiliza o termo módulo para referenciar os c-adendos, c-métodos, c-construtores, métodos simples, construtores simples e métodos introduzidos. Assim, um tipo de teste intramódulo, por exemplo, é o teste de um c-adendo.

Para possibilitar a utilização dessa abordagem, é proposto um modelo estrutural baseado nas abordagens tradicionais. Para cada um dos módulos (c-método, c-adendo, etc.) é construído um grafo de programa com as informações de definições e usos de variáveis. Para cada um dos c-aspectos e c-classes também é construído um *grafo interprocedimental*, que é formado por um grafo de chamadas que representa as possíveis chamadas de cada módulo somado ao

grafo de programa de cada um dos módulos. Além disso é proposta a utilização do *grafo de fluxo de controle com frame* (FCFG – *Framed Control Flow Graph*) baseado no CCFG proposto por Harrold e Rothermel (1994), porém considerando as unidades como sendo os c-aspectos ou c-classes. A partir desse grafo pode ser conduzido o teste intraclasse ou intra-aspecto.

Um problema que pode ser encontrado na abordagem de Zhao é que não se pode testar aspectos e adendos separadamente. No primeiro nível de teste – o intramódulo – os adendos já estão combinados com as unidades que afetam, sem que se possa testá-los isoladamente. Tal falha é decorrente da abordagem ser baseada em uma estratégia de implementação anterior de AspectJ, na qual os aspectos não eram compilados em módulos separados. Nas mais recentes implementações do AspectJ, os aspectos e classes são compilados em classes separadas, de maneira que as classes ficam mais limpas e tem-se a possibilidade de identificar os pontos onde os aspectos afetam, sem que esses fiquem misturados com as classes.

Com base nessas limitações do trabalho de Zhao, no projeto de mestrado de Lemos (2005) foi proposta uma abordagem de teste estrutural de unidade para programas OA com base na linguagem AspectJ (Lemos *et al.*, 2005, 2004, 2007). Assim como neste texto, consideraram-se os métodos (inclusive os introduzidos) e os adendos como as menores unidades a serem testadas. Com essa perspectiva, em oposição ao trabalho de Zhao (2003), os métodos e adendos são tratados isoladamente no teste de unidade.

Em programas OA, quando os aspectos definem comportamento adicional em determinados pontos de junção por meio dos adendos, ocorre um novo tipo de interação no nível de unidade. Antes, depois, ou no lugar do ponto de junção alcançado, o fluxo de controle é passado para o adendo do aspecto que afeta o ponto de junção, retornando ao final da execução. Essa interação pode ser comparada com as chamadas a métodos, nas quais o fluxo de controle é passado para o método chamado, retornando ao final de sua execução (Lemos, 2005). Porém, é interessante que se faça uma distinção explícita entre os pontos de chamada e os pontos nos quais os aspectos definem comportamento por meio dos adendos. No primeiro caso o desenvolvedor do método insere explicitamente a chamada ao método para os propósitos do método em programação, enquanto que no segundo caso são os aspectos os responsáveis por definir o comportamento adicionado desejado em pontos de junção específicos do sistema. Dessa forma, tem-se um novo tipo de interação que torna necessária a adaptação do grafo de fluxo de controle, para representar adequadamente o fluxo das unidades afetadas por adendos de aspectos.

Para esse fim, foi definido um novo tipo de nó no grafo de fluxo – o nó transversal (*crosscutting node*) – para representar os pontos das unidades onde os aspectos definem comportamentos transversais, tomando o fluxo de controle para si. O Grafo de Fluxo de Controle Orientado a Aspectos (*AOCFG*) foi definido com o objetivo de apoiar o teste de unidade de programas OA (Lemos *et al.*, 2007). Como é possível que em uma linguagem OA os adendos sejam afe-

tados por outros adendos, o $\mathcal{AOCFG}$ pode ser usado para representar tanto métodos quanto adendos.

Um grafo $\mathcal{AOCFG}$ de uma dada unidade u é definido como um grafo dirigido $\mathcal{AOCFG}(u) = (N, E, s, C, T)$, tal que:

- N representa o conjunto de nós de um grafo $\mathcal{AOCFG}$: $N = \{n | n \text{ corresponde a uma sequência linear de computações de } u\}$;
- $E \subseteq N \times N$ é o conjunto completo de arestas do $\mathcal{AOCFG}$. Cada aresta $e \in E$ representa a transferência de controle que pode ocorrer entre dois nós;
- $s \in N$ é o nó de entrada de u . s é o nó do grafo que contém a primeira instrução da unidade;
- $C \subseteq N$ é o conjunto (possivelmente vazio) de nós transversais que representam um nó no qual ocorre uma interação com um adendo de um dado aspecto;
- $T \subseteq N$ é o conjunto de nós de saída. Ou seja, cada $t \in T$ não possui nenhuma aresta de saída.

O Grafo Def-Use Orientado a Aspectos ($\mathcal{AODU}$) é o $\mathcal{AOCFG}$ com informações de definições e usos de variáveis, para a aplicação de critérios baseados em fluxo de dados. Como o grafo $\mathcal{AODU}$ é um $\mathcal{AOCFG}$ estendido, é necessária apenas a construção do $\mathcal{AODU}$ para se derivar requisitos de teste tanto para o fluxo de controle quanto para o fluxo de dados do programa.

Para uma dada linguagem orientada a aspectos, o grafo $\mathcal{AODU}$ deve ser instanciado para representar as unidades de programas escritas nessa linguagem e para que possa ser definido um modelo de dados a partir do qual identifiquem-se as definições/usos de variáveis e quais tipos de variáveis são tratadas. No caso de AspectJ – linguagem na qual o trabalho de Lemos *et al.* (2007) foi baseado, e na qual também esta tese é baseada – estendeu-se o trabalho de Vincenzi (2004), que definiu modelos e critérios de teste para o *bytecode* Java. Essa decisão foi baseada no fato de que o compilador/combinador de AspectJ produz *bytecode* Java comum como resultado da compilação/combinação.

Na Figura 5.1 é mostrada uma aplicação OA simples escrita em AspectJ. Parte do *bytecode* do método `affectedMethod` é mostrada na Figura 5.2 e o grafo $\mathcal{AODU}$ referente ao método é mostrado na Figura 5.3.

A representação gráfica do $\mathcal{AODU}$ foi definida da seguinte forma:

- Nós regulares são representados por círculos com o rótulo contendo a primeira instrução de *bytecode* do bloco;

- Nós de chamada são representados por círculos duplos;
- Nós transversais são representados por elipses tracejadas, com informação de qual tipo de adendo afeta aquele ponto (posterior, anterior ou de contorno – *before*, *after* ou *around*), e a qual aspecto pertence. Por exemplo, se há uma interação com um adendo posterior de um aspecto *AnAspect* em um certo ponto, o nó transversal correspondente é adicionado com a informação $\ll after - AnAspect \gg$;
- Nós de saída são representados por nós negritados;
- Arestas de exceção (E_e) são representadas por arestas tracejadas, representando o fluxo de controle do ponto onde uma exceção é gerada até o primeiro nó correspondente ao tratador daquela exceção.

```

public class Point {
    public int x;
    public int y;
    public AClass a;

    public Point(int _x, int _y) {
        x = _x; y = _y;
    }

    public void affectedMethod(Point p, int
        _x, int _y) {
        try {
            if (p.x <= 10 && p.y <= 10) {
                p.x = _x; p.y = _y;
                p.a = new AClass(10, 20);
            }
            p.printPoint(p);
            System.out.println(p.x);
        } catch (AnException ae) {
            System.out.println("Exception " +
                "caught!");
        }
    }

    public void printPoint(Point p)
        throws AnException {
        if (p.x == 0) {
            AnException ae = new AnException();
            throw ae;
        }
        ...
    }
    ...
    public static class AnException
        extends Exception {
        ...
    }
}

public aspect AnAspect {
    pointcut exec(Point p, int i, int j):
        execution(void
            Point.affectedMethod(Point, int, int))
            && args(p, i, j);
    pointcut settingA(AClass a):
        set(AClass Point.a) && args(a) &&
        !within(aspects.*);
    pointcut handlerPC(Point p) :
        handler(Point.AnException) && this(p);
    before(Point p, int i, int j):
        exec(p, i, j) {
            if (p.x >= 0) p.x = i + 3;
            if (p.y >= 0) p.y = j + 4;
        }
    after (Point p, int i, int j) returning():
        exec(p, i, j) {
            System.out.println("after " +
                "returning exec");
            if (i > 10)
                System.out.println("i > 10");
            else
                System.out.println("i <= 10");
            if (p.x > 10)
                System.out.println("p.x > 10");
            else
                System.out.println("p.x <= 10");
        }
    void around(AClass a) : settingA(a) {
        System.out.println("around settingA");
        a.a = 20; a.b = 30;
        proceed(a);
    }
    before(Point p) : handlerPC(p) {
        p.x = 40;
    }
}

```

Figura 5.1: Exemplo de um programa orientado a aspectos escrito em AspectJ.

```

0 aload_1
...
18 invokevirtual #116 <Method void
   ajc$before$aspects_AnAspect$1$dcd6c0af(
   components.Point, int, int)>
21 aload_1
22 getfield #17 <Field int x>
25 bipush 10
27 if_icmpgt 109
30 aload_1
31 getfield #19 <Field int y>
34 bipush 10
36 if_icmpgt 109
39 aload_1
...
106 invokevirtual #110 <Method void
   ajc$around$aspects_AnAspect$2$f8ad947f(
   components.AClass, org.aspectj.runtime.
   internal.AroundClosure)>
109 aload_1
110 aload_1
111 invokevirtual #37 <Method void
   printPoint(components.Point)>
114 getstatic #43 <Field java.io.PrintStream
   java.lang.System.out>
117 aload_1
118 getfield #17 <Field int x>
128 invokestatic #106 <Method
   aspects.AnAspect aspects.AnAspect.
   aspectOf()>
131 aload_0
132 invokevirtual #113 <Method void
   ajc$before$aspects_AnAspect$4$e6001804(
   components.Point)>
135 astore 4
137 getstatic #43 <Field java.io.PrintStream
   java.lang.System.out>
140 ldc #51 <String "Exception caught!">
142 invokevirtual #54 <Method void println(
   java.lang.String)>
145 goto 148
...
157 invokevirtual #119 <Method void
   ajc$afterReturning$aspects_AnAspect$3$
   dcd6c0af(components.Point, int, int)>
160 return

```

Figura 5.2: Parte do *bytecode* do método `affectedMethod`.

Figura 5.3: *AODU* do método `affectedMethod` com a presença do aspecto `AnAspect`.

Critérios de teste baseados tanto no fluxo de controle quanto no fluxo de dados da aplicação foram propostos. Seja Π o conjunto de caminhos exercitados para um dado conjunto de casos de teste T ; aresta de exceção uma aresta que liga um nó ao tratador de exceção referente a ele; e E_c o conjunto de arestas transversais cujos nós início ou destino são nós transversais. Para o fluxo de controle os seguintes critérios foram definidos:

- todos-nós-transversais (Todos-Nós_C)

- Π satisfaz o critério todos-nós-transversais se cada nó $n_i \in C$ está incluído em Π . Em outras palavras, este critério requer que cada nó transversal, e portanto cada execução de adendo que ocorre na unidade afetada, seja exercitado pelo menos uma vez por algum caso de teste de $\mathcal{T}$.
- todas-arestas-transversais ($Todas-Arestas_C$)
 - Π satisfaz o critério todas-arestas-transversais se cada aresta $e_c \in E_c$ está incluída em Π . Em outras palavras, este critério requer que cada aresta do grafo $\mathcal{AODU}$ que tem um nó transversal como nó início ou destino seja exercitada por algum caso de teste de $\mathcal{T}$.

Para o fluxo de dados o seguinte critério foi definido:

- todos-usos-transversais ($Todos-Usos_C$)
 - Π satisfaz o critério todos-usos-transversais se para cada nó x no qual existe uma definição da variável v , Π inclui um caminho livre de definição para v de x a cada nó transversal em que ocorre um c-uso de v e a cada aresta cujo nó início é um nó transversal e na qual ocorre um p-uso de v .

Teste de integração par-a-par

No trabalho de Franchin *et al.* (2007) et al., foi considerado o teste estrutural de integração de pares de unidades de programas OO e OA. A ideia principal é analisar a cobertura de um conjunto de casos de teste no que diz respeito à cobertura de uma unidade no contexto da unidade que a chama ou é afetada por ela (quando se trata de um adendo). Isso é importante porque alguns defeitos podem não ser encontrados apenas considerando a cobertura individual de cada uma das unidades. Com os critérios definidos, a interface entre as duas unidades é exercitada, e assim aumenta-se a probabilidade de encontrar defeitos na integração entre elas.

Para o estabelecimento dos critérios par-a-par, como denominados por Franchin (2007), primeiramente foi definido um grafo de fluxo de controle e de dados que modela a integração entre duas unidades. O grafo foi denominado *PairWise Def-Use graph* ($PWDU$). Esse modelo foi definido com base no trabalho de Linnenkugel e Müllerburg (1990), que definiram critérios de teste par-a-par baseados em fluxo de dados para programas procedimentais, e no trabalho de Lemos *et al.* (2007), discutido anteriormente nesta seção. O modelo é gerado juntando-se os dois $\mathcal{AODUs}$ das unidades que se interagem. Na Figura 5.4 é apresentado um exemplo de $PWDU$. As unidades se referem a um método chamado `pmf` e a um adendo chamado `printOperands`, em que o adendo afeta o método em dois pontos de junção. O $PWDU$

é gerado para a segunda execução de `printOperands` em `pmf`. Nesse trabalho, unidade base foi definida como a unidade que chama um método ou é afetada por um adendo; e unidade integrada como o método chamado ou o adendo que afeta a unidade base. Repare que os rótulos dos nós da unidade integrada são prefixados com ‘i.’ para evitar repetição, já que o *offset* das instruções de *bytecode* pode repetir de unidade para unidade. Nessa abordagem é necessário fazer o mapeamento das variáveis de comunicação, ou seja, as variáveis que são compartilhadas entre as unidades. No caso do exemplo, o adendo captura duas informações de contexto do método e as utiliza com nomes diferentes dos definidos no método.

Figura 5.4: *PWDU* do método `pmf` sendo afetado pelo adendo `printOperands`.

Com base no *PWDU*, três critérios foram definidos: todos-nós-integrados e todas-arestas-integradas, que requerem a cobertura dos nós e arestas da unidade integrada no contexto da unidade base; e todos-usos-integrados, que requer a cobertura dos pares def-uso das variáveis de comunicação, tanto quando são definidas na unidade base e usadas na unidade integrada quanto quando definidas na unidade integrada e usadas na unidade base. Esse trabalho também é utilizado como base para a abordagem de teste baseado em conjuntos de junção definida no Capítulo 6.

Teste de descritores de conjuntos de junção

Em um trabalho preliminar à abordagem apresentada no Capítulo 6, Lemos *et al.* (2006a) tratam do teste de DCJs, descrevendo o contexto em que se insere e a sua importância, e propondo uma solução inicial para o problema. Sempre que há um comportamento transversal implementado por meio de um adendo, é necessário que o desenvolvedor defina em quais pontos do sistema o adendo deve executar. Essa definição é feita por meio dos DCJs, como descrito no Capítulo 2. O DCJ selecionará os pontos de junção adequados baseando-se em uma propriedade comum para que o comportamento transversal seja aplicado a eles. Entretanto, como o DCJ é um predicado sobre pontos de junção, se ele for erroneamente formulado, defeitos são injetados no sistema. Isto é, o comportamento pode ser aplicado a pontos de junção não desejados ou deixar de ser aplicado a pontos de junção adequados. Assim, é necessário que o DCJ seja testado para que se aumente a confiança de que ele está adequadamente formulado no que diz respeito a não selecionar mais nem menos pontos de junção do que deveria.

Em AspectJ, como o modelo de pontos de junção é dinâmico, os pontos de junção são definidos como pontos na execução de um programa, entretanto, em linguagens que utilizam modelos estáticos, os pontos de junção são definidos como regiões estáticas do programa. Independentemente do modelo, um DCJ pode ser erroneamente formulado das seguintes maneiras:

- (A) selecionar um conjunto de pontos de junção que tem alguma intersecção com o conjunto adequado mas também selecionar outros pontos de junção; em outras palavras, o DCJ seleciona alguns dos pontos de junção adequados mas também alguns inadequados;
- (B) selecionar um conjunto diferente, com nenhuma intersecção com o conjunto adequado; em outras palavras, o DCJ não seleciona nenhum dos pontos de junção adequados;
- (C) selecionar um conjunto maior do que o adequado; em outras palavras, o DCJ seleciona todos os pontos de junção adequados e também alguns inadequados;
- (D) selecionar um conjunto menor do que o conjunto adequado; em outras palavras, o DCJ seleciona alguns pontos de junção adequados mas não todos.

Na Figura 5.5 são representados os quatro tipos de defeitos em DCJs. Os pontos dentro dos círculos representam o universo de pontos de junção em um determinado programa. O conjunto adequado de pontos de junção para um dado DCJ é representado pelos pontos de junção delimitados por linhas tracejadas, e o conjunto realmente selecionado pelo DCJ é representado pelos pontos de junção delimitados por linhas normais. Os tipos de defeitos C e D foram previamente definidos por Alexander *et al.* (2004) como padrões de CJ muito fracos (*too weak pointcut patterns*) e padrões de CJ muito restritivos (*too strong pointcut patterns*), respectivamente.

Figura 5.5: Classe de defeitos em DCJs.

Ao examinar a Figura 5.5, pode-se notar que o DCJ pode estar incorreto não somente por selecionar pontos de junção não desejados mas também por negligenciar pontos de junção adequados. O DCJ pode também estar incorreto pelos dois motivos. Para auxiliar a encontrar esses tipos de defeitos, poderiam ser utilizadas técnicas de qualidade de software para aumentar a confiança de que os DCJs estão corretamente formulados (como por exemplo, inspeção de código). Em uma estratégia baseada em duas etapas, Lemos *et al.* (2006a) propuseram o uso de teste estrutural e baseado em defeitos como uma solução para resolver esse problema. A primeira etapa consiste em utilizar teste estrutural com a intenção de detectar pontos de junção inadequados selecionados pelo DCJ; e a segunda consiste em utilizar o teste de mutação com a intenção de detectar pontos de junção adequados negligenciados pelo DCJ. A ideia do teste estrutural de DCJ é desenvolvida nesta tese; porém, no caso desta tese, o teste baseado em conjuntos de junção não é feito somente para detectar pontos de junção inadequados, mas também para detectar outros tipos de defeitos relacionados com a execução do adendo em determinado ponto de junção. Já o teste de mutação de programas OA está sendo desenvolvido no projeto de doutorado de Ferrari (Ferrari *et al.*, 2008).

Outras abordagens relacionadas com o teste estrutural

Mortensen e Alexander (2005) também apresentam uma abordagem mista de teste estrutural e de mutação para programas AspectJ (Mortensen e Alexander, 2004, 2005). Os autores tentam adequar os tipos de defeitos da taxonomia apresentada por Alexander *et al.* (2004), definindo um conjunto de critérios estruturais para atuarem em diferentes níveis de granularidade, desde a cobertura de comandos e pares definição-uso de variáveis, até a cobertura das execuções dos adendos em cada ponto de junção – similar à abordagem descrita nesta tese – e de todas as introduções de um aspecto. Apesar disso, os critérios não são formalmente definidos, eles são apenas discutidos em alto nível, e nenhuma implementação da abordagem é apresentada. No contexto de teste de mutação os autores definem três operadores de mutação para atuar nas definições de escopo dos DCJs e nas definições de precedência de aspectos. Também nesse caso, os operadores não são formalmente definidos, e nenhuma implementação da abordagem é apresentada.

Deursen *et al.* (2005) apresentam uma estratégia para refatoração de software OO com o apoio da POA. Esse trabalho também discute uma estratégia de teste para garantir o correto comportamento do software independentemente da refatoração. A estratégia, intitulada *BETTAR (Better Evolvability Through Tested Aspect Refactorings)*, consiste de cinco passos, podendo-se destacar o projeto de casos de testes capaz de exercitar as porções de código refatoradas; é neste ponto que a abordagem se relaciona com teste estrutural. Uma taxonomia de defeitos também é proposta no trabalho, incluindo defeitos relacionados a declarações intertipos, definição de conjuntos de junção e definição e implementação de adendos.

5.3.2 Teste Baseado em Defeitos

Ferrari *et al.* (2008) apresentam uma abordagem de teste de mutação para programas OA. Nesse trabalho é feita uma exploração mais concreta e abrangente do que aquela encontrada no trabalho de Alexander *et al.* (2004) sobre os tipos de defeitos que podem ser encontrados em programas OA, e desenvolvem-se as ideias preliminares exploradas no trabalho de teste de DCJs descrito anteriormente (Lemos *et al.*, 2006a). O conjunto de defeitos foi identificado por meio de uma extensa revisão sobre teste de programas OA, mais especificamente: modelos de falhas candidatos (Alexander *et al.*, 2004; Bækken, 2006; Ceccato *et al.*, 2005; Eaddy *et al.*, 2007), a classificação de falhas apresentadas anteriormente neste capítulo (Lemos *et al.*, 2006a) e padrões de *bugs* (Zhang e Zhao, 2007).

As falhas foram distribuídas em quatro grupos relacionados com: (1) DCJs; (2) declarações intertipos e outros tipos de declarações; (3) definição e implementação de adendos; e (4) programa base. Para cada um desses grupos, Ferrari *et al.* (2008) conceberam conjuntos de

operadores de mutação. Por exemplo, para o grupo (1), defeitos em DCJs, entre outros, são definidos os seguintes operadores: PWSR, que troca os tipos utilizados no DCJ pelos seus super-tipos imediatos, possivelmente enfraquecendo o conjunto de junção; PSWR, que remove os caracteres coringa utilizados nos DCJs, possivelmente aumentando a restrição do conjunto de junção; PCTT, que troca o DCJ primitivo `this` pelo `target` e vice-versa. Depois da definição dos operadores os autores discutem o seu custo de aplicação por meio de um estudo com duas aplicações de grande porte. Apesar de discutir teoricamente a implementação dos operadores, até o momento nenhuma ferramenta de apoio foi apresentada.

No futuro, a abordagem apresentada nesta tese poderia ser utilizada em complemento à abordagem de Ferrari *et al.* (2008), já que o teste de mutação e o teste estrutural podem ser utilizados em conjunto para obter uma atividade de teste mais completa.

5.3.3 Teste Baseado em Modelos

Xu *et al.* (2004) e Xu e Xu (2006) propuseram uma abordagem de teste baseada em modelos de estados (e estrutural) para programas OA. A ideia principal é estender o modelo baseado em estados FREE (*Flattened Regular Expression*) proposto por Binder (1999), para um modelo de estados aspectual (*Aspectual State Model – ASM*). A partir desse modelo, é gerada uma árvore de transição para a geração de casos de teste. A segunda parte da abordagem mistura a técnica estrutural à técnica baseada em estados. A ideia é substituir as transições e interações de aspectos por grafos de fluxo de controle referentes aos métodos e adendos em um modelo chamado grafo de fluxo de aspectos (*Aspect Flow Graph – AFG*). Com esse modelo podem ser realizada a análise de cobertura a partir da execução dos casos de teste gerados com auxílio da árvore de transição, para auxiliar na geração de novos casos de teste e para se ter ideia da suficiência da atividade de teste.

No trabalho mais recente, Xu e Xu (2006) enfatizam o impacto dos aspectos nos comportamento das classes base, por meio da extensão no modelo de estados. A partir desse modelo também auxiliam a determinar quais testes criados para testar o programa base podem ser reutilizados para o teste da aplicação combinada com os aspectos. No trabalho mais recente, (Xu e Xu, 2006) desconsideram a parte relacionada com teste estrutural de sua abordagem e tratam somente do teste baseado em modelos. O ASM é formalizado juntamente com as gerações de casos de teste baseados nos ASMs. Os autores argumentam que a abordagem proposta apresenta duas contribuições básicas: (1) a modelagem rigorosa de programas OA em modelos de estado que facilita a especificação do impacto dos aspectos nos estados e transições das classes base; e (2) uma investigação de como utilizar casos de teste previamente criados para testar as classes base para testar os mesmos programas com as extensões apresentadas pelos aspectos. O

trabalho mostra que a maioria dos casos de teste criados para testar as classes base podem ser reusados para os aspectos, porém algumas modificações sutis podem ser necessárias.

Os modelos de estados são construídos para as classes e para os aspectos e são combinados (*woven*) para formar um único modelo. Por exemplo, na Figura 5.6 é mostrado como um modelo de uma classe de contas bancárias – similar ao FREE mostrado na Seção 4.3.1 – é combinado com um aspecto que modela contas devedoras (*Overdraft*). O aspecto adiciona um estado novo às contas: *Overdrawn*, que ocorre quando o cliente retira mais dinheiro do que possui na conta. As flechas tracejadas indicam quais operações são relacionadas e as duas regiões ressaltadas indicam que o aspecto sobrecarrega a parte da aplicação referente ao estado *Open* da conta. A partir desse modelo é derivada uma árvore de transições da qual podem ser gerados casos de teste que definem sequências de chamadas a métodos.

Figura 5.6: Combinação de modelos de estados no trabalho de Xu e Xu (2006).

Além da abordagem de Xu e Xu (2006), Badri *et al.* (2005) também apresentam uma abordagem de teste baseado em modelos de estados cujo foco é o teste de classes que têm seu comportamento afetado por um ou mais aspectos. O modelo de estados OO é estendido para incluir os estados resultantes da atuação dos aspectos, contendo também novas notações para representar os pontos em que os aspectos afetam a classe. Partindo-se do teste de uma classe isolada, adiciona-se os estados resultantes da atuação de um aspecto por vez. A geração dos casos de teste tem como base uma árvore de transições de estados obtida a partir do modelo, ou seja, bem similar a abordagem de Xu e Xu (2006).

Esses trabalhos objetivam a geração automática de casos de teste para programas OA baseados nos modelos de estados. Em contrapartida, não é garantido que os casos de teste são

efetivos em revelar defeitos, pois nenhuma medida de cobertura do código, por exemplo, é utilizada (ao menos na abordagem mais recente; e no caso da abordagem menos recente de Xu *et al.* (2004) que considerava o teste estrutural, não é apresentada nenhuma implementação). Além do mais, não é comum que todo software seja projetado utilizando modelos formais – o que é considerada uma desvantagem do teste baseado em modelos –, e também não se pode garantir que os próprios modelos estejam corretos (a não ser que eles mesmos sejam testados). Em geral, as técnicas de geração de casos de teste deveriam ser utilizadas em conjunto com os critérios de cobertura de casos de teste; como, por exemplo, critérios estruturais.

5.3.4 Outras abordagens

Outras abordagens de teste não necessariamente baseadas nas técnicas abordadas anteriormente foram propostas para o teste de programas OA. Alexander *et al.* (2004) discutem questões envolvidas no teste de programas OA, apresentando quatro potenciais fontes de defeitos em um programa OA que já passou pelo processo de combinação. Baseados nessas fontes de defeitos, os autores propõem uma taxonomia de defeitos candidata para programas OA, considerando também características específicas da linguagem AspectJ (essa taxonomia foi apresentada no início deste capítulo). Além disso, apresentam um critério de teste definido em alto nível, destinado a apoiar a identificação dos tipos de defeitos apresentados.

Ceccato *et al.* propõem algumas adaptações aos critérios de teste tradicionais de forma a cobrir os novos requisitos de teste introduzidos pela POA. Estendem a taxonomia de defeitos de Alexander *et al.* (2004), incluindo outros três tipos de defeitos, e avaliam o emprego dos critérios de teste no apoio à sua identificação. Propõem também outros dois critérios específicos para tratar de defeitos relacionados ao fluxo de controle de aplicações OA e precedência de aspectos. Além disso, apresentam uma estratégia de teste similar a outras que se inicia pelo teste do programa base e depois inclui os aspectos incrementalmente.

Zhou *et al.* (2004) apresentam uma abordagem de teste incremental, na qual se inicia pelo teste do programa base sem aspectos, e depois se testa os aspectos combinados incrementalmente até chegar ao teste da aplicação completa (similar a abordagem de Ceccato *et al.* (2005)). Um algoritmo para a identificação de testes relevantes para executar os aspectos é proposto. Os autores definem ainda um critério de cobertura de teste para programas OA que requer a execução de todos os métodos interceptados por um determinado aspecto (similar ao teste de unidade proposto por Lemos *et al.* (2007)). Uma ferramenta que implementa o algoritmo de seleção de casos de teste e calcula a cobertura do teste também é sucintamente descrita.

Lesiecki (2005) apresenta uma abordagem prática para o teste de unidade de programas desenvolvidos em AspectJ, considerando como unidade uma classe ou um aspecto. Entretanto, o

autor não emprega técnicas ou critérios de teste específicos. Um conjunto de pequenos padrões de teste é apresentado, envolvendo teste funcional, uso de ferramentas de visualização, transferência de lógica dos adendos para outras classes e uso de objetos que simulam as classes base, chamados objetos *mock*. De acordo com os padrões apresentados, a abordagem é direcionada para o teste de adendos e conjuntos de junção.

McEachen e Alexander (2005) discutem possíveis tipos de defeitos decorrentes da combinação de aspectos em código reutilizado que já possui aspectos combinados, chamados aspectos estrangeiros. Consideram características específicas da linguagem AspectJ que permite, por exemplo, a combinação de aspectos com aplicações já compiladas. Os tipos de defeitos destacados estão principalmente relacionados à definição dos conjuntos de junção e à precedência de aspectos. Os autores fornecem algumas diretrizes sobre como proceder no caso de reúso de código que possui aspectos estrangeiros.

Xie e Zhao (2006) propuseram uma infraestrutura para geração automática de teste de comportamento aspectual como extensão de ferramentas que já existiam para o teste de programas Java. Da mesma forma que para o trabalho de Xu e Xu (2006) citado na seção anterior, a análise de cobertura proposta nesta tese pode ser utilizada para complementar essa abordagem, para avaliar os conjuntos de casos de teste gerados a partir do framework, já que os critérios propostos são mais fracos.

No contexto de modelos de defeitos, Bækken (2006) apresenta o início de um trabalho que visa a elaborar um modelo de defeitos para programas AspectJ. Nesse trabalho, o foco é na classificação de tipos de defeitos relacionados a conjuntos de junção. A partir daí, um exemplo de tipo de defeito é descrito de acordo com um formato sugerido pelo autor, destacando-se as possíveis formas sintáticas nas quais o defeito pode aparecer em um programa e o impacto semântico que o defeito pode causar na execução do programa.

Xu e Rountev (2008) propuseram uma abordagem de teste de regressão para programas AspectJ. Eles também utilizam grafos de fluxo de controle para analisar comportamento adicional introduzido por aspectos, mas no caso dessa abordagem a análise é realizada para gerar requisitos de teste de regressão. Um exemplo de grafo de interação (IG), como definido por Xu e Rountev (2008), é mostrado na Figura 5.7. Note que o grafo também modela a interação com adendos dinâmicos utilizando nós especiais e arestas de decisão (*ph_decision*). Os mesmos autores propuseram um framework para análise de fluxo de dados interprocedimental para programas AspectJ (Xu e Rountev, 2007). Esse framework também poderia ser utilizado em uma abordagem de teste estrutural, porém os autores apenas o exploram para fins de análise de impacto das estruturas dos aspectos no programa.

Figura 5.7: Exemplo de grafo gerado a partir da abordagem proposta por Xu e Rountev (2008).

5.4 Considerações Finais

Neste capítulo foram apresentadas algumas questões fundamentais relacionadas com o teste de programas OA e os trabalhos mais relevantes relacionados com a contribuição principal desta tese. De acordo com essa pesquisa bibliográfica, é importante ressaltar a necessidade de abordagens de teste estrutural de programas OA que enfatizam a integração dos aspectos com o programa base por meio dos DCJs. Outro fator importante é a implementação de ferramentas de apoio, já que o teste de software é uma atividade dificilmente realizada manualmente, ainda mais com a complexidade introduzida pela POA.

Como mostrado no Capítulo 2, o mecanismo de conjunto de junção da POA apoia a seleção de pontos de junção onde adendos podem ser aplicados para a realização de uma característica transversal de um sistema. O conjunto de junção e/ou o adendo podem conter defeitos que são revelados somente quando caminhos específicos do adendo são exercitados em pontos de junção específicos do sistema. O teste estrutural pode auxiliar o testador a encontrar esses defeitos, garantindo a cobertura da lógica do adendo ao testar o sistema. Entretanto, as abordagens existentes de teste estrutural adaptadas para a POA apresentadas na Tabela 5.1, e que definem algum critério de teste, requerem apenas a execução de um caminho no adendo em cada ponto de junção e, dessa forma, não garantem a descoberta de defeitos relacionados com partes particulares

da lógica do adendo ou com pontos de junção específicos. Além do mais, observa-se a carência de ferramentas de apoio para o teste estrutural de programas OA e de estudos que verifiquem o custo de aplicação e eficácia dos critérios propostos (por exemplo, da Tabela 5.1, os trabalhos de Ferrari *et al.* (2008) e de Xu e Rountev (2008) são os únicos que apresentam estudos de custo de aplicação). Dos trabalhos discutidos neste capítulo, apenas o trabalho de Franchin *et al.* (2007) trata do teste estrutural de integração de programas OA. Porém, nesse trabalho, não é enfatizado o teste dos adendos integrados com os métodos que afetam, os pares de unidades integradas são tratados da mesma forma, independente de serem métodos ou adendos.

No próximo capítulo é apresentada uma abordagem de teste estrutural de integração para programas OA que trata das carências indicadas. A abordagem é baseada nos conjuntos de junção, e garante que a lógica do adendo seja exercitada em cada um dos pontos de junção selecionados, tanto do ponto de vista do fluxo de controle quanto do ponto de vista do fluxo de dados. Dessa forma, propõem-se critérios mais rigorosos do que os disponíveis na literatura. Além disso, no Capítulo 7 é apresentada uma ferramenta que apoia o uso dos critérios propostos e no Capítulo 8 são apresentados estudos que avaliam tanto o custo quanto a eficácia desses critérios.

Teste Estrutural de Programas Orientados a Aspectos Baseado em Conjuntos de Junção

6.1 Considerações Iniciais

Para o teste de programas orientados a aspectos, uma importante medida de adequação para os casos de teste é a cobertura global (ou seja, considerando todos os pontos em que pode ser executado) de cada interesse transversal do programa implementado como adendo. Por exemplo, para ter maior confiança de que um interesse transversal de controle de acesso foi adequadamente testado em um programa, é importante saber a cobertura da estrutura do adendo correspondente em cada ponto de junção em que ele pode atuar. Neste capítulo é apresentada uma abordagem de teste estrutural de integração que considera esse tipo de cobertura.

Na Seção 6.2 são apresentadas as ideias básicas do teste estrutural baseado em conjuntos de junção, juntamente com exemplos que serão utilizados ao longo deste capítulo e nos próximos. Na Seção 6.3 é apresentado o modelo adjacente utilizado para representar o comportamento de cada adendo de um programa OA integrado em cada um dos pontos de junção em que pode ser executado. Na Seção 6.4 é apresentada uma família de critérios de teste estrutural baseada em conjuntos de junção para o teste de programas OA, e na Seção 6.5 é apresentada uma

estratégia básica de teste para ser utilizada na aplicação dos critérios propostos. Na Seção 6.6 são apresentadas algumas limitações relacionadas com a abordagem proposta e, por fim, na Seção 3.7 são apresentadas as considerações finais deste capítulo.

6.2 Fundamentos do Teste Baseado em Conjuntos de Junção

Como um defeito pode estar relacionado com a execução de uma parte particular do adendo em um ponto de junção específico, garantir a cobertura da estrutura do adendo em cada ponto de junção aumenta a confiança de que as interações adicionadas estão corretas. Para a aplicação desse tipo de análise de cobertura, é necessário um modelo para representar o fluxo de controle e de dados de cada adendo do sistema em cada possível ponto de junção. Seguindo as abordagens de teste propostas anteriormente pelo grupo de teste do autor desta tese, e a investigação inicial discutida na Seção 5.3.1, é definido um grafo de fluxo de controle e de dados para programas OA. Baseado nesse grafo, é definida uma família de critérios de fluxo de controle e de dados.

Para facilitar o entendimento da abordagem, simplificam-se os termos DCJ e conjunto de junção (ver Capítulo 2), utilizando-se conjunto de junção no lugar de ambos, especificando um deles quando necessário para a explicação. Além disso, o termo ponto de junção é utilizado para designar sua versão estática, ou seja a sombra estática correspondente. Dessa forma, utiliza-se uma abordagem conservadora, já que para uma sombra estática pode ou não se configurar um ponto de junção.

6.2.1 Exemplos

Os seguintes exemplos de programas OA serão utilizado em algumas partes desta tese para ilustrar a abordagem e a ferramenta proposta.

Sistema de Pagamento

Considere um sistema de pagamento que armazena informações sobre empregados, informações sobre o pagamento dos empregados (salário semanal e pagamento por hora), e informações sobre o trabalho dos empregados (datas e horas trabalhadas). Para cada empregado são armazenados seu nome, quanto vale sua hora e/ou semana de trabalho e uma lista de horas trabalhadas. Para cada empregado podem ser armazenadas as suas horas trabalhadas juntamente com a data em que aquelas horas de trabalho foram realizadas. O sistema também gerencia os usuários que

o acessam e as sessões de acesso. Cada usuário possui um nível de acesso que vai de usuário convidado a super usuário, passando por usuário básico. O sistema contém uma política de controle de acesso na qual apenas super usuários podem modificar informações de pagamento e trabalho. Apesar de somente super usuários poderem alterar informações, todos os usuários devem poder visualizar informações sobre as horas trabalhadas de todos empregados.

Na Figura 6.1 é apresentada parte de uma implementação em AspectJ do sistema de pagamento descrito anteriormente. Note que, por meio do conjunto de junção `superOperacoes`, o adendo anterior `controleSuperOperacoes` do aspecto `ControleAcesso` afeta a execução dos métodos cujos nomes contêm a cadeia de caracteres ‘Detalhes’ e do método `adicionaTrabalho`, que supostamente seriam as operações restritas aos super usuários.

Sistema de Envio de Mensagens

Considere uma aplicação simples de envio de mensagens que possui a funcionalidade de registro de mensagens enviadas e a possibilidade de criptografia das mensagens antes do envio. O sistema pode tanto enviar mensagens personalizadas quanto uma mensagem padrão. Além disso considere também que mensagens criptografadas não devem ser registradas. Na Figura 6.2 é apresentada parte de uma implementação em AspectJ desse sistema.

A classe `Mensagem` possui um atributo que armazena a mensagem e os seguintes métodos: `enviaPersonalizada`, que envia mensagens personalizadas; `enviaPadrao`, que envia uma mensagem padrão; `criptografa` que faz a criptografia de uma cadeia de caracteres (no código, por simplicidade, é apenas adicionada a cadeia de caracteres “criptografada” para indicar que a mensagem está criptografada); `envia`, que faz o envio da mensagem em si (este também somente um stub); e `setCriptografa` e `setNaoCriptografa`, que modificam o atributo `criptografa` indicando se a mensagem deve ou não ser criptografada antes do envio.

O aspecto `Registro` é responsável por registrar as mensagens enviadas. O adendo anterior `registra` afeta as chamadas ao método `envia` da classe `Mensagem` por meio do conjunto de junção `envio`, adicionando a mensagem ao atributo `registro`. O aspecto também possui os seguintes métodos: `registra`, que realiza o registro da mensagem em si; `setRegistra` e `setNaoRegistra`, que modificam o atributo `registra` indicando se as mensagens devem ou não serem registradas; `limpaRegistro`, que limpa o registro; e `getRegistro`, que dá acesso ao registro.

```

public class Empregado {
    private String nome;
    private ArrayList<InfoTrabalho> listaTrabalho;
    private InfoPagamento infoPagamento;
    public Empregado(String n, InfoPagamento ip) {
        nome = n;
        infoPagamento = ip;
        listaTrabalho = new ArrayList<InfoTrabalho>();
    }
    public void adicionaTrabalho(InfoTrabalho t) {
        listaTrabalho.add(t);
    }
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
    public ArrayList<InfoTrabalho> getListaTrabalho() {
        return listaTrabalho;
    }
}

public class InfoPagamento {
    private double salarioHora, salarioSemana;
    public InfoPagamento(double sh, double ss) {
        salarioHora = sh;
        salarioSemana = ss;
    }
    public void setDetalhesPagamento(double sh, double ss) {
        salarioHora = sh;
        salarioSemana = ss;
    }
    public double getSalarioHora() { return salarioHora; }
    public double getSalarioSemana() { return salarioSemana; }
}

public class InfoTrabalho {
    private double horasTrabalhadas;
    private Date data;
    public InfoTrabalho(double h, Date d) {
        horasTrabalhadas = h;
        data = d;
    }
    public void setDetalhesTrabalho(double h, Date d) {
        horasTrabalhadas = h;
        data = d;
    }
    public void visualizaDetalhesTrabalho() {
        System.out.println("Empregado trabalhou por " + horasTrabalhadas + " hora(s) em " +
            data.toString() + ".");
    }
}

public aspect ControleAcesso {
    public pointcut operacoesSuper() :
        execution(* model.*.*Detalhes*(..)) ||
        execution(* *.Empregado.adicionaTrabalho(..));
    @AdviceName("controleOperacoesSuper")
    before() : operacoesSuper() {
        Usuario usuario = (Usuario)Sessao.instancia().getValor("usuarioAtual");
        if(usuario.getNivel() < Usuario.SUPER)
            throw new SecurityException("Somente super usuarios podem acessar esta operacao.");
    }
}

```

Figura 6.1: Parte de uma implementação em AspectJ do sistema de pagamento.

```

public class Mensagem {
    private boolean criptografa = false;
    private String mensagem = new String();
    public void enviaPersonalizada(String s) {
1   if (criptografa)
2       this.mensagem = criptografa(s);
3   else this.mensagem = s;
4       envia(this.mensagem);
    }
    public void enviaPadrao() {
5       this.mensagem = "mensagem padr\~{a}o";
6       if (criptografa)
7           this.mensagem = criptografa(this.mensagem);
8       envia(this.mensagem);
    }
    private void envia(String msg) { // envia a mensagem }
    public void setCriptografa() { criptografa = true; }
    public void setNaoCriptografa() { criptografa = false; }
    private String criptografa(String msg) {
        msg = msg + " criptografada";
        return msg;
    }
}

public aspect Registro {
    private boolean registra = false;
    private static String registro = new String();
    pointcut envio(String msg) :
        call(void Mensagem.envia(String)) && args(msg);
    @AdviceName("registra")
    before(String msg) : envio(msg) {
9       if (registra)
10          registra(msg);
    }
    private void registra(String s) { registro = registro + s; }
    public void setRegistra() { registra = true; }
    public void setNaoRegistra() { registra = false; }
    public void limpaRegistro() { registro = new String(); }
    public String getRegistro() { return registro; }
}

```

Figura 6.2: Parte de uma implementação em AspectJ do sistema de envio de mensagens.

6.3 Grafo de Fluxo Baseado em Conjuntos de Junção

Para a utilização de critérios estruturais baseados em conjuntos de junção é necessário definir um modelo que represente o fluxo de execução nas regiões afetadas por um adendo por meio do conjunto de junção correspondente. Neste trabalho é definido um grafo que deve ser construído para cada par adendo-conjunto de junção, chamado grafo Def-Use Baseado em Conjuntos de Junção (ou *PointCut-based Def-Use graph* – *PCDU*). O *PCDU* compreende os *AODUs* das unidades afetadas pelo adendo, e o *AODU* do adendo repetido em cada possível ponto de junção. O *AODU* do adendo deve ser repetido pois a intenção é rastrear sua execução em cada ponto de junção.

O $PCDU$ é definido como um grafo dirigido $PCDU(P_b, a, p) = (N, E, T, s, t)$, construído para um adendo a e seu conjunto de junção correspondente p e programa base P_b , com as informações do fluxo de dados em cada ponto de junção. Considere que a afeta n pontos de junção selecionados por p em P_b e que $adA = \{aAODU_1, \dots, aAODU_n\}$ é o conjunto formado pelo $AODU$ de a repetido em cada ponto de junção $(1, \dots, n)$ selecionado por p em P_b (os *advice AODUs*). N_a e E_a são os conjuntos de nós e arestas de cada $aAODU_i$, sendo N_{ai} e E_{ai} os nós e arestas de $aAODU_i$, com $1 \leq i \leq n$. Note que o $PCDU$ é composto pelos $AODUs$ das unidades que contêm pontos de junção selecionados por p – o que é definido nesta tese como *regiões de execução do programa afetadas pelo conjunto de junção* –, juntamente com o $AODU$ do adendo expandido em cada um dos pontos de junção. As componentes do grafo são definidas como segue:

- $N = N_b \cup N_a$ representa o conjunto completo de nós do $PCDU$, tal que:
 - N_b é o conjunto de nós dos $AODUs$ das unidades afetadas por a por meio de p (referidos a partir daqui por $AODUs$ base);
 - N_a é o conjunto de nós dos $AODUs$ em adA , isto é, $N_a = N_{a1} \cup \dots \cup N_{an}$.
- $E = E_b \cup E_a \cup E_c \cup E_g - E_d$ é o conjunto completo de arestas do $PCDU$, tal que:
 - $E_b \subseteq N_b \times N_b$ é o conjunto de arestas dos $AODUs$ base;
 - $E_a \subseteq N_{ai} \times N_{ai}$, $1 \leq i \leq n$; é o conjunto de arestas dos $AODUs$ em adA , isto é, $E_a = E_{a1} \cup \dots \cup E_{an}$;
 - E_c é o conjunto de arestas de integração criadas para conectar cada $AODU$ base com o $AODU$ correspondente em adA , tanto quando o fluxo de controle passa do $AODU$ base para o $AODU$ correspondente em adA , quanto no retorno;
 - E_g é o conjunto de arestas que conectam o nó de entrada do $PCDU$ (s) com os nós de entrada dos $AODUs$ base; e o conjunto de arestas que conectam os nós de saída de cada $AODU$ base (T) – ou algum dos $AODUs$ em adA quando exceções são lançadas dentro do adendo – com o nó de saída do $PCDU$ (t); e
 - E_d é o conjunto de arestas originais dos $AODUs$ base que conectam os nós transversais com os nós subsequentes. Essas arestas são removidas porque os nós transversais são agora conectados aos $AODUs$ correspondentes em adA .
- T é o conjunto de nós de saída dos $AODUs$ base;
- s é o nó de entrada do $PCDU$ e modela o fluxo de execução para uma unidade afetada por a por meio de p ;

- t é o nó de saída do $\mathcal{PCDU}$ e modela o fluxo de execução para fora das unidades afetadas por a por meio de p .

Os nós do $\mathcal{PCDU}$ são rotulados com duas cadeias de caracteres: um prefixo e um sufixo. O sufixo sempre corresponde ao deslocamento da primeira instrução de *bytecode* do bloco correspondente, como nos $\mathcal{AODUs}$. O prefixo é: (1) uma letra maiúscula, se o nó pertence a um $\mathcal{AODU}$ base ($n \in N_b$); ou (2) o número do ponto de junção correspondente, se o nó pertence a um $\mathcal{AODU}$ em adA ($n \in N_a$). Note que se existem múltiplos pontos de junção em uma mesma unidade afetada, o $\mathcal{AODU}$ dessa unidade também aparecerá múltiplas vezes no $\mathcal{PCDU}$.

Considere os grafos $\mathcal{AODU}$ do método `setDetalhesPagamento` e do adendo `controleOperacoesSuper` do sistema de pagamento apresentado na Seção 6.2.1, mostrados na Figura 6.3. Um exemplo de $\mathcal{PCDU}$ sem informações de fluxo de dados que modela a região de execução do sistema afetada pelo adendo `controleOperacoesSuper` por meio do conjunto de junção `superOperacoes` do aspecto `ControleAcesso` é apresentado na Figura 6.4 (JP = *Join Point*, ou seja, ponto de junção). Note que o adendo afeta quatro pontos de junção nos seguintes métodos: `setDetalhesPagamento` da classe `InfoPagamento`; `setDetalhesTrabalho` e `visualizaDetalhesTrabalho` da classe `InfoTrabalho`; e `adicionaTrabalho` da classe `Empregado`. Para facilitar o entendimento, a representação das unidades afetadas é feita utilizando uma cor mais clara. Note que algumas arestas conectam os nós com sufixo 20 ao nó de saída do $\mathcal{PCDU}$, porque nesses pontos são lançadas exceções, ou seja, o fluxo não retorna ao método afetado.

Figura 6.3: $\mathcal{AODUs}$ de duas unidades do sistema de pagamento.

Figura 6.4: *PCDU* construído a partir do adendo anterior `controleSuperOperacoes` e conjunto de junção `superOperacoes`.

Para facilitar a compreensão da definição do grafo de fluxo baseado em conjuntos de junção, considere o *PCDU* apresentado na Figura 6.4. As componentes do grafo são formadas pelos seguintes elementos:

- $N_b = \{ A:0, A:3, A:6, B:0, B:3, B:6, C:0, C:3, C:6, D:0, D:3, D:6 \}$
- $N_a = \{ 1:0, 1:3, 1:8, 1:16, 1:20, 1:30, 2:0, 2:3, 2:8, 2:16, 2:20, 2:30, 3:0, 3:3, 3:8, 3:16, 3:20, 3:30, 4:0, 4:3, 4:8, 4:16, 4:20, 4:30 \}$
- $E_b = \{ (A:0, A:3), (B:0, B:3), (C:0, C:3), (D:0, D:3) \}$
- $E_a = \{ (1:0, 1:3), (1:3, 1:8), (1:8, 1:16), (1:16, 1:30), (1:16, 1:20), (2:0, 2:3), (2:3, 2:8), (2:8, 2:16), (2:16, 2:30), (2:16, 2:20), (3:0, 3:3), (3:3, 3:8), (3:8, 3:16), (3:16, 3:30), (3:16, 3:20), (4:0, 4:3), (4:3, 4:8), (4:8, 4:16), (4:16, 4:30), (4:16, 4:20) \}$
- $E_c = \{ (A:3, 1:0), (1:30, A:6), (B:3, 2:0), (2:30, B:6), (C:3, 3:0), (3:30, C:6), (D:3, 4:0), (4:30, D:6) \}$
- $E_g = \{ (s, A:0), (1:20, t), (A:6, t), (s, B:0), (2:20, t), (B:6, t), (s, C:0), (3:20, t), (C:6, t), (s, D:0), (4:20, t), (D:6, t) \}$

- $E_d = \{ (A:3, A:6), (B:3, B:6), (C:3, C:6), (D:3, D:6) \}$
- $T = \{ A:6, B:6, C:6, D:6 \}$

É importante notar que o $\mathcal{PCDU}$ é definido genericamente para abranger qualquer tipo de adendo, conjunto de junção, ou programa base. Por exemplo, na Figura 6.5 é mostrado um programa simples que contém um adendo de contorno que afeta três pontos de junção através de um conjunto de junção que utiliza o descritor `call` em vez de `execution` e o $\mathcal{PCDU}$ correspondente. Note que os nós com sufixo 16 representam a chamada ao `proceed`, que retorna à execução ao ponto de junção original. Outro exemplo é mostrado na Figura 6.6. Dessa vez o programa contém um adendo posterior do tipo *after throwing* que afeta também três pontos de junção.

Figura 6.5: Exemplo de um programa simples com adendo de contorno e o $\mathcal{PCDU}$ correspondente.

Quando os métodos que podem ser afetados por um adendo são polimórficos, adota-se uma estratégia conservadora, ou seja, todos os possíveis métodos que podem ser afetados pelo adendo são representados no $\mathcal{PCDU}$ resultante. Essa estratégia segue a mesma linha do trabalho de Franchin (2007) para o teste par-a-par: quando um método chamado por outro em algum ponto é polimórfico, é criado um par de métodos para cada possibilidade. Um exemplo

Figura 6.6: Exemplo de um programa simples com adendo posterior *after throwing* e o *PCDU* correspondente.

em que pode ocorrer polimorfismo e o *PCDU* resultante para o adendo e conjunto de junção correspondente é mostrado na Figura 6.7. Repare que o conjunto de junção afeta a execução do método `operacao(. .)` da classe A, mas como esse método é sobrecarregado nas classes B e C, esses últimos também são representados no grafo.

6.4 Critérios de Teste

Critérios de teste são importantes para apoiar a seleção e avaliação sistemática de conjuntos de casos de teste. Para aumentar a confiança de que cada comportamento transversal de um programa OA implementado como adendo se integra corretamente com o resto do sistema, definem-se três critérios de teste, dois baseados no fluxo de controle e um baseado no fluxo de dados. A ideia é garantir que a lógica de cada adendo seja exercitada adequadamente em cada ponto de junção selecionado pelo conjunto de junção correspondente.

6.4.1 Critérios de Fluxo de Controle

Seja T um conjunto de teste para um programa P constituído de um programa base P_b e aspectos, $PCDU$ o grafo para um adendo a e conjunto de junção p de um determinado aspecto, e Π o conjunto de caminhos executado por T em P . Um nó $x \in N$ está incluído em Π se Π contém

Figura 6.7: Exemplo de um programa simples com adendo anterior que afeta métodos polimórficos e o *PCDU* correspondente.

um caminho $(n_1, \dots, n_m)$ tal que $x = n_j$ para algum j , $1 \leq j \leq m$. Similarmente, uma aresta (x_1, x_2) está incluída em Π se Π contém um caminho $(n_1, \dots, n_m)$ tal que $x_1 = n_j$ e $x_2 = n_{j+1}$ para algum j , $1 \leq j < m$.

Os critérios de fluxo de controle baseados em conjunto de junção são definidos como segue:

- **todos-nós-baseados-em-conjunto-de-junção** (todos-nós-CJ ou *all-pc-nodes*): Π satisfaz o critério todos-nós-CJ se cada nó do *ACDU* do adendo a está incluído em Π para cada ponto de junção selecionado pelo conjunto de junção p , ou seja, cada nó dos *ACDUs* em adA . Em outras palavras, este critério requer que cada nó $n \in N_a$ esteja incluído em Π .
- **todas-arestas-baseadas-em-conjunto-de-junção** (todas-arestas-CJ ou *all-pc-edges*): Π satisfaz o critério todas-arestas-CJ se cada aresta do *ACDU* do adendo está incluída em Π , para cada ponto de junção selecionado pelo conjunto de junção p , ou seja, cada aresta dos *ACDUs* em adA . Em outras palavras, este critério requer que cada aresta $e \in E_a$ esteja incluída em Π .

Exemplo de Aplicação

Para o sistema de pagamento apresentado na Seção 6.2.1, na Tabela 6.1 são apresentados

os requisitos de teste para os critérios todos-nós-CJ e todas-arestas-CJ para o adendo `controleOperacoesSuper` do aspecto `ControleAcesso` que afeta o programa por meio do conjunto de junção `superOperacoes`. Os requisitos são gerados a partir do *PCDU* mostrado na Figura 6.4.

Tabela 6.1: Requisitos de teste para cada critério de fluxo de controle baseado em conjuntos de junção, para o adendo `controleOperacoesSuper` e conjunto de junção `superOperacoes`.

Critério	Conjunto de Requisitos
todos-nós-CJ	$R_n = \{ 1:0, 1:3, 1:8, 1:16, 1:20, 1:30, 2:0, 2:3, 2:8, 2:16, 2:20, 2:30, 3:0, 3:3, 3:8, 3:16, 3:20, 3:30, 4:0, 4:3, 4:8, 4:16, 4:20, 4:30 \}$
todas-arestas-CJ	$R_e = \{ (1:0, 1:3), (1:3, 1:8), (1:8, 1:16), (1:16, 1:30), (1:16, 1:20), (2:0, 2:3), (2:3, 2:8), (2:8, 2:16), (2:16, 2:30), (2:16, 2:20), (3:0, 3:3), (3:3, 3:8), (3:8, 3:16), (3:16, 3:30), (3:16, 3:20), (4:0, 4:3), (4:3, 4:8), (4:8, 4:16), (4:16, 4:30), (4:16, 4:20), \}$

6.4.2 Critério de Fluxo de Dados

Em alguns casos, cobrir as sentenças e condicionais dos adendos nos pontos de junção não é suficiente para encontrar um defeito de integração. Por exemplo, considere um adendo anterior que afeta uma unidade em um dado programa. Considere def_v e uso_v locais em que a variável de comunicação v é definida e subsequentemente utilizada. Um grafo de fluxo de dados simples representando parte da unidade afetada e o adendo é apresentado na Figura 6.8. Note que um conjunto de casos de teste que execute os caminhos $1, 2, 4, i.1, i.2, i.4$ e $1, 2, 4, i.1, i.3, i.4$ cobre todos os nós e arestas do grafo do adendo; entretanto, se um defeito de integração está relacionado com a definição de v no nó 3 e seu subsequente uso no nó $i.3$ (caminho enfatizado no grafo), o conjunto de casos de teste não revelaria o defeito. Nesse caso, um critério de fluxo de dados como o critério definido nesta seção provavelmente revelaria o defeito, pois requereria a execução do caminho relacionado com o defeito.

Antes da definição do critério de fluxo de dados é necessário adotar um modelo de fluxo de dados para determinar o que caracteriza uma definição ou uso de variável nas instruções do programa, quais tipos de variáveis são tratadas e como são tratadas. Nesta tese, para o teste de programas AspectJ, utiliza-se o modelo definido por Franchin (Franchin, 2007; Franchin *et al.*, 2007) para *bytecode* Java¹, que foi adaptado do trabalho de Vincenzi (2004).

¹Parte desta seção foi adaptada da dissertação de Franchin (2007).

Figura 6.8: Exemplo que motiva a utilização de critério de fluxo de dados.

Modelo de Dados

No modelo definido considera-se apenas classes e métodos, já que depois da compilação e combinação, no *bytecode* resultante, os programas AspectJ são idênticos aos programas Java. Para representar cada aspecto é compilada uma classe e para representar cada adendo é compilado um método. Dessa forma, nas definições que seguem, quando houver uma referência a métodos, entenda-se métodos e adendos. O mesmo para classes e aspectos.

Vincenzi (2004) classificou as instruções de *bytecode* Java, relacionando cada uma com o fluxo de dados envolvido, ou seja, se uma instrução caracteriza uma definição, um uso, uma definição e uso (no caso da instrução *inc*) ou uma instrução que não tem implicações no fluxo de dados. Essa classificação foi apresentada na Tabela 4.1. Em seguida, Vincenzi (2004) apresentou algumas regras para a identificação de definições e usos dos diferentes tipos de variáveis em Java. Uma variável é um local de armazenamento que possui um tipo associado que pode ser primitivo ou de referência. Uma variável primitiva sempre contém um valor compatível com seu tipo e uma variável de referência armazena uma referência nula ou uma referência para qualquer objeto cuja classe (ou array) seja compatível com o tipo da variável. Na linguagem Java existem cinco tipos de variáveis: variáveis locais, variáveis agregadas, atributos estáticos, atributos de instância e parâmetros. Os cinco tipos de variáveis são discutidos a seguir.

1. Variáveis locais são unicamente visíveis dentro do escopo dos métodos nos quais elas são declaradas. Elas podem ser do tipo primitivas ou de referência. Uma variável local não é iniciada até que sua sentença de declaração seja executada.
2. Variáveis Agregadas (ou elementos de array) são variáveis anônimas que são criadas e iniciadas sempre que um novo array é criado. Existem também arrays de arrays, que são denominados multi-arrays (matrizes ou vetores multidimensionais). Para tratar variáveis agregadas

foi utilizada a abordagem proposta por Horgan e London (1991), que consideram uma variável agregada unidimensional como sendo uma única localização de memória de tal modo que, quando ocorre uma definição (uso) de qualquer um de seus elementos, considera-se que a variável agregada em si está sendo definida (usada) e não apenas o elemento particular.

3. Atributos de instância são atributos declarados em uma classe para os propósitos de cada objeto. Se uma classe *Cl* tem um atributo de instância *a*, então esse atributo é criado e iniciado assim que um novo objeto da classe *Cl* é criado. Diferentemente dos atributos estáticos, os atributos de instância armazenam valores que são associados à cada instância da classe.
4. Atributos estáticos são atributos declarados em uma classe utilizando a palavra reservada `static`. Eles são criados e recebem um valor inicial quando a classe é carregada. O valor de um atributo estático é associado à classe onde ele é definido e é compartilhado com todas as instâncias da classe.
5. Parâmetros formais representam valores de argumentos passados para um método. Para todo parâmetro presente na declaração de um método, uma nova variável local é criada no corpo do método toda vez que ele for invocado. Um parâmetro formal é iniciado com o correspondente valor de argumento passado na chamada.

A partir daí, são definidas as seguintes regras para a identificação de definições e usos de variáveis:

1. Não é considerado o uso de um literal *c*.
2. A definição ou uso de uma variável primitiva *p* é considerada como sendo unicamente a definição ou uso de *p*.
3. A definição de uma variável de referência *r* pode envolver uma referência nula ou uma referência a um objeto que está sendo construído ou que já reside na memória, que pode ser um array ou uma instância de uma classe. A definição de uma variável de referência *r* envolvendo uma referência nula é considerada como sendo somente a definição de *r*. A definição de uma variável de referência *r* envolvendo uma referência a um objeto é considerada como sendo a definição de *r* e, no caso de um array, a definição das variáveis agregadas *r[i]*. No caso de uma instância da classe *Cl* que possua *f* como atributo de instância, é considerada a definição do atributo de instância *r.f*.

4. Variáveis agregadas são consideradas como sendo uma única posição de memória. Assim, a definição de uma variável agregada $a[]$ que é um elemento de um array referenciado pela variável de referência a é considerada como sendo a definição de $a[]$ e do array referenciado por a (representado como definição de a). O uso de uma variável agregada $a[]$ é considerado como sendo o uso da variável de referência a , que permite o acesso ao elemento, e o uso da variável agregada $a[]$.
5. A definição de um atributo de instância f de uma variável de referência r do tipo da classe Cl é considerada como sendo o uso da variável de referência r , que permite o acesso ao atributo, a definição do atributo de instância (representado por $r.f$) e a definição do objeto referenciado pela variável de referência r (representado como definição de r). O uso de um atributo de instância f é considerado como sendo o uso da variável de referência r (para acessar o atributo) e uso do atributo de instância $r.f$.
6. O acesso aos atributos estáticos (ou de classe) é feito sem a necessidade de se ter uma variável de referência. Assim, a definição ou uso de qualquer atributo estático s de uma classe Cl é considerada como sendo somente a definição ou uso do atributo estático representado por $Cl.s$. Mesmo que a definição ou uso do atributo estático seja feita por meio de uma variável de referência r do tipo da classe Cl , no nível de *bytecode*, tal variável de referência é automaticamente convertida para o nome da classe, não sendo caracterizado um uso da variável de referência nesse caso.
7. Na invocação de um método de instância mi , tal como $r.mi(t_1; t_2; \dots; t_n)$, onde t_i é um parâmetro que pode ser um literal ou um dos tipos de variáveis, considera-se que ocorre um uso da variável de instância r e uso dos parâmetros $t_1, t_2, \dots, t_n$ segundo as regras descritas nos itens 1 a 6. Na invocação de um método estático ms , tal como $Cl.ms(t_1; t_2; \dots; t_n)$, considera-se que ocorrem usos dos parâmetros $t_1, t_2, \dots, t_n$ segundo as regras descritas nos itens 1 a 6.
8. Em uma atribuição de uma expressão a uma variável v da forma $v = e_1 \text{ op } e_2 \text{ op } \dots e_n$, onde e_i é um item da expressão que pode ser um literal ou um dos tipos de variáveis e op é um operador, considera-se que ocorre uso de $e_1, e_2, \dots, e_n$ conforme as regras descritas nos itens 1 a 6, e definição de v .

Na Tabela 6.2 são apresentados alguns exemplos de sentenças genéricas de Java e as correspondentes implicações no fluxo de dados. Para o entendimento dos exemplos considere os seguintes termos:

- c : um literal;

- n : um valor primitivo do tipo inteiro;
- p : uma variável primitiva;
- a : uma variável de referência a um array;
- $a[]$ uma variável agregada (ou elemento do array) que pode ser do tipo primitivo ou referência;
- Cl : uma classe que possui: um atributo de instância f , um atributo estático s , um método de instância mi e um método estático ms .
- $Cl.s$: um atributo estático da classe Cl que pode ser do tipo primitivo ou referência;
- r : uma variável de referência a um objeto do tipo da classe Cl ;
- $r.f$: um atributo de instância de r que pode ser do tipo primitivo ou referência;
- $null$: referência a nenhum objeto ou array;
- v : uma variável que pode ser p , a , $a[]$, r , $r.f$ ou $Cl.s$;
- t : um parâmetro que pode ser c ou v ;
- e : uma expressão simples ou complexa;

Além das sentenças usadas para ilustrar o modelo de fluxo de dados na Tabela 6.2, podem ser consideradas quaisquer sentenças válidas na linguagem, como por exemplo: $r.a[n] = p_1 + Cl.ms(c, p_2)$. Neste caso tem-se usos de p_1 , p_2 , r e a , e definições de $a[]$, do array referenciado por a e do objeto referenciado por r (representado como definição de r). Essa sentença é ilustrada pelas linhas 26 e 15 da Tabela 6.2.

Definição do Critério

Para o teste de fluxo de dados, decidiu-se por adaptar o critério tradicional todos-usos, que requer que todas as associações entre uma definição de variável e seus subsequentes usos sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição. Esse critério foi revisado no contexto do teste estrutural de programas OA baseado em conjuntos de junção. A ideia é levar em consideração as variáveis de interface, ou seja, dados que são compartilhados entre as unidades-base e o adendo, em cada ponto de junção. Para definir o fluxo de dados entre cada par de unidades, tomou-se como base a abordagem de teste de interfaces proposta por Linnenkugel e Müllerburg (1990) e retomada por Franchin (2007) no teste par-a-par de programas Java OO e OA.

Durante o teste de integração deve-se testar as interfaces entre as unidades que se relacionam em um programa, ou seja, testar as variáveis que influenciam diretamente a comunicação entre as unidades. Essas variáveis são denominadas variáveis de comunicação (Linnenkugel e Müllerburg, 1990). Note que ainda está-se considerando o adendo como um método compilado,

Tabela 6.2: Exemplos de sentenças Java genéricas e suas implicações no fluxo de dados.

Tipo	Sentença	Implicações no Fluxo de Dados
1	$p = c$	definição de p
2	$p = v$	uso de v e definição de p
3	$v = p$	uso de p e definição de v
4	$a = \text{new type}[n]$	definição do array referenciado por a (representado como definição de a) e definição de $a[]$
5	$a = \text{null}$	definição de a
6	$a_1 = a_2$	uso de a_2 , definição de a_1 e definição de $a_1[]$
7	$a[n] = c$	uso de a , definição de $a[]$, definição do array referenciado por a (representado como definição de a)
8	$a[n] = v$	uso de v , uso de a , definição de $a[]$ e definição do array referenciado por a (representado como definição de a)
9	$v = a[n]$	uso de a , uso de $a[]$ e definição de v
10	$r = \text{null}$	definição de r
11	$r = \text{new Cl}()$	definição do objeto referenciado por r (representado como definição de r) e definição de $r.f$
12	$r = \text{new Cl}(t_1, \dots, t_n)$	uso de $t_1, \dots$, uso de t_n , definição do objeto referenciado por r (representado como definição de r) e definição de $r.f$
13	$r_1 = r_2$	uso de r_2 , definição de r_1 e definição de $r_1.f$
14	$r.f = c$	uso de r , definição de $r.f$ e definição do objeto referenciado por r (representado como definição de r)
15	$r.f = v$	uso de v , uso de r , definição de $r.f$ e definição do objeto referenciado por r (representado como definição de r)
16	$v = r.f$	uso de r , uso de $r.f$ e definição de v
17	$Cl.s = c$	definição de $Cl.s$
18	$Cl.s = v$	uso de v e definição de $Cl.s$
19	$v = Cl.s$	uso de $Cl.s$ e definição de v
20	$v = Cl.ms(c)$	definição de v
21	$v_1 = Cl.ms(v_2)$	uso de v_2 e definição de v_1
22	$v = Cl.ms(t_1, \dots, t_n)$	uso de $t_1, \dots$, uso de t_n e definição de v
23	$v = r.mi(c)$	uso de r e definição de v
24	$v_1 = r.mi(v_2)$	uso de r , uso de v_2 e definição de v_1
25	$v = r.mi(t_1, \dots, t_n)$	uso de r , uso de $t_1, \dots$, uso de t_n e definição de v
26	$v = e_1 \text{ op } \dots e_n$	uso dos fatores de $e_1, \dots$, uso dos fatores de e_n e definição de v

de tal forma que as variáveis de comunicação são aquelas que são passadas como parâmetro para o método em *bytecode* referente ao adendo, ou variáveis que podem ser alteradas dentro desse método e impactar a unidade afetada no retorno. No caso dos adendos, essas variáveis se referem a dados que são capturados no contexto dos pontos de junção; por exemplo, pelo uso do DCJ `args`, que captura os parâmetros sendo passados para um método; ou pelo uso do DCJ `this` que captura o objeto corrente no ponto de junção. No método em *bytecode* referente ao adendo, essas variáveis podem ser de qualquer tipo da linguagem Java, isto é, de tipo primitivo ou de referência. As seguintes variáveis de comunicação são tratadas:

- Parâmetros formais (*FP — Formal Parameters*);

- Parâmetros reais (AP — *Actual Parameters*);
- Atributos estáticos das classes do programa (SF — *Static Field*);

O teste de fluxo de dados baseado em conjuntos de junção deve considerar somente os caminhos no $\mathcal{PCDU}$ (ou relações definição-uso) que influenciam diretamente a comunicação entre as unidades afetadas e as execuções do adendo relacionadas:

- para as variáveis de comunicação v que são usadas como entrada, ou seja, que são capturadas pelo adendo, consideram-se os caminhos compostos dos sub-caminhos a partir das definições de v precedentes ao ponto de junção até o ponto de junção (quando existe pelo menos um caminho livre de definição de v até esse ponto) e dos sub-caminhos a partir da entrada no adendo até o uso de v na execução do adendo relacionada; e
- para as variáveis de comunicação v que são usadas como saída, ou seja, que são manipuladas durante a execução do adendo e podem surtir efeito fora dele, consideram-se os caminhos compostos dos sub-caminhos a partir das definições de v na execução do adendo relacionada até a saída do adendo (quando existe pelo menos um caminho livre de definição de v até esse ponto) e dos sub-caminhos a partir do retorno da execução do adendo para a unidade afetada até o uso de v na unidade afetada.

Um programa OA consiste de unidades U (métodos ou adendos). Para cada unidade U são definidos os seguintes conjuntos:

- $FP-IN(U)$ é o conjunto de parâmetros formais de U usados como entrada;
- $FP-OUT(U)$ é o conjunto de parâmetros formais de U usados como saída;
- $SF-IN(U)$ é o conjunto de atributos estáticos usados em U ; e
- $SF-OUT(U)$ é o conjunto de atributos estáticos definidos em U .

Considere o contexto das unidades afetadas por um adendo a por meio do conjunto de junção p , p afetando n pontos de junção no programa. Seja Uaf_i , $1 \leq i \leq n$, uma unidade afetada por a e Ua_i , $1 \leq i \leq n$, o adendo sendo executado no ponto de junção i em Uaf_i ². O ponto de junção em Uaf_i referente ao adendo a é representado por PJ_i . Para esse ponto de junção são definidos os seguintes conjuntos:

²Note que a unidade afetada em si pode aparecer mais de uma vez no $\mathcal{PCDU}$ se possui mais de um ponto de junção relacionado com o adendo a . Nesse caso, tem-se uma Uaf para cada ponto de junção na unidade afetada.

- $AP-IN(PJ_i)$ é o conjunto de parâmetros reais usados como entrada no ponto de junção PJ_i . Ou seja, consiste nos parâmetros reais de entrada de Ua_i .
- $AP-OUT(PJ_i)$ é o conjunto de parâmetros reais usados como saída no ponto de junção PJ_i . Ou seja, consiste nos parâmetros reais de saída de Ua_i .

Para descrever as relações entre parâmetros reais e seus correspondentes parâmetros formais e entre os atributos estáticos usados pelas unidades, são definidos dois mapeamentos I_{PJ_i} e O_{PJ_i} . Antes da definição dos mapeamentos é importante ressaltar que ao fazer o mapeamento dos parâmetros e dos atributos estáticos que são do tipo de referência, ocorre tanto o mapeamento deles com seus correspondentes quanto o mapeamento das suas variáveis agregadas (caso sejam uma referência a um array). Além disso, os atributos estáticos possuem sempre a mesma identificação tanto na unidade afetada quanto no adendo.

O mapeamento I_{PJ_i} relaciona cada parâmetro real de entrada do ponto de junção PJ_i com o correspondente parâmetro formal de entrada do adendo que está sendo executado em Uaf_i , Ua_i , e cada atributo estático usado como entrada, com ele mesmo.

- $I_{PJ_i} : AP-IN(PJ_i) \cup SF-IN(Ua_i) \rightarrow FP-IN(Ua_i) \cup SF-IN(Ua_i)$, com
 - $AP-IN(PJ_i) \rightarrow FP-IN(Ua_i)$ e $SF-IN(Ua_i) \rightarrow SF-IN(Ua_i)$

O mapeamento O_{PJ_i} relaciona cada parâmetro real de saída do ponto de junção PJ_i com o correspondente parâmetro formal de saída do adendo Ua_i e cada atributo estático usado como saída, com ele mesmo.

- $O_{PJ_i} : AP-OUT(PJ_i) \cup SF-OUT(Ua_i) \rightarrow FP-OUT(Ua_i) \cup SF-OUT(Ua_i)$, com
 - $AP-OUT(PJ_i) \rightarrow FP-OUT(Ua_i)$ e $SF-OUT(Ua_i) \rightarrow SF-OUT(Ua_i)$

Com base nessas definições e no grafo $\mathcal{PCDU}$ de um adendo a e correspondente conjunto de junção p , alguns conjuntos são definidos. Para isso considere as seguintes definições: $def(x)$ é o conjunto de variáveis definidas no nó x ; $c-uso(x)$ é o conjunto de variáveis para as quais existem $c-usos$ em x ; e $p-uso(x, y)$ é o conjunto de variáveis para as quais existem $p-usos$ na aresta (x, y) (Weyuker, 1982). Assim, para cada ponto de junção PJ_i são definidos os seguintes conjuntos:

- $DEF-AFFECTED(PJ_i, v)$ é o conjunto de nós x de Uaf_i tal que $v \in def(x)$ e existe um caminho livre de definição com relação a v a partir de x até o ponto de junção, sendo que $v \in AP-IN(PJ_i)$ ou $v \in SF-IN(UA_i)$.

- $C-USE-AFFECTED(PJ_i, v)$ é o conjunto de nós x de Uaf_i tal que $v \in c-uso(x)$ e existe um caminho livre de definição com relação a v a partir do retorno da execução de Ua_i em Uaf_i até x , sendo que $v \in AP-OUT(PJ_i)$ ou $v \in SF-OUT(Ua_i)$.
- $P-USE-AFFECTED(PJ_i, v)$ é um conjunto de arestas (x, y) de Uaf_i tal que $v \in p-use(x, y)$ e existe um caminho livre de definição com relação a v a partir dos nós de retorno até (x, y) , sendo que $v \in AP-OUT(PJ_i)$ ou $v \in SF-OUT(Ua_i)$.

Para cada possível execução de adendo, ou seja, para cada Ua_i , são definidos os seguintes conjuntos:

- $C-USE-ADVICE(Ua_i, v)$ é o conjunto de nós x de Ua_i tal que $v \in c-uso(x)$ e existe um caminho livre de definição com relação a v a partir do nó de entrada de Ua_i até o nó x , sendo que $v \in FP-IN(Ua_i)$ ou $v \in SF-IN(Ua_i)$.
- $P-USE-ADVICE(Ua_i, v)$ é o conjunto de arestas (x, y) de Ua_i tal que $v \in p-uso(x, y)$ e existe um caminho livre de definição com relação a v a partir do nó de entrada de Ua_i até a aresta (x, y) , sendo que $v \in FP-IN(Ua_i)$ ou $v \in SF-IN(Ua_i)$.
- $DEF-ADVICE(Ua_i, v)$ é o conjunto de nós x de Ua_i tal que $v \in def(x)$ e existe um caminho livre de definição com relação a v a partir do nó x até o nó de saída de Ua_i , sendo que $v \in FP-OUT(Ua_i)$ ou $v \in SF-OUT(Ua_i)$.

A partir das definições anteriores é definido o critério **todos-usos-baseados-em-conjunto-de-junção** (*all-pc-uses*) utilizado para derivar requisitos de teste de fluxo de dados baseado em conjuntos de junção para as interfaces entre as unidades afetadas e as execuções do adendo.

- **todos-usos-baseados-em-conjunto-de-junção** (todos-usos-CJ ou all-pc-uses): Π satisfaz o critério todos-usos-CJ se:
 1. para cada $v \in AP-IN(PJ_i) \cup SF-IN(Ua_i)$, $1 \leq i \leq n$, Π inclui um caminho livre de definição com relação a v a partir de cada nó $x \in DEF-AFFECTED(PJ_i, v)$, $1 \leq i \leq n$, até cada nó $y \in C-USE-ADVICE(Ua_i, I_{PJ_i}(v))$, $1 \leq i \leq n$, e até cada aresta $(x, y) \in P-USE-ADVICE(Ua_i, I_{PJ_i}(v))$, $1 \leq i \leq n$. Em outras palavras, este critério requer a execução de um caminho livre de definição com relação a cada variável de comunicação a partir de cada definição relevante na unidade afetada até todo uso computacional e todo uso predicativo na execução do adendo relacionada.

2. para cada $v \in AP-OUT(PJ_i) \cup SF-OUT(Ua_i)$, $1 \leq i \leq n$, Π inclui um caminho livre de definição com relação a v a partir de cada nó $x \in DEF-ADVICE(Ua_i, O_{PJ_i}(v))$, $1 \leq i \leq n$, até cada nó $y \in C-USE-AFFECTED(PJ_i, v)$, $1 \leq i \leq n$, e até cada aresta $(x, y) \in P-USE-AFFECTED(PJ_i, v)$, $1 \leq i \leq n$. Em outras palavras, este critério requer a execução de um caminho livre de definição com relação a cada variável de comunicação a partir de cada definição relevante na execução do adendo relacionada até todo uso computacional e todo uso predicativo na unidade afetada.

Há uma exceção à cláusula (2) da definição do critério todos-usos-CJ, com relação às definições dos parâmetros formais dentro do adendo e seus posteriores usos, após o retorno da execução do adendo na unidade afetada. Em Java, a passagem de parâmetros é feita por valor, ou seja, se o parâmetro real for do tipo primitivo, o parâmetro formal correspondente recebe e armazena o dado do parâmetro real. Caso o parâmetro real seja do tipo de referência, o parâmetro formal correspondente recebe e armazena o endereço do objeto em memória referenciado pelo parâmetro real. Pode-se dizer que o parâmetro formal é uma cópia do parâmetro real. Assim, qualquer alteração no valor da cópia de um parâmetro real não o afeta, seja o parâmetro real do tipo primitivo ou de referência. Deste modo, caso haja um uso posterior do parâmetro real depois da interação com o adendo, não será caracterizado um par definição-uso.

O mesmo não ocorre se um parâmetro real for do tipo de referência e sua cópia definir/alterar, por meio do endereço de referência, os dados do objeto referenciado pelo parâmetro real. Em tal situação, a definição surtirá efeito no parâmetro real, pois o objeto que ele referencia foi modificado. Desse modo, caso haja um uso posterior do parâmetro real depois da interação com o adendo, um par definição-uso será caracterizado.

Exemplo de Aplicação

Considere o sistema de envio de mensagens apresentado na Seção 6.2.1. Na Figura 6.9 é mostrado o *PCDU* incluindo as informações de fluxo de dados para as variáveis de comunicação, para o adendo *registra* e conjunto de junção *envia*. Note que o adendo afeta os métodos *enviaPersonalizada* e *enviaPadrao*. As informações de fluxo de dados referem-se ao atributo *mensagem* da classe *Envio*, que é capturado pelo conjunto de junção antes da chamada ao método *envio* (os pontos de junção se referem às linhas 4 e 8 do código-fonte mostrado na Figura 6.2). Dentro do adendo, o valor do atributo é atribuído a variável *msg*.

O adendo anterior *registra* afeta o programa em dois pontos de junção – PJ_1 e PJ_2 –, localizados nos métodos *enviaPersonalizada* e *enviaPadrao*. As unidades afetadas

Figura 6.9: PCDU construído a partir do adendo anterior *registra* e conjunto de junção *envio*.

(Uaf) são esses dois métodos. Assim, $Uaf_1 = enviaPersonalizada$ e $Uaf_2 = enviaPadrao$. Ua_1 e Ua_2 correspondem a execução do adendo *registra* em cada um dos pontos de junção. Para os pontos de junção PJ_1 e PJ_2 , os conjuntos de variáveis de comunicação e o mapeamento do parâmetro de entrada (não existe mapeamento de parâmetros de saída pois não há implicação no fluxo de dados nos retornos às unidades afetadas), e os conjuntos de variáveis definidas são os seguintes:

- $AP-IN(PJ_1) = \{ \text{this.mensagem} \}$ e $AP-IN(PJ_2) = \{ \text{this.mensagem} \}$.
- $AP-OUT(PJ_1) = \emptyset$ e $AP-OUT(PJ_2) = \emptyset$, pois nenhuma variável é definida dentro do adendo e possivelmente utilizada posteriormente.
- $I_{PJ_1}(\text{this.mensagem}) = \text{msg}$ e $I_{PJ_2}(\text{this.mensagem}) = \text{msg}$. Este mapeamento correlaciona o atributo de instância `this.mensagem` em ambos os métodos afetados pelo adendo ao parâmetro `msg` dentro do adendo, que se refere ao valor capturado nos pontos de junção.
- $DEF-AFFECTED(PJ_1, \text{this.mensagem}) = \{ A:13, A:19 \}$ e $DEF-AFFECTED(PJ_2, \text{this.mensagem}) = \{ B:0, B:22 \}$.

Para cada Ua , os conjuntos de informações de fluxo de dados são formados pelos seguintes elementos:

- $C-USE-ADVICE(Ua_i, v) = \emptyset$ para qualquer variável v , para Ua_1 e Ua_2 , com exceção de `msg`. $C-USE-ADVICE(Ua_1, \text{msg}) = \{ 1:7 \}$ e $C-USE-ADVICE(Ua_2, \text{msg}) = \{ 2:7 \}$.
- $P-USE-ADVICE(Ua_i, v) = \emptyset$ para qualquer variável v , para Ua_1 e Ua_2 , pois não há uso predicativo de nenhuma variável de comunicação.
- $DEF-ADVICE(Ua_i, v) = \emptyset$ para qualquer variável v , para Ua_1 e Ua_2 . Isso acontece porque não há nenhuma variável definida dentro do adendo que possa ser usada fora dele, no retorno às unidades afetadas.

Para esse exemplo, os outros conjuntos de fluxo de dados definidos anteriormente não possuem nenhum elemento. Com esses conjuntos, para o critério todos-usos-CJ, são gerados os requisitos mostrados na Tabela 6.3 (no formato <variável, definição, uso>). Note que o nome utilizado para a variável nos requisitos é o nome dado a ela nas unidades afetadas e não no adendo. Além disso, é importante verificar que o par def-uso <`this.mensagem`, B:0, 2:7> é gerado como requisito porque existe um caminho que não passa pela redefinição de `this.mensagem` no nó B:22.

6.4.3 Relação de Inclusão entre os Critérios

No contexto dos critérios estruturais tradicionais de teste de unidade, foi discutida na Seção 3.6 uma hierarquia que define que, desconsiderando os caminhos não executáveis e sob certas restrições nos programas, o critério todos-usos inclui o critério todas-arestas que, por sua vez, inclui

Tabela 6.3: Requisitos de teste do critério todos-usos-CJ para o adendo `registra` e conjunto de junção `envio`.

Conjunto de Requisitos
$R_u = \{ \langle \text{this.mensagem}, A:13, 1:7 \rangle; \langle \text{this.mensagem}, A:19, 1:7 \rangle; \langle \text{this.mensagem}, B:0, 2:7 \rangle; \langle \text{this.mensagem}, B:22, 2:7 \rangle \}$

o critério todos-nós. Esses mesmos resultados poderiam ser aplicados aos critérios baseados em conjuntos de junção; ou seja, sob as mesmas condições definidas para os critérios de teste de unidade, o critério todos-usos-CJ incluiria o critério todas-arestas-CJ que, por sua vez, incluiria o critério todos-nós-CJ.

Entretanto, no que diz respeito à inclusão do critério todas-arestas-CJ pelo critério todos-usos-CJ, percebe-se que as restrições impostas são muito fortes quando consideradas neste contexto. Isso porque as variáveis levadas em conta pelo critério todos-usos-CJ são apenas as variáveis de interface que fluem nos pontos de junção. A restrição imposta para os critérios de unidade inclui as seguintes propriedades: informalmente, NSUP, que define que para cada uso predicativo de uma variável em um nó, existe um caminho livre de definição do nó inicial para esse nó; e NSL, que define que deve existir pelo menos um nó e um uso predicativo no programa. Dessa forma, no contexto do teste baseado em conjuntos de junção, isso implicaria que os programas deveriam conter pelo menos uma variável de interface em cada ponto de junção, e também um uso predicativo da mesma variável dentro de cada adendo correspondente. Essa restrição é muito forte: apenas um conjunto limitado de programas OA obedecem tais propriedades (considere-se, por exemplo, o *PCDU* apresentado na Figura 6.10(c)). Dessa forma, não se considera que o critério todos-usos-CJ inclui o critério todas-arestas-CJ. Como o contrário também não é verdadeiro, ou seja, o critério todas-arestas-CJ não pode incluir o critério todos-usos-CJ (considere-se, por exemplo, o *PCDU* da Figura 6.10(a), onde é possível cobrir todas as arestas sem cobrir todos os pares def-uso), pode-se afirmar que esses critérios são incomparáveis do ponto de vista da relação de inclusão.

No que diz respeito às relações de inclusão entre os critérios baseados em conjuntos de junção e os outros critérios definidos anteriormente, pode-se dizer que o critério todos-nós-CJ inclui o critério de unidade todos-nós, e que o critério todas-arestas-CJ inclui o critério de unidade todas-arestas, quando se considera o adendo para o qual o critério está sendo aplicado. Isso pode ser afirmado porque se um conjunto de casos de teste cobre a lógica do adendo em todos os pontos de junção em que pode executar, fica claro que, do ponto de vista da unidade, ou seja, do adendo em questão, a lógica também será coberta isoladamente.

Apesar de se poder definir essas relações entre os critérios de fluxo de controle, o mesmo não pode ser feito para os critérios de fluxo de dados todos-usos-CJ e todos-usos, porque o

conjunto de variáveis consideradas por cada um é diferente. Note-se que, como comentado anteriormente, as variáveis consideradas pelo critério todos-usos-CJ são apenas as variáveis de comunicação, ou seja, as que fluem nos pontos de junção. Já as variáveis consideradas pelo critério todos-usos são todas as que estão presentes na unidade. Dessa forma pode acontecer que um conjunto de casos de teste seja adequado para o critério todos-usos-CJ mas não seja adequado para o critério de unidade todos-usos. Por outro lado pode acontecer também que um conjunto de casos de teste seja adequado para o critério todos-usos mas não seja adequado para o critério todos-usos-CJ. Considere-se, por exemplo, quando existe uma definição de variável dentro do adendo, e uso em alguma unidade afetada, após o retorno. É possível que as entradas utilizadas para cobrir o critério todos-usos no adendo não executem um caminho que leve ao uso da variável fora do adendo, após o retorno. Dessa forma, os critérios em questão são incomparáveis do ponto de vista da relação de inclusão.

As demonstrações formais das principais considerações anteriores são apresentadas a seguir. Seja $\mathcal{T}$ um conjunto de casos de teste para um programa P , adendo a e conjunto de junção p , $\mathcal{PCDU}$ sendo o grafo para a e p em P , e seja Π o conjunto de caminhos executados por $\mathcal{T}$. Para essas demonstrações, consideram-se também a propriedade NSL definida na Seção 3.6 para o $\mathcal{PCDU}$ e também para o $\mathcal{AODU}$ do adendo a (ou seja para cada $aAODU \in adA$). Essa propriedade aplicada ao contexto dos critérios baseados em conjunto de junção deve ser entendida como: cada $aAODU \in adA$ no $\mathcal{PCDU}$ contém ao menos um comando condicional ou de repetição; pelo menos um nó dos grafos $aAODU \in adA$ tem mais de um sucessor; e pelo menos uma variável de comunicação tem um p-uso em cada $aAODU \in adA$.

- todas-arestas-CJ $\Rightarrow$ todos-nós-CJ

Suponha que $\mathcal{T}$ é todas-arestas-CJ-adequado para P , no que diz respeito ao adendo a e conjunto de junção p . Como $\mathcal{T}$ é todas-arestas-CJ-adequado, isso implica que $\forall (n_1, n_2) \in E_a$, existe um caminho $(n_1, \dots, n_m)$ em Π , tal que $n_1 = n_j$ e $n_2 = n_{j+1}$, para algum j , $1 \leq j \leq m - 1$, ou seja, todas as arestas dos grafos em adA estão incluídas em Π . Se $\mathcal{T}$ não é todos-nós-CJ-adequado, então existe pelo menos um nó $n_a \in N_a$ tal que n_a não está incluído em nenhum dos caminhos de Π . Entretanto, considerando a restrição NSL para cada $aAODU \in adA$ de a , cada nó $n_a \in N_a$ é origem ou destino de uma aresta em E_a . Desse modo, deve existir uma aresta $(n_1, n_a) \in E_a$ ou uma aresta $(n_a, n_2) \in E_a$. Portanto, se $\mathcal{T}$ é todas-arestas-CJ-adequado, ele também deve ser todos-nós-CJ-adequado.

Para provar que todos-nós-CJ $\nRightarrow$ todas-arestas-CJ, basta considerar o $\mathcal{PCDU}$ apresentado na Figura 6.10(b). Um conjunto de casos de teste que inclua os caminhos (1:0, 1:1,

1:2), (2:0, 2:1, 2:2) e (3:0, 3:1, 3:2) satisfaz o critério todos-nós-CJ, porém não satisfaz o critério todas-arestas-CJ, pois não inclui as arestas (1:0, 1:2), (2:0, 2:2) e (3:0, 3:2).

- todos-usos-CJ $\nRightarrow$ todos-usos

Para provar que o critério todos-usos-CJ não inclui o critério todos-usos-CJ, basta considerar o grafo $PCDU$ da Figura 6.10(c). Os conjuntos d correspondem às definições de variáveis e u aos usos. Um conjunto de casos de teste que inclua o caminho (A:0, A:3, 1:0, 1:11, 1:22, 1:25, 1:30, 1:35, 1:40, A:25) é adequado para o critério todos-usos-CJ, porém, não é adequado para o critério todos-usos, pois a associação def-uso $\langle y, 0, 37 \rangle$ do $AODU$ de a considerado isoladamente não é exercitada.

Figura 6.10: Grafos $PCDU$ utilizados como contraexemplos para as provas.

6.5 Estratégia Básica de Teste

Como comentado nos capítulos anteriores, a maioria dos processos de teste divide a atividade de teste em três níveis: (1) Teste de Unidade, (2) Teste de Integração e (3) Teste de Sistema (Bertolino, 2007). Seguindo esta estratégia, os critérios de teste baseados em conjunto de junção seriam mais eficientemente aplicados após o teste de unidade do sistema. Assim, a estratégia mais lógica a ser seguida nesse contexto seria: (1) enfatizar cada unidade testando cada método e adendo em isolamento (utilizando, por exemplo os critérios propostos anteriormente (Lemos *et al.*, 2007; Vincenzi *et al.*, 2006)); (2) enfatizar os interesses transversais testando cada adendo

em cada um dos pontos de junção afetados, utilizando os critérios propostos neste capítulo; e (3) se o testador desejar obter maior confiança no resto das interfaces do sistema, enfatizar cada uma delas utilizando os critérios par-a-par definidos por Franchin *et al.* (2007). Note também que os conjuntos de teste utilizados nas fase anteriores podem servir de ponto de partida para os próximos níveis, sendo melhorados conforme necessário. Desta forma pode-se ter uma atividade de teste mais eficiente.

Como existe uma hierarquia de inclusão entre os critérios baseados em conjunto de junção, desconsiderando os caminhos não executáveis e considerando algumas restrições no programa, dentro do passo (2) da estratégia apresentada ainda pode ser imposta uma ordem de aplicação dos critérios, começando pelo mais fraco e incrementando os casos de teste em direção à adequação aos critérios mais fortes. Assim, a ordem mais lógica seria aplicar primeiramente o critério todos-nós-CJ, depois o critério todas-arestas-CJ, e depois o critério todos-usos-CJ, conforme a necessidade de se construir mais casos de teste para adequar os conjuntos aos critérios mais fortes.

6.6 Limitações

Uma limitação da abordagem proposta, no que diz respeito ao teste de fluxo de dados, é que não se considera as associações def-uso de variáveis de comunicação que são ao mesmo tempo atributos de classe ou de aspecto, quando são definidas em uma unidade afetada e utilizadas em execuções do adendo em outros pontos de junção, ou quando são definidas na execução do adendo em determinado ponto de junção e utilizadas em outras unidades afetadas. No caso do exemplo de envio de mensagens apresentado na Seção 6.2.1 e o *PCDU* do adendo *registra* e conjunto de junção *envio* apresentado na Figura 6.9, nenhuma definição do atributo *mensagem* em uma das unidades afetadas poderia surtir efeito na execução do adendo na outra unidade afetada, pois não há possíveis caminhos livres de definição para essa variável. No entanto em outra aplicação esse poderia ser o caso, e poderia existir um defeito relacionado com essa associação. Para considerar tais associações, seria necessário envolver o *PCDU* com um frame, de forma que fossem rastreadas as definições de variáveis em unidades afetadas que poderiam surtir efeito no contexto de outros pontos de junção (teste sequencial – ver Seção 5.2 – similar à abordagem de Harrold e Rothermel (1994)). Entretanto é importante ficar claro que o custo de implementação e aplicação do critério seria mais alto, pois mais associações poderiam ser geradas, e a estrutura de dados adjacente é mais complexa.

Outro problema que afeta a técnica de teste estrutural em geral é a geração de requisitos relacionados com caminhos não executáveis (ou requisitos não executáveis). No caso da abor-

dagem proposta neste capítulo, eles também podem ser gerados quando a cobertura de certas estruturas no adendo (nós, arestas ou associações def-uso) exigem passar por caminhos para os quais não existem entradas possíveis para os executar. Essa característica das abordagens de teste estrutural é problemática pois, em geral, é indecidível se um requisito é executável ou não, ou seja, não existe uma maneira geral de automatizar a identificação de requisitos não executáveis. De qualquer forma, para amenizar o problema, a ferramenta que implementa a abordagem pode possibilitar que o testador indique os requisitos não executáveis, fazendo com que sejam descartados. A ferramenta que implementa a abordagem descrita neste capítulo e que será apresentada no próximo capítulo, possui essa opção.

6.7 Considerações Finais

Neste capítulo foi apresentada uma abordagem de teste estrutural para programas OA. A abordagem inclui um modelo para a representação do comportamento de cada adendo presente em um programa OA integrado ao contexto de cada possível ponto de junção. Com base neste modelo foi definida uma família com três critérios de teste – dois baseados no fluxo de controle e um baseado no fluxo de dados.

Com a abordagem de teste definida, é necessário implementá-la em uma ferramenta de teste, para viabilizar a análise de cobertura a partir dos critérios propostos. No próximo capítulo são definidas as extensões da ferramenta JaBUTi (Vincenzi, 2004), que levaram à criação da versão JaBUTi/PC-AJ (*PointCut-based for AspectJ*), para apoiar o teste estrutural baseado em conjuntos de junção de programas OA escritos em AspectJ. Para ilustrar a aplicabilidade da abordagem são apresentados alguns exemplos de aplicação da ferramenta no teste de programas OA.

Automação do Teste Estrutural Baseado em Conjuntos de Junção

7.1 Considerações Iniciais

Como observado no Capítulo 3, a atividade de teste, se realizada manualmente, é geralmente impraticável e propensa a erros. Nesse contexto, ferramentas de teste podem auxiliar a automatizar essa tarefa, permitindo a aplicação dos critérios de teste estruturais de maneira mais eficiente e prática. Além disso, as ferramentas de teste possibilitam a realização de *teste de regressão*, quando manutenções são feitas no programa e os casos de teste armazenados podem ser executados novamente para a validação das modificações realizadas.

Para implementar a abordagem de teste proposta no capítulo anterior, é necessária uma ferramenta que possibilite a instrumentação do código-objeto de Java – o *bytecode* – e derive os modelos de fluxo de controle e de dados a partir da análise estática do *bytecode*. A ferramenta JaBUTi proposta por Vincenzi (2004) e estendida no trabalho de Lemos *et al.* (2007) e de Franchin *et al.* (2007) apoia essas funcionalidades, necessitando de algumas extensões para possibilitar o teste de programas OA de acordo com a abordagem proposta neste trabalho.

Na Seção 7.2 são descritas as extensões realizadas na ferramenta JaBUTi/PW-AJ (de *PairWise testing for AspectJ*) para a obtenção da JaBUTi/PC-AJ (de *PointCut-based for AspectJ*), incluindo uma ideia quantitativa do esforço de implementação. Na Seção 7.3 são apresentadas

as operações disponíveis na ferramenta estendida, juntamente com o seu modo de utilização por meio da utilização de dois exemplos. Esses exemplos também evidenciam a eficácia dos critérios e da ferramenta proposta para encontrar um tipo de defeito específico de programas OA. Na Seção 7.4 são discutidas algumas limitações da implementação apresentada e, por fim, na Seção 7.5 são apresentadas as considerações finais do capítulo.

7.2 Extensão da Ferramenta JaBUTi para Apoio ao Teste Estrutural Baseado em Conjuntos de Junção

A ferramenta JaBUTi/PW-AJ para teste de integração par-a-par de programas OA (Franchin, 2007) foi estendida para permitir a utilização dos critérios definidos neste trabalho; ou seja, os critérios todos-nós-CJ, todas-arestas-CJ e todos-usos-CJ. Essa versão da ferramenta foi nomeada JaBUTi/PC-AJ. O grafo *PCDU* foi implementado a partir do agrupamento de grafos *PWDU* construídos para os pares que possuem o adendo como par integrado. Os critérios também foram implementados a partir do agrupamento dos requisitos gerados para os critérios par-a-par, levando também em conta os pares referentes ao adendo. Além disso, um novo ambiente de teste chamado *Integration Testing (PCD)* foi criado, para permitir o teste baseado em conjuntos de junção. Nas seções seguintes explica-se com mais detalhes a extensão da ferramenta.

7.2.1 Identificação de Pontos de Junção para Adendos Específicos

Para a utilização dos critérios baseados em conjunto de junção definidos no Capítulo 6, é necessário identificar os pares de unidades referentes a um adendo em particular, nos quais a unidade base contém um ponto de junção relacionado com o adendo. Por exemplo, para o sistema de pagamento apresentado na Seção 6.2.1, para a construção do *PCDU* do adendo `controleSuperOperacoes` e conjunto de junção `superOperacoes`, é necessário identificar quais métodos (e possivelmente adendos) são afetados pelo adendo. O modelo conceitual simplificado de programas OA mostrado na Figura 7.1 pode ser utilizado para entender melhor a ideia da implementação ¹. Um conjunto de junção seleciona diversos pontos de junção que, por sua vez, são localizados em unidades específicas.

¹Um aspecto pode conter diversos conjuntos de junção e adendos e um adendo é atrelado a um conjunto de junção. O conjunto de junção pode ser composto por outros conjuntos de junção, porém esse tipo de composição não é modelada aqui. Considere um conjunto de junção como a composição final utilizada por um adendo em particular.

Figura 7.1: Modelo conceitual simplificado de programas OA adotado nesta tese.

Na implementação da JaBUTi/PC-AJ, a classe `AdvicePointcut` mantém uma lista de adendos e conjuntos de junção relacionados, isto é, ela modela a relação entre *Adendo* e *Conjunto de junção*. Para cada instância dessa classe, é mantida uma lista de unidades que contêm pontos de junção relacionados (isto é, a relação entre *Ponto de junção* e *Unidade*). Para criar as instâncias de `AdvicePointcut`, como a família JaBUTi é baseada em *bytecode*, o *bytecode* é analisado. Para descobrir os adendos, são utilizadas funções disponibilizadas na própria distribuição do AspectJ, que também trabalham no nível de *bytecode*. O processo de identificação dos pontos de junção é simplificado, pois as possíveis interações entre adendos e outras unidades já estão resolvidas no nível de *bytecode* (isto é, o programa já está combinado). Sendo assim, só é necessário checar as chamadas a métodos no nível de *bytecode* que representam pontos de junção e agrupá-las por adendo.

7.2.2 Geração do Grafo *PCDU*

Para a construção do *PCDU* é necessário agrupar todos os pares de unidades integradas para cada par adendo-conjunto de junção. A ferramenta JaBUTi/PW-AJ já gerava os grafos para cada par de unidades; sendo assim, é necessário somente agrupá-los por adendo em um grafo único, e construir os nós que representam as entradas e saídas das regiões afetadas. Também é necessário adicionar as arestas dos nós de entrada do *PCDU* aos nós de entrada de cada unidade base afetada, e dos nós de saída dessas últimas ao nó de saída do *PCDU*.

7.2.3 Implementação dos Critérios Baseados em Conjuntos de Junção

A implementação dos critérios é feita a partir dos critérios par-a-par implementados na JaBUTi/PW-AJ. Como os critérios par-a-par calculam os requisitos para cada par de unidades integradas, para os critérios baseados em conjuntos de junção é necessário somente agrupar os requisitos do

conjunto de pares relacionados a cada par adendo-conjunto de junção. Por exemplo, considere o critério todos-nós-CJ e o sistema de pagamento apresentado na Seção 6.2.1. Os requisitos do critério todos-nós-integrados (Franchin *et al.*, 2007) inclui os nós do adendo `controleSuperOperacoes` em cada ponto que ele afeta o programa, entretanto os requisitos são gerados para cada par, separadamente. Para obter os requisitos do critério todos-nós-CJ, pode-se coletar os requisitos de todos os pares de unidades que contêm `controleSuperOperacoes` como a unidade integrada. O mesmo acontece para os outros critérios `todas-arestas-CJ` e `todos-usos-CJ`, considerando as estruturas tratadas por esses critérios.

7.2.4 Implementação do Ambiente de Teste Baseado em Conjuntos de Junção

Para apoiar a abordagem de teste baseada em conjuntos de junção, é necessário um ambiente de teste adicional aos ambientes disponíveis na JaBUTi/PW-AJ. Esse ambiente utiliza os mesmos módulos selecionados para serem testados no ambiente de teste de unidade – o ambiente inicial da JaBUTi – e, a partir desses módulos, são identificados os adendos e todas as suas interações no sistema.

O ambiente de teste baseado em conjuntos de junção contém seus próprios dados de teste, de forma que tudo o que acontece neste ambiente não afeta os outros. Um exemplo refere-se à execução dos casos de teste importados: quando se executa um caso de teste particular no ambiente de teste de unidade, por exemplo, essa execução não deveria afetar os ambientes de teste de integração.

O ambiente de teste baseado em conjuntos de junção apoia as seguintes atividades: checagem dos requisitos derivados para cada critério, importação dos casos de teste JUnit, análise da cobertura obtida pelos casos de teste importados, e visualização dos grafos para cada par adendo-conjunto de junção.

7.2.5 Esforço de Implementação

Para a extensão da ferramenta JaBUTi/PC-AJ, as seguintes classes foram implementadas no pacote `project` (responsável pelo gerenciamento dos projetos de teste): `AdviceImpl`, `AdvicePCDDData`, `AdvicePointcut`, `Aspect`, `NonAspectException`. Além disso, as seguintes classes foram implementadas no pacote `gvf` (responsável pela geração dos grafos na interface): `GVFPCCFGEntryNode` e `GVFPCCFGExitNode`. As classes que sofreram modificações foram as seguintes: `ClassFile`, `ClassIntegrationData`, `ClassMethod`, `Jabutiproject` e `TestCase` do pacote `project`; `DialogIntegration-`

Selection, JabutiGUI e TableSorterPanel do pacote gui (responsável pela interface da ferramenta); CFGFrame e GVFDDrawPanel do pacote gvff; e CFG e CFGNode do pacote graph (responsável pela implementação da estrutura dos grafos). Para dar uma ideia do esforço empreendido na extensão da ferramenta e da funcionalidade associada a cada modificação, na Tabela 7.1 são listadas as classes criadas/modificadas com uma breve descrição da funcionalidade relacionada, se a classe foi criada ou modificada (C/M) e o número de linhas de código (LDC) criadas e/ou modificadas em cada classe. Ao todo 7 classes foram criadas e 12 foram modificadas. Note que nenhum pacote foi criado, apenas foram criadas/modificadas classes dentro dos pacotes existentes.

Tabela 7.1: Arquivos criados na extensão da JaBUTi/PW-AJ para JaBUTi/PC-AJ.

Pacote	Classe	Descrição	C/M	LDC
project	AdviceImpl	Implementação dos adendos (código do AspectJ). Mapeia os métodos referentes no <i>bytecode</i> aos adendos.	C	172
	AdvicePCDDData	Classe de infra-estrutura. Armazena os pares método-adendo referentes a um conjunto de junção.	C	50
	AdvicePointcut	Similar à anterior mas utilizada somente na interface.	C	45
	Aspect	Identifica se uma classe do <i>bytecode</i> se refere a um aspecto (código do AspectJ).	C	224
	NonAspectException	Exceção lançada quando uma classe de <i>bytecode</i> analisada não é um aspecto.	C	9
	ClassFile	Adiciona lista de pares adendo-conjunto de junção às classes de <i>bytecode</i> referentes a aspectos.	M	14
	ClassIntegrationData	Classe de infra-estrutura principal. Armazena informações sobre a integração das unidades.	M	153
	ClassMethod	Identifica se um método do <i>bytecode</i> se refere a um adendo.	M	7
	JabutiProject	Adiciona a possibilidade de selecionar os pares adendo-conjunto de junção para testar.	M	7
	TestCase	Adiciona a possibilidade de verificar se um caminho executado pelo caso de teste se refere a algum caminho no grafo <i>PCDU</i> .	M	19
gui	DialogIntegrationSelection	Adiciona a possibilidade de escolher os pares adendo-conjunto de junção para testar pela interface.	M	59
	JabutiGUI	Adiciona a parte de teste baseado em conjuntos de junção à interface da JaBUTi.	M	500
	TableSorterPanel	Interface que possibilita ao usuário selecionar os pares adendo-conjunto de junção a serem testados	M	11
gvff	CFGFrame	Adiciona a parte relacionada com o teste baseado em conjuntos de junção à interface que mostra os grafos de fluxo.	M	34
	GVFDDrawPanel	Adiciona a parte relacionada com a exibição do grafo <i>PCDU</i> ao painel de exibição de grafos de fluxo.	M	7
	GVFPCCFGEntryNode	Implementa o nó de entrada do grafo <i>PCDU</i> .	C	13
	GVFPCCFGExitNode	Implementa o nó de saída do grafo <i>PCDU</i> .	C	14
graph	CFGNode.java	Adiciona os prefixos aos rótulos dos nós do grafo <i>PCDU</i> .	M	2
	CFG.java	Adiciona a opção de inserir os prefixos aos rótulos dos nós do grafo <i>PCDU</i> .	M	2
Total				1342

7.3 Exemplos de Uso da Ferramenta JaBUTi/PC-AJ

Para ilustrar o funcionamento da ferramenta JaBUTi/PC-AJ e mostrar evidências de sua efetividade, são utilizados os sistemas de pagamento e de envio de mensagens apresentados na

Seção 6.2.1. Em particular, o primeiro exemplo motiva a utilização dos critérios de fluxo de controle – todos-nós-CJ e todas-arestas-CJ – enquanto que o segundo motiva a utilização do critério de fluxo de dados – todos-usos-CJ.

Sistema de pagamento

Seguindo a estratégia de teste básica sugerida na Seção 6.5, deve-se primeiramente testar as unidades de cada uma das classes e aspectos presentes no sistema. Para isso, pode-se utilizar uma abordagem de teste funcional e analisar a cobertura utilizando os critérios de teste de unidade propostos anteriormente (Lemos *et al.*, 2007; Vincenzi *et al.*, 2006). Foi criado um conjunto de teste funcional completo em JUnit para o sistema de pagamento mostrado na Seção 6.2.1, que é adequado para os critérios de teste de unidade estruturais (todos-nós, todas-arestas e todos-usos, dependentes e independentes de exceção (Vincenzi *et al.*, 2006)). O código-fonte dos casos de teste é resumido na Figura 7.2.

Para verificar se o conjunto é adequado para os critérios de teste de unidade pode ser utilizado o ambiente de teste de unidade da JaBUTi/PC-AJ, previamente existente nas outras versões da ferramenta. Na figura 7.3 é mostrada uma tela da ferramenta com a cobertura para os critérios de teste de unidade implementados, após a execução do conjunto de casos de teste inicial. Apesar do conjunto ser adequado para os critérios mencionados, ou seja, obter 100% de cobertura, nenhum defeito é revelado a partir de sua execução. O conjunto obtém 0% de cobertura para os critérios dependentes de exceção porque não existem requisitos dependentes de exceção no sistema testado.

Para a utilização dos critérios baseados em conjuntos de junção, o ambiente de teste baseado em conjuntos de junção da JaBUTi/PC-AJ pode ser acessado. A partir daí, o conjunto de casos de teste inicial pode ser importado e a cobertura para cada critério pode ser verificada. Na Figura 7.4 é mostrada a cobertura do adendo `controleSuperOperacoes` e conjunto de junção `superOperacoes` – único par adendo-conjunto de junção do sistema –, para o critério todos-nós-CJ, após a execução do conjunto inicial. Note que alguns nós e arestas não são cobertos pelo conjunto de teste, 87% é a cobertura para o critério todos-nós-CJ. Na figura pode ser visto que a ferramenta apresenta os requisitos por ponto de junção: no caso da tela mostrada, o ponto de junção selecionado é o terceiro (dos quatro selecionados pelo conjunto de junção), referente ao método `InfoTrabalho.visualizaDetalhesTrabalho()`. O *PCDU* gerado pela JaBUTi/PC-AJ para o adendo e conjunto de junção referidos após a execução dos casos de teste é mostrado na Figura 7.5. Nesse *PCDU* também podem ser analisadas as estruturas que não foram cobertas a partir dos nós representados com cores mais escuras (os caminhos que foram executados pelos casos de teste são mostrados em branco). Nenhum


```

public class TestesIniciais
    extends TestCase {
    ...
    public void
        testAdicionaEmpregado() {
        Sessao sessao =
            Sessao.instancia();
        Usuario u = new Usuario("u1");
        session.putValor("usuarioAtual",
            u);
        u.setNivel(Usuario.BASICO);
        InfoPagamento p1 =
            new InfoPagamento(10, 20);
        Empregado e1 =
            new Empregado("John", p1);
        assertEquals("John", e1.getNome());
    }

    public void
        testEmpregadoSetNome() {
        ...
        u.setNivel(Usuario.BASICO);
        ...
        e1.setNome("Michael");
        assertEquals("Michael",
            e1.getNome());
    }

    public void
        testEmpregadoGetListTrabalho() {
        ...
        u.setNivel(Usuario.SUPER);
        ...
        InfoTrabalho w1 =
            new InfoTrabalho(5, d);
        e1.addWork(w1);
        assertEquals(w1,
            e1.getListTrabalho().get(0));
    }

    public void
        testSetDetalhesPagamento() {
        ...
        u.setNivel(Usuario.SUPER);
        ...
        assertEquals(20.0,
            p1.getSalarioHora());
        assertEquals(10.0,
            p1.getSalarioSemana());
    }

    public void
        testSetHorasTrabalhadas() {
        ...
        u.setNivel(Usuario.SUPER);
        ...
        w1.setWorkDetails(6.0, d);
        ...
    }

    public void
        testVisualizaDetalhesTrabalho() {
        ...
        u.setNivel(Usuario.SUPER);
        ...
        w1.visualizaDetalhesTrabalho();
        ...
    }

    public void
        testAdicionaTrabalhoComUsuarioNaoSuper() {
        ...
        InfoPagamento p1 =
            new InfoPagamento(10, 20);
        Empregado e1 =
            new Empregado("John", p1);
        Date d = new Date();
        InfoTrabalho w1 =
            new InfoTrabalho(5, d);
        try {
            e1.adicionaTrabalho(w1);
            fail();
        } catch (SecurityException e) {
        }
    }

    public void
        testSessionRemoveValor() {
        ...
        Usuario r =
            (Usuario)Sessao.instancia().
                getValue("usuarioAtual");
        assertNull(r);
    }
}

```

Figura 7.2: Conjunto de casos de teste adequado para os critérios de teste de unidade, para o sistema de pagamento.

requisito é gerado para o critério todos-usos-CJ pois não existem variáveis de comunicação nos pontos de junção.

Em particular, o nó 20 do adendo controleSuperOperacoes, que corresponde ao lançamento de uma exceção quando o usuário não é um super usuário, é executado somente no quarto ponto de junção: a execução do método `Empregado.adicionaTrabalho(...)` (veja o caso de teste `testAdicionaTrabalhoComNaoSuperUsuario` na Figura 7.2).

☐ All-Nodes-ei
 ☒ All-Edges-ei
 ☐ All-Uses-ei
 ☐ All-Nodes-c
 ☐ All-Edges-c
 ☐ All-Uses-c
 ☐ All-Nodes-ed
 ☐ All-Edges-ed

Overall Coverage Summary by Criterion

Testing Criterion	Coverage	Percentage
All-Nodes-ei	34 of 34	100%
All-Nodes-ed	0 of 1	0%
All-Edges-ei	12 of 12	100%
All-Edges-ed	0 of 1	0%
All-Uses-ei	27 of 27	100%
All-Uses-ed	0 of 0	0%
All-Nodes-c	4 of 4	100%
All-Edges-c	4 of 4	100%
All-Uses-c	0 of 0	0%

JaBUTi-AJ: Coverage
 Bytecode Files: 6 of 6
 Active Test Cases: 8 of 8

Figura 7.3: Cobertura do sistema de pagamento para os critérios de unidade implementados na JaBUTi, para o conjunto de testes funcionais.

☒ All-PC-Nodes
 ☐ All-PC-Edges
 ☐ All-PC-Uses

All-PC-Nodes Testing Requirements

(PCD) controle.ControleAcesso

controleOperacoesSuper before: operacoesSuper() (1)
 (modelo.InfoTrabalho) visualizaDetalhesTrabalho(V - (controle.ControleAcesso) controleOperacoesSuper before: operacoesSuper() (1)

(modelo.InfoPagamento) setDetalhesPagamento(DD)V - (controle.ControleAcesso) controleOperacoesSuper before: operacoesSuper() (1)
 (modelo.InfoTrabalho) setDetalhesTrabalho(DL java.util.Date; V - (controle.ControleAcesso) controleOperacoesSuper before: operacoesSuper() (1)
 (modelo.InfoTrabalho) visualizaDetalhesTrabalho(V - (controle.ControleAcesso) controleOperacoesSuper before: operacoesSuper() (1)
 (modelo.Empregado) adicionaTrabalho(Lmodelo.InfoTrabalho; V - (controle.ControleAcesso) controleOperacoesSuper before: operacoesSuper() (1)

Activate All Deactivate All Feasible All

Covered	Active	Infeasible	
☒	☒	☐	3:3
☒	☒	☐	3:0
☐	☒	☐	3:20
☒	☒	☐	3:30
☒	☒	☐	3:8
☒	☒	☐	3:16

ADVICE-POINTCUT COVERAGE 21 of 24 87%
 TOTAL COVERAGE 21 of 24 87%

JaBUTi-AJ: Coverage
 Coverage: All-PC-Nodes
 Active Test Cases: 8 of 8

Figura 7.4: Requisitos e cobertura do adendo `controleSuperOperacoes` e conjunto de junção correspondente para o critério todos-nós-CJ.

Os outros métodos afetados pelo adendo somente são testados a partir de super usuários. Quando o testador cria casos de teste para cobrir a parte remanescente do adendo no terceiro ponto de junção (ou seja, o ponto de junção localizado em `InfoTrabalho.visualizaDetalhesTrabalho()`), um defeito é revelado. De acordo com a descrição do sistema de pagamento apresentada na Seção 6.2.1, todos os usuários deveriam poder visualizar informações sobre as horas trabalhadas dos empregados. Como o conjunto de junção `superOperacoes` também seleciona a execução do método `InfoTrabalho.visualizaDetalhesTrabalho()`, esse método é tido como uma operação restrita a somente super usuários, quando não deveria; ou seja, nem todos os métodos cujos nomes contêm a cadeia de caracteres ‘Detalhes’ deveriam

Figura 7.5: *PCDU* do adendo `controleSuperOperacoes` e conjunto de junção correspondente após a execução de casos de teste de unidade.

ser afetados. Entretanto, como no teste de unidade essa condição somente foi testada para o quarto ponto de junção selecionado, o defeito não foi revelado nessa fase. Utilizando os critérios implementados na JaBUTi/PC-AJ, pode-se dizer que o defeito tem alta probabilidade de ser revelado, já que esses critérios requerem caminhos diretamente relacionados com o defeito. Na Figura 7.6 é mostrado o caso de teste criado para cobrir o nó 3 : 20 que revela o defeito, e nas Figuras 7.7 e 7.8 são mostrados a cobertura e requisitos para o critério e estruturas referidas, e o *PCDU* gerado pela JaBUTi/PC-AJ, depois da execução desse caso de teste adicional (note que o nó referente ao defeito é coberto). Para cobrir 100% dos requisitos dos critérios todos-nós-CJ e todas-arestas-CJ ainda é necessário cobrir as outras partes do adendo no 1º e no 2º pontos de junção selecionados. Quando essas partes são cobertas, nenhum outro defeito é revelado.

```

public class RevelaDefeito extends TestCase {
    public RevelaDefeito(String name) {
        super(name);
    }
    public void testVisualizaDetalhesTrabalhoComUsuarioBasico() {
        Sessao sessao = Sessao.instancia();
        Usuario u = new Usuario("u1");
        u.setNivel(Usuario.BASICO);
        sessao.putValor("usuarioAtual", u);

        InfoPagamento p1 = new InfoPagamento(10, 20);
        Empregado e1 = new Empregado("John", p1);
        Date d = new Date();
        InfoTrabalho w1 = new InfoTrabalho(5, d);
        try {
            w1.visualizaDetalhesTrabalho();
        } catch (SecurityException e) { fail(); }
    }
}

```

Figura 7.6: Caso de teste para cobrir requisito do critério todos-nós-CJ, e que revela o defeito do conjunto de junção.

Figura 7.7: Requisitos e cobertura do adendo `controleSuperOperacoes` e conjunto de junção correspondente para o critério todos-nós-CJ, após a execução do caso de teste que revela um defeito.

Sistema de envio de mensagens

Para o sistema de envio de mensagens, seguindo novamente a mesma estratégia de teste básica sugerida na Seção 6.5, deve-se primeiramente testar as unidades de cada uma das classes e aspectos presentes no sistema. Para isso, novamente pode-se utilizar uma abordagem de teste funcional e analisar a cobertura utilizando os critérios de unidade propostos anteriormente (Lemos *et al.*, 2007; Vincenzi *et al.*, 2006). Foi criado um conjunto de teste funcional completo em JUnit para o sistema de envio de mensagens mostrado na Seção 6.2.1, que é adequado para os critérios de teste de unidade estruturais (todos-nós, todas-arestas e todos-usos, dependentes

Figura 7.8: *PCDU* do adendo `controleSuperOperacoes` e conjunto de junção correspondente após a execução do caso de teste que revela um defeito.

e independentes de exceção (Vincenzi *et al.*, 2006)) e, além disso, adequado para os critérios baseados em conjuntos de junção todos-nós-CJ e todas-arestas-CJ. O código-fonte dos casos de teste é parcialmente mostrado na Figura 7.9. O ambiente de teste de unidade da JaBUTi/PC-AJ pode ser utilizado para verificar a cobertura das unidades e o ambiente de teste baseado em conjuntos de junção para verificar a cobertura dos critérios todos-nós-CJ e todas-arestas-CJ. Na Figura 7.10 é mostrada a tela da JaBUTi/PC-AJ com o sumário de cobertura para os critérios baseados em conjuntos de junção após a execução do conjunto de casos de teste inicial. Apesar desse conjunto ser adequado para os critérios de teste de unidade e para os critérios todos-nós-CJ e todas-arestas-CJ, nenhum defeito é revelado a partir de sua execução.

Como o conjunto de casos de teste obtém cobertura de 100% para os critérios baseados em conjuntos de junção de fluxo de controle, é interessante verificar o critério de fluxo de dados todos-usos-CJ. De acordo com a tela da JaBUTi/PC-AJ mostrada na Figura 7.10, o conjunto de casos de teste obtém cobertura de 66% para esse critério, indicando que faltam dois requisitos a serem cobertos. Ao visualizar esses requisitos, pode-se perceber que os pares def-uso `<this.mensagem, A:13, 1:7>` e `<this.mensagem, B:22, 2:7>` não são cobertos pelos casos de teste. Na JaBUTi/PC-AJ esses requisitos podem ser visualizados tanto no grafo *PCDU* quanto na tela que lista os requisitos gerados para cada critério. Na Figura 7.11 é mostrado o *PCDU* do adendo `registra` e conjunto de junção `envio` – único par adendo-conjunto de

```

public class TestesIniciais
    extends TestCase {
    ...
    public void testEnvio1() {
        Registro.aspectOf().limpaRegistro();
        Registro.aspectOf().setRegistra();

        Mensagem e = new Mensagem();
        e.setNaoCriptografa();
        e.enviaPersonalizada("minha mensagem");
        assertTrue(Registro.aspectOf().
            getRegistro().
            contains("minha mensagem"));
        assertFalse(Registro.aspectOf().
            getRegistro().
            contains("criptografada"));
    }

    public void testEnvio2() {
        Registro.aspectOf().limpaRegistro();
        Registro.aspectOf().setNaoRegistra();
        Mensagem e = new Mensagem();
        e.setCriptografa();
        e.enviaPersonalizada("outra mensagem");
        assertFalse(Registro.aspectOf().
            getRegistro().
            contains("outra mensagem"));
    }

    public void testEnvio3() {
        Registro.aspectOf().limpaRegistro();
        Registro.aspectOf().setRegistra();
        Mensagem e = new Mensagem();
        e.setNaoCriptografa();
        e.enviaPadrao();
        assertTrue(Registro.aspectOf().
            getRegistro().
            contains("mensagem padr\~{a}o"));
        assertFalse(Registro.aspectOf().
            getRegistro().
            contains("criptografada"));
    }

    public void testEnvio4() {
        Registro.aspectOf().limpaRegistro();
        Registro.aspectOf().setNaoRegistra();
        Mensagem e = new Mensagem();
        e.setCriptografa();
        e.enviaPadrao();
        assertFalse(Registro.aspectOf().
            getRegistro().
            contains("mensagem padr\~{a}o"));
    }
}

```

Figura 7.9: Conjunto de casos de teste adequado para os critérios de teste de unidade, para o sistema de envio de mensagens.

Figura 7.10: Cobertura do adendo registra e conjunto de junção envio, após a execução do conjunto de casos de teste inicial.

junção no sistema –, após a execução do conjunto de casos de teste inicial. Os nós mais escuros nas unidades afetadas – nós 13 e 22, com a definição original do grafo A:13 e B:22 – indicam que existem definições de variáveis de comunicação naqueles pontos para as quais um ou mais usos não foram cobertos ainda; no caso, a variável `this.mensagem`. Se o testador pressionar o botão esquerdo do mouse sobre esses nós, é possível visualizar os nós que contêm usos da variável para os quais essas definições obtêm cobertura; no caso, os nós 1:7 e 2:7. Esses pares def-uso se referem aos caminhos percorridos no grafo *PCDU* quando a mensagem é criptografada nos métodos `enviaPersonalizada` e `enviaPadrao` (linhas 2 e 7 no código-fonte

da Figura 6.2) e registrada no adendo (linha 10 no código-fonte da Figura 6.2). No *PCDU* são destacados os dois caminhos que se referem a esses pares def-uso, e que com alta probabilidade revelam o defeito quando executados.

Figura 7.11: *PCDU* do adendo `registra` e conjunto de junção `envio` após a execução de casos de teste iniciais, para o critério todos-usos-CJ.

Ao criar casos de teste para percorrer esses caminhos, é revelado um defeito: de acordo com a descrição do sistema na Seção 6.2.1, mensagens criptografadas não deveriam ser registradas. Apesar do conjunto de casos de teste inicial ser adequado para os critério de teste de unidade e baseados em conjuntos de junção de fluxo de controle, o defeito não é revelado na execução desse conjunto, pois os caminhos que revelam o defeito não são requeridos por esses critérios. Utilizando o critério todos-usos-CJ, pode-se dizer que o defeito tem alta probabilidade de ser revelado, já que esse critério requer a execução desses caminhos diretamente relacionados com o defeito. Na Figura 7.12 são mostrados os casos de teste criados para cobrir esses pares def-uso remanescentes, e que revelam o defeito.

```

...
public void testEnvio1() {
    Registro.aspectOf().limpaRegistro();
    Registro.aspectOf().setRegistra();

    Mensagem e = new Mensagem();
    e.setCriptografa();
    e.enviaPersonalizada("minha mensagem");
    assertTrue(Registro.aspectOf().getRegistro().contains("minha mensagem"));
    assertFalse(Registro.aspectOf().getRegistro().contains("criptografada"));
}

public void testEnvio2() {
    Registro.aspectOf().limpaRegistro();
    Registro.aspectOf().setRegistra();

    Mensagem e = new Mensagem();
    e.setCriptografa();
    e.enviaPadrao();
    assertTrue(Registro.aspectOf().getRegistro().contains("mensagem padrao"));
    assertFalse(Registro.aspectOf().getRegistro().contains("criptografada"));
}
...

```

Figura 7.12: Casos de teste para cobrir os pares def-uso remanescentes no *PCDU* do sistema de envio de mensagens que revelam um defeito.

7.4 Limitações

A ferramenta JaBUTi/PC-AJ apresenta algumas limitações no teste baseado em conjuntos de junção. No que diz respeito à escalabilidade, por ser um protótipo, atualmente a ferramenta não suporta o teste de programas muito grandes. Ao tentar testar programas grandes a ferramenta fica lenta, e às vezes não processa todas as informações para a realização da atividade de teste.

Outra limitação diretamente ligada ao teste de programas AspectJ se refere ao teste de adendos de contorno. Quando existe o uso do `proceed` para prosseguir com a execução do ponto de junção original, a representação do *PCDU* não fica completa. A ferramenta mostra o grafo corretamente, porém, no ponto em que há a chamada ao `proceed`, a ferramenta apenas mostra um nó de chamada, ou seja, não é considerado o que acontece depois, na continuação da execução do ponto de junção original (como no *PCDU* mostrado na Figura 6.5).

Na versão atual, JaBUTi/PC-AJ não mostra os prefixos dos rótulos dos nós referentes às unidades afetadas, no *PCDU*, apenas os prefixos dos nós referentes ao adendo em cada ponto de junção. Apesar disso, a cobertura é corretamente calculada, e é possível compreender o grafo e o fluxo envolvido.

7.5 Considerações Finais

Neste capítulo foi apresentada uma implementação da abordagem de teste estrutural de integração proposta neste trabalho para o teste de programas OA escritos na linguagem AspectJ. A ferramenta implementada chama-se JaBUTi/PC-AJ, e estende versões anteriores da ferramenta para o teste de unidade e par-a-par de programas OA. É importante ressaltar que a ferramenta permite realizar as atividades básicas que devem ser apoiadas por uma ferramenta de teste estrutural, como as abordadas na Seção 3.5 e na Seção 4.4: a geração e possibilidade de visualização dos modelos de fluxo de controle e de dados; a instrumentação dos programas a serem testados; a execução de casos de teste; e a análise de cobertura com base nos critérios definidos e na execução dos casos de teste. Assim, a versão JaBUTi/PC-AJ estende a família JaBUTi acrescentando o modelo e os critérios baseados em conjuntos de junção, sem mudar essencialmente seu escopo; ou seja, a ferramenta não adiciona nenhuma funcionalidade, como geração automática de casos de teste, etc.

No capítulo foi dado enfoque às mudanças da ferramenta, deixando de lado detalhes já abordados nos trabalhos de Vincenzi (2004), Lemos *et al.* (2007) e Franchin *et al.* (2007). Em particular, foi enfatizado o uso dos critérios definidos no Capítulo 6, demonstrando evidências de sua eficácia. Apesar de serem demonstradas algumas evidências da eficácia dos critérios propostos, ainda é necessário avaliá-los com maior rigor. Em particular, é necessário verificar também o custo de aplicação dos critérios, tanto do ponto de vista teórico quanto do ponto de vista experimental. No próximo capítulo são apresentados alguns estudos teóricos e experimentais exploratórios para avaliar o custo e eficácia dos critérios propostos nesta tese.

Avaliação da Abordagem de Teste Estrutural Baseado em Conjuntos de Junção

8.1 Considerações Iniciais

Neste capítulo são considerados o custo de aplicação e a eficácia dos critérios estruturais baseados em conjuntos de junção. Esses estudos são importantes porque os critérios devem ter um custo de aplicação viável para permitir seu uso na prática e, além disso, devem ser eficazes na descoberta de defeitos relacionados com a POA para serem úteis neste contexto.

Para a avaliação do custo de aplicação, foram realizados: (1) um estudo teórico sobre o custo de aplicação dos critérios de fluxo de controle baseados em conjuntos de junção; (2) um estudo experimental sobre o custo de aplicação dos critérios de fluxo de controle baseados em conjuntos de junção no qual nove sistemas OA pequenos foram realmente testados utilizando alguns critérios de teste de unidade e os critérios propostos; (3) um estudo experimental no qual três programas OA de grande porte foram utilizados para estimar o custo de aplicação dos critérios de fluxo de controle baseados em conjuntos de junção; e (4) um estudo teórico sobre o custo de aplicação do critério de fluxo de dados baseado em conjuntos de junção utilizando resultados anteriores sobre o custo de critérios de fluxo de dados tradicionais (Weyuker, 1984).

Para a avaliação da eficácia dos critérios propostos foi realizado um estudo experimental envolvendo sete alunos de pós-graduação e dois sistemas OA. Neste estudo, os alunos testaram os dois sistemas utilizando critérios de teste de unidade e os critérios de fluxo de controle baseados em conjuntos de junção em diferentes ordens, para obter evidência sobre a eficácia dos critérios propostos na detecção de defeitos diretamente relacionados à POA. Para os estudos experimentais, procurou-se utilizar quando aplicável a nomenclatura proposta por Wohlin *et al.* (2000).

Na Seção 8.2 são descritos os estudos sobre o custo de aplicação dos critérios propostos, tanto os teóricos quanto os experimentais. Na Seção 8.3 é apresentado o estudo experimental sobre a eficácia dos critérios de fluxo de controle propostos e, por fim, na Seção 8.4 são apresentadas as considerações finais do capítulo.

8.2 Avaliação do Custo de Aplicação dos Critérios Baseados em Conjuntos de Junção

Nesta seção é avaliado o custo para adequar um conjunto de casos de teste inicialmente adequado para o teste de unidade de um programa OA, aos critérios baseados em conjuntos de junção. Considera-se que o programa OA está adequadamente testado do ponto de vista das unidades se o conjunto de casos de teste obtém 100% de cobertura para os critérios todos-nós e todas-arestas adaptados para o teste de programas Java e AspectJ (Lemos *et al.*, 2007; Vincenzi *et al.*, 2006), e que essa cobertura continua sendo obtida após a integração dos módulos do programa (ou seja, consideram-se aplicações nas quais os aspectos são testados a partir dos pontos de junção presentes na própria aplicação). Como o critério todas-arestas é mais forte que o critério todos-nós sob hipóteses básicas (Weyuker, 1988), analisa-se o critério todas-arestas-CJ. Indiretamente, investiga-se também a utilidade da abordagem, mostrando se na maioria dos casos realmente são necessários mais casos de teste para obter uma cobertura de 100% dos critérios baseados em conjuntos de junção, ou se a cobertura dos critérios de teste de unidade é suficiente para cobrir também os critérios baseados em conjuntos de junção.

8.2.1 Estudo Teórico sobre o Custo dos Critérios de Fluxo de Controle

Como comentado no início desta seção, quando um conjunto de casos de teste é adequado para os critérios de teste de unidade, é garantido que cada sentença e cada arestas de cada unidade no programa são cobertos por pelo menos um caso de teste. Assumindo essa cobertura,

em geral sobram alguns caminhos a serem cobertos pelos critérios baseados em conjuntos de junção, caminhos estes que podem revelar defeitos não encontrados durante o teste de unidade do programa. Por exemplo, considere o *PCDU* apresentado na Figura 8.1(a) que representa um adendo que afeta três pontos de junção, após a execução de um conjunto de casos de teste adequado para os critérios de unidade (os caminhos executados são enfatizados no grafo). Note que todas as arestas do grafo do adendo são cobertas, considerando o grafo de fluxo do adendo isoladamente (Figura 8.1(b) – nós mais escuros identificam cobertura), e que ao menos um caminho é executado a partir de cada ponto de junção (por causa da cobertura das unidades afetadas). Para cobrir as arestas remanescentes em cada ponto de junção, são necessários no mínimo três casos de teste e nove no pior caso, isto é, quando somente duas arestas são cobertas por cada caso de teste.

(a) *PCDU* do adendo que afeta três pontos de junção, após a execução de um conjunto de casos de teste.

Figura 8.1: Cobertura de um *PCDU* e do *AODU* do adendo correspondente, considerado isoladamente.

Como comentado na Seção 3.4.2, a complexidade ciclomática (M) mede diretamente o número de caminhos linearmente independentes do código-fonte de um programa (McCabe, 1976). M é um limite superior para o número de casos de teste necessários para a obtenção da cobertura completa das arestas de um programa. Considerando que nesse contexto é necessário cobrir as arestas do adendo em cada ponto de junção afetado, M pode ser utilizado para estimar o número de casos de teste adicionais para os critérios baseados em conjuntos de junção, para cada par adendo-conjunto de junção. M é calculado como $M = E - N + 2$, onde E é o número de arestas do grafo e N é o número de nós. Se o programa tem múltiplos pontos de retorno, a fórmula fica $M = E - N + 1 + R$, onde R é o número de retornos do programa.

Como cada adendo é aplicado a um número de pontos de junção J , o número de casos de teste adicionais para cobrir os critérios propostos (CTa) pode ser calculado como, no pior caso, a complexidade ciclomática do adendo vezes o número de pontos de junção – pois as arestas devem ser cobertas em cada ponto de junção –, menos o número de pontos de junção, porque no mínimo um caminho é executado no adendo em cada ponto de junção (dada a cobertura de 100% dos nós das unidades e dos aspectos terem sido testados a partir dos pontos de junção do próprio programa). A fórmula é a seguinte: $CTa = (M \times J) - J$. Por exemplo, para o grafo apresentado na Figura 8.1(a), no pior caso, o número de casos de teste adicionais pode ser calculado como $CTa = [(12 - 10 + 2) * 3] - 3 = 9$. Uma exceção a esta fórmula acontece quando há apenas um ponto de junção afetado; neste caso, não é necessário nenhum caso de teste adicional pois todos os caminhos requeridos são executados no teste de unidade. Além disso, alguns caminhos também podem ser não executáveis, o que faz com que o número de casos de teste requeridos seja possivelmente menor.

Um experimento conduzido por Souter *et al.* (1999) mostrou evidências de que o projeto de programas OO comumente resulta em métodos com pequeno número de sentenças e fluxo de controle simples. O mesmo pode ser aplicado a programas OA que são baseados no paradigma OO (como programas implementados em AspectJ, que são baseados em Java). Também é notável que os adendos, que são construções similares a métodos, geralmente contêm as mesmas propriedades (de fato, observa-se que os adendos em geral contêm lógica até mais simples). Baseando-se nessas evidências, é possível afirmar que o número de casos de teste adicionais para os critérios de fluxo de controle baseados em conjuntos de junção será relativamente pequeno para programas que contêm adendos simples que não afetam uma quantidade muito grande de pontos de junção, o que pode ser considerado um caso comum. Isso porque os valores dos componentes E , N e J da fórmula definida serão baixos. Esse cálculo também pode ser utilizado por uma ferramenta de teste para avisar o testador quando o número de casos de teste para a cobertura dos critérios é estimado como muito alto.

8.2.2 Estudo Experimental sobre o Custo dos Critérios de Fluxo de Controle

Conduziu-se um estudo exploratório para obter evidências relacionadas com as considerações teóricas feitas anteriormente. Inspirando-se em um estudo conduzido por Xu e Rountev (2007) para avaliar uma técnica de teste de regressão para programas OA, nove programas foram coletados para a condução do estudo. Alguns desses programas também foram utilizados em outros experimentos (Programming Tools Group and Sable Research Group, 2007; Rinard *et al.*, 2004; Xu e Rountev, 2007).

O estudo pode ser definido como segue:

- Analisar a *aplicação de critérios de teste estrutural*;
- Para o propósito de *observação*;
- Com respeito ao *custo de aplicação*;
- No contexto de *teste de integração de programas OA*.

Os *materiais* utilizados no estudo foram coletados a partir de diferentes fontes; as seguintes são aplicações que foram obtidas da distribuição do AspectJ (AspectJ Team, 2003) e também utilizadas como benchmark pelo Programming Tools Group and Sable Research Group (2007): uma aplicação de componentes Java Bean utilizando aspectos (*bean*); uma aplicação 2D que modela figuras e contém um aspecto de rastreamento (*shape*); uma implementação do padrão sujeito-observador utilizando aspectos (*subj-obs*); e uma simulação de um sistema de telefonia utilizando aspectos (*telecom*). As três aplicações restantes foram obtidas do trabalho de Rinard *et al.* (2004) – uma implementação de pilha com um aspecto que previne que números negativos sejam passados como parâmetros (*stack*); do sistema de pagamento utilizado na Seção 6.2.1 e subsequentes (*payroll*); e de um tutorial de AspectJ para aplicações *Enterprise* apresentado por Bodkin e Laddad (2005) – duas versões de um serviço de música (*music1* e *music2*).

O *procedimento* do estudo foi o seguinte: primeiramente foram testadas as unidades isoladamente, guiando-se pelos critérios de teste de unidade definidos anteriormente (Lemos *et al.*, 2007; Vincenzi *et al.*, 2006) e utilizando a ferramenta JaBUTi/AJ. Depois disso, utilizando a ferramenta JaBUTi/PC-AJ, foram adicionados casos de teste aos conjuntos iniciais para adequá-los aos critérios de fluxo de controle baseados em conjuntos de junção. Nas duas etapas utilizou-se a mesma estratégia, procurando-se obter conjuntos de casos de teste otimizados com o mínimo número de casos de teste possível.

As seguintes informações foram *coletadas* para cada programa: número aproximado de linhas de código (*#LdC*); número de classes (*#C*); número de aspectos (*#A*); número de métodos (*#m*); número de adendos (*#a*); número de casos de teste requeridos para cobrir os critérios de teste de unidade todos-nós e todas-arestas (*#CTu*); número de casos de teste adicionais estimados (segundo a fórmula descrita anteriormente) para cobrir os critérios de fluxo de controle baseados em conjuntos de junção (*#ECTa*); e número de casos de teste realmente adicionados ao conjunto inicial para cobrir os critérios de fluxo de controle baseados em conjuntos de junção (*#CTa*). Considera-se o número de casos de teste adicionais como uma medida de esforço para cobrir os critérios de fluxo de controle baseados em conjuntos de junção. Na Tabela 8.1 são mostrados os *resultados* do estudo.

Tabela 8.1: Dados coletados a partir do teste de nove aplicações OA.

Subjeito	#LdC	#C	#A	#m	#a	#CTu	#ECTa	#CTa
bean	66	1	1	13	2	5	0	0
stack	39	4	2	10	2	5	0	0
shape	189	5	1	44	4	25	0	0
subj-obs	46	5	2	13	1	6	0	0
banking	21	1	2	5	4	5	2	1
telecom	289	6	3	77	6	22	2	1
payroll	109	5	1	18	1	9	4	2
music1	206	10	2	35	3	19	6	3
music2	178	7	1	32	1	14	12	5
Média	127	4,89	1,67	27,44	2,67	12,22	2,89	1,33

Note que o número de casos de teste adicionais foi de 1,33, na média, para cada aplicação (ou seja, um número pequeno). Os números de casos de teste adicionais também estão de acordo com as estimativas, ou seja, ficaram abaixo das estimativas, considerando que se estima o pior caso, e proporcionais às estimativas – quanto maior o número de casos de teste estimados, maior o número de casos de teste realmente adicionados. As aplicações que não requereram casos de teste adicionais não continham transferências condicionais dentro dos adendos, confirmando assim que, para esses casos, o teste de unidade é suficiente para cobrir os critérios de fluxo de controle baseados em conjuntos de junção.

Estudo Experimental de Estimativa para Aplicações Maiores

Estudos experimentais apresentam limitações que precisam ser consideradas quando da interpretação dos resultados. No contexto do estudo descrito anteriormente, as principais ameaças à validade das interpretações estão relacionadas com a representatividade dos programas, o que pode afetar a habilidade de generalização dos resultados. As aplicações consideradas são

pequenas e utilizadas principalmente para introduzir a POA. Aplicações comerciais com características diferentes podem estar sujeitas a diferentes níveis de custo de aplicação.

Preocupando-se com esse problema, decidiu-se analisar aplicações OA de maior porte, para checar se as estimativas de casos de teste adicionais para cobrir os critérios de fluxo de controle baseados em conjuntos de junção de fato resultam em atividades de teste praticáveis, quando comparados com o teste de unidade. Os programas analisados são aplicações reais que foram utilizadas em outros estudos (Ferrari *et al.*, 2008; Figueiredo *et al.*, 2008; Hoffman e Eugster, 2008). iBATIS (iB) para Java (iBA, 2008) é um framework de persistência que possibilita o mapeamento de consultas SQL a objetos Java. A versão do iB utilizada aqui contém alguns interesses implementados como aspectos (como tratamento de exceções e sincronização), que foram refatorados da versão OO em um estudo conjunto (Lan, 2008). *Health Watcher* (HW) é uma aplicação Web que permite que cidadãos registrem reclamações sobre problemas de saúde (Soares *et al.*, 2006). Alguns interesses implementados como aspectos no HW incluem distribuição, persistência e tratamento de exceção. *Toll System Demonstrator* (TSD) inclui um subconjunto de requisitos de um sistema de pedágio. TSD foi desenvolvido no contexto do projeto AOSD-Europe (AOSD-Europe, 2008) e contém interesses funcionais e não-funcionais implementados como aspectos (como variabilidades de cobrança, distribuição e registro).

Para cada aplicação foram coletados os seguintes dados: número aproximado de linhas de código (**#LdC**); número de classes (**#C**); número de aspectos (**#A**); número de métodos (**#m**); número de adendos (**#a**); número de casos de teste estimados para cobrir os critérios de teste de unidade discutidos anteriormente (**#ECTu**); número de casos de teste estimados como adicionais para cobrir os critérios de fluxo de controle baseados em conjuntos de junção (**#ECTa**); e porcentagem do número de casos de teste adicionais estimado em relação ao número de casos de teste estimado para o teste de unidade (**%ECTa**). Para a estimativa do número de casos de teste de unidade, foram somadas as complexidades ciclomáticas das unidades, como um limite superior para o critério todas-arestas. Para os critérios propostos, utilizou-se a fórmula discutida na Seção 8.2.1. Deve ficar claro que as estimativas são de pior caso, e que não se leva em conta as dependências das unidades. Na realidade, o número de casos de teste requeridos nos dois casos deve ser muito menor, apesar de que provavelmente as proporções devem ser mantidas (como nos exemplos do estudo relatado na seção anterior). Na Tabela 8.2 são mostrados os resultados das estimativas.

Esses resultados oferecem indícios de que os critérios de fluxo de controle baseados em conjuntos de junção são praticáveis: em média, menos de 6% do total de número de casos de teste estimados como requeridos para os critérios de teste de unidade são estimados como adicionalmente requeridos para os critérios propostos. Além disso, notou-se que a intuição discutida na Seção 8.2.1 de que os adendos geralmente possuem fluxo de controle simples e

Tabela 8.2: Estimativa de casos de teste para o teste de três aplicações OA de grande porte.

Programa	#LdC	#C	#A	#m	#a	#ECTu	#ECTa	%ECTa
iB	7278	180	38	1237	92	2229	87	4,0%
HW	4396	100	13	414	18	1064	98	9,2%
TSD	1340	80	22	322	35	544	35	6,4%
Média	4338	120	24,33	657,66	48,33	1279	73,33	5,73%

que geralmente não afetam uma quantidade muito grande de pontos de junção se verifica na prática. Para todas as aplicações analisadas, a contagem de ramos foi menor do que 1 na média (0,6) e o número de pontos de junção selecionados foi por volta de 3 na média (3,23).

8.2.3 Estudo Teórico sobre o Custo do Critério de Fluxo de Dados

Os indícios apresentados nas seções anteriores foram obtidos para os critérios de fluxo de controle baseados em conjuntos de junção. No que diz respeito ao critério todos-usos-CJ, o seu custo é estudado teoricamente nesta seção. Para estimar o custo de aplicação desse critério, utilizam-se os resultados obtidos por Weyuker (1984), que se assemelham ao estudo dos critérios de fluxo de controle descrito anteriormente, porém, também inclui critérios de fluxo de dados.

Como comentado na Seção 3.6, para um dado programa, Weyuker (1984) afirma que, no pior caso, o custo do critério tradicional todos-usos pode ser calculado como $\frac{1}{4}(t^2 + 4t + 3)$ casos de teste, onde t refere-se ao número de *condicionais* contidas no programa. No contexto de programas OA, os condicionais podem estar presentes tanto nas unidades afetadas quanto no adendo, assim, considere um adendo a que afeta n pontos de junção em um programa por meio do conjunto de junção p , e Uaf_i a unidade afetada por a que contém o i -ésimo ponto de junção (similar as definições tomadas na Seção 6.4.2). Considere t_i o número de condicionais contidas em Uaf_i , e t_a o número de condicionais contidas em a . Considere $T_{af} = \sum_{i=1}^n t_i$ o número total de condicionais das unidades afetadas por a e $T_a = t_a \times n$ o número total de condicionais do adendo somados em todos os pontos de junção. O número total de condicionais em um grafo $PCDU$ do adendo a e conjunto de junção p será $T = T_{af} + T_a$. Assim pode-se dizer também que, no pior caso, o número de casos de teste necessários para adequar um conjunto de teste ao critério todos-usos-CJ é $CT = \frac{1}{4}(T^2 + 4T + 3)$. Essa fórmula pode ser utilizada para estimar o número de casos de teste necessários para cobrir o critério todos-usos-CJ para um dado par adendo-conjunto de junção¹. Para facilitar o entendimento da fórmula, considere o

¹Como comentado na Seção 3.6, Maldonado (1991) discute que, na realidade, a complexidade do critério todos-usos é de 2^t . Nesse caso, levando em conta o formato de pior caso utilizado por Maldonado (1991), o custo

PCDU exemplo apresentado na Figura 8.2. Os condicionais das unidades afetadas e do adendo são indicados na figura. Note que o formato desse grafo é incomum, em geral encontram-se menos condicionais em métodos e adendos; ele só é utilizado a título de exemplo.

Figura 8.2: Um *PCDU* genérico construído para um adendo que afeta quatro pontos de junção.

Substituindo os números de condicionais na fórmula indicada, para esse exemplo obtém-se os seguintes valores:

- Tomando-se $n = 4$, $t_1 = 3$, $t_2 = 1$, $t_3 = 4$, $t_4 = 2$ e $t_a = 2$;

- $$T_{af} = \sum_{i=1}^4 t_i = 3 + 1 + 4 + 2 = 10;$$

do critério todos-usos-CJ também se tornaria exponencial. Entretanto, mais uma vez é importante lembrar que tais casos dificilmente aconteceriam, especialmente no contexto da POO e POA.

- $T_a = 2 \times 4 = 8$;
- $T = 10 + 8 = 18$;
- $CT = \frac{1}{4}(18^2 + 4 \times 18 + 3) = 99,75$.

Esse número de casos de teste necessários – por volta de 100 – não é muito realista, pois o exemplo é bastante incomum. Para se ter uma ideia de quantos casos de teste seriam necessários em média para o critério em questão e um programa OA comum, considere os indícios que foram obtidos para as características de programas OO por Souter *et al.* (1999) e os três programas OA de grande porte estudados na seção anterior. Utilizando esses dados, pode-se estimar aproximadamente o número de casos de teste necessários para adequar um conjunto de teste ao critério em questão, para um dado par adendo-conjunto de junção, e para um programa OA comum. Segundo os estudos de Souter *et al.* (1999), os métodos de programas OO contêm entre 0 e 3 condicionais e, nas aplicações estudadas na seção anterior, os adendos contêm em média por volta de 0,5 condicionais e em média afetam por volta de 3 pontos de junção. Dessa forma, substituindo esses valores na fórmula (tomando-se 1,5 como a média para t_i , número de condicionais das unidades afetadas, os métodos) de estimativa de casos de teste anteriormente definida, obtém-se os seguintes valores:

- Tomando-se $n = 3$, $t_i = 1,5$, para $1 \leq i \leq 3$, $t_a = 0,5$;
- $T_{af} = \sum_{i=1}^3 t_i = 1,5 + 1,5 + 1,5 = 4,5$;
- $T_a = 0,5 \times 3 = 1,5$;
- $T = 4,5 + 1,5 = 6$;
- $CT = \frac{1}{4}(6^2 + 4 \times 6 + 3) = 18,75$.

Dessa forma, para uma programa OA comum, por volta de 19 casos de teste seriam necessários para adequar um conjunto de teste ao critério todos-usos-CJ, para um dado par adendo-conjunto de junção. Repare que o programa pode conter diversos adendos, de forma que o número de adendos deve ser multiplicado por 19 para se estimar o número de casos de teste total para todo o programa. Note que esse número pode crescer bastante se a aplicação contém muitos adendos. Entretanto, o critério pode ser aplicável se a aplicação não contém muitos adendos.

Para melhor entender o custo de aplicação do critério todos-usos-CJ, é preciso recordar também que a fórmula definida por Weyuker (1984) leva em conta o pior caso dos grafos de

fluxo de dados, ou seja, o caso em que os condicionais estão todos aninhados e existe um laço que envolve uma grande parte do grafo (a forma apresentada na Figura 3.4, Seção 3.6). Segundo Weyuker (1984), como comentado anteriormente, para que um programa real necessite o número de casos de teste ditado pela fórmula de pior caso adotada, é necessário que, além do programa ter a forma particular mostrada na Figura 3.4, as definições e usos das variáveis estejam estrategicamente localizadas no grafo. De fato, poderia existir um programa com essas características, tornando o critério impraticável; entretanto, o mero fato do critério ter o limite de $O(t^2)$ não deveria ser suficiente para que fosse descartado. Por exemplo, se o grafo tivesse a forma apresentada na Figura 3.5, ainda assim bastante incomum, o critério todos-usos-CJ requereria $T + 1$ casos de teste. Dessa forma, apesar de ser interessante avaliar o pior caso para considerações de custo de aplicação, fica claro que, na prática, deve-se avaliar o custo individualmente, por programa.

Afora essas considerações, é importante notar também que a fórmula original é definida com base nas variáveis contidas em um programa procedimental comum; no caso do critério todos-usos-CJ, apenas as variáveis de comunicação, ou seja, variáveis que são compartilhadas entre as unidades afetadas e o adendo, são consideradas. Como nem todos os conjuntos de junção necessitam capturar dados disponíveis nos pontos de junção, as variáveis de comunicação não aparecem em todos os *PCDUs* gerados em um programa OA. E, mais importante, para o caso em que essas variáveis de comunicação de fato estejam presentes, o critério todos-usos-CJ se torna interessante para verificar a existência de defeitos associados a elas.

8.3 Avaliação da Eficácia dos Critérios de Fluxo de Controle Baseados em Conjuntos de Junção

Para obter indícios da eficácia dos critérios de fluxo de controle propostos nesta tese, conduziu-se um estudo exploratório com sete alunos de pós-graduação e dois sistemas OA. A ideia principal foi analisar se os critérios baseados em conjuntos de junção são eficazes para encontrar defeitos específicos de programas OA.

O estudo pode ser definido como segue:

- Analisar a *aplicação de critérios de teste estrutural*;
- Para o propósito de *observação*;
- Com respeito à *eficácia*;
- No contexto de *teste de estrutural de programas OA*.

O *material* do experimento consistiu em dois sistemas, nos quais foi injetado um defeito no conjunto de junção e comparou-se a utilização de critérios de teste de unidade com a utilização dos critérios baseados em conjuntos de junção, verificando se estes auxiliavam o testador em encontrar o defeito. Uma das aplicações é o sistema de pagamento que foi mostrado na Seção 6.2.1; a outra é um sistema similar de serviço de acesso a canções no qual um aspecto realiza a cobrança dos usuários, toda vez que uma operação cobrada é acessada. Os defeitos injetados são da mesma natureza e se referem a conjuntos de junção que afetam um ponto não desejado na aplicação. No sistema de pagamento, o conjunto de junção `superOperacoes` afeta um método que não deveria, sendo que os usuários são impedidos de visualizar informações de trabalho, que é uma operação disponível para todos os usuários segundo a descrição do sistema (como mostrado na Seção 7.3). No sistema de música, o conjunto de junção que faz a cobrança afeta um ponto de junção indesejado, cobrando do usuário por uma operação identificada como grátis na descrição do sistema (apresentar o nome dos compositores de uma música). Na Figura 8.3 são mostradas as descrições dos sistemas entregues aos alunos (o sistema de pagamento foi chamado de *Employee* e o sistema de música foi chamado de *Music Online*). O código-fonte do sistema de pagamento é o mesmo que o parcialmente mostrado na Figura 6.1, porém em Inglês; e o código-fonte do sistema de música é parcialmente mostrado na Figura 8.4 (com destaque para a parte do conjunto de junção que contém o defeito).

Descrição – Programa Employee	Descrição – Programa Music Online
<p>Considere um sistema de pagamento que armazena informações sobre empregados, informações sobre o pagamento dos empregados (salário semanal e pagamento por hora), e informações sobre o trabalho dos empregados (datas e horas trabalhadas). Para cada empregado, armazena-se seu nome, quanto vale sua hora e/ou semana de trabalho e uma lista de horas trabalhadas. Para cada empregado, podem ser armazenadas as suas horas trabalhadas, juntamente com a data em que aquelas horas de trabalho foram realizadas. O sistema também gerencia os usuários que o acessam, e as sessões de acesso. Cada usuário possui um nível de acesso que vai de usuário convidado a super usuário, passando por usuário básico. O sistema contém uma política de controle de acesso na qual apenas super usuários podem modificar informações de pagamento e trabalho. Apesar de somente super usuários poderem alterar informações, todos os usuários devem poder visualizar informação sobre as horas trabalhadas de todos os empregados.</p>	<p>Considere um sistema online para tocar e exibir informações sobre canções de uma base de dados de música. Cada usuário do sistema possui uma conta com um determinado crédito e pode acessar canções disponíveis na base de dados. A um preço específico, estipulado por canção, usuários podem tocar canções, ler letras de canções, e mostrar canções relacionadas. O mesmo preço estipulado por canção é debitado do usuário para qualquer operação (tocar canção, ler letra de canção e exibir canções relacionadas). Se um usuário tentar acessar uma operação sem ter créditos, o sistema deve emitir um erro. Os usuários também devem poder exibir os nomes dos compositores de uma canção, livre de taxas. Finalmente, uma canção pode ser um bônus, o que faz com que qualquer operação sobre ela seja livre de taxas. O sistema também gerencia as sessões de acesso dos usuários.</p>

Figura 8.3: Descrições dos sistemas utilizados no estudo exploratório.

```

public class Song implements Playable {
    private String name;
    private int price;
    private boolean bonus;
    private String composer;
    private ArrayList<Song> related = new ArrayList<Song>();
    public Song(String name, String composer, int price, boolean bonus) {
        ...
    }
    public Song() { ... }
    public String getName() { return name; }
    public String getComposer() { return composer; }
    public void play() { System.out.println("Playing song " + getName()); }
    public void showLyrics() { System.out.println("Displaying lyrics for " + getName()); }
    public void showRelated() {
        if (!related.isEmpty()) {
            System.out.println("Related songs:");
            for (Iterator<Song> it = related.iterator (); it.hasNext ();) {
                System.out.println(((Song)it.next()).getName());
            }
        } else System.out.println("No related songs.");
    }
    public void addRelated(Song s) { related.add(s); }
    public void showComposer() {
        System.out.println("The composer of the song is " + getComposer());
    }
    public boolean equals(Object o){
        Song other = (Song) o;
        return this.name.equals(other.name);
    }
    public int hashCode() { return name.hashCode(); }
    public boolean isBonus() { return bonus; }
    public void setBonus(boolean bonus) { this.bonus = bonus; }
    public void setComposer(String composer) { this.composer = composer; }
    public int getPrice() { return price; }
    public void setPrice(int price) { this.price = price; }
}

public aspect Billing {
    public pointcut useTitle() :
        execution(* Song.play(..)) || execution(* Song.show*(..));
    @AdviceName("bill")
    void around(Song song) throws InsufficientBalanceException :
        useTitle() && this(song)
        && !cflowbelow(useTitle()){
        if (song.isBonus()) {
            proceed(song);
            return;
        }
        User user = (User)Session.instance().getValue("currentUser");
        int amount = song.getPrice();
        if (amount > user.getAccount().getBalance())
            throw new InsufficientBalanceException(
                "Insufficient available balance");
        proceed(song);
        user.getAccount().debit(amount);
        System.out.println("Charge: " + user + " " + amount);
    }
}

```

Figura 8.4: Parte de uma implementação em AspectJ do sistema de música online.

8.3.1 Caracterização do Estudo

O experimento foi realizado no contexto de uma disciplina de pós-graduação realizada no ICMC-USP. Os alunos foram instruídos a partir de aulas nos tópicos de teste estrutural, teste

estrutural de programas OO, POA, e teste estrutural de programas OA. Após adquirirem esse conhecimento inicial, foi ministrada uma aula sobre a abordagem de teste baseada em conjuntos de junção para programas OA. Além disso, antes do estudo, os alunos realizaram um exercício para se familiarizarem com a JaBUTi/PC-AJ.

O *procedimento* do estudo foi o seguinte. O experimento foi realizado em duas fases. Na primeira fase pediu-se que os alunos testassem uma das aplicações, metade deles – escolhida aleatoriamente – utilizando somente os critérios de teste de unidade anteriormente implementados na JaBUTi/PC-AJ (todos-nós, todas-arestas e todos-usos) e a outra metade utilizando os critérios de fluxo de controle baseados em conjuntos de junção (todos-nós-CJ e todas-arestas-CJ). Na verdade, inicialmente o estudo continha oito alunos, de forma que as metades seriam de tamanhos iguais; porém, um dos alunos não pode realizar o estudo. Entretanto, mesmo com os grupos de tamanhos diferentes, o estudo em si não foi invalidado, pois ambos os grupos fizeram as duas atividades previstas.

Na segunda fase a segunda aplicação foi utilizada. A metade dos alunos que realizou teste de unidade na primeira fase utilizou os critérios baseados em conjuntos de junção nessa fase; e a segunda metade utilizou apenas os critérios de teste de unidade. Esse esquema foi utilizado para minimizar as diferenças entre as aplicações, entre as ordens de aplicação dos critérios, e entre as diferenças de conhecimento dos alunos. Na Figura 8.5 é mostrado o esquema utilizado para a aplicação do estudo.

Aluno	1ª fase	2ª fase
1	TU PA	TCJ PB
2		
3	TCJ PA	TU PB
4		
5	TU PA	TCJ PB
6		
7	TCJ PA	TU PB

TU = Teste de Unidade

TCJ = Teste Baseado em Conjuntos de Junção

PA = Programa A - *Employee*

PB = Programa B - *Music Online*

Figura 8.5: Esquema utilizado no estudo de avaliação de eficácia dos critérios baseados em conjuntos de junção.

No que diz respeito à *coleta de dados*, os alunos deveriam marcar os tempos gastos com as atividades de teste – e somente com elas – para dar uma ideia do esforço, e deveriam informar os defeitos encontrados ao longo da atividade, e em que momento os encontraram.

8.3.2 Resultados

Na Tabela 8.3 são mostrados os resultados do estudo. A coluna ‘Encontrou?’ indica se o aluno encontrou o defeito do conjunto de junção no teste de unidade e/ou no teste baseado em conjuntos de junção. ‘Tempo Def.’ indica em minutos quanto tempo da atividade de teste foi necessário para encontrar o defeito injetado, quando aplicável; e ‘Tempo Total’ indica o tempo para adequar os casos de teste aos critérios correspondentes.

Tabela 8.3: Resultados do estudo de eficácia dos critérios baseados em conjuntos de junção.

Aluno	Teste de Unidade			Teste baseado em CJ		
	Encontrou?	Tempo Def.	Tempo Total	Encontrou?	Tempo Def.	Tempo Total
1	N	--	55,00	S	15,00	50,00
2	S	48,00	115,00	S	25,00	36,00
3	N	--	47,00	S	48,00	82,00
4	N	--	94,00	S	19,00	39,00
5	S	35,00	50,00	S	10,00	35,00
6	S	10,00	98,00	S	28,00	90,00
7	N	--	140,00	S	20,00	60,00
Média	--	31,00	85,57	--	23,57	56,00
Moda	N	--	--	S	--	--
Taxa	43%	--	--	100%	--	--

-- = o dado não faz sentido

S/N = Sim/Não

Note que todos os alunos encontraram o defeito quando realizaram o teste baseado em conjuntos de junção; porém, menos da metade encontrou o defeito quando realizando o teste de unidade (43%). Uma observação importante é que isso aconteceu mesmo com os programas contendo um defeito similar nos conjuntos de junção, e com alunos que fizeram primeiro o teste de conjuntos de junção – encontrando o defeito – e depois o teste de unidade. Outro fato importante é que, na média, o tempo para criar os casos de teste adequados aos critérios baseados em conjuntos de junção foi menor do que aos critérios de teste de unidade (56 minutos contra 85,57 minutos). Além disso, na média, o tempo para encontrar o defeito – para os que de fato o encontraram durante a atividade de teste de unidade – também foi menor para os critérios baseados em conjuntos de junção (23,57 minutos contra 31 minutos).

8.3.3 Limitações do Estudo

É importante ressaltar que o estudo realizado é apenas exploratório e avalia os critérios de fluxo de controle baseados em conjuntos de junção somente no que diz respeito a um tipo defeito, ou seja, conjuntos de junção que selecionam pontos de junção indesejados. Além disso, os sistemas são de pequeno porte, o que dificulta a generalização dos resultados. Entretanto, como avaliação preliminar, e em conjunto com os outros estudos realizados e exemplos apresentados, esse estudo representa uma primeira evidência da eficácia dos critérios baseados em conjuntos de junção.

8.4 Considerações Finais

Neste capítulo foram apresentados alguns estudos para avaliar o custo de aplicação e a eficácia dos critérios propostos nesta tese. Apesar das limitações, os estudos apresentam evidências da aplicabilidade dos critérios – apesar do cuidado indicado para o uso do critério todos-usos-CJ, que pode requerer muitos casos de teste em alguns casos – e de sua eficácia em encontrar defeitos específicos de programas OA. Ressalta-se que para uma validação mais rigorosa dos critérios propostos outros estudos experimentais podem ser conduzidos, principalmente em direção a resultados com validade estatística.

No próximo capítulo são apresentadas as conclusões finais deste trabalho, apresentando as principais contribuições e os trabalhos futuros relacionados com esta tese.

Conclusão

9.1 Considerações Finais

Neste trabalho foi apresentada uma abordagem de teste estrutural de integração para programas OA escritos em AspectJ. A abordagem inclui um modelo subjacente para representar a integração de comportamentos transversais implementados como adendos ao resto do sistema. Com base neste modelo foi definida uma família de critérios de teste que avalia um conjunto de casos de teste no que diz respeito à cobertura dos adendos em cada ponto de junção em que podem ser executados. A família contém três critérios, dois baseados no fluxo de controle e um baseado no fluxo de dados.

Como a atividade de teste é dificilmente realizada sem o auxílio de ferramentas automatizadas, foi implementada uma ferramenta chamada JaBUTi/PC-AJ que apoia a utilização dos critérios definidos. A ferramenta é parte da família de ferramentas de teste JaBUTi, e foi implementada a partir das versões JaBUTi/PW-AJ e JaBUTi/AJ para o teste de integração par-a-par de programas OA (Franchin *et al.*, 2007) e para o teste de unidade de programas OA (Lemos *et al.*, 2007). JaBUTi/PC-AJ ainda possibilita a realização de experimentos para avaliar os critérios propostos.

Para validar a abordagem de teste proposta, foram realizados cinco estudos: dois teóricos e três experimentais. Os estudos avaliam os critérios propostos no que diz respeito ao seu

custo de aplicação e a sua eficácia em encontrar defeitos relacionados com a POA. A seguir, na Seção 9.2 são apresentadas as principais contribuições deste trabalho e, por fim, na Seção 9.3 são apresentados alguns trabalhos futuros que podem ser realizados a partir desta tese.

9.2 Contribuições

A contribuição principal desta tese é uma abordagem de teste de integração de programas OA baseada em conjuntos de junção. Essa abordagem inclui as seguintes partes relevantes:

1. Um modelo de representação da integração de adendos às outras estruturas presentes em um programa OA (o grafo $PCDU$);
2. Uma família de três critérios de teste de integração de programas OA baseados no modelo proposto, e que são mais rigorosos do que os presentes na literatura (os critérios todos-nós-CJ, todas-arestas-CJ e todos-usos-CJ); e
3. Uma ferramenta que implementa os critérios propostos (chamada JaBUTi/PC-AJ).

Foram também realizados cinco estudos de avaliação dos critérios propostos que avaliam o seu custo e eficácia, tanto do ponto de vista teórico quanto do ponto de vista experimental. A seguir são detalhadas as principais contribuições dentro do conjunto que compõe a abordagem de teste e os estudos de avaliação.

9.2.1 Grafo $PCDU$

O grafo $PCDU$ foi definido a partir das definições de outros modelos para o teste estrutural de programas OA. Em particular, foram utilizadas as definições dos grafos $AODU$ e $PWDU$ para a representação de unidades e da integração de pares de unidades em programas AspectJ. Além de possibilitar a aplicação dos critérios definidos com base no modelo, o $PCDU$ também auxilia o entendimento do comportamento transversal presente em um programa OA, pois representa o impacto da aplicação de cada adendo no programa. Nesse sentido, o $PCDU$ pode ser visto como um modelo de fluxo de controle e de dados transversal do sistema, a partir do qual é possível visualizar como cada adendo interfere nas unidades afetadas.

9.2.2 Família de Critérios Baseada em Conjuntos de Junção

As definições dos critérios todos-nós-CJ, todas-arestas-CJ e todos-usos-CJ estendem as definições de outras famílias de critérios de teste estrutural de programas OA. Em particular,

são consideradas as definições dos critérios todos-nós-transversais, todas-arestas-transversais, todos-usos-transversais (Lemos *et al.*, 2007), todos-nós-integrados, todas-arestas-integradas e todos-usos-integrados (Franchin *et al.*, 2007), que são utilizados para o teste de unidade e de integração par-a-par de programas OA. Os critérios propostos são mais rigorosos do que os presentes na literatura e, a partir da sua utilização, podem ser obtidos conjuntos de casos de teste que aumentam a confiança nos programas OA. Especificamente, garante-se que esses casos de teste exercitem com mais rigor a lógica de cada adendo em cada ponto de junção em que pode ser executado.

9.2.3 A Ferramenta JaBUTi/PC-AJ

A ferramenta JaBUTi/PC-AJ apoia a atividade de teste de integração de programas OA utilizando os critérios definidos. Além de auxiliar o testador a obter maior confiança nos programas OA a partir da utilização dos critérios, por meio da visualização dos grafos *PCDU* a ferramenta possibilita também um melhor entendimento da lógica envolvida nos programas, no que diz respeito aos comportamentos transversais implementados como adendos. A ferramenta ainda possibilita a realização de experimentos de avaliação dos critérios propostos.

9.2.4 Estudos de Avaliação da Abordagem Proposta

Os cinco estudos realizados e apresentados nesta tese oferecem indícios sobre a aplicabilidade e eficácia dos critérios propostos. Os dois estudos teóricos seguem e tomam como base a linha de estudos clássicos de avaliação de custo de critérios estruturais, principalmente os estudos realizados por Weyuker (1984). Dessa forma, permitem que outros pesquisadores sigam as mesmas linhas de raciocínio utilizadas para aproveitar os estudos teóricos anteriores. Os três estudos experimentais também oferecem subsídios para que outros pesquisadores possam realizar estudos similares que sigam a mesma linha de avaliação.

9.3 Trabalhos Futuros

Como trabalhos futuros decorrentes desta tese, destacam-se: a extensão dos critérios e modelo definidos para o teste de integração de diferentes interesses transversais que afetam o mesmo programa e a extensão dos critérios e modelo definidos para o teste de comportamentos excepcionais presentes em programas OA. Além disso, como os estudos experimentais de avaliação dos critérios apresentados nesta tese são apenas exploratórios, poderiam ser realizados outros

estudos com o objetivo de obter avaliações mais rigorosas, especialmente aquelas com validade estatística.

No que diz respeito ao teste de integração de diferentes interesses transversais presentes em um programa OA, a ideia é verificar a cobertura dos diferentes adendos que se aplicam a pontos de junção coincidentes, para aumentar a confiança de que eles interagem entre si de forma correta. Na abordagem apresentada nesta tese considera-se apenas a integração dos adendos individuais em cada ponto de junção em que eles podem executar. É possível que algum defeito esteja relacionado com a interação de mais de um adendo em pontos de junção coincidentes. Assim, garantir a cobertura mais rigorosa de cada um dos adendos nos contextos em que os dois interagem entre si pode aumentar a confiança de que essas interações estão corretas.

Em programas OA, o tratamento de exceções é uma característica que pode gerar problemas quando aspectos são utilizados em contextos em que exceções podem ser lançadas. Por exemplo, no trabalho de Coelho *et al.* (2008) é abordado o tratamento de exceções no contexto de bibliotecas de aspectos. A ferramenta desenvolvida descobre os possíveis caminhos percorridos entre o lançamento de exceções em aspectos de uma biblioteca e os tratadores de exceções da aplicação; e sugere, por exemplo, a criação de um tratador para uma exceção não esperada. Ela descobre quais exceções podem “escapar” de um aspecto e serem tratadas pela aplicação. Entretanto, a análise é feita utilizando grafos de chamadas, que contêm nós que representam métodos inteiros, e arestas conectando métodos que interagem entre si; ou seja, a estrutura interna dos métodos não é considerada em um grafo desse tipo. Dessa forma, não é possível saber quais são os possíveis caminhos que podem ser percorridos dentro do tratador de exceção: só se sabe qual método pode ou não tratar uma exceção. Além disso, como a ferramenta faz apenas análise estática, não é possível tampouco saber quais desses possíveis caminhos foram realmente percorridos por determinados casos de teste. É possível estender o modelo e critérios propostos nesta tese para auxiliar neste problema.

Para uma avaliação mais rigorosa dos critérios propostos nesta tese, seria necessária a condução de experimentos controlados, nos quais uma quantidade maior de programas fosse utilizada. Em particular, um estudo que verificasse a habilidade dos critérios em encontrar defeitos a partir do uso de uma quantidade maior de programas com defeitos de diversos tipos relacionados à POA seria interessante para obter mais evidências de sua eficácia. Em tais estudos poderiam ser obtidos resultados com validade estatística. Especificamente, seria interessante realizar estudos para verificar a eficácia do critério de fluxo de dados baseado em conjuntos de junção (o critério todos-usos-CJ), já que é mostrado apenas um exemplo que indica a sua eficácia.

Referências Bibliográficas

- iBATIS AO Joint Case Study. Disponível em: <http://sourceforge.net/projects/ibatislancaaster/> (acessado em 28/08/2008)., 2008.
- iBATIS Data Mapper framework. Disponível em: <http://ibatis.apache.org/index.html> (acessado em 28/08/2008)., 2008.
- ACREE, A. T. *On Mutation*. Tese de Doutorado, School of Information and Computer Science, Georgia Institute of Technology, Atlanta/GA - USA, 1980.
- ADRION, W. R.; BRANSTAD, M. A.; CHERNIAVSKY, J. C. Validation, Verification, and Testing of Computer Software. *ACM Comput. Surv.*, v. 14, n. 2, p. 159–192, 1982.
- AGRAWAL, H. Dominators, super blocks, and program coverage. In: *Proceedings of the 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1994)*, Portland/Oregon - USA: ACM Press, 1994, p. 25–34.
- AGRAWAL, H.; ALBERI, J.; HORGAN, J. R.; LI, J.; LONDON, S.; WONG, W. E.; GHOSH, S.; WILDE, N. Mining system tests to aid software maintenance. *IEEE Computer*, v. 31, n. 7, p. 64–73, 1998.
- ALEXANDER, R. Aspect-Oriented Programming: the Real Costs? *IEEE Software*, v. 20, n. 6, p. 90–93, 2003.
- ALEXANDER, R. T.; BIEMAN, J. M.; ANDREWS, A. A. *Towards the systematic testing of aspect-oriented programs*. Technical report, Department of Computer Science, Colorado State University, 2004.
- ALEXANDER, R. T.; BIEMAN, J. M.; GHOSH, S.; JI, B. Mutation of Java objects. In: *13th International Symposium on Software Reliability Engineering (ISSRE 2002)*, Annapolis, MD, 2002, p. 341–351.
- ALEXANDER, R. T.; OFFUTT, A. J. Analysis techniques for testing polymorphic relationships. In: *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 1999)*, Washington, DC, USA: IEEE Computer Society, 1999, p. 104.

- AOSD-EUROPE AOSD-Europe project home page. Disponível em: <http://www.aosd-europe.net/> (acessado em 20/08/2008)., 2008.
- ARMOUR, P. G. The five orders of ignorance. *Commun. ACM*, v. 43, n. 10, p. 17–20, 2000.
- ARMOUR, P. G. The unconscious art of software testing. *Commun. ACM*, v. 48, n. 1, p. 15–18, 2005.
- ASPECTJ TEAM The AspectJ programming guide. Online, disponível em: <http://www.eclipse.org/aspectj/doc/released/progguide/index.html> (acessado em 27/11/2007), 2003.
- BADRI, M.; BADRI, L.; BOURQUE-FORTIN, M. Generating unit test sequences for aspect-oriented programs: towards a formal approach using uml state diagrams. In: *Information and Communications Technology, 2005. In: ITI 3rd International Conference on Enabling Technologies for the New Knowledge Society*, 2005, p. 237–253.
- BÆKKEN, J. S. *A fault model for pointcuts and advice in AspectJ programs*. Dissertação de Mestrado, School of Electrical Engineering and Computer Science, Pullman, WA, EUA, 2006.
- BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Introdução ao teste de software. cap. 4 - Teste Estrutural, Campus-Elsevier, p. 47–76, 2007.
- BECKER, U.; HAUCK, F. J.; KLEINÖDER, J. D²al - a design-based aspect language for distribution control. In: *Workshop on Object-Oriented Technology – in conjunction with ECOOP 1998*, London, UK: Springer-Verlag, 1998, p. 420–421.
- BECKMAN, O.; GRUPTA, B. Developing test cases from use cases for web applications. In: *International Conference on Practical Software Testing Techniques (PSTT 2002)*, New Orleans, 2002.
- BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: *Future of Software Engineering (FOSE 2007)*, Washington, DC, USA: IEEE Computer Society, 2007, p. 85–103.
- BIEMAN, J. M.; GHOSH, S.; ALEXANDER, R. T. A technique for mutation of java objects. In: *Proceedings of the 16th IEEE international conference on Automated software engineering (ASE 2001)*, Washington, DC, USA: IEEE Computer Society, 2001a, p. 337.
- BIEMAN, J. M.; GHOSH, S.; ALEXANDER, R. T. A technique for mutation of Java objects. In: *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, San Diego, CA, 2001b, p. 23–26.
- BINDER, R. V. *Testing object-oriented systems: models, patterns, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

- BINNS, A.; MCGRAW, G. Building a Java software engineering tool for testing applets. In: *IntraNet 96 NY Conference*, New York – USA, 1996.
- BODKIN, R. Commercialization of AOSD: The Road Ahead. In: *Proceedings of the Workshop on Commercialization of AOSD Technology – in conjunction with AOSD 2003*, Boston, Massachussets, USA, 2003.
- BODKIN, R.; LADDAD, R. Enterprise AspectJ tutorial using eclipse. Online, eclipseCon 2005. Available from: http://www.eclipsecon.org/2005/presentations/EclipseCon2005_EnterpriseAspectJTutorial9.pdf (accessed 12/3/2007), 2005.
- BROOKS, F. P. No silver bullet: essence and accidents of software engineering. *Computer*, v. 20, n. 4, p. 10–19, 1987.
- BUDD, T. A. *Mutation analysis: Ideas, example, problems and prospects*, cáp. Computer Program Testing North-Holand Publishing Company, 1981.
- BUDKOWSKI, S.; DEMBINSKI, P. An introduction to estelle: a specification language for distributed systems. *Comput. Netw. ISDN Syst.*, v. 14, n. 1, p. 3–23, 1987.
- BUY, U.; ORSO, A.; PEZZE, M. Automated testing of classes. In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, New York, NY, USA: ACM Press, 2000, p. 39–48.
- BYBRO, M. *A mutation testing tool for Java programs*. Dissertação de Mestrado, Stockholm University, Stockholm, Sweden, 2003.
- CECCATO, M.; TONELLA, P.; RICCA, F. Is AOP code easier or harder to test than oop code? In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD 2005*, Chicago/IL, USA, 2005.
- CHAIM, M. L. *PokeTool – Uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados*. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, SP, 1991.
- CHAIM, M. L.; CARNIELLO, A.; JINO, M. Teste baseado em casos de uso. Boletim de Pesquisa e Desenvolvimento, disponível em: <http://www.cnptia.embrapa.br/files/bp10.pdf>. (acessado em: 12/2/2009), 2003.
- CHEESMAN, J.; DANIELS, J. *Uml components: a simple process for specifying component-based software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.
- CHEN, M.-H.; KAO, H. M. Testing object-oriented programs: An integrated approach. In: *Proceedings of the 10th International Symposium on Software Reliability Engineering (IS-SRE 1999)*, Washington, DC, USA: IEEE Computer Society, 1999, p. 73.

- CHEVALLEY, P. Applying mutation analysis for object-oriented programs using a reflective approach. In: *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, Macao - China: IEEE Computer Society Press, 2001, p. 267–272.
- CHOW, T. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*, v. SE-4, n. 3, p. 178–187, 1978.
- CLARKE, S.; BANIASSAD, E. *Aspect-oriented analysis and design : The theme approach*. Addison-Wesley Professional, 2005.
- COELHO, R.; KULESZA, U.; RASHID, A.; VON STAA, A.; LUCENA, C. J. P. Unveiling and taming liabilities of aspect libraries reuse. In: *Anais do 22º Simpósio Brasileiro de Engenharia de Software (SBES 2008)*, 2008.
- DELAMARO, M. E. *Proteum: Um ambiente de teste baseado na análise de mutantes*. Dissertação de Mestrado, ICMC/USP, São Carlos/SP - Brasil, 1993.
- DELAMARO, M. E. *Mutação de Interface: Um critério de adequação interprocedimental para o teste de integração*. Tese de Doutorado, IFSC/USP, 1997.
- DELAMARO, M. E.; BARBOSA, E. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Introdução ao teste de software. cap. 5 - Teste de Mutação, Campus-Elsevier, p. 77–118, 2007a.
- DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. Introdução ao teste de software. cap. 1 - Conceitos Básicos, Campus-Elsevier, p. 1–7, 2007b.
- DEMILLO, R. A. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, n. 4, p. 34–43, 1978.
- DEURSEN, A.; MARIN, M.; MOONEN, L. A systematic aspect-oriented refactoring and testing strategy, and its application to jhotdraw. *CoRR*, v. abs/cs/0503015, 2005.
- DEUTSCH, M. S. *Software verification and validation*. Englewood Cliffs, New Jersey: Prentice-Hall, 1982.
- DIJKSTRA, E. W. *A discipline of programming*. Prentice-Hall, 1976.
- DOMINGUES, A. L. S. *Avaliação de critérios e ferramentas de teste para programas oo*. Dissertação de Mestrado, ICMC/USP, São Carlos/SP - Brasil, 2002.
- EADDY, M.; AHO, A.; HU, W.; McDONALD, P.; BURGER, J. Debugging aspect-enabled programs. In: *SC2007: Proceedings of Software Composition*, 2007, p. 200–215.
- ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Communications of the ACM*, v. 44, n. 10, p. 29–32, 2001.
- FABBRI, S. C. P. F.; MALDONADO, J. C.; DELAMARO, M. E.; MASIERO, P. C. Proteum/FSM: A tool to support finite state machine validation based on mutation testing. In: *XIX SCCC – International Conference of the Chilean Computer Science Society*, Talca, Chile, 1999a, p. 96–104.

- FABBRI, S. C. P. F.; MALDONADO, J. C.; MASIERO, P. C.; DELAMARO, M. E. Mutation analysis applied to validate specifications based on petri nets. In: *8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols (FORTE 1995)*, Montreal, Canada: Kluwer Academic Publishers, 1995, p. 329–337.
- FABBRI, S. C. P. F.; MALDONADO, J. C.; SUGETA, T.; MASIERO, P. C. Mutation testing applied to validate specifications based on statecharts. In: *ISSRE – International Symposium on Software Reliability Systems*, 1999b, p. 210–219.
- FERRARI, F. C.; MALDONADO, J. C. *Teste de software orientado a aspectos: Uma revisão sistemática*. Relatório Técnico 291, ICMC/USP, 2007.
- FERRARI, F. C.; MALDONADO, J. C.; RASHID, A. Mutation testing for aspect-oriented programs. In: *Proc. of the 1st International Conference on Software Testing, Verification, and Validation (ICST 2008)*, 2008, p. 52–61.
- FIGUEIREDO, E.; CACHO, N.; SANT’ANNA, C.; MONTEIRO, M.; KULESZA, U.; GARCIA, A.; SOARES, S.; FERRARI, F.; KHAN, S.; FILHO, F. C.; DANTAS, F. Evolving software product lines with aspects: an empirical study on design stability. In: *Proc. of the 30th International Conference on Software Engineering (ICSE 2008)*, ACM Press, 2008, p. 261–270.
- FONSECA, R. P. *Suporte ao teste estrutural de programas FORTRAN no ambiente poke-tool*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas, SP, 1993.
- FRANCHIN, I. G. Estudo e implementação de uma abordagem de teste de integração para programas orientados a aspectos. Exame de Qualificação de Mestrado – Instituto de Ciências Matemáticas e de Computação – USP – São Carlos, SP, 2006.
- FRANCHIN, I. G. *Teste estrutural de integração par-a-par de programas orientados a objetos e a aspectos: Critérios e automatização*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, 2007.
- FRANCHIN, I. G.; LEMOS, O. A. L.; MASIERO, P. C. Pairwise structural testing of object and aspect-oriented java programs. In: *Anais do 21º Simpósio Brasileiro de Engenharia de Software (SBES 2007)*, Porto Alegre, RS, Brasil: SBC Press, 2007, p. 377–393.
- FRANKL, F. G. *The use of data flow information for the selection and evaluation of software test data*. Tese de Doutorado, University of New York, NY, 1987.
- FRANKL, F. G.; WEYUKER, E. J. Data flow testing tools. In: *Softfair II*, San Francisco, CA, 1985, p. 46–53.
- FRANKL, P. G.; WEYUKER, E. J. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, v. 14, n. 10, p. 1483–1498, 1988.
- FRANKL, P. G.; WEYUKER, E. J. Testing software to detect and reduce risk. *Journal of Systems and Software*, v. 53, n. 3, p. 275–286, 2000.

- FRIEDMAN, A. D. *Logical design of digital systems*. Computer Science Press, 1975.
- HAREL, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, v. 8, n. 3, p. 231–274, 1987.
- HARROLD, M. J.; ROTHERMEL, G. Performing data flow testing on classes. In: *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, New York, NY: ACM Press, 1994, p. 154–163.
- HERMAN, P. M. A data flow analysis approach to program testing. *Australian Computer Journal*, v. 8, n. 3, 1976.
- HEUMANN, J. Is a use case a test case? In: *International Conference on Practical Software Testing Techniques (PSTT 2001)*, St. Paul, 2001.
- HILSDALE, E.; HUGUNIN, J. Advice Weaving in AspectJ. In: *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, 2004, p. 26–35.
- HOFFMAN, D.; STROOPER, P. Classbrench: A framework for automated class testing. *Software Practice and Experience*, v. 27, n. 5, p. 573–597, 1997.
- HOFFMAN, K.; EUGSTER, P. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In: *Proc. of the 30th International Conference on Software Engineering (ICSE 2008)*, USA: ACM Press, 2008, p. 91–100.
- HORGAN, J. R.; LONDON, S. Data Flow Coverage and the C Language. In: *Proceedings of the Symposium on Testing, Analysis, and Verification*, Victoria, British Columbia, Canada: ACM Press, 1991, p. 87–97.
- HOWDEN, W. E. *Functional program testing and analysis*. New York/NY - USA: McGraw-Hill, Inc., 1986.
- HUANG, J. Experience Using AspectJ to Implement Cord. In: *Proc. OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2000.
- IEEE *IEEE standard glossary of software engineering terminology*. Standard 610.12, Institute of Electric and Electronic Engineers, 1990.
- KARR, D. A.; RODRIGUES, C.; LOYALL, J. P.; SCHANTZ, R. E.; KRISHNAMURTHY, Y.; PYARALI, I.; SCHMIDT, D. C. Application of the quo quality-of-service framework to a distributed video application. In: *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, Washington, DC, USA: IEEE Computer Society, 2001, p. 299.
- KICZALES, G. Episode 11: Interview Gregor Kiczales. Disponível em: http://www.se-radio.net/index.php?post_id=78154, Software Engineering Radio, 2006.

- KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, 2001, p. 327–353.
- KICZALES, G.; MEZINI, M. Aspect-oriented programming and modular reasoning. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, ACM Press, 2005, p. 49–58.
- KIM, S.; CLARK, J. A.; MCDERMID, J. A. Assessing test set adequacy for object-oriented programs using class mutation. In: *Proc. of the Symposium on Software Technology (SoST 1999)*, 1999, p. 72–83.
- KIM, S.; CLARK, J. A.; MCDERMID, J. A. Class mutation: mutation testing for object-oriented programs. In: *Proc. of the Net.Objectdays 2000*, 2000, p. 9–12.
- KOREL, B.; LASKI, J. W. A tool for data flow oriented program testing. In: *Softfair II*, San Francisco, CA, 1985, p. 34–38.
- KUNG, D.; LU, Y.; VENUGOPALAN, N.; HSIA, P.; TOYOSHIMA, Y.; CHEN, C.; GAO, J. Object state testing and fault analysis for reliable software systems. In: *7th International Symposium on Software Reliability Engineering*, White Plains, NY: IEEE Computer Society Press, 1996, p. 76–85.
- LADDAD, R. *Aspectj in action*. Manning, 2003.
- LEITÃO, P. S. J. *Suporte ao teste estrutural de programas COBOL no ambiente poke-tool*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas, SP, 1992.
- LEMONS, O. A. L. Is ‘advice’ adequate? AOSD discussion list, disponível em: http://aosd.net/pipermail/discuss_aosd.net/2004-November/001178.html, 2004.
- LEMONS, O. A. L. *Teste de programas orientados a aspectos: Uma abordagem estrutural para AspectJ*. Dissertação de Mestrado, ICMC/USP, São Carlos/SP - Brasil, 2005.
- LEMONS, O. A. L.; FERRARI, F. C.; MASIERO, P. C.; LOPES, C. V. Testing aspect-oriented programming pointcut descriptors. In: *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs – in conjunction with ISSSTA 2006*, New York, NY, USA: ACM Press, 2006a, p. 33–38.
- LEMONS, O. A. L.; JUNQUEIRA, D. C.; SILVA, M. A. G.; DE MATTOS FORTES, R. P.; STAMEY, J. Using aspect-oriented PHP to implement crosscutting concerns in a collaborative web system. In: *Proceedings of the 24th annual ACM international conference on Design of communication (SIGDOC 2006)*, New York, NY, USA: ACM, 2006b, p. 134–141.
- LEMONS, O. A. L.; MALDONADO, J. C.; MASIERO, P. C. Structural unit testing of AspectJ programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD 2005*, Chicago/IL, USA, 2005.

- LEMONS, O. A. L.; VINCENZI, A. M. R.; MALDONADO, J. C.; MASIERO, P. C. Teste de unidade de programas orientados a aspectos. In: *Anais do 18th Simpósio Brasileiro de Engenharia de Software (SBES 2004)*, Brasília/DF - Brasil, 2004, p. 55–70.
- LEMONS, O. A. L.; VINCENZI, A. M. R.; MALDONADO, J. C.; MASIERO, P. C. Control and data flow structural testing criteria for aspect-oriented programs. *J. Syst. Softw.*, v. 80, n. 6, p. 862–882, 2007.
- LESIECKI, N. Unit test your aspects. *IBM developerWorks Homepage*, disponível em <http://www-128.ibm.com/developerworks/java/library/j-aopwork11/> - último acesso em 10/01/2006, 2005.
- LINNENKUGEL, U.; MÜLLERBURG, M. Test data selection criteria for (software) integration testing. In: *Proceedings of the first international conference on systems integration (ISCI 1990)*, Piscataway, NJ, USA: IEEE Press, 1990, p. 709–717.
- LOPES, C. V. *D: A Language Framework for Distributed Programming*. Tese de Doutorado, College of Computer Science, Northeastern University, Boston, MA, 1997.
- LOPES, C. V. AOP: A historical perspective (What's in a name?). In: FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKŞIT, M., eds. *Aspect-Oriented Software Development*, Boston: Addison-Wesley, p. 97–122, 2005.
- LOPES, C. V.; DOURISH, P.; LORENZ, D. H.; LIEBERHERR, K. Beyond AOP: toward naturalistic programming. *SIGPLAN Notices*, v. 38, n. 12, p. 34–43, 2003.
- MA, Y.-S.; KWON, Y.-R.; OFFUTT, J. Inter-class mutation operators for java. In: *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002)*, Washington, DC, USA: IEEE Computer Society, 2002, p. 352.
- MALDONADO, J. C. *Crítérios potenciais usos: Uma contribuição ao teste estrutural de software*. Tese de Doutorado, DCA/FEE/UNICAMP, Campinas, SP - Brasil, 1991.
- MALDONADO, J. C.; CHAIM, M. L.; JINO, M. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos. In: *XXII Congresso Nacional de Informática*, São Paulo, SP, 1989.
- MATHUR, A. P.; WONG, W. E. Evaluation of the cost of alternative mutation strategies. In: *VII Simpósio Brasileiro de Engenharia de Software (SBES 1993)*, Rio de Janeiro, RJ, Brazil, 1993, p. 320–335.
- MATHUR, A. P.; WONG, W. E. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, v. 4, n. 1, p. 9–31, 1994.
- MCCABE, T. J. A complexity measure. In: *Proc. of the 2nd International Conference on Software Engineering (ICSE 1976)*, USA: IEEE Computer Society Press, 1976, p. 407.

- MCEACHEN, N.; ALEXANDER, R. T. Distributing classes with woven concerns: An exploration of potential fault scenarios. In: *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, Illinois: ACM Press, 2005, p. 192–200.
- MCGREGOR, J. D. Functional testing of classes. In: *Proc. 7th International Quality Week*, San Francisco, CA: Software Research Institute, 1994.
- MORELL, L. J. A theory of fault-based testing. *IEEE Trans. Softw. Eng.*, v. 16, n. 8, p. 844–857, 1990.
- MORTENSEN, M.; ALEXANDER, R. T. *Adequate testing of aspect-oriented programs*. Technical Report CS 01-110, Department of Computer Science, Colorado State University, 2004.
- MORTENSEN, M.; ALEXANDER, R. T. An approach for adequate testing of AspectJ programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD 2005*, Chicago/IL, USA, 2005.
- MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. *The art of software testing*. 2nd. ed. John Wiley & Sons, 2004.
- NTAFOS, S. C. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, v. 14, n. 6, p. 868–874, 1988.
- OFFUT, J. Mutation testing. Disponível em: <http://cs.gmu.edu/~offutt/rsrch/mut.html> (acessado em 7/2/2009)., 2009.
- OFFUTT, A. J.; IRVINE, A. Testing object-oriented software using the category-partition method. In: *17th International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA: Prentice-Hall, 1995, p. 293–304.
- OFFUTT, J.; ABDURAZIK, A. Generating tests from uml specifications. In: FRANCE, R.; RUMPE, B., eds. *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Springer, 1999, p. 416–429.
Disponível em <http://citeseer.ist.psu.edu/403942.html>
- OFFUTT, J.; ALEXANDER, R.; WU, Y.; XIAO, Q.; HUTCHINSON, C. A fault model for subtype inheritance and polymorphism. In: *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Hong Kong - China: IEEE Computer Society Press, 2001, p. 84–93.
- OPENSOURCETESTING.ORG Open source functional testing tools. Disponível em: <http://www.opensourcetesting.org/functional.php> (acessado em 7/2/2009)., 2009.
- PANDE, H. D.; LANDI, W.; ; RYDER, B. G. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, v. 20, n. 5, p. 385–403, 1994.

- PARASOFT Parasoftware delivers quality as a continuous process. Disponível em: <http://www.parasoftware.com> (acessado em 12/2/2009), 2009.
- PRESSMAN, R. S. *Software engineering - a practitioner's approach*. 5th. ed. McGraw-Hill, 2000.
- PRICE, A. M.; ZORZO, A. Visualizando o fluxo de controle de programas. In: *IV Simpósio Brasileiro de Engenharia de Software (SBES 1990)*, Águas de São Pedro, SP, 1990.
- PROGRAMMING TOOLS GROUP AND SABLE RESEARCH GROUP abc: The aspectbench compiler for AspectJ. Online, disponível em: <http://abc.comlab.ox.ac.uk/benchmarks> (acessado em 27/11/2007), 2007.
- RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for program test data selection. In: *Proceedings of the 6th International Conference on Software Engineering (ICSE 1982)*, Tokio, Japan, 1982, p. 272–278.
- RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. 11, n. 4, p. 367–375, 1985.
- RATIONAL SOFTWARE CORPORATION PureCoverage. Disponível em: <http://www-306.ibm.com/software/awdtools/purifyplus/>. (acessado em: 12/2/2009), 2000.
- REISMAN, S. Management and integrated tools. *IEEE Software*, v. 7, n. 3, 1990.
- RINARD, M.; SALCIANU, A.; BUGRARA, S. A classification system and analysis for aspect-oriented programs. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2004/FSE-12)*, Newport Beach/CA - USA: ACM Press, 2004, p. 147–158.
- ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. *Qualidade de software*. Prentice Hall, 2001.
- RYDER, B. G.; SOFFA, M. L. Influences on the design of exception handling: ACM SIGSOFT project on the impact of software engineering research on programming language design. *SIGPLAN Not.*, v. 38, n. 6, p. 16–22, 2003.
- RYDER, B. G.; SOFFA, M. L.; BURNETT, M. The impact of software engineering research on modern programming languages. *ACM Trans. Softw. Eng. Methodol.*, v. 14, n. 4, p. 431–477, 2005.
- SIMÃO, A. S. Introdução ao teste de software. cap. 3 - Teste Baseado em Modelos, Campus-Elsevier, p. 27–45, 2007.
- SIMÃO, A. S.; MALDONADO, J. C. Mutation based test sequence generation for Petri nets. In: *Proceedings of III Workshop of Formal Methods*, João Pessoa/PB - Brasil, 2000, p. 68–79.

- SOARES, S.; BORBA, P.; LAUREANO, E. Distribution and persistence as aspects. *Softw. Pract. Exper.*, v. 36, n. 7, p. 711–759, 2006.
- SOUTER, A. L. *Context-driven testing of object-oriented systems*. Tese de Doutorado, University of Delaware, professor In Charge-Lori L. Pollock, 2002.
- SOUTER, A. L.; POLLOCK, L. L. Omen: A strategy for testing object-oriented software. In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, New York, NY, USA: ACM Press, 2000, p. 49–59.
- SOUTER, A. L.; POLLOCK, L. L. The construction of contextual def-use associations for object-oriented systems. *IEEE Trans. Softw. Eng.*, v. 29, n. 11, p. 1005–1018, 2003.
- SOUTER, A. L.; POLLOCK, L. L.; HISLEY, D. Inter-class def-use analysis with partial class representations. In: *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 1999)*, New York, NY, USA: ACM Press, 1999, p. 47–56.
- SOUZA, S. R. S.; FABBRI, S. C. P. F.; BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Introdução ao teste de software. cap. 10 - Estudos Teóricos e Experimentais, Campus-Elsevier, p. 251–268, 2007.
- STAFF, S. M. L. *Fiscal 1997 project portfolio report*. Relatório Técnico, Mountain View, CA, USA, 1997.
- SULLIVAN, K.; GU, L.; CAI, Y. Non-modularity in aspect-oriented languages: integration as a crosscutting concern for aspectj. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, New York, NY, USA: ACM Press, 2002, p. 19–26.
- TURNER, C. D.; ROBSON, D. J. The state-based testing of object-oriented programs. In: *Conference on Software Maintenance*, Montreal Quebec, Canada: IEEE Computer Society Press, 1993, p. 302–310.
- VINCENZI, A. M. R. *Orientação a objeto: Definição, implementação e análise de recursos de teste e validação*. Tese de Doutorado, ICMC/USP, São Carlos, SP - Brasil, 2004.
- VINCENZI, A. M. R.; DELAMARO, M. E.; MALDONADO, J. C.; WONG, W. E. Establishing structural testing criteria for java bytecode. *Softw. Pract. Exper.*, v. 36, n. 14, p. 1513–1541, 2006.
- VINCENZI, A. M. R.; DOMINGUES, A. L. S.; DELAMARO, M. E.; MALDONADO, J. C. Introdução ao teste de software. cap. 6 - Teste Orientado a Objetos e de Componentes, Campus-Elsevier, p. 119–174, 2007.
- VINCENZI, A. M. R.; MALDONADO, J. C.; WONG, W. E.; DELAMARO, M. E. Coverage testing of java programs and components. *Science of Computer Programming*, v. 56, n. 1-2, p. 211–230, 2005.

- VINCENZI, A. M. R.; NAKAGAWA, E. Y.; MALDONADO, J. C.; DELAMARO, M. E.; ROMERO, R. A. R. Bayesian-learning based guidelines to determine equivalent mutants. *International Journal of Software Engineering and Knowledge Engineering - IJSEKE*, v. 12, n. 6, p. 675–689, 2002.
- VINCENZI, A. M. R.; WONG, W. E.; DELAMARO, M. E.; MALDONADO, J. C. Jabuti: A coverage analysis tool for java programs. In: *Anais do 17º Simpósio Brasileiro de Engenharia de Software (SBES 2003), Sessão de Ferramentas*, Manaus/AM - Brasil, 2003, p. 79–84.
- WEYUKER, E. J. On testing non-testable programs. *Computer Journal*, v. 25, n. 4, p. 465–470, 1982.
- WEYUKER, E. J. The complexity of data flow criteria for test data selection. *Inf. Process. Lett.*, v. 19, n. 2, p. 103–109, 1984.
- WEYUKER, E. J. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, v. 31, n. 6, p. 668–675, 1988.
- WEYUKER, E. J. The cost of data flow testing: An empirical study. *IEEE Trans. Softw. Eng.*, v. 16, n. 2, p. 121–128, 1990.
- WHALEY, J.; RINARD, M. Compositional pointer and escape analysis for java programs. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999)*, New York, NY, USA: ACM Press, 1999, p. 187–206.
- WHITE, L. J.; SAHAY, P. N. A computer system for generating test data using the domain strategy. In: *Softfair II*, San Francisco, CA, 1985, p. 38–45.
- WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
- XIE, T.; ZHAO, J. A framework and tool supports for generating test inputs of aspectj programs. In: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, New York, NY, USA: ACM, 2006, p. 190–201.
- XIE, T.; ZHAO, J.; MARINOV, D.; NOTKIN, D. Automated test generation for AspectJ programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD 2005*, Chicago/IL, USA, 2005.
- XU, D.; XU, W. State-based incremental testing of aspect-oriented programs. In: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, New York, NY, USA: ACM Press, 2006, p. 180–189.
- XU, D.; XU, W.; NYGARD, K. *A state-based approach to testing aspect-oriented programs*. Technical report, Computer Science Department, North Dakota State University, 2004.

- XU, D.; XU, W.; NYGARD, K. A state-based approach to testing aspect-oriented programs. In: *Proc. of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE 2005)*, Taiwan, 2005, p. 14–16.
- XU, G.; ROUNTEV, A. Regression test selection for AspectJ software. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, Washington, DC, USA: IEEE Computer Society, 2007, p. 65–74.
- XU, G.; ROUNTEV, A. AJANA: a general framework for source-code-level interprocedural dataflow analysis of aspectj software. In: *Proc. of the 7th International Conference on Aspect-Oriented Software Development (AOSD 2008)*, USA: ACM Press, 2008, p. 36–47.
- ZHANG, S.; ZHAO, J. On identifying bug patterns in aspect-oriented programs. In: *Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1 (COMPSAC 2007)*, Washington, DC, USA: IEEE Computer Society, 2007, p. 431–438.
- ZHAO, J. Tool support for unit testing of aspect-oriented software. In: *OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development*, Seattle/WA - USA, 2002.
- ZHAO, J. Data-flow-based unit testing of aspect-oriented programs. In: *Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2003)*, Dallas/Texas - USA: IEEE Computer Society, 2003, p. 188–197.
- ZHOU, Y.; RICHARDSON, D.; ZIV, H. Towards a practical approach to test aspect-oriented software. In: *Proceedings of the 2004 Workshop on Testing Component-based Systems (TECOS 2004)*, 2004, p. 1–16.
- ZHU, H.; HALL, P.; MAY, J. Software unit test coverage and adequacy. *ACM Computing Surveys*, v. 29, n. 4, p. 366–427, 1997.